Parasoft Runtime Error Detection Starter Kit

"Integrated Error Detection Techniques - Java" White Paper

"Integrated Error Detection Techniques - C" White Paper

Parasoft Data Sheets

Integrated Error-Detection Techniques: Find More Bugs in Java Applications

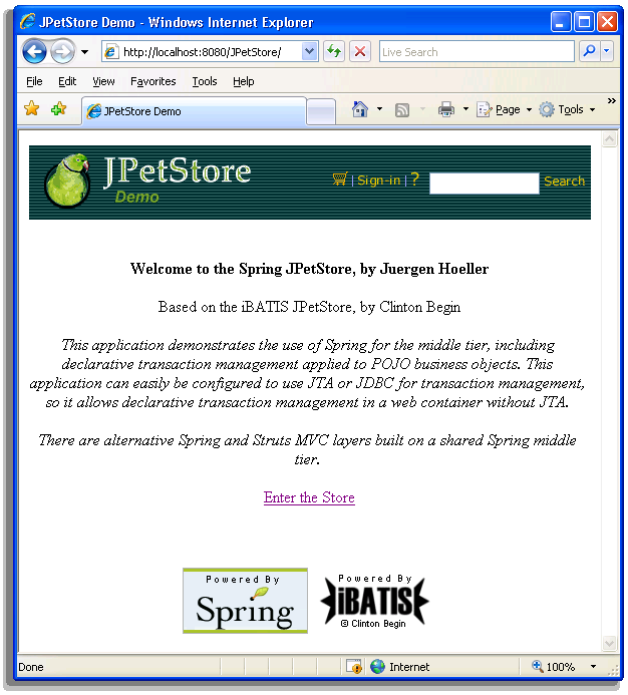
Software verification techniques such as pattern-based static code analysis, runtime error detection, unit testing, and flow analysis are all valuable techniques for finding bugs in Java web applications. On its own, each technique can help you find specific types of errors. However, if you restrict 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.

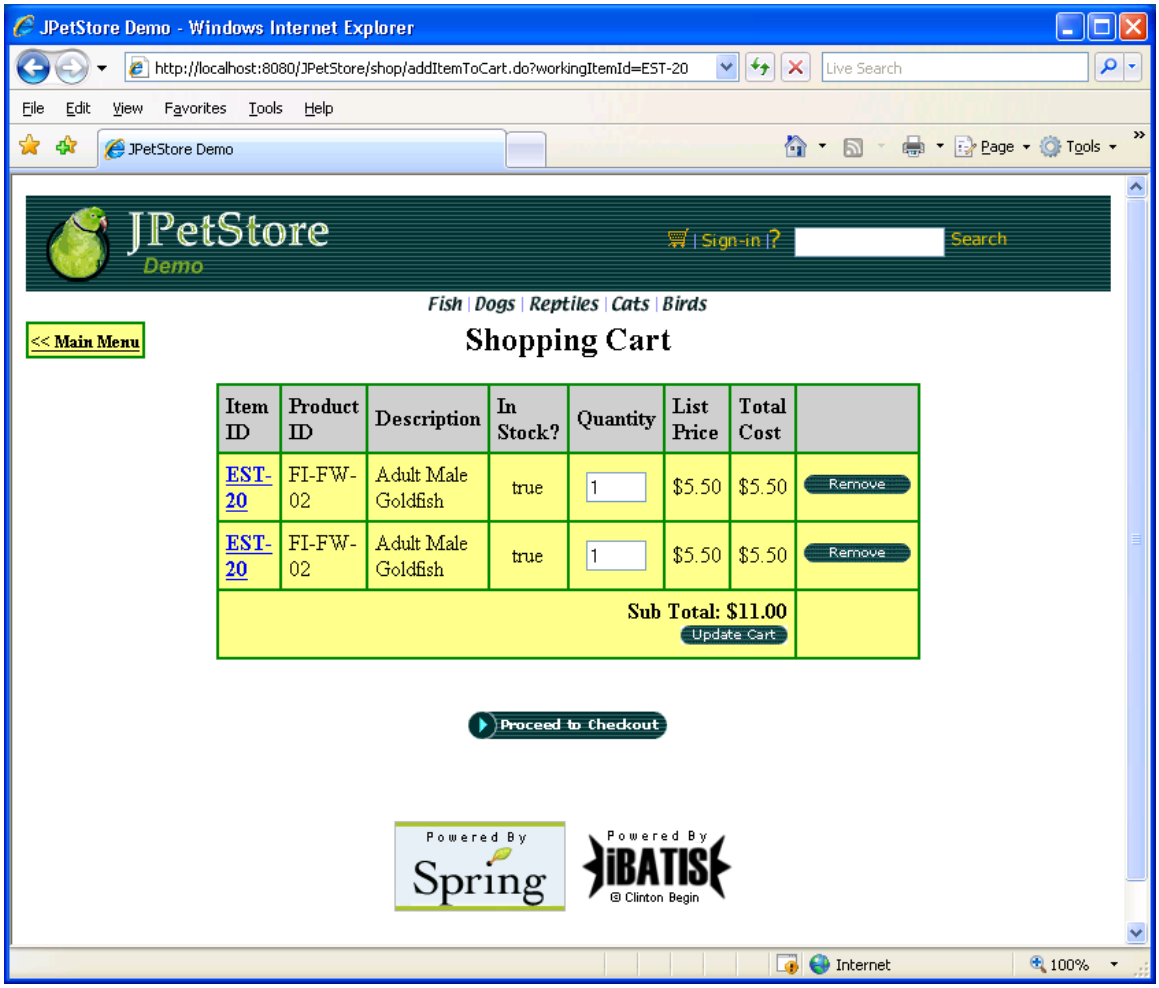
Development can try to debug it, but debugging on the production server is time-consuming and tedious. They would need to step through each statement of business logic as it executes in hopes of spotting the point where the runtime behavior deviates from their plan. Even if they find the point where 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.

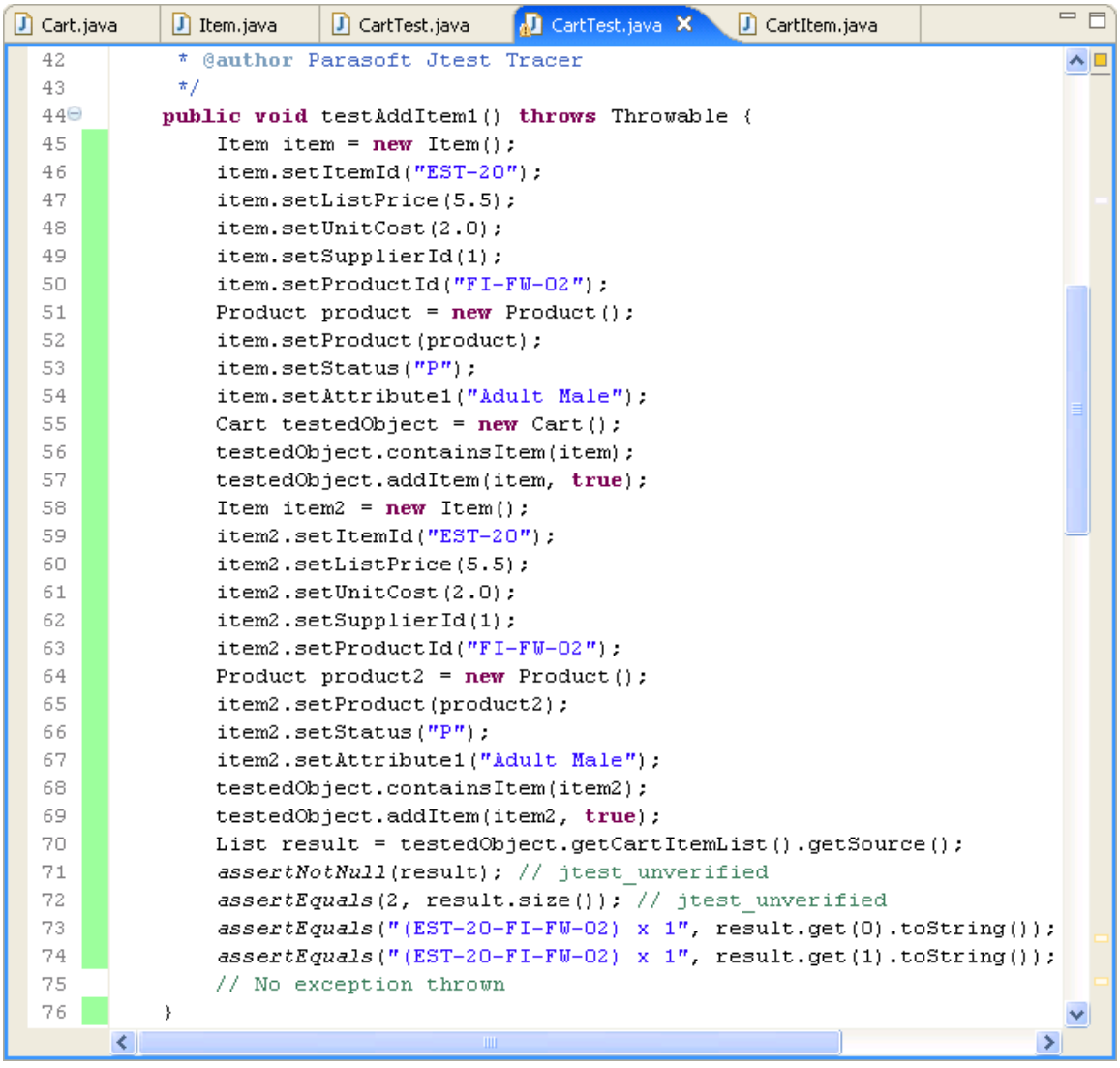


The intended behavior is for the shopping cart to show a single line item for goldfish, and for that line to have a quantity of two.

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

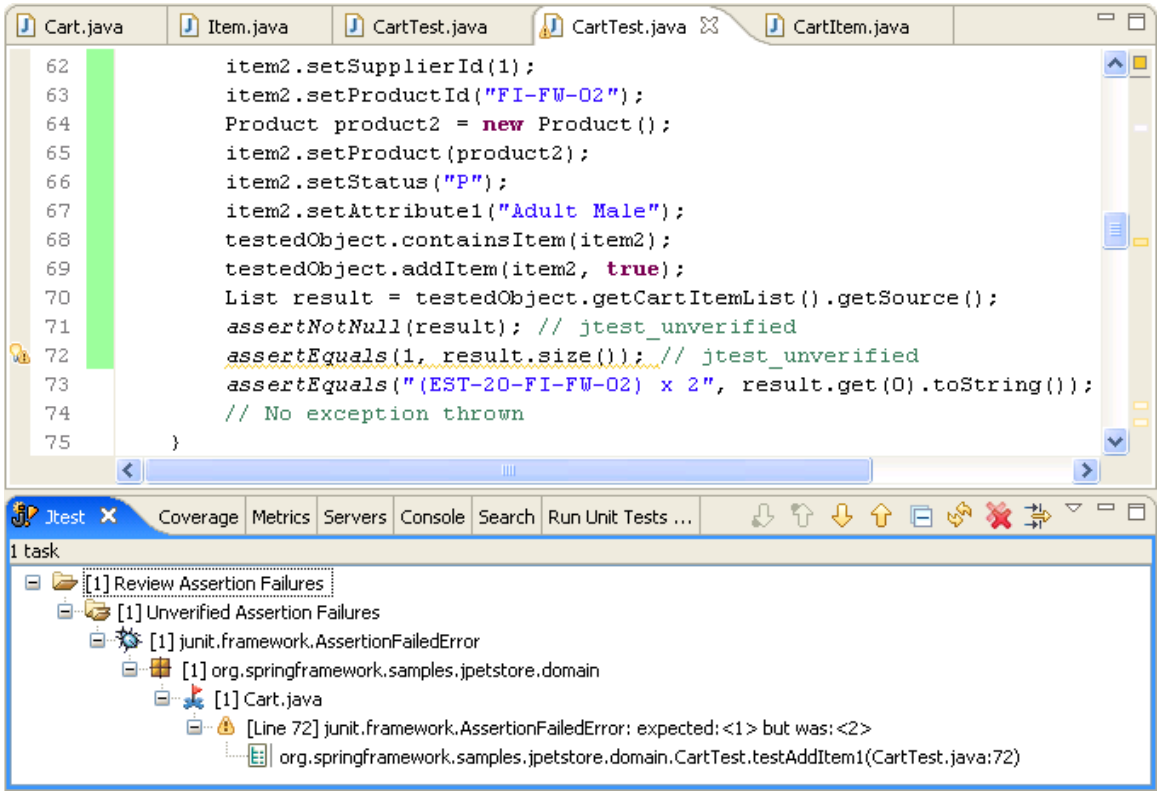
behavior for two cart items with a quantity of 1 for each. It is a JUnit functional test that automates the steps to reproduce the reported problem:

```
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:

```
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.

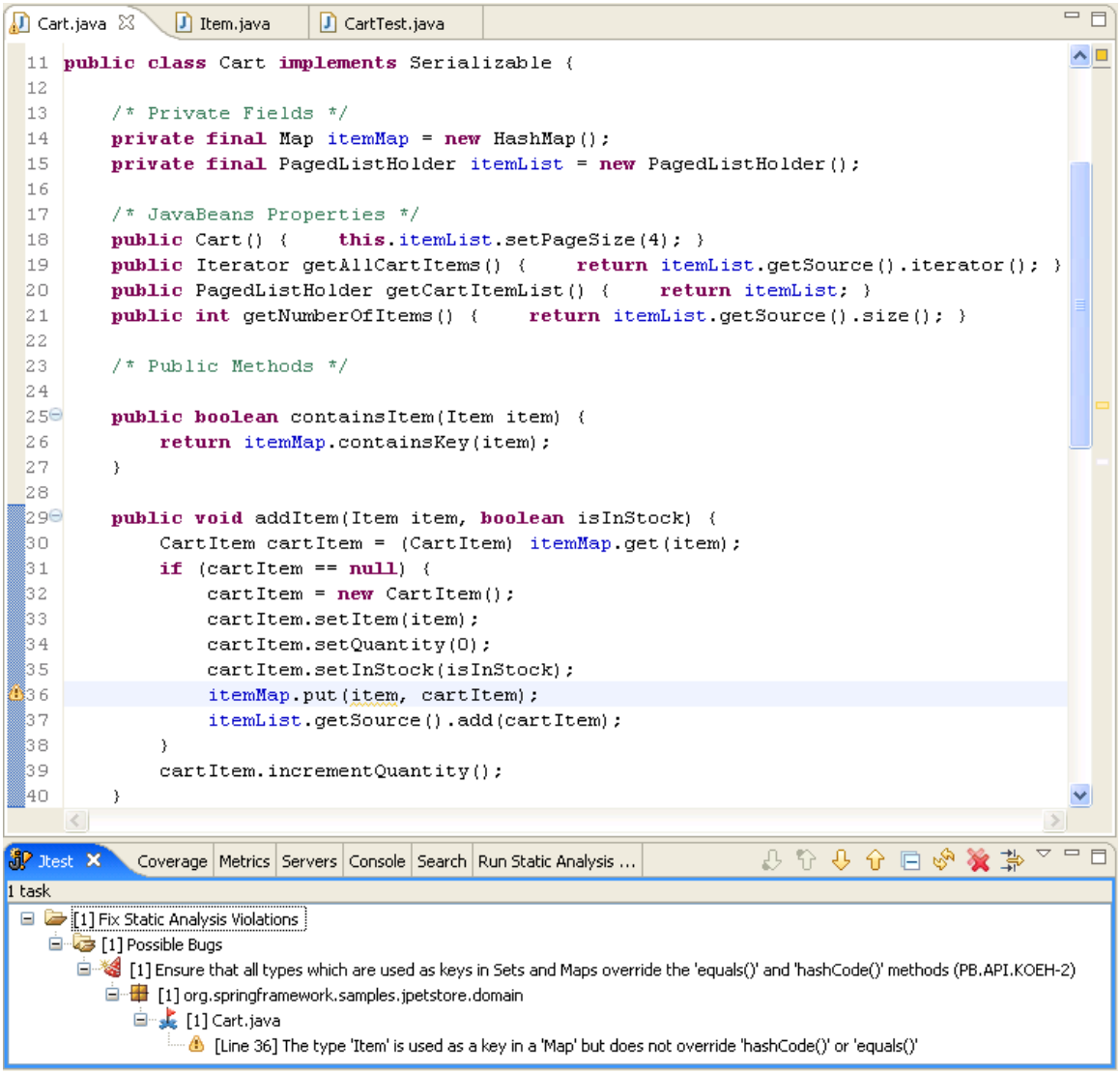


This test will be helpful for quickly checking which code changes resolve the problem. After a resolution is implemented, the test will continue to ensure that future code changes do not introduce a regression in this use case.

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:



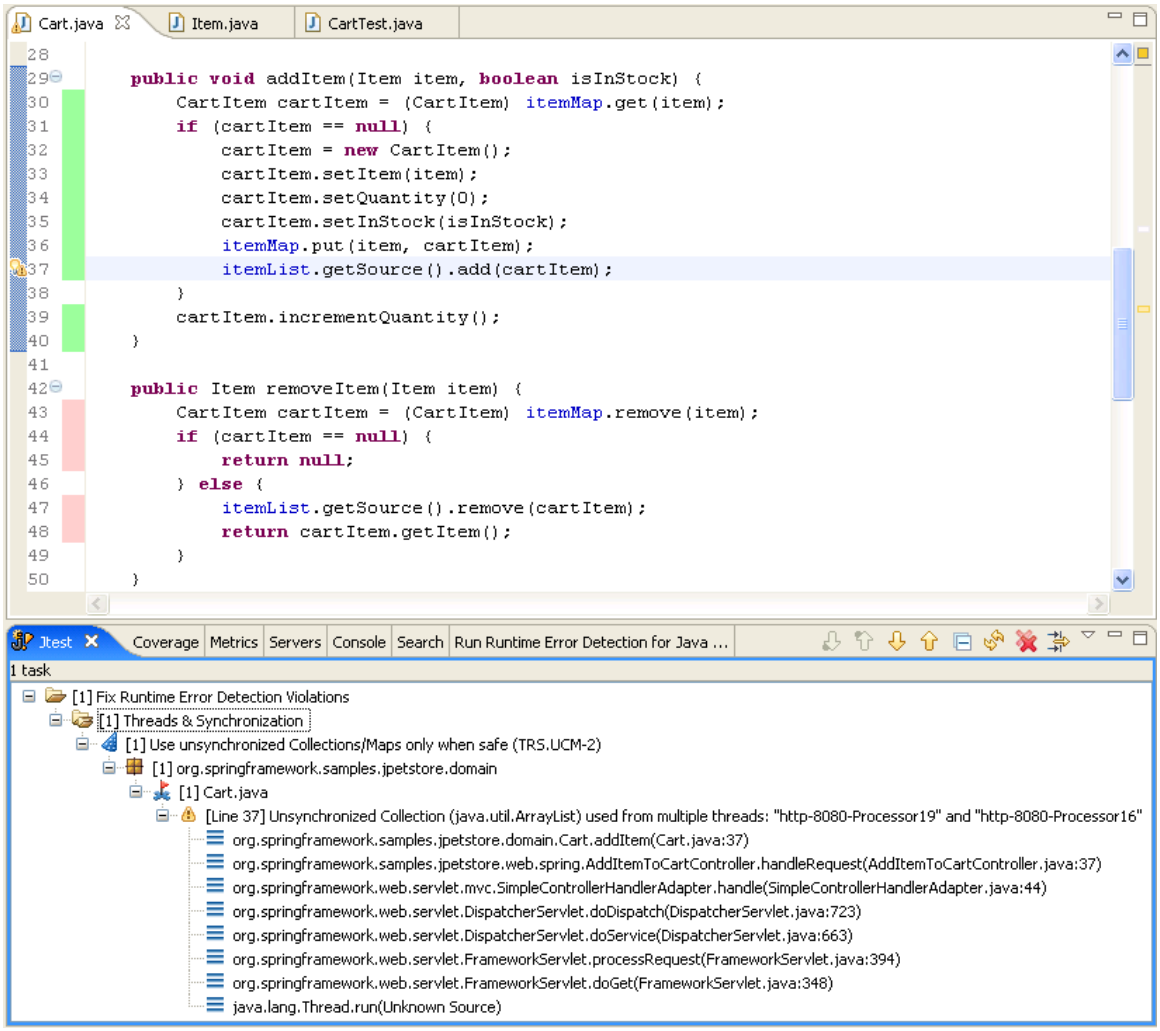
This is a violation of a Java Collections API rule that says all object keys to HashMaps should implement 'equals()' and 'hashCode()'. Indeed, this is exactly the problem. A new instance of Item is allocated for each web request, so two requests to add the same item do not actually use the same item 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.

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.

```
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);
            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.

Unit Testing with Runtime Error Detection

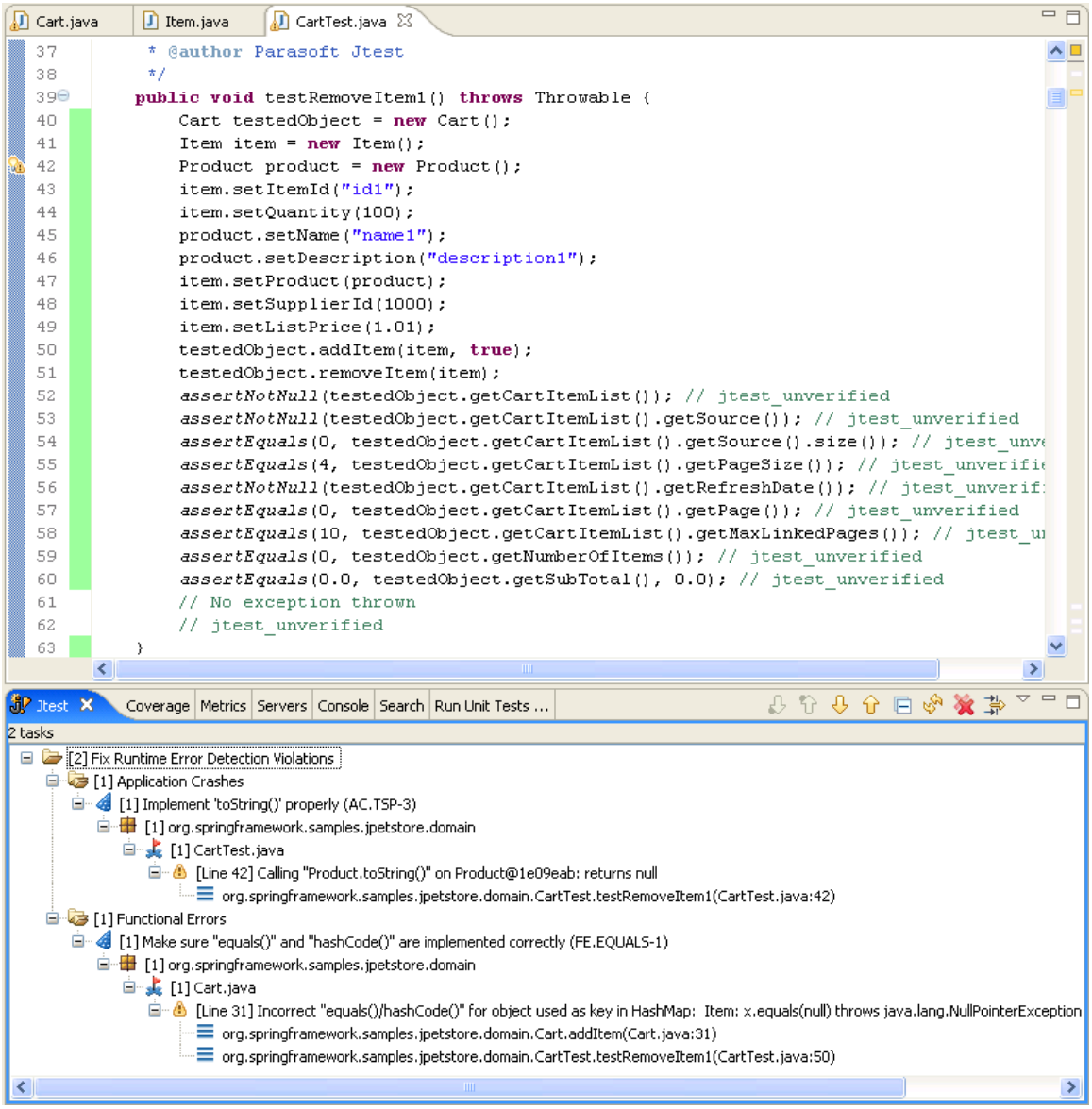
The problem report thus far has focused around adding the same item to a shopping cart twice… but what about other scenarios? Jtest reported that the 'removeItem()' method was still not covered. Removing items from the cart needs to be tested as well, but manual tests can be tedious—even manual tests 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.

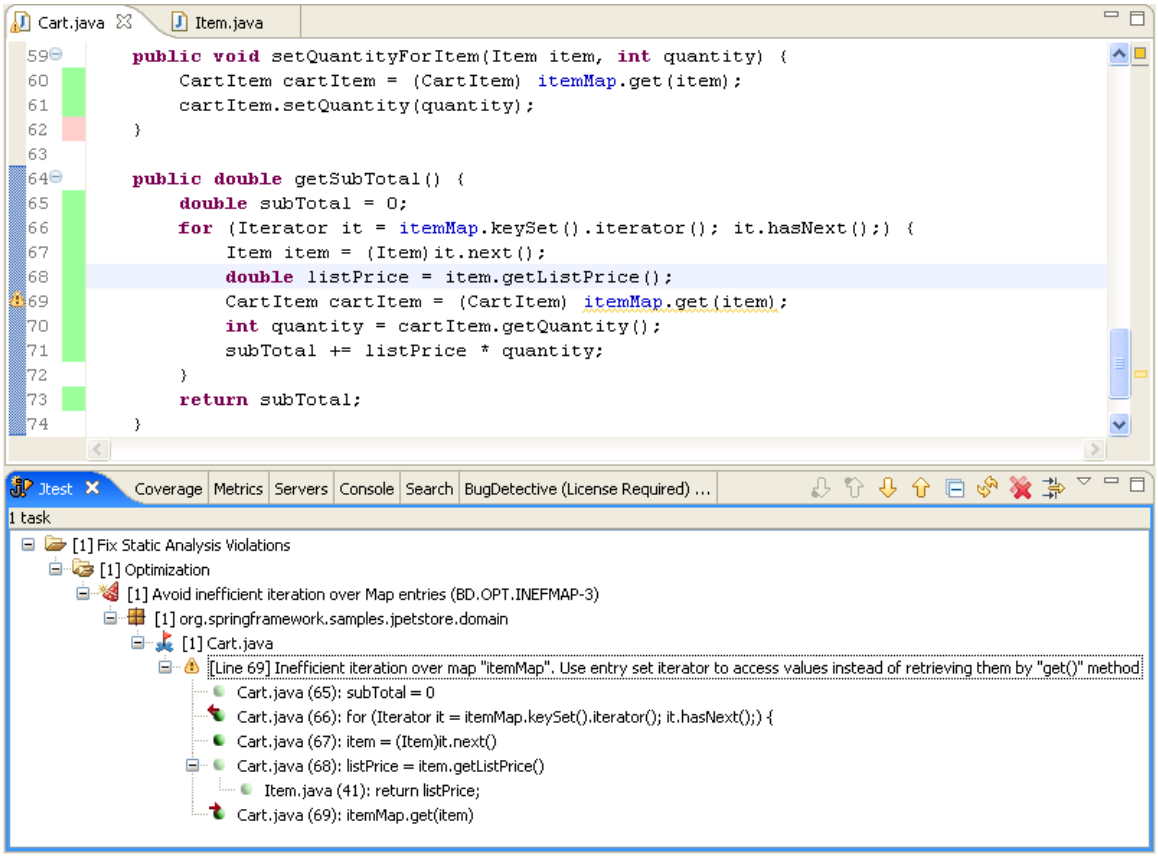


To satisfy the Java API contracts, the developers add the necessary checks to these core methods. Running the test case one more time ensures that the code is robust and no more runtime errors are reported.

```
public int hashCode() {
    return itemId == null? 0 : itemId.hashCode();
}
public boolean equals(Object o) {
    if ((o == null) || (o.getClass() != Item.class)) {
        return false;
    }
    return itemId.equals(((Item)o).itemId);
}
```

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 (using Jtest's BugDetective static analysis rules) 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.



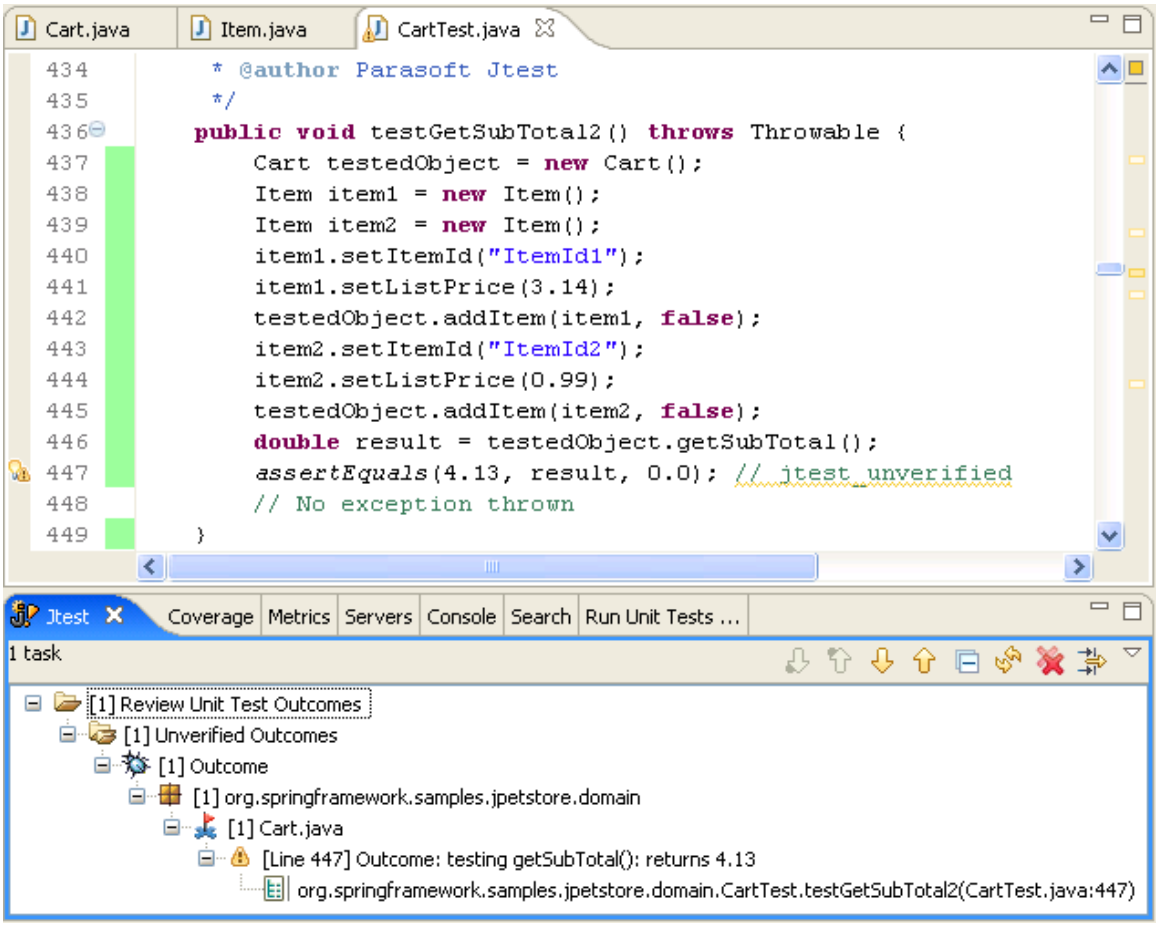
Here, Jtest's flow analysis found a path that could easily be optimized. The map of items to quantity is not used efficiently while computing the total price for the shopping cart. Specifically, a lookup into the map (order log n) is done for each item (order n) in the map, resulting in order n * log 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.

```
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;
}
```

Performing flow analysis again shows that the issue has been fixed.

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 solution center.

About Parasoft

For 21 years, Parasoft has investigated how and why software defects are introduced into applications. Our solutions leverage this research to dramatically improve SDLC productivity and application quality. Through an optimal combination of quality tools, configurable workflow, and automated infrastructure, Parasoft seamlessly integrates into your development environment to drive SDLC tasks to a predictable outcome. Whether you are delivering code, evolving and integrating business systems, or improving business processes—draw on our expertise and award-winning products to ensure that quality software can be delivered consistently and efficiently. 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)208 263 6005
Germany: Tel: +49 731 880309-0
Email: info-europe@parasoft.com

Asia

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

Other Locations

See http://www.parasoft.com/contacts

Integrated Error-Detection Techniques: Find More Bugs in Embedded C Software

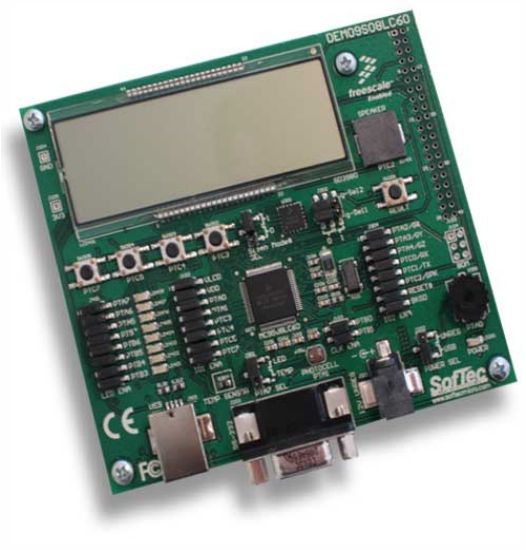
Software verification techniques such as pattern-based static code analysis, runtime memory monitoring, unit testing, and flow analysis are all valuable techniques for finding bugs in embedded C software. On its own, each technique can help you find specific types of errors. However, if you restrict 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 memory monitoring, unit testing, and flow analysis can be used together to find bugs in an embedded C application. These techniques will be demonstrated using Parasoft C++test, 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 memory corruption 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 a scenario that we recently encountered: a simple sensor application that runs on an ARM board.



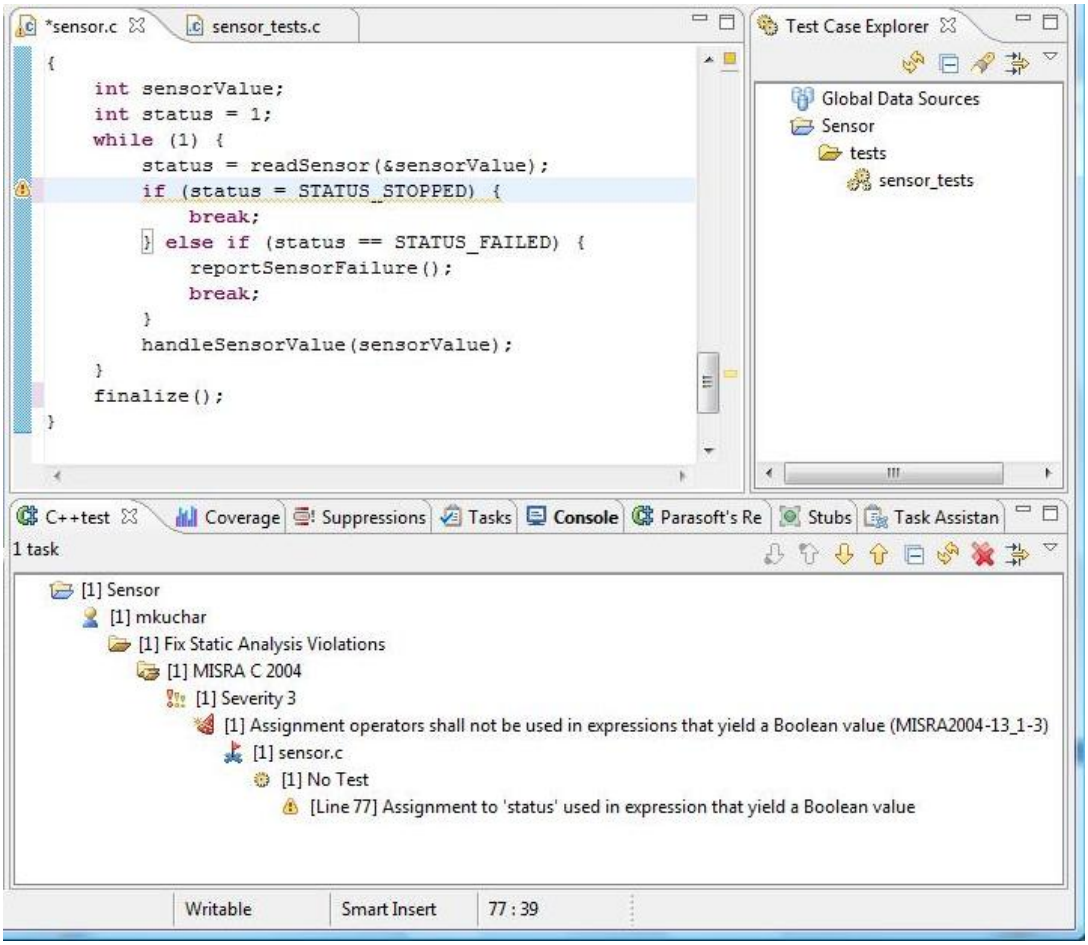
Assume that so far, we have created an application and uploaded it to the board. When we tried to run it, we did not see an expected output on the LCD screen.

It's not working, but we're not sure why. We can try to debug it, but debugging on the target board is time-consuming and tedious. We would need to manually analyze the debugger results and try to determine the real problems on our own. Or, we might apply certain tools or techniques proven to pinpoint errors automatically.

At this point, we can start crossing our fingers as we try to debug the application with the debugger. Or, we can try to apply an automated testing strategy in order to peel errors out of the code. If it's still not working after we try the automated techniques, we can then go to the debugger as a last resort.

Pattern-Based Static Code Analysis

Let's assume that we don't want to take the debugging route unless it's absolutely necessary, so we start by running pattern-based static code analysis. It finds one problem:

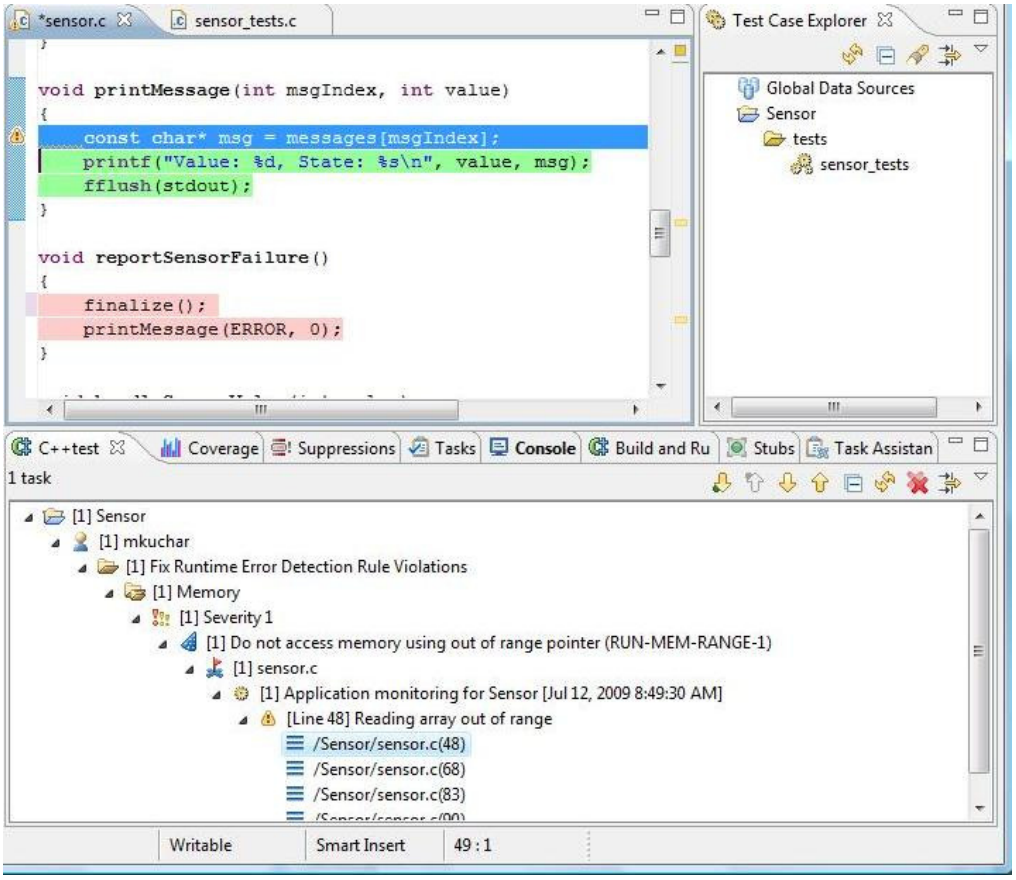


This is a violation of a MISRA rule that says that there is something suspicious with this assignment operator. Indeed, our intention was not to use an assignment operator, but rather a comparison operator. So, we fix this problem and rerun the program.

There is improvement: some output is displayed on the LCD. However, the application crashes with an access violation. Again, we have a choice to make. We could try to use the debugger, or we can continue applying automated error detection techniques. Since we know from experience that automated error detection is very effective at finding memory corruptions such as the one we seem to be experiencing, we decide to try runtime memory monitoring.

Runtime Memory Monitoring of the Complete Application

To perform runtime memory monitoring, we have C++test instrument the application. This instrumentation is so lightweight that it is suitable for running on the target board. After uploading and running the instrumented application, then downloading results, the following error is reported:



This indicates reading an array out of range at line 48. Obviously, the `msgIndex` variable must have had a value that was outside the bounds of the array. If we go up the stack trace, we see that we came here with this print message with a value that was out of range (because we put an improper condition for it before calling function `printMessage() )`. We can fix this by taking away unnecessary conditions `(value <= 20)`.

```
void handleSensorValue(int value)
{
	initialize();
	int index = -1;
	if (value >= 0 && value <= 10) {
		index = VALUE_LOW;
	} else if ((value > 10) && (value <= 20)) {
		index = VALUE_HIGH;
	}
	printMessage(index, value);
}
```

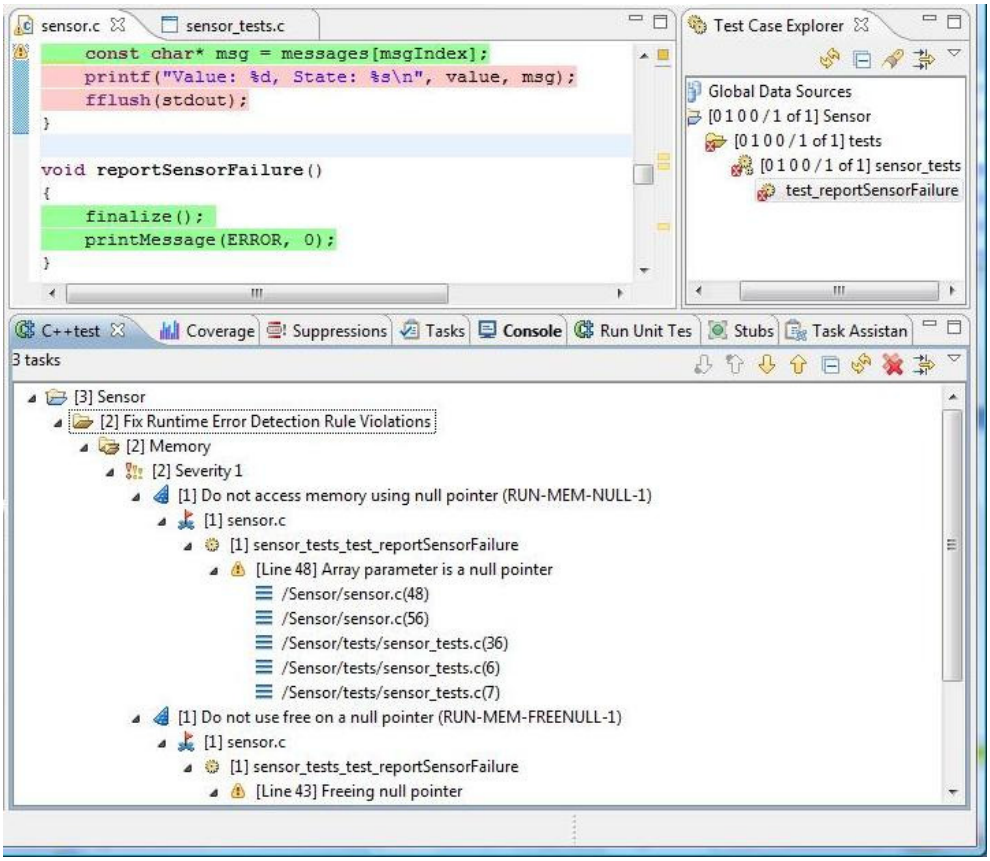
Now, when we rerun the application, no more memory errors are reported. After the application is uploaded to the board, it seems to work as expected. However, we are still a bit worried.

We just found one instance of a memory overwrite in the code paths that we exercised…but how can we rest assured that there are no more memory overwrites in the code that we did not exercise? If we look at the coverage analysis, we see that one of the functions, `reportSensorFailure()`, has not been exercised at all. We need to test this function…but how? One way is to create a unit test that will call this function.

Unit Testing with Runtime Memory Monitoring

We create a test case skeleton using C++test's test case wizard, then we fill in some test code. Then, we run this test case—exercising just this one previously-untested function—with runtime memory monitoring enabled. With C++test, this entire operation takes just seconds.

The results show that the function is now covered, but new errors are reported:



Our test case uncovered more memory-related errors. We have a clear problem with memory initialization (null pointers) when our failure handler is being called. Further analysis leads us to realize that in `reportSensorValue()` we mixed an order of calls. `finalize()` is being called before `printMessage()` is called, but `finalize()` actually frees memory used by `printMessage()`.

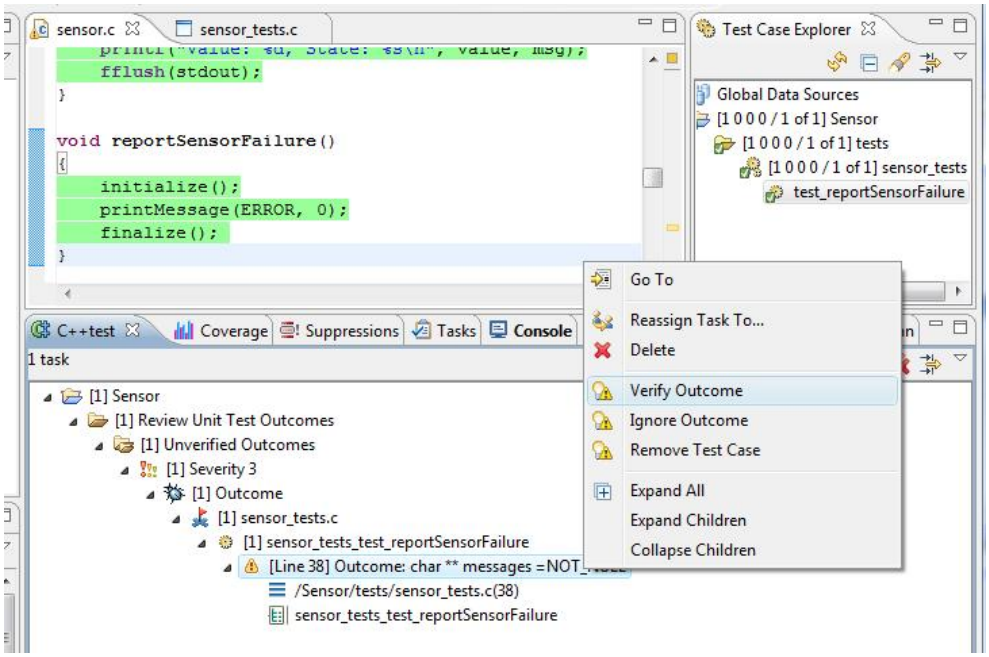
```
void finalize()
{
	if (messages) {
		free(messages[0]);
		free(messages[1]);
		free(messages[2]);
	}
	free(messages);
}
```

We fix this order, then rerun the test case one more time.

That resolves one of the errors reported. Now, let's look at the second problem reported: AccessViolationException in the print message. This occurs because these table messages are not initialized. To resolve this, we call the `initialize()` function before printing the message. The repaired function looks as follows:

```
void reportSensorFailure()
{
	initialize();
	printMessage(ERROR, 0);
	finalize();
}
```

When we rerun the test, only one task is reported: an unvalidated unit test case, which is not really an error. All we need to do here is verify the outcome in order to convert this test into a regression test. C++test will do this for us automatically by creating an appropriate assertion.



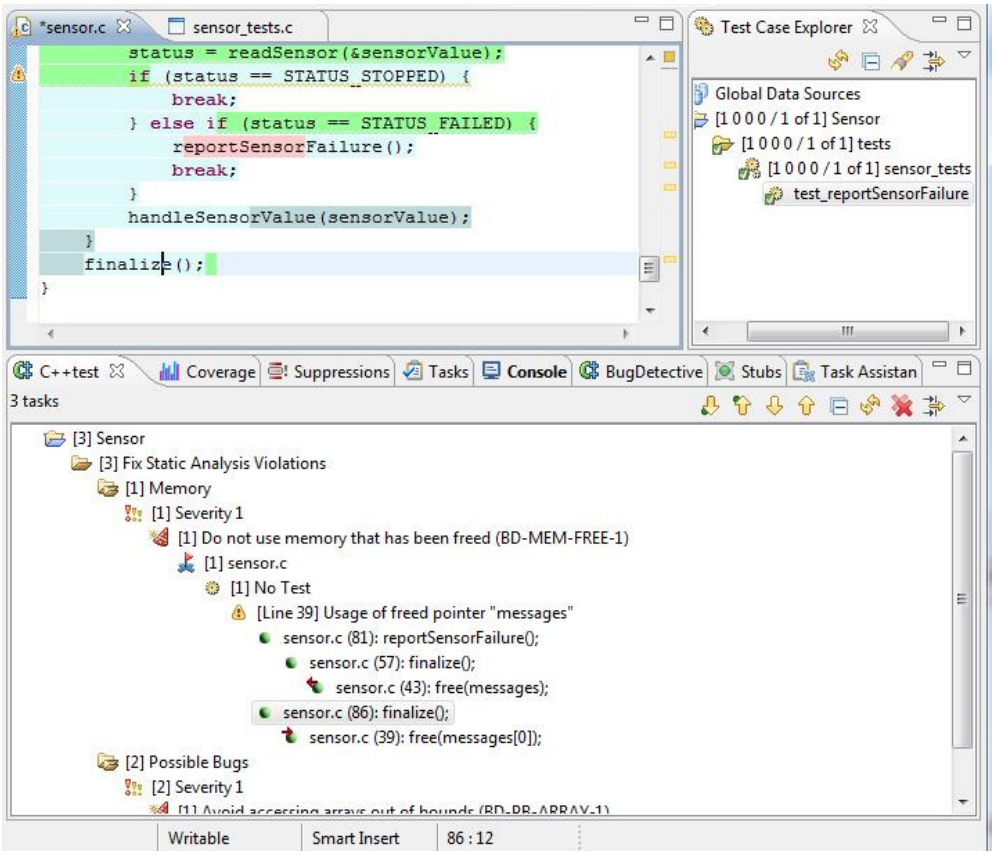
Next, we run the entire application again. The coverage analysis shows that almost the entire application was covered, and the results show that no memory error problems occurred.

Are we done now? Not quite. Even though we ran the entire application and created unit tests for an uncovered function, there are still some paths that are not covered. We can continue with unit

test creation, but it would take some time to cover all of the paths in the application. Or, we can try to simulate those paths with flow analysis.

Flow Analysis

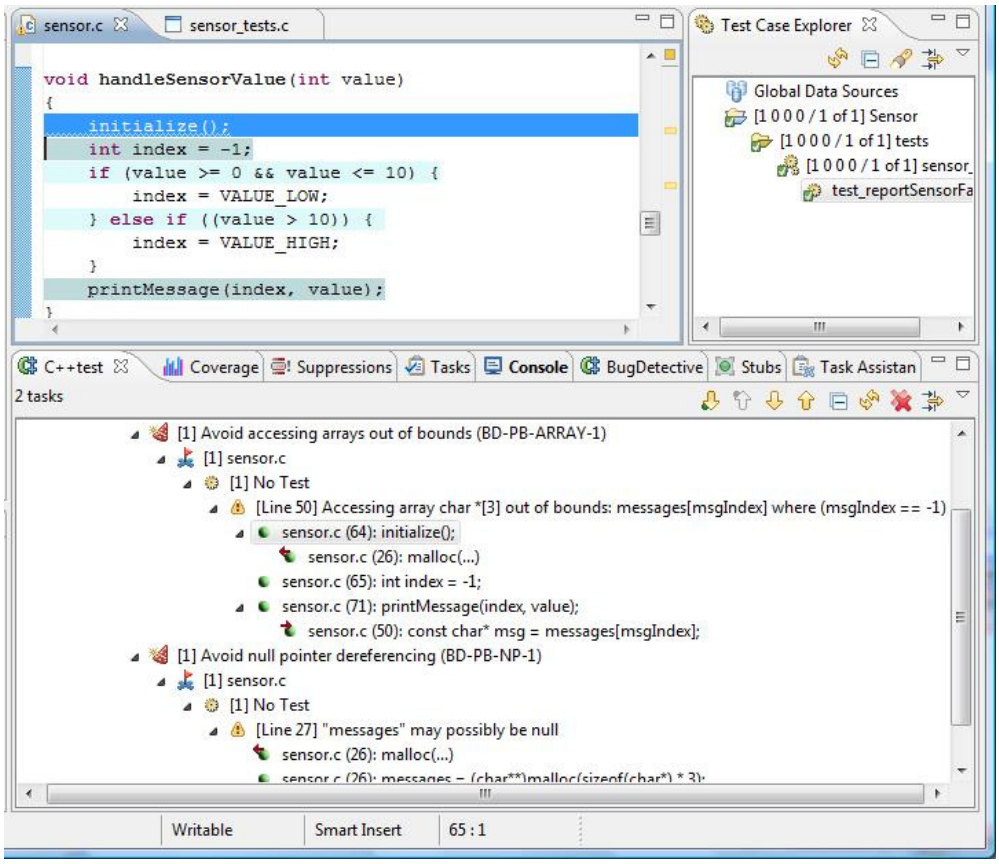
We run flow analysis with C++test's BugDetective, which tries to simulate different paths through the system and check if there are potential problems in those paths. The following issues are reported:



If we look at them, we see that there is a potential path—one that was not covered—where there can be a double free in the `finalize()` function. The `reportSensorValue()` function calls `finalize()`, then the `finalize()` calls `free()`. Also, `finalize()` is called again in the `mainLoop()`. We could fix this by making `finalize()` more intelligent, as shown below:

```
void finalize()
{
	if (messages) {
		free(messages[0]);
		free(messages[1]);
		free(messages[2]);
		free(messages);
		messages = 0;
	}
}
```

Now, let's run flow analysis one more time. Only two problems are reported:



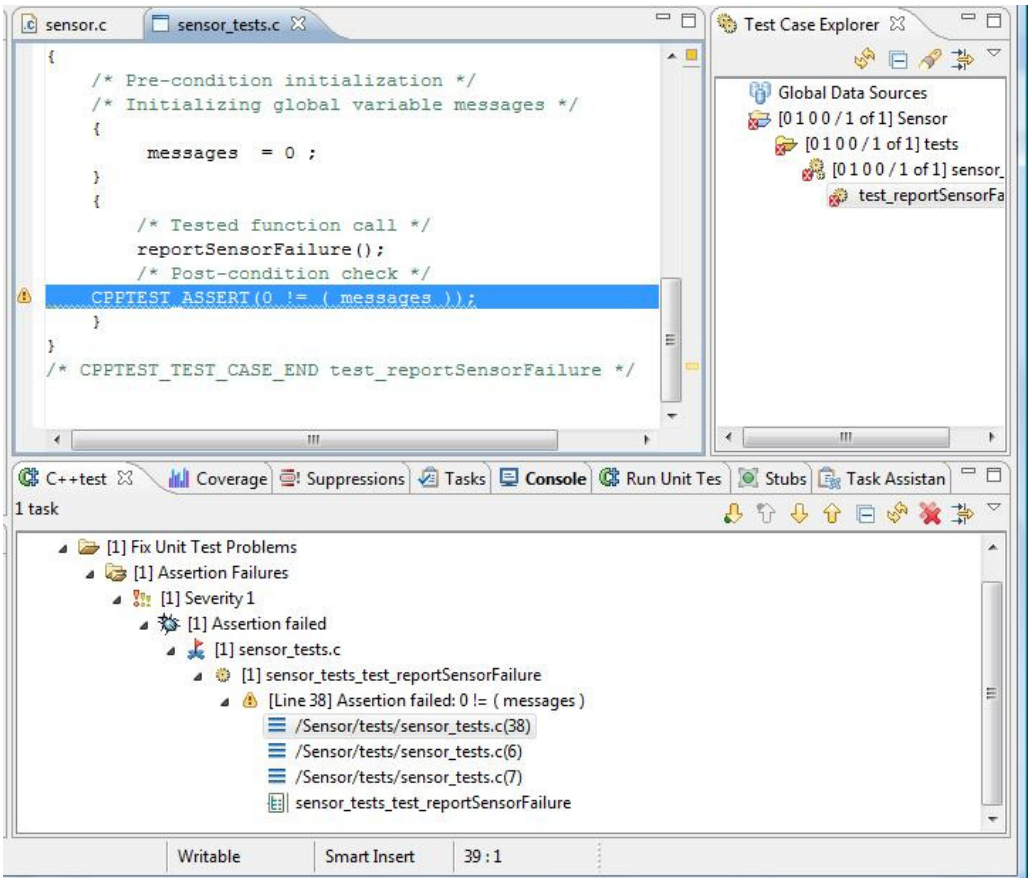
We might be accessing a table with the index -1 here. This is because the integral `index` is set initially to -1 and there is a possible path through the `if` statement that does not set this integral to the correct value before calling `printMessage()`. Runtime analysis did not lead to such a path, and it might be that such path would never be taken in real life. That is the major weakness of static flow analysis in comparison with actual runtime memory monitoring: flow analysis shows potential paths, not necessarily paths that will be taken during actual application execution. Since it's better to be safe than sorry, we fix this potential error easily by removing the unnecessary condition `(value >= 0)`.

```
void handleSensorValue(int value)
{
	initialize();
	int index = -1;
	if (value <= 10) {
		index = VALUE_LOW;
	} else {
		index = VALUE_HIGH;
	}
	printMessage(index, value);
}
```

In a similar way, we fix the final error reported. Now, we rerun the flow analysis, and no more issues are reported.

Regression Testing

To ensure that everything is still working, let's re-run the entire analysis. First, we run the application with runtime memory monitoring, and everything seems fine. Then, we run unit testing with memory monitoring, and a task is reported:



Our unit test detected a change in the behavior of the `reportSensorFailure()` function. This was caused by our modifications in `finalize()`—a change that we made in order to correct one of the previously-reported issues. This task alerts us to that change, and reminds us that we need to review the test case and then either correct the code or update the test case to indicate that this new behavior is actually the expected behavior. After looking at the code, we discover that the latter is true and we update the assertion correct condition.

```
/* CPPTEST_TEST_CASE_BEGIN test_reportSensorFailure */
/* CPPTEST_TEST_CASE_CONTEXT void reportSensorFailure(void) */
void sensor_tests_test_reportSensorFailure()
{
    /* Pre-condition initialization */
    /* Initializing global variable messages */
    {
        messages  = 0 ;
    }
    {
        /* Tested function call */
        reportSensorFailure();
```

```
        /* Post-condition check */
    CPPTEST_ASSERT(0 == ( messages ));
    }
}
/* CPPTEST_TEST_CASE_END test_reportSensorFailure */
```

As a final sanity check, we run the entire application on its own—building it in the IDE without any runtime memory monitoring. The results confirm that it is working as expected.

Wrap Up

To wrap up, let's take a bird's-eye view of the steps we just went over…

We had a problem with 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: some variable always being assigned the same value. We would have had to deduct from this that the problem was actually caused by an assignment operator being used instead of comparison. Static analysis found this logical error for us automatically. This type of error could not have been found by runtime memory analysis because it has nothing to do with memory. It probably would not be found by flow analysis either because flow analysis traverses the execution rather than validate whether conditions are proper.

After we fixed this problem, the application ran, but it still had memory problems. Memory problems are very difficult to see under a debugger; when you are in a debugger, you don't really remember the sizes of memory. Automated tools do, however. So, to find these memory problems, we instrumented the entire application, and ran it with runtime memory analysis. This told us exactly what chunk of memory was being overwritten.

However, upon reviewing the coverage analysis results, we learned that some of the code was not covered while testing on the target. Getting this coverage information was simple since we had it tracked automatically, but if we were using a debugger, we would have had to try 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 to add additional execution coverage to our testing efforts. Indeed, this revealed yet another problem. During normal testing on the target, covering those functions may be almost impossible because they might be hardware error-handling routines—or something else that is only executed under very rare conditions. This can be extremely important for safety critical applications. Imagine that there is a software error in code that should handle a velocity sensor problem in an airplane: instead of flagging a single device as non-working, we have a system corrupt. Creating a unit test case to cover such an execution path is very often the only way to effectively test it.

Next, we cleaned those problems and also created a regression test case by verifying the outcome (as one of the reported tasks guided us to do). Then, we ran flow analysis to penetrate paths that were not executed during testing on the target—even with the unit test. Before this, we had nearly 100% line coverage, but we did not have that level of path coverage. BugDetective uncovered some potential problems. They didn't actually happen and they might have never happened. They would surface only under conditions that were not yet met during actual

execution and might never be met in real life situations. However, there's no guarantee that as the code base evolves, the application won't end up in those paths.

Just to be safe, we fixed the reported problem to eliminate the risk of it ever impacting actual application execution. While modifying the code, we also introduced a regression, which was immediately detected when we re-ran unit testing. Among these automated error detection methods, regression testing is unique in its ability to detect functional changes and verify that code modifications do not introduce functional errors or unexpected side effects. Finally, we fixed the regression, retested the code, and it all seems fine.

As you can see, all of the testing methods we applied—pattern-based static code analysis, memory analysis, unit testing, flow analysis, and regression 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 embedded C software.**

***

In sum, by automatically finding many bugs related to memory and other coding errors, we were able to get the application up and running successfully. However, it's important to remember that the most deadly bugs are actually functional errors: instances where the application is not working according to specification. Unfortunately, these errors are much more difficult to find.

One of the best ways to find 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, enabling you to verify that it continues to adhere to specification. In the sample scenario described above, unit testing was used to force execution of code that was not covered by application-level runtime memory monitoring: it framed the current functionality of the application, then later, as we modified the code, it alerted us to a functionality problem. In fact, such unit test cases should be created much earlier: ideally, as you are implementing the functionality of your application. This way, you achieve higher coverage and build a much stronger "safety net" for catching critical changes in functionality.

Parasoft C++test 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 test suite.

About Parasoft C++test

Parasoft C++test is an integrated solution for automating a broad range of best practices proven to improve software development team productivity and software quality. C++test enables coding policy enforcement, static analysis, runtime memory monitoring, automated peer code review, and unit and component testing to provide teams a practical way to ensure that their C and C++ code works as expected. C++test 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 C++test results and other key process metrics.

C++test reduces the time, effort, and cost of testing embedded systems applications by enabling extensive testing on the host and streamlining validation on the target. As code is built on the host, an automated framework enables developers to start testing and improving code quality before the target hardware is available. This significantly reduces the amount of time required for testing on the target. The test suite built on the host can be reused to validate software functionality on the simulator or the actual target.

For more details on C++test, visit Parasoft's C and C++ Testing Tool center.

About Parasoft

For 20 years, Parasoft has investigated how and why software errors are introduced into applications. Our solutions leverage this research to deliver quality as a continuous process throughout the SDLC. This promotes strong code foundations, solid functional components, and robust business processes. Whether you are delivering Service-Oriented Architectures (SOA), evolving legacy systems, or improving quality processes—draw on our expertise and award-winning products to increase productivity and the quality of your business applications. 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)208 263 6005
Germany: Tel: +49 731 880309-0
Email: info-europe@parasoft.com

Asia

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

Other Locations

See http://www.parasoft.com/contacts

PARASOFT

Jtest®

DATA SHEET

Parasoft® Jtest® is an integrated solution for automating a broad range of practices proven to improve development team productivity and software quality. It focuses on practices for validating Java code and applications, and it seamlessly integrates with Parasoft SOAtest to enable end-to-end functional and load testing of today's complex, distributed applications and transactions.

Parasoft's customers, including the majority of the Fortune 500, rely on Jtest for:

- Preventing defects that impact application security, reliability, and performance
- Complying with internal or regulatory quality initiatives
- Ensuring consistency across large and distributed teams
- Increasing productivity by automating tedious yet critical defect-prevention practices
- Successfully implementing popular development methods like TDD, Agile, and XP

Capabilities

Static code analysis	Facilitates regulatory compliance (FDA, PCI, etc.). Ensures that the code meets uniform expectations around security, reliability, performance, and maintainability. Eliminates entire classes of programming errors by establishing preventive coding conventions.
Data flow static analysis	Detects complex runtime errors related to resource leaks, exceptions, SQL injections, and other security vulnerabilities without requiring test cases or application execution.
Metrics analysis	Identifies complex code, which is historically more error-prone and difficult to maintain.
Peer code review process automation	Automates and manages the peer code review workflow- including preparation, notification, and tracking- and reduces overhead by enabling remote code review on the desktop.
Unit test generation and execution	Enables the team to start verifying reliability and functionality before the complete system is ready, reducing the length and cost of downstream processes such as debugging.
Runtime error detection	Automatically exposes defects that occur as the application is exercised-including race conditions, exceptions, resource & memory leaks, and security attack vulnerabilities.
Test case "tracing"	Generates unit test cases that capture actual code behavior as an application is exercised providing a fast and easy way to create the realistic test cases required for functional/regression testing.
Automated regression testing	Generates and executes regression test cases to detect if incremental code changes break existing functionality or impact application behavior.
Coverage analysis	Assesses test suite efficacy and completeness using a multi-metric test coverage analyzer. This helps demonstrate compliance with test and validation requirements such as FDA.
Team deployment and workflow	Establishes a sustainable process that ensures software verification tasks are ingrained into the team's existing workflow and automated so team members can focus on tasks that truly require human intelligence.

These core capabilities are also available for C, C++, .NET languages.

Error assignment and distribution	Facilitates error review and correction. Each issue detected is prioritized, assigned to the developer who wrote the related code, and distributed to his or her IDE with direct links to the problematic code.
Centralized reporting	Ensures real-time visibility into quality status and processes. This helps managers assess and document trends, as well as determine if additional actions are needed for regulatory compliance.
Continuous "On-the-fly" static analysis	Automatically run static analysis in the background as developers review, add, and modify code. This helps the team identify and fix problems as soon as they are introduced.

Key Features

- Built-in support for Google Android, Spring, Hibernate, Eclipse plug-ins, TDD, JSF, Struts, JDBC, EJBs, JSPs, servlets, and more (mobile, embedded, Java EE...)
- Integrates with Parasoft SOAtest for end-to-end functional and load testing for web, SOA, and cloud development.
- Exposes runtime defects that occur as the application is exercised by unit, manual, or scripted tests–including race conditions, exceptions, resource leaks, and security attack vulnerabilities
- Without requiring execution, identifies execution paths that can trigger runtime defects
- Checks compliance to configurable sets of over 1000 built-in static analysis rules for Java
- Provides templates for OWASP Top 10, CWE-SANS Top 25, PCI DSS, and other security static standards
- Automatically corrects violations of 350+ rules with QuickFix
- Allows easy GUI-based customization of built-in rules
- Identifies and prevents concurrency defects such as deadlocks, race conditions, missed notification, infinite loops, data corruption other threading problems
- Automatically creates robust low-noise regression test suites–even for large code bases
- Generates functional JUnit test cases that capture actual code behavior as a deployed application is exercised
- Generates extendable JUnit and Cactus (in-container) tests that expose reliability problems and achieve high coverage using branch coverage analysis
- Integrates and extends manually-written unit test cases
- Continuously executes the test suite to identify regressions and unexpected side effects
- Performs runtime error detection as tests execute
- Parameterizes test cases for use with varied, controlled test input values (runtime-generated, user-defined, or from data sources)
- Monitors test coverage with multiple metrics
- Tracks code coverage from manual tests and test scripts
- Steps through tests with the debugger
- Tests individual methods, classes, or large, complex applications
- Calculates metrics such as Inheritance Depth, Lack Of Cohesion, Cyclomatic Complexity, Nested Blocks Depth, Number Of Children
- Identifies and refactors duplicate and unused code
- Automates the peer code review process (including preparations, notifications, and routing)
- Shares test settings and files team-wide or organization-wide
- Generates HTML, PDF, XML, and custom reports
- Tracks how test results and code quality change over time
- Provides GUI (interactive) and command-line (batch) mode

Infrastructure Support

- Full integration with Eclipse 3.2-3.7, IBM Rational Application Developer 7.0-8.0
- Integration with Ant, Maven, CruiseControl, Hudson, and other build & release tools
- Integration with most popular source control systems
- Open Source Control API, which allows teams to integrate any other source control system

System Requirements

Operating System

- Windows: 7, Vista, 2000, XP, or 2003 (x86 or x86_64)
- Linux: Red Hat E.L. 3, 4, 5 or equivalent (x86 or x86_64)
- Solaris: Solaris 10 (SPARC)
- Mac: OS X 10.5 or higher

Hardware

- Intel® Pentium® III 1.0 GHZ or higher recommended
- 512 MB RAM minimum; 2 GB RAM recommended
- JRE 1.3 or higher

www.parasoft.com

Contact info:
Parasoft Corporation, 101 E. Huntington Dr., 2nd Flr., Monrovia, CA 91016
Ph: (888)305.0041, Fax: (626)256.6884, Email: info@parasoft.com

PARASOFT®
We make software work.™

C++test™

C/C++ Development

DATA SHEET

Parasoft® C++test™ – Comprehensive Code Quality Tools for C/C++ Development

Parasoft C++test enables teams to produce better code, test it more efficiently, and consistently monitor progress towards their quality goals. With C++test, critical time-proven best practices—such as static analysis, comprehensive code review, runtime error detection, unit and component testing with integrated coverage analysis—are automated on the developer's desktop, early in the development cycle. A command line interface enables fully automated execution within regression and continuous integration environments, providing data for monitoring and analyzing quality trends. Moreover, C++test integrates with Parasoft's Concerto, which provides interactive Web-based dashboards with drill-down capability. This allows teams to track project status and trends based on C++test results and other key process metrics.

For embedded and cross-platform development, C++test can be used in both host-based and target-based code analysis and test flows. See page 3 for details.

Automate Code Analysis for Monitoring Compliance

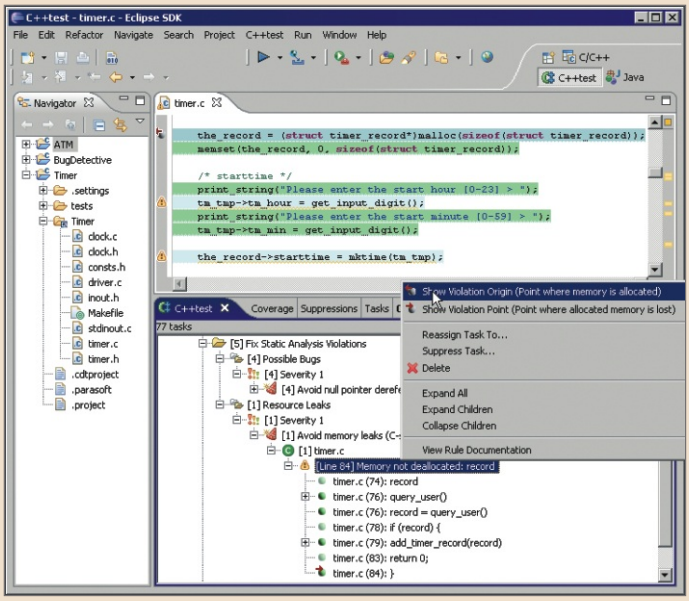
A properly implemented coding policy can eliminate entire classes of programming errors by establishing preventive coding conventions. C++test statically analyzes code to check compliance with such a policy. To configure C++test to enforce a coding standards policy specific to their group or organization, users can define their own rule sets with built-in and custom rules. Code analysis reports can be generated in a variety of formats, including HTML and PDF.

Hundreds of built-in rules—including implementations of MISRA, MISRA 2004, and the new MISRA C++ standards, as well as guidelines from Meyers' Effective C++ and Effective STL books, and other popular sources—help identify potential bugs from improper C/C++ language usage, enforce best coding practices, and improve code maintainability and reusability. Custom rules, which are created with a graphical RuleWizard editor, can enforce standard API usage and prevent the recurrence of application-specific defects after a single instance has been found.

Identify Runtime Bugs without Executing Software

BugDetective, the advanced interprocedural static analysis module of C++test, simulates feasible application execution paths—which may cross multiple functions and files—and determines whether these paths could trigger specific categories of runtime bugs. Defects detected include using uninitialized or invalid memory, null pointer dereferencing, array and buffer overflows, division by zero, memory and resource leaks, and various flavors of dead code.

C++test greatly simplifies defect analysis by providing a complete path trace for each potential defect in the developer's IDE. Automatic cross-links to code help users quickly jump to any point in the highlighted analysis path.



C++test's static analysis identifies critical bugs without executing the code (Eclipse version shown)

Benefits

- **Increase team development productivity** – Apply a comprehensive set of best practices that reduce testing time, testing effort, and the number of defects that reach QA.
- **Achieve more with existing development resources** – Automatically vet known coding issues so more time can be dedicated to tasks that require human intelligence.
- **Build on the code base with confidence** – Efficiently construct, continuously execute, and maintain a comprehensive regression test suite that detects whether updates break existing functionality.
- **Gain instant visibility into C and C++ code quality and readiness** – Access on-demand objective code assessments and track progress towards quality and schedule targets.
- **Reduce support costs** – Automate negative testing on a broad range of potential user paths to uncover problems that might otherwise surface only in "real-world" usage.

DATA SHEET

Features

- Static analysis of code for compliance with user-selected coding standards
- Graphical RuleWizard editor for creating custom coding rules
- Static code path simulation for identifying potential runtime errors
- Automated code review with a graphical interface and progress tracking
- Application monitoring/memory analysis
- Automated generation and execution of unit and component-level tests
- Flexible stub framework
- Full support for regression testing
- Code coverage analysis with code highlighting
- Code coverage analysis for unit testing and beyond (including application-level tests)
- Full team deployment infrastructure for desktop and command line usage

Runtime Error Detection

- Identify complex memory-related problems through simple functional testing–for example:
  - memory leaks
  - null pointers
  - uninitialized memory
  - buffers overflows
- Collect code coverage from application runs
- Increase test result accuracy through execution of the monitored application in a real target environment

Streamline Code Review

Code review is known to be the most effective approach to uncover code defects. Unfortunately, many organizations underutilize code review because of the extensive effort it is thought to require. The C++test Code Review module automates preparation, notification, and tracking of peer code reviews, enabling a very efficient team-oriented process. Status of all code reviews, including all comments by reviewers, is maintained and automatically distributed by the C++test infrastructure. C++test supports two typical code review flows:

- **Post-commit code review.** This mode is based on automatic identification of code changes in a source repository via custom source control interfaces, and creating code review tasks based on pre-set mapping of changed code to reviewers.
- **Pre-commit code review.** Users can initiate a code review from the desktop by selecting a set of files to distribute for the review, or automatically identify all locally changed source code.

The effectiveness of team code reviews is further enhanced through C++test's static analysis capability. The need for line-by-line inspections is virtually eliminated because the team's coding policy is monitored automatically. By the time code is submitted for review, violations have already been identified and cleaned. Reviews can then focus on examining algorithms, reviewing design, and searching for subtle errors that automatic tools cannot detect.

Monitor the Application for Memory Problems

Runtime Error Detection is the best known approach to eliminating serious memory-related bugs with zero false positives. The running application is constantly monitored for certain classes of problems–like memory leaks, null pointers, uninitialized memory, and buffer overflows–and results are visible immediately after the testing session is finished.

Without requiring advanced and time-consuming testing activities, the prepared application goes through the standard functional testing and all existing problems are flagged. The application can be executed on the target device, simulated target, or host machine. The collected problems are presented directly in the developer's IDE with the details required to understand and fix the problem (including memory block size, array index, allocation/deallocation stack trace etc.)

Coverage metrics are collected during application execution. These can be used to see what part of the application was tested and to fine tune the set of regression unit tests (complementary to functional testing).

Unit and Integration Test with Coverage Analysis

C++test's automation greatly increases the efficiency of testing the correctness and reliability of newly-developed or legacy code. C++test automatically generates complete tests, including test drivers and test cases for individual functions, purely in C or C++ code in a format similar to CppUnit. These tests, with or without modifications, are used for initial validation of the functional behavior of the code. By using corner case conditions, these automatically-generated test cases also check function responses to unexpected inputs, exposing potential reliability problems.

Test creation and management is simplified via a set of specific GUI widgets. A graphical Test Case Wizard enables developers to rapidly create black-box functional tests for selected functions without having to worry about their inner workings or embedded data dependencies. A Data Source Wizard helps parameterize test cases and stubs–enabling increased test scope and coverage with minimal effort. Stub analysis and generation is facilitated by the Stub View, which presents all functions used in the code and allows users to create stubs for any functions not available in the test scope–or to alter existing functions for specific test purposes. Test execution and analysis are centralized in the Test Case Explorer, which consolidates all existing project tests and provides a clear pass/fail status. These capabilities are especially helpful for supporting automated continuous integration and testing as well as "test as you go" development.

A multi-metric test coverage analyzer, including statement, branch, path, and MC/DC coverage, helps users gauge the efficacy and completeness of the tests, as well as demonstrate compliance with test and validation requirements, such as DO-178B. Test coverage is presented via code highlighting for all supported coverage metrics—in the GUI or color-coded code listing reports. Summary coverage reports including file, class, and function data can be produced in a variety of formats.

Automated Regression Testing

C++test facilitates the development of a robust regression test suite that detects if incremental code changes break existing functionality. Whether teams have a large legacy code base, a small piece of just-completed code, or something in between, C++test can generate tests that capture the existing software behavior via test assertions produced by automatically recording the runtime test results. As the code base evolves, C++test reruns these tests and compares the current results with those from the originally captured "golden set." It can easily be configured to use different execution settings, test cases, and stubs to support testing in different contexts (e.g., different continuous integration phases, testing incomplete systems, or testing specific parts of complete systems). This type of regression testing is especially critical for supporting agile development and short release cycles, and ensures the continued functionality of constantly evolving and difficult-to-test applications.

Configurable Detailed Reporting

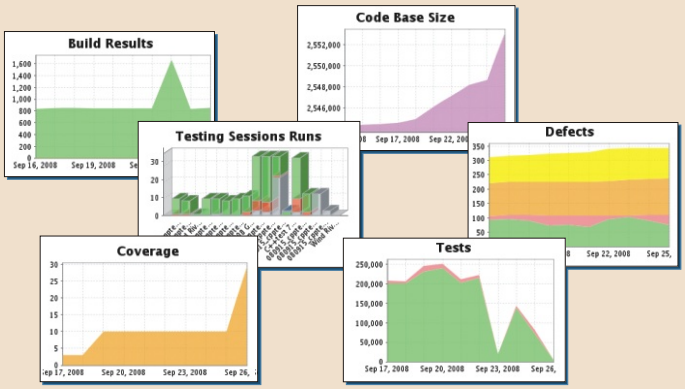
C++test's HTML, PDF, and custom format reports can be configured via GUI controls or an options file. The standard reports include a pass/fail summary of code analysis and test results, a list of analyzed files, and a code coverage summary. The reports can be customized to include a listing of active static analysis checks, expanded test output with pass/fail status of individual tests, parameters of trend graphs for key metrics, and full code listings with color-coding of all code coverage results. Generated reports can be automatically sent via email, based on a variety of role-based filters. In addition to providing data directly to the developers responsible for the code flagged for defects, C++test sends summary reports to managers and team leads.

Efficient Team Deployment

C++test establishes an efficient process that ensures software verification tasks are ingrained into the team's existing workflow and automated—enabling the team to focus on tasks that truly require human intelligence. Defect review and correction are facilitated through automated task assignment and distribution. Each defect detected is prioritized, assigned to the developer who wrote the related code, and distributed to his or her IDE with full data and cross-links to code. To help managers assess and document trends, centralized reporting ensures real-time visibility into quality status and processes. This data also helps determine if additional actions are needed to satisfy internal goals or demonstrate regulatory compliance.

Advanced Unit Test Features

- Automatic generation of tests and stubs
- Automatic generation of assertions based on observed test results
- Graphical Test Case Wizard for interactive definition of tests
- Complete visibility into test and stub source code
- Intelligent, test-case-sensitive stubs
- Parameterization of tests and stubs
- Multi-metric coverage analysis for DO-178B (including MC/DC)
- Flexible support for continuous regression testing
- Annotation of tests against bug and requirement IDs
- Execution of tests under debugger
- Special mode for testing template code

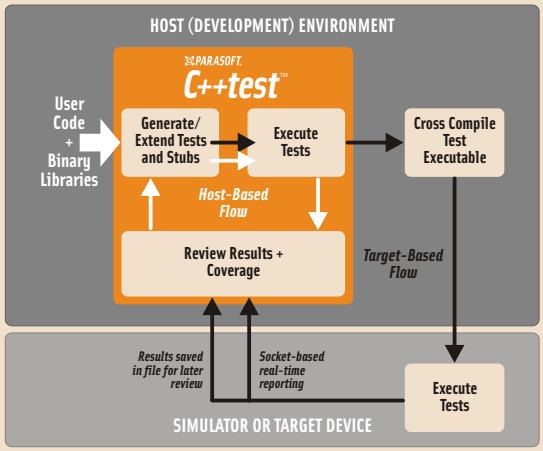


Dashboards track key development metrics

Embedded and Cross-Platform Development

As the software components in embedded systems are becoming increasingly critical, the attention to quality in embedded software increases across the board. Long-standing quality strategies such as testing with a debugger are no longer efficient or sufficient. To further complicate matters, many developers cannot readily run a test program in the actual deployment environment because they lack access to the final system hardware. To address these challenges, code quality needs to be realized throughout the development lifecycle—using a synergy of time-proven techniques for early defect prevention, assisted by automation for implementation and monitoring.

For highly quality-sensitive industries, such as avionics, medical, automobile, transportation, and industrial automation, the addition of Parasoft's Web-based audit and reporting system, with interactive Web-based dashboards and drill-down capability powered by a SQL database, enables an efficient and auditable quality process with complete visibility into compliance efforts.



C++test's customizable workflow allows users to test code as it's developed, then use the same tests to validate functionality/reliability in target environments

Test on the Host, Simulator, and Target

C++test automates the complete test execution flow, including test case generation, cross-compilation, deployment, execution, and loading results (including coverage metrics) back into the GUI. Testing can be driven interactively from the GUI or from the command line for automated test execution, as well as batch regression testing. In the interactive mode, users can run tests individually or in selected groups for easy debugging or validation. For batch execution, tests can be grouped based either on the user code they are liked with, or their name or location on disk.

High Degree of Customization

C++test allows full customization of its test execution sequence. In addition to using the built-in test automation, users can incorporate custom test scripts and shell commands to fit the tool into their specific build and test environment.

C++test can be utilized with a wide variety of embedded OS and architectures, by cross-compiling the provided runtime library for a desired target runtime environment. All test artifacts of C++test are source code, and therefore completely portable.

DATA SHEET

Supported Host Environments

Host Platforms
- Windows NT/2000/XP/2003/Vista/7
- Linux kernel 2.4 or higher with glibc 2.3 or higher and an x86-compatible processor
- Linux kernel 2.6 or higher with glibc 2.3 or higher and an x86_64-compatible processor
- Solaris 7, 8, 9, 10 and an UltraSPARC processor
- IBM AIX 5.3 and a PowerPC processor

IDEs
- Eclipse IDE for C/C++ Developers 3.2, 3.3, 3.4, 3.5, 3.6, 3.7
- Microsoft Visual Studio 2003, 2005, 2008, 2010 with Visual C++
- Wind River Workbench 2.6 or 3.0-3.3
- ARM Workbench IDE for RVDS 3.0, 3.1, 4.0, 4.1
- QNX Momentics IDE 4.5 (QNX SDP 6.4) or 4.7 (QNX SDP 6.5)
- Texas Instruments Code Composer Studio 4LB

Debuggers
- Lauterbach TRACE32

IDEs with Project Import Support
- ARM ADS 1.2
- Green Hills MULTI 4.0.x
- IAR Embedded Workbench 5.3/5.4
- Keil RealView MDK 3.40/uVision3
- Microsoft eMbedded Visual C++ 4.0
- Microsoft Visual Studio 6
- Texas Instruments Code Composer 3.1 and 3.3
- Wind River Tornado 2.0, 2.2

Host Compilers
- Windows
  - Microsoft Visual C++ 6.0, .NET (7.0), .NET 2003 (7.1), 2005 (8.0), 2008 (9.0), 2010 (10.0)
  - GNU and MingW gcc/g++ 2.95.x, 3.2.x, 3.3.x, 3.4.x
  - GNU gcc/g++ 4.0.x, 4.1.x, 4.2.x, 4.3.x, 4.4.x, 4.5.x
  - Green Hills MULTI for Windows x86 Native v4.0.x
- Linux (x86 target platform)
  - GNU gcc/g++ 2.95.x, 3.2.x, 3.3.x, 3.4.x, 4.0.x, 4.1.x, 4.2.x, 4.3.x, 4.4.x, 4.5.x
- Linux (x86_64 target platform)
  - GNU gcc/g++ 3.3.x, 3.4.x, 4.0.x, 4.1.x, 4.2.x, 4.3.x, 4.4.x, 4.5.x
- Solaris
  - Sun C++ 5.3 (Sun Forte C++ 6 Update 2), Sun C++ 5.5 (Sun ONE Studio 8), Sun C++ 5.6 (Sun ONE Studio 9), Sun C++ 5.7 (Sun ONE Studio 10), Sun C++ 5.8 (Sun ONE Studio 11), Sun C++ 5.9 (Sun ONE Studio 12)
  - GNU gcc/g++ 2.95.x, 3.2.x, 3.3.x, 3.4.x, 4.0.x, 4.1.x, 4.2.x, 4.3.x, 4.4.x, 4.5.x
  - Green Hills MULTI for SPARC Solaris Native v4.0.x
- AIX
  - IBM XL C/C++ compiler 8.0
  - GNU gcc/g++ 4.1.x

Target/Cross Compilers
- Altera (Linux hosted)
  - Nios GCC 2.9 (static analysis only)
  - NIOS II 5.1 GCC 3.4 (static analysis only)
- ARM (Windows hosted)
  - ARM RVCT 2.2, 3.x, 4.x
  - ARM ADS 1.2
- Cosmic (Windows hosted)
  - Cosmic Software 68HC08 C Cross Compiler V4.6.x (static analysis only)
- eCosCentric (Linux hosted)
  - GCC 3.4.x (static analysis only)
- Fujitsu (Windows hosted)
  - FR Family SOFTUNE C/C++ Compiler V6
- GNU (Windows, Linux, Solaris hosted)
  - gcc 2.9 - 4.5
- Green Hills
  - Windows hosted
    - Green Hills MULTI v5.1.x optimizing compilers for Embedded V800
  - Windows, Solaris hosted
    - Green Hills MULTI v4.0.x optimizing compilers
- IAR (Windows hosted)
  - IAR ANSI C/C++ Compiler 5.3x, 5.4x, 5.5x for ARM
  - IAR ANSI C/C++ Compiler 6.1x for ARM (C only)
- Keil (Windows hosted)
  - ARM/Thumb C/C++ Compiler, RVCT3.1 for uVision
  - ARM C/C++ Compiler, RVCT4.0 for uVision
  - C51 Compiler V8.18 (static analysis only)
- Microsoft (Windows hosted)
  - Microsoft Visual C++ for Windows Mobile 8.0, 9.0
  - Microsoft Embedded Visual C++ 4.0
- QNX (Windows hosted)
  - GCC 2.9.x, 3.3.x, 4.2.x, 4.4.x
- Renesas (Windows hosted)
  - Renesas SH SERIES C/C++ Compiler V9.03
- STMicroelectronics (Windows hosted)
  - ST20 (static analysis only)
  - ST40 (static analysis only)
- TASKING
  - Windows and Solaris hosted
    - 80C196 C Compiler v6.0 (static analysis only)
  - Windows hosted
    - TriCore VX-toolset C/C++ Compiler 2.5 (C only)
    - TriCore VX-toolset C/C++ Compiler 2.5, 3.3, 3.4, 3.5
- Texas Instruments (Windows hosted)
  - Windows hosted - CCS 4.x:
    - TMS320C6x C/C++ Compiler v6.1.x
    - TMS320C2000 C/C++ Compiler v5.2.x
    - TMS320C55x C/C++ Compiler v4.3
    - TMS320C54x C/C++ Compiler v4.2 (static analysis only)
    - MSP430 C/C++ Compiler v3.2.x
  - Windows hosted - CCS 3.x:
    - TMS320C6x C/C++ Compiler v5.1
    - TMS320C6x C/C++ Compiler v6.0
    - TMS320C2000 C/C++ Compiler v4.1 (static analysis only)
  - Solaris hosted:
    - TMS320C2x/C2xx/C5x Version 7.00 (static analysis only)
    - TMS320C6x C/C++ Compiler v. 4.3 (static analysis only)
    - TMS320C6x C Compiler v. 4.00 (static analysis only)
    - TMS320C6x C/C++ Compiler v. 5.1 (static analysis only)
- Wind River
  - Windows, Solaris, Linux hosted
    - GCC 2.96, 3.4.x, 4.1.x, 4.3.x
    - DIAB 5.0, 5.5, 5.6, 5.7, 5.8, 5.9
  - Windows hosted
    - GCC 3.3.x for VxWorks 653 (static analysis only)
    - EGCS 2.90

Build Management
- GNU make
- Sun make
- Microsoft nmake
- JAM
- Other build scripts that can provide an option of overriding a compiler via an environment variable

Source Control
- AccuRev SCM
- Borland StarTeam
- CVS
- Git
- IBM/Rational ClearCase
- Microsoft Team Foundation Server 2005, 2008 (only VS-based tools)
- Microsoft Visual SourceSafe
- Perforce SCM
- Serena Dimensions
- Subversion (SVN)
- Telelogic Synergy

www.parasoft.com

Contact info:
Parasoft Corporation, 101 E. Huntington Dr., 2nd Flr., Monrovia, CA 91016
Ph: (888) 305.0041, Fax: (626) 256.6884, Email: info@parasoft.com