Bulletproof Java Code: A Practical Strategy for Developing Functional, Reliable, and Secure Java Code

Java software drives the application logic for everything from Web services, to traditional Web applications, to desktop applications, to embedded devices. In these contexts, reliability, security, and performance problems can be just as serious as functionality problems. Yet, few Java development teams have the resources to ensure that their code is free of functionality problems, let alone also worry about reliability, security, and performance. Whether or not your team currently performs functional testing, you're taking several significant risks if you have not yet implemented a comprehensive team-wide quality control strategy:

- Your code might cause the application to become unstable, produce unexpected results, or even crash when the application is used in a way that you and QA did not anticipate (and did not test).
- Your code might open the only door that an attacker needs to manipulate the system and/or access privileged information.
- The functionality that you worked so hard to design, implement, and verify might be broken when other team members add and modify code.

This paper explains a simple four-step strategy that has been proven to make Java code more reliable, more secure, and easier to maintain—as well as less likely to experience functionality problems:

1. As you write code, comply with development rules for improving code functionality, security, performance, and maintainability.
2. Immediately after each piece of code is completed or modified, use unit-level reliability testing to verify that it's reliable and secure.
3. Immediately after each piece of code is completed or modified, use unit-level functional testing to verify that it's implemented correctly and functions properly.
4. Use regression testing to ensure that each piece of code continues to operate correctly as the code base evolves.

All four steps can be automated to promote a quick implementation and allow your team to reap the potential benefits without disrupting your development efforts or adding overhead to your already hectic schedule. When appropriate, this paper introduces ways that Parasoft Jtest facilitates the recommended tasks; however, it focuses on explaining the general four-step strategy, not discussing Jtest features or usage in depth. If you would like a general overview of Jtest or specific details on how to use it, visit http://www.parasoft.com.

As you read this paper, remember that the strategy and practices discussed are designed for team-wide application. Unless they are applied consistently across an entire development team, they will not significantly improve the software that the team is building. Having a development team inconsistently apply software development standards/best practices as it implements code is like having an electrician team inconsistently apply electrical standards/best practices as it implements a new building's electrical system. In both cases, the team members' work will interact to form a single system. Consequently, any hazards, problems, or even quirks introduced by one "free spirit" team member who ignores the applicable guidelines and best practices can make the entire system unsafe, unreliable, or difficult to maintain/upgrade.

1. Comply with development rules for improving code functionality, security, performance, and maintainability

The first step in ensuring that your code will not jeopardize the application's functionality, reliability, or security is to comply with applicable development rules as you write it. Many developers think that complying with development rules involves just beautifying code. However, there is actually a wealth of available Java development rules that have been proven to improve code functionality, security, performance, and maintainability. In addition, each team's experienced developers have typically developed their own (often informal) rules that codify the application-specific lessons they've learned over the course of the project.

Why is it important?

The key benefits of complying with applicable development rules are:

- *It cuts development time and cost by reducing the number of problems that need to be identified, diagnosed, and corrected later in the process.*

  Complying with meaningful development rules prevents serious functionality, security, and performance problems. Each defect that is prevented by complying with development rules means one less defect that the team needs to identify, diagnose, correct, and recheck later in the development process (when it's exponentially more time-consuming, difficult, and costly to do so). Or, if testing does not expose every defect, each prevented defect could mean one less defect that will impact the released/deployed application. On average, one defect is introduced for each ten lines of code (A. Ricadela, "The State of Software", *InformationWeek*, May 2001) and over half of a project's defects can be prevented by complying with development rules (R.G. Dromey, "Software Quality – Prevention Versus Cure", *Software Quality Institute*, April 2003). Do the math for a typical program with millions of lines of code, and it's clear that preventing errors with development rules can save a significant amount of resources. And considering that it takes only 3 to 4 defects per 1,000 lines of code to affect the application's reliability (A. Ricadela, "The State of Software", *InformationWeek*, May 2001), it's clear that ignoring the defects is not an option.

- *It makes code easier to understand, maintain, and reuse.*

  Different developers naturally write code in different styles. Code with stylistic quirks and undocumented assumptions probably makes perfect sense to the developer as he's writing it, but may confuse other developers who later modify or reuse that code—or even the same developer, when his original intentions are no longer fresh in his mind. When all team members write code in a standard manner, it's easier for each developer to read and understand code. This not only prevents the introduction of errors during modifications and reuse, but also improves developer productivity and reduces the learning curve for new team members.

What's required to do it?

a. Decide which development rules to comply with.

First, review industry-standard Java development rules and decide which ones would prevent your project's most serious or common defects. The rules implemented by automated Java static analysis tools offer a convenient place to start. If needed, you can supplement these rules with the ones listed in books and articles by Java experts. As you are deciding which rules to comply with, aim for quality over quantity to ensure that the team members get the greatest benefit for

the least work. The point of having the team comply with development rules is to help the team write better code faster. Checking compliance with too many insignificant rules could defeat that purpose.

Next, think about rules that are unique to your organization, team, and project. Are there explicit rules for formatting or naming conventions that your team is expected to comply with? Do your most experienced team developers have an informal list of lessons learned from past experiences?

Simplifying this task

If you're using Parasoft Jtest to check whether your code complies with development rules, you can take a shortcut on this step: adopt Jtest's core set of critical rules, then add to it any critical organizational and application-specific rules that your team is currently following.

The core set of critical rules includes approximately 50 rules that have been proven to make an immediate and significant improvement to code. By working with Java development teams worldwide, we have learned that Java software that complies with this core set of rules will be faster, more secure, easier to maintain, and less likely to experience functional problems. The core set includes rules such as:

- Reliability rules: Avoid dangling "else" statements, Avoid "try", "catch" and "finally" blocks with empty bodies, Do not assign loop control variables in the body of a "for" loop
- Security rules: Do not compare Class objects by name, Do not pass byte arrays to DataOutputStream in the 'writeObject()' method, Make your 'clone ()' method "final" for security
- Performance rules: Close input and output resources in "finally" blocks, Prevent potential memory leaks in ObjectOutputStreams by calling 'reset ()', Use 'String' instead of 'StringBuffer' for constant strings
- Maintainability rules: Provide Javadoc comments and descriptions for classes and interfaces, Use chain constructors in classes with multiple constructors, Declare package-private methods as inaccessible as possible

If you want to move beyond the core set of rules, Jtest includes 500+ development rules developed by Java gurus such as Sun Microsystems, Scott Ambler, and Joshua Bloch. Most rules prevent serious functionality, security, or performance problems; others make code easier to maintain and reuse. To help you determine which rules to comply with, rules are categorized by topic (for instance, security, optimization, initialization, garbage collection, JDBC, EJB, servlets, JSP, and so on) as well as ranked by severity.

b. Configure all team tools to check the designated rules consistently.

To fully reap the potential benefits of complying with development rules, the entire development team must check the designated set of rules consistently. Consistency is required because even a slight variation in tool settings among team members could allow non-compliant code to enter the team's shared code base. If the team has carefully selected a set of meaningful development rules to comply with, just one overlooked rule violation could cause serious problems. For instance, assume a developer checks in code that does not comply with rules for closing external resources. If your application keeps temporary files open until it exits, normal testing—which can

last a few minutes or run overnight—won't detect any problems. However, when the deployed application runs for a month, you can end up with enough temporary files to overflow your file system, and then your application will crash.

Simplifying this task

Jtest automatically manages the sharing and updating of standard team test settings and files to provide all team members a hassle-free way to perform tests uniformly. With Jtest, the team-wide configuration process begins with the team architect or lead developer configuring a test scenario to check the exact set of development rules that the team has decided to comply with. First, the architect enables/disables Jtest's built-in rules according to the team's designated rule list, then customizes those rules' parameters, names, and categories as needed. Most teams prefer to select and customize one of the preconfigured sets of rules included with Jtest rather than define a custom set of rules from scratch. The architect can also configure Jtest to check only code added or modified after a designated cutoff date, and/or limit the number of rule violations reported per test. This prevents rule compliance from becoming an overwhelming task.

If it's necessary to create additional rules to check application-specific or organizational rules, these rules can be created graphically or automatically with Jtest's RuleWizard module.

The next step is ensuring that all team developers are equipped to check the same set of rules in the same way. With Jtest, the necessary sharing and updating is managed automatically through Parasoft Team Configuration Manager (TCM). The architect adds the team-standard development rule settings and files to TCM, then they are automatically distributed to all developer Jtest installations that are connected to TCM. If the architect later decides to modify, add, or remove settings and related files, TCM makes the appropriate updates on all of the team's Jtest installations. This way, developers do not need to waste time configuring settings or copying/pasting files. Further, there is no risk of developers checking compliance with an outdated or incomplete set of development rules.

c. Check new/modified code before adding it to source control.

Study after study has shown that the earlier a problem is found, the faster, easier, and cheaper it is to fix. That's why the best time to check whether code complies with development rules is as soon as it's written or updated. If you check whether each piece of code complies with the designated development rules immediately, while the code is still fresh in your mind, you can then quickly resolve any problems found and add it to source control with increased confidence.

Simplifying this task

You can use your desktop version of Jtest to test your "in progress" code and ensure that it complies with the designated team development rules before you add it to source control. These tests can be configured to run automatically each night, or they can be run as needed during the work day. Either way, Jtest provides the type of "constructive criticism" you would hope to receive from code reviews or pair programming—but without requiring your team members to take time away from their own work. Plus, Jtest's Quick Fix feature makes correcting most violations as fast and easy as identifying them.

For example, assume that you want to check whether a servlet complies with your team's designated development rules. After you click Jtest's Play button, Jtest automatically checks whether the code complies with the designated team development rules. In this case, Jtest identifies a violation of the "Specify an initial 'StringBuffer' capacity" rule. `StringBuffer` allocates only a 16-character buffer by default; if that capacity is exceeded, the `StringBuffer` class allocates a longer array and copies the contents to the new array. If we specify the initial size—by manually modifying the code, or by using the Quick Fix feature—all those allocations, copies, and garbage collections are avoided, and the code is optimized. Because this servlet is used repeatedly in the application, this optimization will have a significant effect on overall application performance.

d. Check all new/modified code in the team's shared code base on a nightly basis.

Even if all team members intend to check and correct code before adding it to source control, code with problems might occasionally slip into the team's shared code base. To maintain the integrity of the team's shared code base, you schedule your testing tool to automatically check the team's code base at a scheduled time each night.

Simplifying this task

With Jtest, this nightly checking can be set up easily. First, configure a team "Jtest Server" installation to check the code from the command line interface. Then, use cron or the Windows Scheduled Tasks functionality to ensure that the test runs automatically each night. If a problem is detected, Jtest will email the responsible developer a report that explains the discovered problem. In addition, a comprehensive report will be sent to managers and be available for team review.

2. Use reliability testing to verify that each piece of code is reliable and secure

The next step toward reliable and secure code is to perform unit-level reliability testing (also known as white-box testing or construction testing). In Java, this involves exercising each method as thoroughly as possible and checking for uncaught runtime exceptions.

Why is it important?

If your unit testing only checks whether the unit functions as expected, you can't predict what could happen when untested paths are taken by well-meaning users exercising the application in unanticipated ways—or taken by attackers trying to gain control of your application or access to privileged data. It's hardly practical to try to identify and verify every possible user path and input. However, it's critical to identify the possible paths and inputs that could cause uncaught runtime exceptions because:

- *Uncaught runtime exceptions can cause application crashes and other serious runtime problems.*
  Uncaught runtime exceptions—exceptions that are automatically thrown by the Java runtime system when a program violates the syntax/semantics of Java—usually indicate software bugs. They typically stem from problems related to arithmetic, pointers, and indexing, and can occur at any point in a program. If these exceptions surface in the field, the resulting unexpected flow transfer and potential thread termination could lead to instability, unexpected results, or crashes. In fact, Parasoft has worked with many Java development teams who had trouble with Java-based applications crashing for unknown reasons. Once these teams started identifying and correcting the uncaught runtime exceptions that they previously overlooked, their applications stopped crashing.

- *Uncaught runtime exceptions can open the door to security attacks.*

  Many developers don't realize that uncaught runtime exceptions can also create significant security vulnerabilities. For instance, a NullPointerException in login code could allow an attacker to completely bypass the login procedure.

What's required to do it?

a. Design, implement, and execute reliability test cases.

To identify potential uncaught runtime exceptions, you test each class's methods with a large number and range of potential inputs, then check whether uncaught runtime exceptions are thrown.

> *Simplifying this task*
>
> Jtest generates reliability test cases automatically. To prompt Jtest to automatically generate test cases, tell it which class or set of classes to test, then click the Start button. Jtest will then use its patented test case generation technology to try to exercise the code as thoroughly as possible and expose uncaught runtime exceptions. Jtest aims to create test cases that execute every possible branch of each method it tests. For example, if the method contains a conditional statement (such as an `if` block), Jtest will generate test cases that test the `true` and `false` outcomes of the `if` statement. Jtest understands Java code and creates intelligent test cases that make realistic method invocations.

b. Review and address all reported exceptions.

All uncaught runtime exceptions exposed by the tests should be reviewed and addressed before proceeding. Each method should be able to handle any valid input without throwing an undocumented uncaught runtime exception. If code should not throw an uncaught runtime exception for a given input, the code should be corrected now, before you (or a team member) unwittingly introduce additional errors by adding code that builds upon or interacts with the problematic code. If the exception is expected or if the test inputs are not expected/permissible, document those requirements in the code. When other developers working with the code know exactly how the code is supposed to behave, they will be less likely to introduce errors.

c. Check all new/modified code in the team's shared code base on a nightly basis.

Same reason and procedure as step 1.d.

3. Use functional testing to verify that each piece of code is implemented correctly and operates properly

Next, extend your reliability test cases to verify each unit's functionality. The goal of unit-level functional testing is to verify that each unit is implemented according to specification before that unit is added to the team's shared code base.

Why is it important?

The key benefit of verifying functionality at the unit level is that it allows you to identify and correct functionality problems as soon as they are introduced, reducing the number of problems that need to be identified, diagnosed, and corrected later in the process. Finding and fixing a unit-level functional error immediately after coding is easier, faster, and from 10 to 100 times less costly than finding and fixing that same error later in the development process. When you perform functional testing at the unit level, you can quickly identify simple functionality problems (such as a "+" substituted for a "-") because you are verifying the unit directly. If the same problem entered the shared code base and became part of a multi-million line application, it might surface only as strange behavior during application testing. Here, finding the problem's cause would be like searching for a needle in a haystack. Even worse, the problem might not be exposed during application testing and remain in the released/deployed application.

What's required to do it?

a. Leverage Design by Contract to automatically create basic functional test cases (recommended, but not required).

Many Java developers have found that by adding specification requirements to the code in the form of simple Design by Contract (DbC) comments, then having an automated unit testing tool such as Jtest automatically generate test cases that verify this functionality, they can produce a foundational functional unit test suite with minimal effort. DbC comments are like Javadoc comments, and are simple to add. For example, if your specification requires that the `month` variable is always between 1 and 12, you simply add the comment `/** @pre month >= 1 && month <= 12 */`

> *Simplifying this task*
>
> Jtest reads DbC-format specification information during the test case generation process, then automatically develops and executes test cases that verify the documented requirements. To make the contract-writing process as fast and effortless as possible, Jtest's automated DbC checking will help you determine where to add DbC comments and how much DbC documentation is enough.

b. Add and execute more functional test cases as needed to fully verify the specification.

Next, compare the automatically-generated test cases to the unit-level requirements detailed in your specification. To ensure that each requirement is implemented correctly now and is not broken by future modifications, extend the functional test suite as needed until it contains at least one test case for each requirement.

Simplifying this task

With Jtest, you can jumpstart your functional testing by reviewing the unverified results of Jtest's automatically-generated reliability test cases. If you choose to review these test cases, you tell Jtest whether they produced the correct outcomes and—if not—specify the correct outcomes. When the classes are retested after test case validation, Jtest will report these test cases as "failed" if and only if the given inputs do not produce the correct outputs. The verified test cases will be treated specially by Jtest, and they will not be overwritten. They will become a permanent part of your functional regression test suite, where they are used to verify that modifications do not introduce problems.

There are several ways to add the realistic functional test cases needed to ensure that new or modified code implements the functionality described in the specification. The recommended technique is to use Jtest Tracer, which monitors a running application and automatically generates unit test cases that capture application behavior. Simply exercise the application functionality you want to test, and Jtest Tracer automatically designs JUnit test cases with real data that represents the paths taken through the application. No coding or scripting is required. The result is a library of test cases against which new code can be tested to ensure it meets specifications and does not “break” existing functionality.

Alternatively, you can use create functional test cases using Jtest's graphical test case editor, which lets you specify test cases without writing or modifying code. If running the same test case with different method inputs will help you verify functionality, you can define specific input values in a data source or define a value range in the GUI, then set up one "template" test case. During testing, Jtest will cycle through the full input set, creating and running multiple test cases based on the one "template" test case that you defined. In this way, you can test a large number of user-defined inputs without having to configure more than one test case.

Or, if you're familiar with JUnit, you might prefer to add your additional test cases by manually extending the automatically-generated JUnit test classes.

4. Use regression testing to ensure that each piece of code continues to operate correctly as the code base evolves

Finally, collect all of the project's existing test cases to create an automated regression test suite that verifies whether each unit continues to function as expected when the code base grows and evolves.

Why is it important?

The key benefit of performing unit-level regression testing is to ensure that code additions and changes do not break the verified functionality. In the rush to accommodate last minute requests, developers often unknowingly change or break previously-verified functionality. Moreover, previously-verified functionality is often impacted by the code modifications made during routine maintenance. In fact, studies have shown that, on average, 25 percent of software defects are introduced while developers are changing and fixing existing code during maintenance. (R.B. Grady, *Software Process Improvement*, Prentice Hall, 1997).

Why are code additions and modifications so problematic? With complex software, even a seemingly innocuous change in one part of the application can impact other functionality. However, these changes are difficult to detect without a thorough unit-level regression test suite. Application testing might catch obvious problems that affect the application's interface, but subtle internal problems could easily go unnoticed.

What's required to do it?

a. Configure the regression test to run nightly in the background.

The regression test should check the team's shared code base by executing all applicable functional and reliability test cases. To ensure that regression testing is performed regularly and unobtrusively, schedule your unit testing tool to automatically run the complete test suite at a scheduled time each night.

> *Simplifying this task*
>
> With Jtest, automated nightly regression testing can be set up by configuring a designated team "Jtest Server" installation to run a regression test from the Jtest command line interface (jtestcli), then using cron or the Windows Scheduled Tasks functionality to run the test automatically each night. To ensure that the complete regression test suite is run, Jtest will automatically access all applicable test cases, test objects, user-defined stubs, and other test assets as long as they are added to source control along with the project.
>
> When the team arrives at work each morning, all testing will be completed and results will be ready for review. If a problem is introduced, Jtest will email the responsible developer a report that explains the problem found. In addition, a comprehensive report will be sent to managers and be available for team review.

b. Update the regression test suite as needed.

If the regression test suite is not kept current, it might report annoying false positives or overlook true errors. When code functionality changes intentionally, some of the existing test cases may need to be updated to reflect new expected outcomes, reflect removed or renamed classes/methods, and so on. When new code is added, new test cases will need to be written for that code.

> *Simplifying this task*
>
> With Jtest, you do not need to manually update the regression test suite. Jtest automatically monitors code modifications, then updates and refactors the test suite as needed when classes/methods are renamed, moved, and so on. For instance, if new methods are added to the code, new test cases are added automatically; you do not have to tell Jtest to recreate test cases or alert it to the code modifications. If methods are removed, obsolete unverified test cases are deleted. If code is modified in a way that should not affect the test suite, Jtest understands this and does not try to recreate new test cases. Instead, it reruns the existing functional regression test suite so that it can alert you if modifications cause the class behavior to change. Jtest works closely with the source control system, so it not only recognizes what classes were modified, but also what specific lines were modified. As a result, it's sensitive to code changes, but does not recreate its test suite needlessly, which wastes resources and reduces the effectiveness of your regression testing.

Conclusion

This paper explained a four-step strategy that has been proven to make Java code more reliable, more secure, easier to maintain, and less likely to experience functional problems. Other desirable side effects of performing this strategy consistently across a Java development team include:

- Developers spend less time finding and fixing errors, which means less "crunch time" at the end of the project and more time for more challenging and interesting tasks such as developing and implementing new technologies.

- The QA team no longer has to chase implementation-level errors, and has more time to dedicate to higher-level verification.

- Releases are more likely to occur on time, on budget, and with the expected functionality.

By applying this strategy with an automated technology such as Parasoft Jtest, development teams can significantly improve the quality of their software without disrupting their development efforts or adding overhead to their already hectic schedule. With Jtest, implementing the recommended strategy is fast and simple, regardless of your team's size/organization, projects, or development methodology. Essentially, there is nothing to lose, and much to gain. To learn more about how you can use Parasoft Jtest to implement the strategies discussed in this paper, or to try Jtest for free, contact Parasoft.

Jtest can also be used to help development and QA organizations test large and complex code bases. For details on how Jtest supports this strategy, see the Parasoft white paper "Best Practices for Improving the Functionality, Reliability, Security, and Performance of a Large and Complex Java Code Base," which is available at http://www.parasoft.com/jtest.

About Parasoft

Parasoft is the leading provider of innovative solutions for automated software test and analysis and the establishment of software error prevention practices as an integrated part of the software development lifecycle. Parasoft's product suite enables software development and IT organizations to significantly reduce costs and delivery delays, ensure application reliability and security and improve the quality of the software they develop and deploy through the practice of Automated Error Prevention (AEP). Parasoft has more than 10,000 clients worldwide including: Bank of America, Boeing, Cisco, Disney, Ericsson, IBM, Lehman Brothers, Lockheed, Lexis-Nexis, Sabre Holdings, SBC and Yahoo. Founded in 1987, Parasoft is headquartered in Monrovia, CA. For more information visit: http://www.parasoft.com.

Contacting Parasoft

USA

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

Europe

France: Tel: +33 (1) 64 89 26 00
UK: Tel: +44 (0)1923 858005
Germany: Tel: +49 7805 956 960
Email: info-europe@parasoft.com

Asia

Tel: +886 2 6636-8090
Email: info-psa@parasoft.com