Agile Software Development Management

PARASOFT
We make software work.

Parasoft Learning Series

Agile

Software Development Management

Parasoft Learning Series

Increase Agile Productivity through Policy-Driven Management

By Adam Kolawa, Parasoft CEO and co-founder

Parasoft has been developing software for more than 20 years. During this time, we have come to understand that there is no silver bullet for the creation and maintenance of software. Although we are great proponents of automation and analysis, we are the first to admit that there is no single tool, technology, or technique that can, on its own, improve productivity, guarantee quality, and ensure that project deadlines are met.

Over the years, Parasoft explored many different tools and technologies that held promise for improving code, application, or project quality. Parasoft's research and development not only explored these options from an internal productivity perspective, but also from the business value perspective—given our history of selling development organizations tools that assist in preventing software defects.

Our bottom line was that if we could not find a way for our own team to adopt a specific technique or technology as a process, then we deemed it superfluous and moved on. More importantly, we discovered that the "good" practices transcend methodology. It does not matter if you declare your process to be waterfall, agile, or iterative. The common denominator is that developers must write code and technology must complement the individual. It must be non-intrusive, automated, and follow the human workflow.

Personally, given our industry's appetite for technology, I'm a bit amazed that software development organizations have not adopted tighter, more disciplined processes leveraging the available automation. However, we have reached an inflection point in the business of developing software: a point where there is a distinct need for greater productivity and a sense of urgency for better software quality.

In 2009, Parasoft released Parasoft Concerto. Parasoft Concerto is a project planning and management tool that seamlessly integrates into any development environment tool to completely enable Agile software development methodologies—from Agile, to Scrum, to Extreme Programming, or hybrid.

I feel that Parasoft Concerto is an optimal combination of tools, processes, and infrastructure that can increase development productivity by orders of magnitude. We refer to this category of development infrastructure as Software Development Management (SDM). In the remainder of this document, I'd like to highlight how the Parasoft Concerto SDM solution assists to increase productivity throughout the SDLC.

Productivity Achieved Naturally

Developing software can be supplemented with automation, but we must accept that software development remains a complex, human-oriented task. In order to achieve greater levels of developer productivity, we must supplement human behavior with an unobtrusive process that fits into a human's natural workflow while achieving the objectives of the organization.

Achieving a return on investment by optimizing any resource requires change. This might be a change to a process or a change to the input to the process. From a human perspective, supplementary changes to a process or infrastructure offer the greatest opportunity for success. In software development, radical changes to the organization's current infrastructure can have severely detrimental impacts. The endeavor to increase productivity within the SDLC must leverage current assets and fit seamlessly into the existing environment.

Parasoft Concerto is designed to do just that. It complements the existing technical infrastructure, connecting all the distributed components in order to better facilitate human workflow. Parasoft Concerto accomplishes this by assisting the organization to manage "what" needs to be accomplished and by automatically monitoring "how" it is implemented.

People are more productive when they know exactly what they need to do. Developers are no different. If all the work they need to accomplish is provided right before their eyes—in their natural development environment—their productivity

101 E. Huntington Dr., 2nd Flr., Monrovia, CA 91016, Ph. 626.305.0041 / Fax 626.256.6884
www.parasoft.com

increases. Parasoft Concerto achieves this by managing the correct granularity of developers' tasks directly through the IDE. Simultaneously, as developers evolve code, Parasoft Concerto monitors that developers' work adheres to the project's policies and goals, which are established by management.

Parasoft Concerto manages "what" needs to be accomplished within the context of "how" management expects those tasks to be accomplished. This infrastructure allows for a distinct increase in productivity. More importantly, this platform monitors policies "behind the scenes." It is an unobtrusive, invisible infrastructure that only interacts with (nudges) staff when policies are not being followed. These policies, visible and monitorable, increase efficiency by reducing overhead and removing re-work.

With Parasoft Concerto integrated with the existing infrastructure, managers are automatically provided with project progress:

- Managers can verify whether a project is on budget because it is constantly monitored.
- Managers can verify that the required quality is achieved based on the policies put in place.
- Managers are provided warnings when additional resources are needed (for instance, because the work has become more complex than expected).

Policy Drives Behavior

Establishing management's expectation about what defines a completed requirement is essential for driving greater productivity. With expectations clearly defined, the team does not need to waste time trying to figure out exactly what is expected when—or constantly reworking the code to remedy misunderstandings.

Understanding these expectations is just the beginning though. Actually satisfying them can be a daunting task, considering the complexity of software development. Having an automated infrastructure that continually and objectively measures compliance with management's expectations is vital for making them a reality—without delaying projects or disrupting the team's optimal workflow.

An SDM system like Parasoft Concerto alleviates this burden by centralizing a defined policy and invoking policy-driven tasks across a distributed development environment. The ultimate goal is passive interaction with the user. The system is an invisible infrastructure that guides the user to achieve policy compliance; it works imperceptibly unless a team member does not comply with management's defined expectations. Essentially, it is like an EKG system hooked up to a hospital patient. If everything is fine, it runs inconspicuously in the background. But if the patient starts to flatline, alarms are sounded—and the staff knows to react immediately.

Systems like this ensure the continued productivity of resources that are acutely aware of management's expectations. They also bring new resources up to speed rapidly.

Policy Example—Every Requirement Must Have a Test Case

Software development organizations typically establish formal or informal internal quality policies to ensure the consistent quality of their code. For instance, many internal development policies mandate that every requirement must be validated with a passing test case before it is considered "implemented."

To help teams implement such a policy, Parasoft Concerto automatically correlates requirements to code and test cases. The system can confirm if the artifacts required by the policy are actually completed. Furthermore, it collects the results of the nightly test case execution to ensure that code is behaving correctly, as defined by the requirement.

Without a system like Parasoft Concerto, implementing even a simple policy like this one can be a daunting effort. Parasoft Concerto will not only remind the developer that a task (in this case, creating a test case) must be completed in order to comply with the policy, but it will also keep management informed of policy compliance. Such centralized policy management and compliance reporting is vital to ensuring that management requests are actually achieved.

101 E. Huntington Dr., 2nd Flr., Monrovia, CA 91016, Ph. 626.305.0041 / Fax 626.256.6884
www.parasoft.com

Demonstrating Compliance with Industry- or Government-mandated Policies

Some industry segments must not only define required practices via a policy, but also measure and report on compliance to that policy. In industries where software drives devices that impact human life—for instance, the medical device, defense, and aerospace industries—software development policy definition and compliance is driven by government or industry entities. Traditionally, industries like these have layered in manual practices to accommodate government or industry initiatives, and the lack of centralization or automation of these additional tasks bog down productivity.

For these industries, the challenge is not only achieving compliance, but also improving productivity in order to stay competitive with rapidly-evolving business requirements. Parasoft Concerto takes management expectations, which are defined as a policy, and converts them into actionable, measurable tasks. This helps the organization ensure process consistency while agilely adapting to changing market trends, regulatory environments, and customer demands.

"Right-sized" Tasks Are Presented to Developers and QA

For the QA group, it has become the norm to receive a request to execute an impossible scope of work within an unreasonable timeframe. For developers, abstract requirements pose a very difficult challenge as the implications of changing the code impacts unexpected components of the application. In both cases, lack of visibility and unclear expectations drain productivity and lead to out-of-control deliverables.

These challenges can be eliminated with an SDM system like Parasoft Concerto. Significant gains in developer or QA productivity are achieved when resources better understand the purpose of the code or the business goal of the requirement they are implementing. Parasoft Concerto provides development and QA with a greater level of understanding into the overall project objectives.

Providing an Exact Plan of Attack

A core driver for better understanding is the system's ability to assist development or QA managers to dissect high-level requests into reasonable and manageable tasks. It's the job of technical managers and architects to get tasks in front of the staff in an effective and efficient manner.

When developers are assigned to work on smaller tasks (for example, work tasks that are scoped to be no greater than one day) as opposed to being assigned to participate in one large project, experience has shown that they become much more productive. These granular tasks are scoped so development understands exactly where to start and what to do. The developers become much more efficient with this level of granularity: their tasks are attainable and progress is immediately recognized.

Maintaining Accuracy Throughout Complex, Dynamic Projects

An SDM system like Parasoft Concerto allows technical managers to create and assign tasks from a central console. Tasks are then distributed to the team via business rules or via manual assignment.

More importantly, Parasoft Concerto allows the staff member who was assigned the task the flexibility to submit a new time estimate or create a new task when it is recognized that more work needs to be done. It is the nature of the software development business that initial time estimates for specific tasks are often impossible because there is always hidden work that nobody can foresee… until a team member begins to dig into the code.

In such cases, managers are immediately notified that new work has been added or suggested, and they are given the flexibility to carve out new tasks and distribute the work among the team. This capability allows for better, more accurate information to flow back to the management team.

101 E. Huntington Dr., 2nd Flr., Monrovia, CA 91016, Ph. 626.305.0041 / Fax 626.256.6884
www.parasoft.com

Keeping Everyone on the Same Page—Without Having to be in the Same Room

Information regarding the status of these task assignments is carried from the developer's desktop back into the central Parasoft Concerto Report Center.

Managers know the status of assigned tasks and the overall project, and are thus armed with the information they need to make better decisions. Having access to project information in real time alleviates the need for status meetings, giving the developers more time to do what they like to do: write code.

Automate Human Review

No matter how much automation is in place to assist the development and QA teams to deliver the right code faster, humans must read, understand, and translate information into critical SDLC artifacts. Just as in the famous game of "telephone," a message delivered into one end of a chain of interpreters often comes out much different at the end of the line. This is why Parasoft Concerto has built in critical human review capabilities that are centrally driven by policies.

From Parasoft's 20+ years developing code, we have learned that one of the primary keys to both productivity and the reduction of errors is the automation of the human review process (for example, automating the "double implementation" of a requirement as well as automating the peer code review process).

Essentially, "baking" critical human review tasks into the software development lifecycle yields significant benefits—not only to overall application quality, but also to team productivity. Human tasks generated to remind someone to double-check their own work or to review their peers' work drive developers to gain greater visibility of the broader application. Ultimately, this delivers significant gains in productivity.

Checking it Twice: Double Implementation of Requirements

Getting developers to perform "double implementation" of a requirement ensures that a level of objective interpretation is baked into the software development cycle. Double implementation of the requirement means that the developer is required to have at least one test case for every requirement that is being implemented.

When such a practice is enforced, developers are obliged to think about the requirement from two different perspectives. The first perspective is the actual implementation of code. The second perspective is the creation of a test case. This not only ensures a review of the requirement, but also creates an artifact that holds the requirement's business assumptions stable in a changing application.

A system like Parasoft Concerto drives this practice by reminding developers when they need to create test cases for the second perspective. By flushing functional errors as well as prompting independent code review, this prevents a ton of rework later in the development process. Moreover, the system also ensures that you will see a test case break far before you know that specific code changes have impacted related components of an application. This essentially establishes a baseline for Change-based Testing as well as Change Impact Analysis.

Two Pairs of Eyes Are Better Than One: Peer Code Review

This leads to another very valuable quality component that is delivered as part of Parasoft Concerto: automated peer code review. Just as there is no substitute for an individual reviewing his or her own work, having the individual explain that work to a peer is priceless. Unfortunately, the practice of code review is often delayed, cancelled, or avoided as rework is required and deadlines creep up.

Parasoft Concerto automates the peer code review process and prompts developers to execute peer code review tasks as an essential part of the SDLC. The practice and conditions of peer code review are centrally managed as a policy within Parasoft Concerto, and all comments made during these reviews are recorded for easy retrieval.

101 E. Huntington Dr., 2nd Flr., Monrovia, CA 91016, Ph. 626.305.0041 / Fax 626.256.6884
www.parasoft.com

Contextual peer code review can also be accomplished to meet distinct regulatory requirements such as PCI, CWE/SANS, and MISRA. This can also drive guidelines for initiatives like secure application development.

Leverage Automation in Context of Process

Although every section of this paper has touched upon this topic in some manner, leveraging automation in context of a process deserves to be highlighted as a critical element of an SDM system. The bottom line is that software development is a complex–and in some cases, risky– business.

As professionals, we need to leverage "good" technology to help the business meet its goals. Ad hoc usage of tools can yield some immediate benefits, but in the long run it will most likely degrade productivity. For example, leveraging a memory error detection tool like Parasoft Insure++ will lead you straight to the maddening memory errors that you've spent months chasing after. Yet, leveraging policy-based static code analysis to prevent memory errors will yield much greater productivity.

Integrating Proven Practices into the Infrastructure

One of the core differentiators of Parasoft Concerto is that the proven defect prevention and quality automation capabilities used to monitor compliance to your quality policies are seamlessly integrated into the infrastructure. These capabilities include:

- Pattern Based Code Analysis
- Flow Based Code Analysis
- Code Analysis for Security
- Code Review
- Contextual Code Review
- Unit Testing
- Memory Error Detection
- Message/Protocol Testing
- Web UI Testing
- End-to-end Testing
- Functional Testing
- Load Testing
- Change-based Testing
- Manual Testing
- User Acceptance Testing

From our long history of developing quality tools that prevent software defects, we evangelized that core productivity tools be deployed and managed in context of the software development process. Depending on your application and business environment, the correct mix of the aforementioned quality practices baked into the development process will deliver greater productivity and better business outcomes.

Why Practice Placement Matters

For example, quality practices such as static analysis assist development to prevent frivolous mistakes. If you use this technology as code is being written, fewer defects are introduced, and you significantly reduce the time required for debugging and reworking. This increases productivity significantly.

101 E. Huntington Dr., 2nd Flr., Monrovia, CA 91016, Ph. 626.305.0041 / Fax 626.256.6884
www.parasoft.com

Critical architectural concepts like security cannot be implemented any time later than when the code is actually being written. A core tenet of delivering a secure application is that security needs to be built into the system. A strong security policy, which is enforced as part of the software development process, will yield expected results with greater productivity.

New practices can also be phased in or extended as requirements or policies evolve. For instance, perhaps management decides to allow international access to an application. If development is already performing static analysis, they can then phase in a new static analysis rule to verify that all of the code suits core internationalization requirements (for instance, code is in WL characters). Instead of manually searching through the code by hand, they can automatically identify all of the existing code that does not comply–and in many cases, they can correct it automatically as well. Then, by keeping this rule in place, they can ensure that all new code written from that point forward satisfies the proper internationalization guidelines.

Real-time Information Leads to Greater Productivity and Better Decisions

At some companies, employees spend 20 to 30 percent of their working days in meetings. If you think about it, if you spend 20% of each day in meetings, you've lost one full day per week. By implementing Parasoft Concerto, companies can significantly reduce the time spent on non-value-added status meetings. A direct quote from a development manager at Cisco Systems: "Parasoft Concerto provides a level of understanding and expectations that prevents the need for people to come together in a room in order to get on the same page."

In Parasoft Concerto, managers are presented with real-time information. The project is constantly being updated as developers provide expected completion dates of in-progress tasks and complete their work. Thus, managers always know where the project is going and when it will be finished.

Statistically speaking, this is the only way to get accurate information about the completion of a project. The accumulation of developers updating how much time they expect to spend on their tasks paints an accurate picture of the project.

Converting Data into Actionable Knowledge

To keep projects on track, development managers need to quickly spot potential problems and make strategic decisions to correct those problems. The key to facilitating this is to use an SDM like Parasoft Concerto to collect and correlate data from code activity, test results, code coverage, requirements, and corrected bugs. Ideally, the system will help you answer questions such as:

- Is the organization following prescribed policies?
- Is the software of high quality?
- How well are the new features implemented?
- How secure is the software?
- Does code follow coding standards?

For instance, if compliance to the team's security policy starts to decay, the manager would be alerted to this via policy compliance reports. He might investigate and address the root cause of the problem, then give the team one week to get the code back into compliance. The developers would then refer back to the results, which are delivered as part of their daily process, for details on exactly what needs to be corrected.

Analyzing the Process to Improve Productivity

In today's economy, organizations need to find ways to do more with less in order to remain competitive. Software development is no exception. With the comprehensive metrics delivered by an SDM like Parasoft Concerto, the manager can track productivity, identify what parts of the process could be improved to increase productivity, check whether the team is following process improvement measures, and monitor the impact of such measures on productivity.

101 E. Huntington Dr., 2nd Flr., Monrovia, CA 91016, Ph. 626.305.0041 / Fax 626.256.6884
www.parasoft.com

Parasoft Concerto tracks software development metrics and reports whether those metrics are improving over time. This makes it is easy to analyze statistical data about past projects, and then draw conclusions to help properly plan for the future and improve the process.

A Platform for Productivity and Process Improvement

Over the 20+ years that I have acted as head of development for Parasoft, rarely have I been concerned with which development methodology to follow. In my experience, teams tend to migrate to a work structure that best meets the challenges of the particular project or product.

I've also learned that people who I feel are my most productive or most valuable resources tend to share some common traits. First, they tend to have a better grasp of the business challenges that the software is addressing. Second, they have a broad awareness of the code base and an intellectual curiosity about the interrelated components or systems. Finally, they have a knack for automating common tasks due to their preference or zeal for creating code rather than dealing with mundane tasks that permeate the SDLC.

Over the years, I've tried to automate my software development process to promote these traits in my development staff. Let's face it: When we look to hire development staff, we rarely test their ability to sit in status meetings or write status reports. We test their logic, their ability to analyze a problem or dissect a requirement, and their skills related to evolving our business applications. The bottom line is that the developers' time is optimally spent writing code to meet the changing demand of business. This productivity is essential for innovation and business growth.

Parasoft Concerto takes into account these critical development traits and provides an automated software development management platform focused on driving productivity. Our unique history, which stems from automating defect prevention, allows management to drive the SDLC to a predictable outcome. With Parasoft Concerto, a requirement is managed and measured on "when" and "how" it is implemented and compliance to defined policies is measurable in real-time.

In summary, having an SDM such as Parasoft Concerto enables you to integrate and facilitate your SDLC to ensure that quality software can be produced consistently and efficiently. By carefully orchestrating the development environment as described in this paper, you gain end-to-end process visibility and control. Moreover, by enabling developers and testers to do what is required—faster and more precisely—you increase productivity and reduce overall costs.

Next Steps

- Learn more about the Parasoft Agile Solution
- Start a free evaluation now
- Contact Parasoft for additional information

101 E. Huntington Dr., 2nd Flr., Monrovia, CA 91016, Ph. 626.305.0041 / Fax 626.256.6884
www.parasoft.com

Integrated Defect Prevention for Agile Development

By Matt Love, Parasoft Software Development Manager

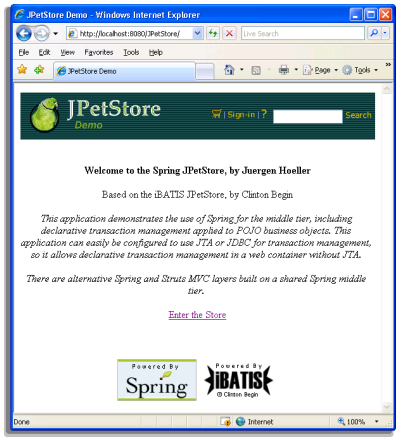
Software verification techniques such as pattern-based static code analysis, runtime error detection, unit testing, and flow analysis are all valuable techniques for finding bugs in Java web applications. On its own, each technique can help you find specific types of errors. However, if you restrict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

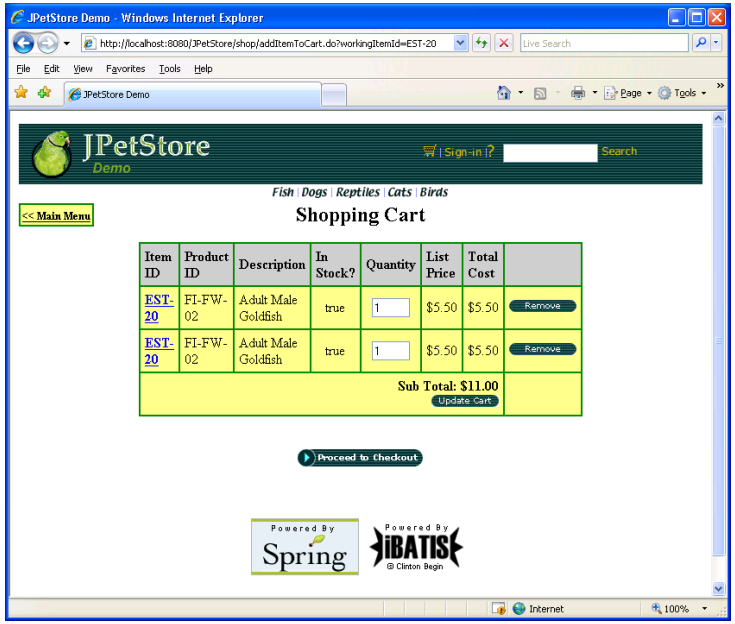
101 E. Huntington Dr., 2nd Flr., Monrovia, CA 91016, Ph. 626.305.0041 / Fax 626.256.6884
www.parasoft.com

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.

Exploring the 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.

101 E. Huntington Dr., 2nd Flr., Monrovia, CA 91016, Ph. 626.305.0041 / Fax 626.256.6884
www.parasoft.com

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.

Cart.java | Item.java | CartTest.java | CartTest.java | CartItem.java

```
42      * @author Parasoft Jtest Tracer
43      */
44     public void testAddItem1() throws Throwable {
45         Item item = new Item();
46         item.setItemId("EST-20");
47         item.setListPrice(5.5);
48         item.setUnitCost(2.0);
49         item.setSupplierId(1);
50         item.setProductId("FI-FW-02");
51         Product product = new Product();
52         item.setProduct(product);
53         item.setStatus("P");
54         item.setAttribute1("Adult Male");
55         Cart testedObject = new Cart();
56         testedObject.containsItem(item);
57         testedObject.addItem(item, true);
58         Item item2 = new Item();
59         item2.setItemId("EST-20");
60         item2.setListPrice(5.5);
61         item2.setUnitCost(2.0);
62         item2.setSupplierId(1);
63         item2.setProductId("FI-FW-02");
64         Product product2 = new Product();
65         item2.setProduct(product2);
66         item2.setStatus("P");
67         item2.setAttribute1("Adult Male");
68         testedObject.containsItem(item2);
69         testedObject.addItem(item2, true);
70         List result = testedObject.getCartItemList().getSource();
71         assertNotNull(result); // jtest_unverified
72         assertEquals(2, result.size()); // jtest_unverified
73         assertEquals("(EST-20-FI-FW-02) x 1", result.get(0).toString());
74         assertEquals("(EST-20-FI-FW-02) x 1", result.get(1).toString());
75         // No exception thrown
76     }
```

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:

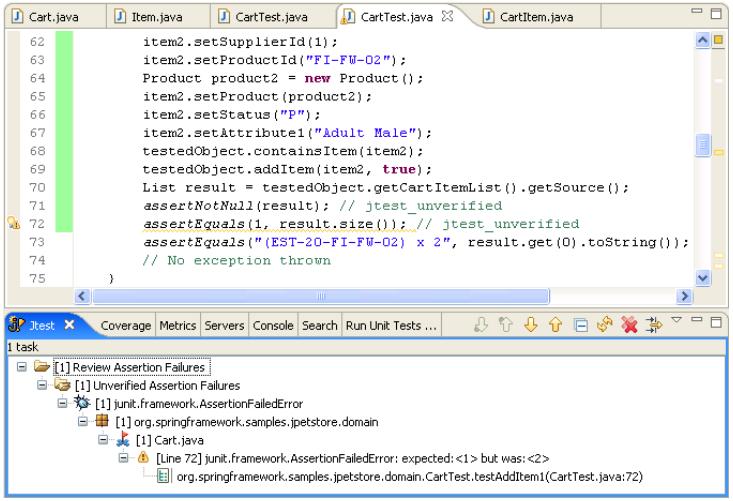
101 E. Huntington Dr., 2nd Flr., Monrovia, CA 91016, Ph. 626.305.0041 / Fax 626.256.6884
www.parasoft.com

```
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.



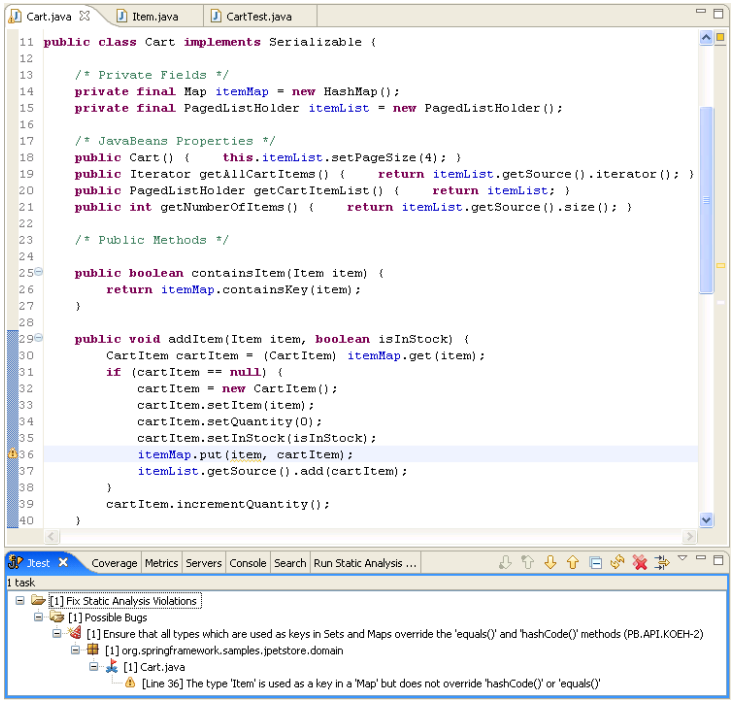
This test will be helpful for quickly checking which code changes resolve the problem. After a resolution is implemented, the test will continue to ensure that future code changes do not introduce a regression in this use case.

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.

101 E. Huntington Dr., 2nd Flr., Monrovia, CA 91016, Ph. 626.305.0041 / Fax 626.256.6884
www.parasoft.com

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.

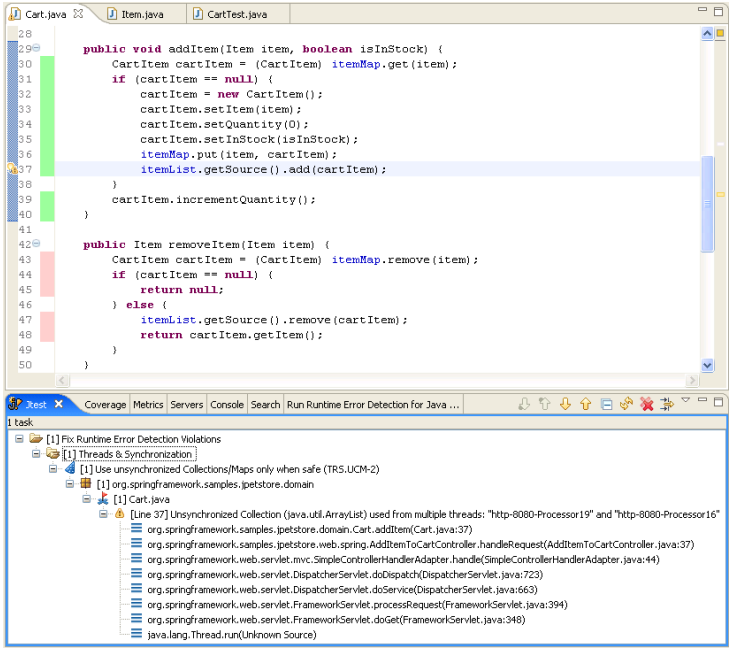
This problem is fixed by adding equals and hashCode methods in Item based on the item id. The functional test created by Jtest Tracer at the start of this exercise now passes.

101 E. Huntington Dr., 2nd Flr., Monrovia, CA 91016, Ph. 626.305.0041 / Fax 626.256.6884
www.parasoft.com

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.

101 E. Huntington Dr., 2nd Flr., Monrovia, CA 91016, Ph. 626.305.0041 / Fax 626.256.6884
www.parasoft.com

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

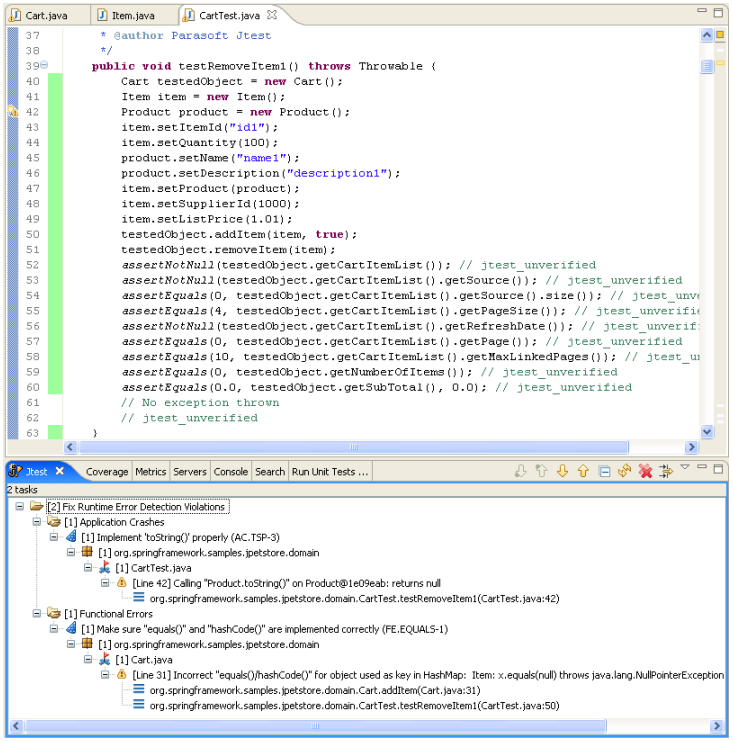
The problem report thus far has focused around adding the same item to a shopping cart twice… but what about other scenarios? Jtest reported that the 'removeItem()' method was still not covered. Removing items from the cart needs to be tested as well, but manual tests can be tedious–even manual tests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

101 E. Huntington Dr., 2nd Flr., Monrovia, CA 91016, Ph. 626.305.0041 / Fax 626.256.6884
www.parasoft.com



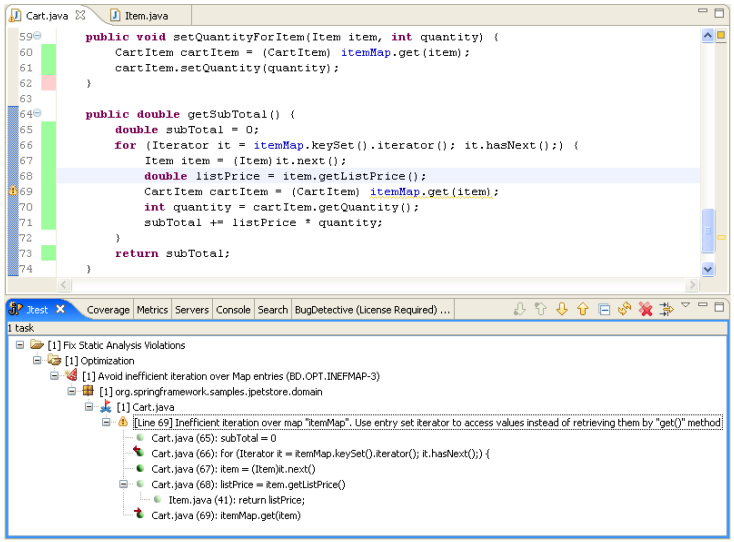
To satisfy the Java API contracts, the developers add the necessary checks to these core methods. Running the test case one more time ensures that the code is robust and no more runtime errors are reported.

```
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);
}
```


101 E. Huntington Dr., 2nd Flr., Monrovia, CA 91016, Ph. 626.305.0041 / Fax 626.256.6884
www.parasoft.com

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.

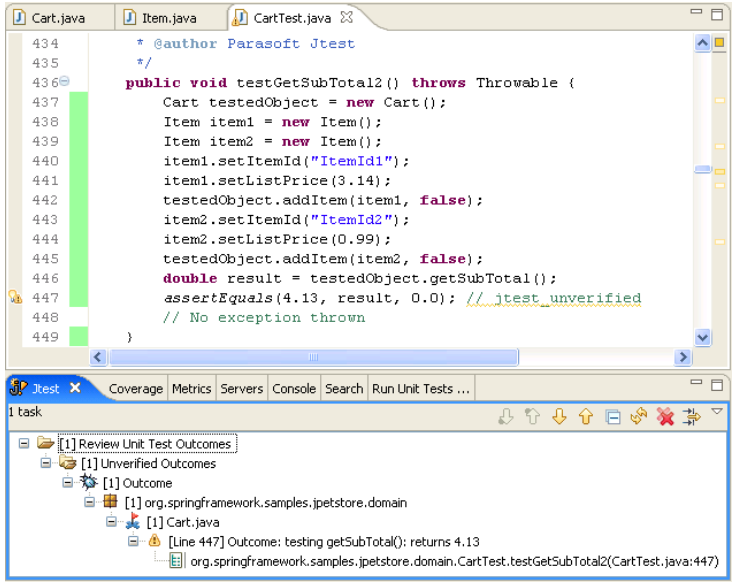
```
public double getSubTotal() {
    double subTotal = 0;
    for (Iterator it = itemMap.entrySet().iterator(); it.hasNext();) {
        Map.Entry entry = (Map.Entry)it.next();
        double listPrice = ((Item)entry.getKey()).getListPrice();
        int quantity = ((CartItem)entry.getValue()).getQuantity();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


101 E. Huntington Dr., 2nd Flr., Monrovia, CA 91016, Ph. 626.305.0041 / Fax 626.256.6884
www.parasoft.com

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.

101 E. Huntington Dr., 2nd Flr., Monrovia, CA 91016, Ph. 626.305.0041 / Fax 626.256.6884
www.parasoft.com

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.

101 E. Huntington Dr., 2nd Flr., Monrovia, CA 91016, Ph. 626.305.0041 / Fax 626.256.6884
www.parasoft.com

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.

Next Steps

- Learn more about the Parasoft Agile Solution
- Start a free evaluation now
- Contact Parasoft for additional information

101 E. Huntington Dr., 2nd Flr., Monrovia, CA 91016, Ph. 626.305.0041 / Fax 626.256.6884
www.parasoft.com

Parasoft Agile Solution

Parasoft's Agile development platform simplifies the process of continuously delivering high quality software that meets expectations. By integrating policy-driven task management with Automated Defect Prevention techniques, it ensures compliance to stakeholder and management expectations while driving unprecedented levels of productivity and application quality.

Agile Development Platform	Agile, Scrum, Extreme Programming, or Hybrid – Parasoft Concerto is a project planning and management tool that seamlessly integrates into any development environment/toolset to completely enable Agile software development methodologies.
Integrated Agile Testing	Parasoft facilitates Agile testing practices with a single comprehensive interface that easily integrates (out-of-the-box) with a wide range of IDEs.
Seamless Integration	Hook-up and go– out of the box. Easily connect with your existing development environments (bug-tracking systems, requirement management systems, etc.).
Policy-Driven Approach	Management expectations for development and testing are converted into actionable, measurable tasks. This helps the organization ensure process consistency while rapidly adapting to changing demands.
Planning and Task Definition	Helps dissect high-level user stories into reasonable and manageable development tasks. Estimate the available resources for each iteration and allocate tasks accordingly.
Task Management and Distribution	Distributes tasks directly to the developer's IDE based on manual assignments or business rules. Development progress is automatically monitored and visible to QA so they know what's ready for testing.
Continuous, Comprehensive Visibility	Provides continuous visibility into "when" and "how" requirements are being implemented. Automated correlation of the code, testing, and builds with requirements provides a comprehensive assessment.
Unobtrusive Monitoring via Automated Infrastructure	Automatically reviews and tests the project nightly, then notifies the appropriate team members if action is needed–ensuring that prescribed practices are applied consistently, without impeding velocity.
Manual Test Optimization	Adds consistency and repeatability to the manual testing process. Also enables change-based testing by identifying the manual test cases impacted by daily source code modifications.
Automated Defect Prevention and Detection	Reduces rework and cuts costs by automating the industry's broadest spectrum of integrated automated defect prevention and detection strategies.
20+ Years of Experience	The Parasoft approach has evolved over our 20+ years in business. We have experience helping 58% of the Fortune 500 companies deliver better software faster.

101 E. Huntington Dr., 2nd Flr., Monrovia, CA 91016, Ph. 626.305.0041 / Fax 626.256.6884
www.parasoft.com

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.

About Parasoft Concerto

Parasoft Concerto is a complete ALM platform that ensures quality software can be produced consistently and efficiently. By integrating policy-driven task management with Automated Defect Prevention techniques, it ensures compliance to management expectations while driving unprecedented levels of productivity and application quality.

Parasoft Concerto helps the team:

- Manage "when" and "how" requirements are implemented–from planning, though test and delivery.
- Determine exactly what tasks are needed to satisfy management expectations and regulatory guidelines.
- Integrate Automated Defect Prevention (ADP) practices across the SDLC as defined by centrally-managed, measurable policies.
- Unobtrusively monitor compliance and dispatch alerts when additional action is needed.
- Reduce rework and significantly improve application quality by automating the industry's broadest spectrum of integrated defect prevention and detection strategies.

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 facilitates:

- **Static analysis** – static code analysis, data flow static analysis, and metrics analysis
- **Peer code review process automation–**preparation, notification, and tracking
- **Unit testing** – JUnit and Cactus test creation, execution, optimization, and maintenance
- **Runtime error detection** – race conditions, exceptions, resource leaks, security attack vulnerabilities, and more

This provides teams a practical way to prevent, expose, and correct errors in order to ensure that their Java code works as expected. To promote rapid remediation, each problem detected is prioritized based on configurable severity assignments, automatically assigned to the developer who wrote the related code, and distributed to his or her IDE with direct links to the problematic code and a description of how to fix it.

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

101 E. Huntington Dr., 2nd Flr., Monrovia, CA 91016, Ph. 626.305.0041 / Fax 626.256.6884
www.parasoft.com