Integrated Error-Detection Techniques:
Find More Bugs in Java Applications
Software verification techniques such as pattern-based static code analysis, runtime error detection, unit testing, and flow analysis are all valuable techniques for finding bugs in Java web applications. On its own, each technique can help you find specific types of errors. However, if you restrict yourself to applying just one or some of these techniques in isolation, you risk having bugs that slip through the cracks. A safer, more effective strategy is to use all of these complementary techniques in concert. This establishes a bulletproof framework that helps you find bugs which are likely to evade specific techniques. It also creates an environment that helps you find functional problems, which can be the most critical and difficult to detect.

This paper will explain how automated techniques such as pattern-based static code analysis, runtime error detection, unit testing, and flow analysis can be used together to find bugs in a Java web application. These techniques will be demonstrated using Parasoft JTest, an integrated solution for automating a broad range of best practices proven to improve software development team productivity and software quality.

As you read this paper—and whenever you think about finding bugs—it’s important to keep sight of the big picture. Automatically detecting bugs such as exceptions, race conditions, and deadlocks is undoubtedly a vital activity for any development team. However, the most deadly bugs are functional errors, which often cannot be found automatically. We’ll briefly discuss techniques for finding these bugs at the conclusion of this paper.

Introducing the Scenario
To provide a concrete example, we will introduce and demonstrate the recommended bug-finding strategies in the context of an e-commerce website: the JPetStore demo.

Assume that an end user reports a bug: although the online shopping cart should be aggregating similar items and increasing the quantity, it actually keeps multiple requests for the same item separate. It is not surprising that this bug made it past QA; it does not block online purchases, and thus could be perceived as a minor annoyance. Development is notified of the problem, but they are not sure why it is occurring. They claim that code was written to handle this exact case.

Development can try to debug it, but debugging on the production server is time-consuming and tedious. They would need to step through each statement of business logic as it executes in hopes of spotting the point where the runtime behavior deviates from their plan. Even if they find the point where plan and reality diverge, the underlying cause may not be apparent. Alternatively, they might apply certain tools or techniques proven to pinpoint errors automatically.
At this point, the developers can start crossing their fingers as they try to debug the application with the debugger. Or, they can apply an automated testing strategy in order to peel errors out of the code. If the application is still not working after they try the automated techniques, they can then go to the debugger as a last resort.

Problem Report

To reproduce this problem in the online pet store application, add a pet to the shopping cart, and then add the same pet again. For example:

1. Add a goldfish to the shopping cart.
2. Add another goldfish to the shopping cart. The cart shows two separate goldfish items—each with a quantity of one.

The intended behavior is for the shopping cart to show a single line item for goldfish, and for that line to have a quantity of two.
Reproducing the Problem Scenario with Jtest Tracer

Based on previous experience fixing defects, the developers know that this problem scenario will need to be recreated several times during the course of troubleshooting, fixing, and verifying the reported problem. This can be done much faster with an automated test that reproduces the problem. With a unit test that fails in isolation until this specific problem is fixed, the developers won’t need to re-deploy to the application server for manual testing.

This functional unit testing can be facilitated with Jtest Tracer, which automates the process of building unit tests based on recorded manual interactions with an application. It attaches to the Java Virtual Machine (JVM) as the application is running and records method calling sequences and input values to be used later in unit test generation. The resulting tests replicate specific usage scenarios that can be repeated in scheduled automated testing.
For example, the following Tracer test shows the exact inputs for the original scenario (adding two items with identical id "EST-20" and price 5.50), and it asserts the current incorrect behavior for two cart items with a quantity of 1 for each. It is a JUnit functional test that automates the steps to reproduce the reported problem:

```java
List result = testedObject.getCartItemList().getSource();
assertNotNull(result); // jtest_unverified
assertEquals(2, result.size()); // jtest_unverified
assertEquals("(EST-20-FI-FW-02) x 1", result.get(0).toString());
assertEquals("(EST-20-FI-FW-02) x 1", result.get(1).toString());
```

Once the JUnit tests are generated for the current incorrect behavior, the assertions can be modified to check for the ideal expected results. For example, to modify the test to check for the correct result rather than the problematic behavior, the developers modify the generated assertions by hand. According to the problem report, the correct result is only one line item in the cart with its quantity set to two, so they modify it as follows:

```java
List result = testedObject.getCartItemList().getSource();
assertNotNull(result); // jtest_unverified
assertEquals(1, result.size()); // jtest_unverified
assertEquals("(EST-20-FI-FW-02) x 2", result.get(0).toString());
```

The test now fails when it is run. This failure is a quick indication that the problem has not yet been resolved.
Performing Pattern-Based Static Code Analysis

Static analysis checks for known anti-patterns in source code constructs and reports each occurrence of a match. Pattern-based static analysis can be performed quickly and is guaranteed to find all cases of code that match a pattern. Avoiding certain software anti-patterns is a great way to eliminate categories of defects from a project.

Let’s assume that the developers for this online pet store don’t want to take the debugging route unless it’s absolutely necessary, so they start trying to track down the problem by running pattern-based static code analysis. It finds one problem:
This is a violation of a Java Collections API rule that says all object keys to HashMaps should implement ‘equals()’ and ‘hashCode()’. Indeed, this is exactly the problem. A new instance of Item is allocated for each web request, so two requests to add the same item do not actually use the same item instance. A proper implementation of ‘equals()’ and ‘hashCode()’ will let the HashMap see that the two instances are equivalent.

This problem is fixed by adding equals and hashCode methods in Item based on the item id. The functional test created by Jtest Tracer at the start of this exercise now passes.

Performing Runtime Error Detection on the Complete Application

Next, the web application is redeployed to a test server. QA tests the scenario repeatedly and reports that it now works correctly most of the time, but occasionally lists the two identical items separately. The problem is now much more difficult to pinpoint because it is no longer deterministic, most likely due to a race condition. The developers could try to use the debugger with breakpoints simulating thread context switches, or they could continue applying automated error detection techniques.

Fortunately, there is an automated technique for uncovering race conditions and other concurrency problems in a running application: automated runtime error detection. Jtest performs runtime error detection on the complete system using a runtime agent in the JVM. This technology acts like an intelligent debugger looking for bad patterns of sequences and values at runtime. The runtime agent instruments Java classes as they are loaded to enforce a number of runtime rules for correctness, robustness, and optimization. If one of these rules is triggered, an error is reported back to the IDE:
This indicates that two threads are accessing the same unsynchronized map. The reported violation shows exactly which two threads accessed which unsynchronized map—and along what runtime path. This is valuable information that would not come easily from a debugger.

Java maps and collections may become corrupted if accessed simultaneously without synchronization. A quick solution is to wrap the block of code that includes map get and put statements with a synchronized block. This way, it will be impossible for two threads to both enter the branch for creating a new CartItem at the same time and for the same Item.

```java
public void addItem(Item item, boolean isInStock) {
    synchronized(itemMap) {
        CartItem cartItem = (CartItem) itemMap.get(item);
        if (cartItem == null) {
            cartItem = new CartItem();
            cartItem.setItem(item);
            cartItem.setQuantity(0);
            cartItem.setInStock(isInStock);
            itemMap.put(item, cartItem);
            itemMap.get().add(cartItem);
        }
        cartItem.incrementQuantity();
    }

    public Item removeItem(Item item) {
        CartItem cartItem = (CartItem) itemMap.remove(item);
        if (cartItem == null) {
            return null;
        } else {
            itemMap.get().remove(cartItem);
            return cartItem.getItem();
        }
    }
```

This indicates that two threads are accessing the same unsynchronized map. The reported violation shows exactly which two threads accessed which unsynchronized map—and along what runtime path. This is valuable information that would not come easily from a debugger.

Java maps and collections may become corrupted if accessed simultaneously without synchronization. A quick solution is to wrap the block of code that includes map get and put statements with a synchronized block. This way, it will be impossible for two threads to both enter the branch for creating a new CartItem at the same time and for the same Item.

```java
public void addItem(Item item, boolean isInStock) {
    synchronized(itemMap) {
        CartItem cartItem = (CartItem) itemMap.get(item);
        if (cartItem == null) {
            cartItem = new CartItem();
            cartItem.setItem(item);
            cartItem.setQuantity(0);
            cartItem.setInStock(isInStock);
            itemMap.put(item, cartItem);
            itemMap.get().add(cartItem);
        }
        cartItem.incrementQuantity();
    }
```

itemMap.put(item, cartItem);
itemList.getSource().add(cartItem);
}
cartItem.incrementQuantity();
}

Now, when the application is rerun, no more runtime threading errors are reported. After redeploying the application, it seems to work as expected. However, the application-level coverage report shows that the ‘removeItem()’ method was not covered.

Performing Unit Testing with Runtime Error Detection

The problem report thus far has focused around adding the same item to a shopping cart twice… but what about other scenarios? Jtest reported that the ‘removeItem()’ method was still not covered. Removing items from the cart needs to be tested as well, but manual tests can be tedious—even manual tests that are traced and automatically converted to JUnit. The team has another choice to make: perform more manual testing or take advantage of automated unit testing to check the remaining business logic code.

Jtest is used to generate unit tests for the uncovered method and other methods. Jtest’s automatically-generated tests cover many input cases and branches that QA typically would not think to cover. Moreover, when these unit tests are executed through Jtest, runtime error detection can expose runtime problems whether the test passes or fails. This entire operation takes just seconds with Jtest.

During test execution, Jtest not only checks the cases explicitly specified in the tests, but also validates that the ‘equals()’ method was implemented according to the Java specification. Even though the unit tests passed, a runtime error was still detected. The equality check between a live object and null should always return false. In this implementation, the ‘equals()’ method that was added earlier threw a NullPointerException instead of returning false. Even though the unit tests did not explicitly test for this case, Jtest’s runtime error detection performed the check on the Item object when it was used as a key to a map. Java HashMaps may contain null keys, so this contract on the ‘equals()’ method is important.

A similar runtime error was detected for the Product class implementation of ‘toString()’. Null values for ‘toString()’ can cause application crashes when extra logging or debugging is enabled. By acting as a more intelligent debugger, Jtest’s runtime error detection points out this problem without crashing as some other debuggers might.
To satisfy the Java API contracts, the developers add the necessary checks to these core methods. Running the test case one more time ensures that the code is robust and no more runtime errors are reported.

```java
class Item {
    private String itemId;

    public int hashCode() {
        return itemId == null ? 0 : itemId.hashCode();
    }

    public boolean equals(Object o) {
        if (o == null) || (o.getClass() != Item.class) {
            return false;
        } else {
            return itemId.equals(((Item)o).itemId);
        }
    }
}
```
Performing Flow Analysis

So far, a lot of defects have been uncovered through static analysis, unit testing, and runtime error detection. It is likely that even more defects are lurking in the code. Data flow analysis expands the scope of testing by simulating different paths through the system and checking for potential problems along those paths. The code under test is not actually executed as with unit testing; rather, all hypothetical paths are explored.

```
public double getSubTotal() {
    double subTotal = 0;
    for (Iterator it = itemMap.entrySet().iterator(); it.hasNext();)
        Map.Entry entry = (Map.Entry)it.next();
        double listPrice = ((Item)entry.getKey()).getListPrice();
        int quantity = ((CartItem)entry.getValue()).getQuantity();
        subTotal += listPrice * quantity;
    return subTotal;
}
```

Here, Jtest’s flow analysis found a path that could easily be optimized. The map of items to quantity is not used efficiently while computing the total price for the shopping cart. Specifically, a lookup into the map (order log n) is done for each item (order n) in the map, resulting in order n * log n performance hit as the number of items in the cart increases. An entry set iterator would reduce the overhead and let the routine scale linearly (order n) with the size of the cart.
Performing Regression Testing

To ensure that everything is still working, the developers re-run the entire analysis. First, they run unit testing with runtime error detection, and no runtime errors are reported. The functional test from tracing the problem scenario passes. Then, they run the application with runtime error detection, and everything seems fine. Finally, they review the regression suite and verify the outcomes before committing the code and tests into source control.

In addition to rerunning the existing test cases (both automatically-generated and Tracer-generated), Jtest noticed that the code changed since tests were last generated, then added a new unit test for the ‘getSubTotal()’ method that had been optimized in response to flow analysis results. The test adds two items of different prices to the cart and computes the sub total. Once verified, this test becomes a valuable asset in checking for regressions to the ‘getSubTotal()’ method.

The combination of smaller unit tests and the functional test from tracing establish a robust regression suite that will detect future changes in both code behavior and use case scenarios. The test artifacts are just as valuable as the code fixes, and the two should be added to source control at the same time. Future development work will benefit from these comprehensive regression tests.
Wrap Up
To wrap up, let’s take a bird’s-eye view of the steps we just went over...

We had a problem report of our application not running as expected, and we had to decide between two approaches to resolving this: running in the debugger, or applying automated error detection techniques.

If we decided to run code through the debugger, we would have seen strange behavior: a map that shows an item to data relation not returning the data during a lookup. We would have had to deduce from this that the problem was actually caused by missing implementations of ‘equals()’ and ‘hashMap()’ in another class. Instead of stepping through a debugger, we traced the application to create a unit test that would reproduce the problem with less manual work. Then, we applied static analysis to detect the problem and propose a fix. The unit test confirmed that the fix actually worked.

After we fixed this problem, the application usually ran correctly, but it still would fail once in a while due to multi-threaded concurrency problems. Threading problems are very difficult to reproduce under a debugger because the debugger influences when the JVM will context switch between threads. Automated tools, however, are able to detect problematic thread patterns at runtime. Thus, runtime error detection was used to find these threading problems, instrumenting the Java classes as they were loaded and injecting code to check for violations of known runtime patterns. This reported exactly which threads were accessing which collections unsynchronized and along which call path.

However, upon reviewing the coverage analysis results, we learned that some of the code was not covered while monitoring the complete application. Getting this coverage information was simple since it was monitored automatically. Using a debugger, it would have been difficult to figure out exactly how much of the application we verified. This is typically done by jotting notes down on paper and trying to correlate everything manually.

Once the tool alerted us to this uncovered code, we decided to leverage unit testing (automated test generation plus runtime error detection) to add additional execution coverage to our testing efforts. Developing a unit test case to recreate partial or unexpected data is very often the only way to effectively test certain code. Indeed, this revealed yet another problem: the new methods added did not have proper null checks. In response to this finding, the new methods were corrected.

Then, flow analysis simulated paths that were not necessarily executed on the server—or even with the unit tests. Before this, testing yielded nearly 100% line coverage, but path coverage was not at the same level. Flow analysis uncovered a potential optimization around a loop. The optimization would not affect our simple tests with one or two shopping cart items, but a theoretically large online purchase would not scale well at all.

Just to be safe in case we ever receive large online orders, we fixed the reported problem to optimize calculating the shopping cart subtotal. Afterwards, we reviewed a regression test case for the optimized method by verifying the outcome (as one of the reported tasks guided us to do). Finally, we re-ran all the unit tests for the application, and everything seemed fine.

As you can see, all of the testing methods we applied—pattern-based static code analysis, runtime error detection, unit testing, flow analysis, regression testing, and functional testing—are not in competition
with one another, but rather complement one another. Used together, they are an amazingly powerful tool that provides an unparalleled level of automated error detection for Java web applications.

***

In sum, by automatically finding many bugs related to functional, multi-threading, performance and other coding errors, we were able to resolve problems reported against the application and make other improvements along the way. However, it would be best to catch functional errors (instances where the application is not working according to specification) before end users do. Unfortunately, these errors are much more difficult to find.

One of the best ways to find such bugs is through peer code reviews. With at least one other person inspecting the code and thinking about it in context of the requirements, you gain a very good assessment of whether the code is really doing what it’s supposed to.

Another helpful strategy is to create a regression test suite that frames the code with a specific use case in mind, enabling you to verify that it continues to adhere to specification. In the sample scenario described above, the application was not working properly, so we used Tracer to capture the incorrect behavior, then modified the test case to check for the expected behavior (so that later, as the code was modified, it would confirm that the problem had been fixed). Such unit test cases should really be created much earlier: ideally, before any changes are made to application code. Even if the functionality is not yet implemented, you can still start by developing a test that will frame the expected behavior…and fail until the related application code is implemented and functioning as expected. This strategy is the essence of test-driven development.

Parasoft Jtest assists with both of these tasks: from automating and managing the peer code review workflow, to helping the team establish, continuously run, and maintain an effective regression and functional test suite.

About Parasoft Jtest

Parasoft Jtest is an integrated solution for automating a broad range of best practices proven to improve software development team productivity and software quality. Jtest enables coding policy enforcement, static analysis, runtime error detection, automated peer code review, and unit and component testing to provide teams a practical way to ensure that their Java code works as expected. Jtest can be used both on the desktop under common development IDEs, as well as in batch processes via command line interface for regression testing. It also integrates with Parasoft’s reporting system, which provides interactive Web-based dashboards with drill-down capability, allowing teams to track project status and trends based on Jtest results and other key process metrics.

For more details on Jtest, visit Parasoft’s [Java Testing](http://www.parasoft.com/products/java-testing) solution center.
About Parasoft

Parasoft researches and develops software solutions that help organizations deliver defect-free software efficiently. To combat the risk of software failure while accelerating the SDLC, Parasoft offers a Development Testing Platform and Continuous Testing Platform. Parasoft’s enterprise and embedded development solutions are the industry's most comprehensive—including static analysis, unit testing, requirements traceability, coverage analysis, API testing, dev/test environment management, service virtualization and more. The majority of Fortune 500 companies rely on Parasoft in order to produce top-quality software consistently and efficiently as they pursue agile, lean, DevOps, compliance, and safety-critical development initiatives.

Contacting Parasoft

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

Global Offices
Visit www.parasoft.com/contact for contacting Parasoft in EMEAI, APAC, and LATAM.