HOW TO INCREASE CODE COVERAGE & ROI WITH JAVA UNIT TESTING

Strategy Guide for Managers & Leaders

Table of Contents

3 Improve Java Unit Testing Processes

- 3 The Value of Unit Testing
- 3 Benefits & Challenges of Unit Testing
- 4 AI-Powered Unit Testing Tools

5 Strategies for Managers & Leaders

- 5 How Does Your Unit Testing Process Stack Up?
- 11 Best Practices for Updating Legacy Code With Confidence
- 16 The Two Big Traps of Code Coverage
- 20 The Simple Way to Increase Java Unit Testing ROI

Improve Java Unit Testing Processes

THE VALUE OF UNIT TESTING

Most development teams will agree that, although they don't like it, unit testing is, in fact, extremely valuable. Unit testing helps catch defects earlier in the development pipeline where they are easier to remediate, making the software better and accelerating delivery.

The challenge is that unit testing inherently requires technical, developer-oriented skills, as it is closer to the code. Despite the many tools that help with functional test creation, creating and maintaining unit tests still requires a lot of manual, time-consuming, often mind-numbing effort. However, the process of creating a unit test is a beneficial activity in and of itself, helping developers to look at their code through a different lens. Essentially doing an additional code review.

BENEFITS & CHALLENGES OF UNIT TESTING

Unit testing is iterative. In Agile processes, unit testing has a good track record of improving project outcomes. It's well-proven that testing earlier in the life cycle is the best way to detect and remove expensive and time-consuming system-level bugs. More specifically, some key benefits of unit testing include:

» **Increasing test coverage and efficiency.** Java unit tests are small and can execute quickly before the code is done. They are also much easier to remediate than other kinds of tests because of their limited scope.

» **Providing agility for agile processes.** Agile processes depend on efficient, repeatable, and automated test suites to ensure that each iteration isn't bogged down in a "big bang" test cycle. The success of agile and DevOps very much depends on development teams creating test suites that can be run efficiently, as well as testing as much functionality as possible.

» **Improved quality and security.** Software teams understand that testing is the only way to ensure software is doing what's required, but teams often fail to do enough testing or do it too late in the development life cycle. Security, in particular, needs to be built into an application at the earliest stages, so testing for security must be done as early as possible.

» **Reducing costs in the long run.** Software bugs found in released software can be 1000 times more expensive to fix than during development. Bugs found at early coding stages in unit testing are much cheaper than those found later. Short-term investment in testing automation and unit test infrastructure pays off in better product quality, security, and on-time delivery.

Unfortunately, regardless of these benefits, developers are still struggling with unit testing, despite the desire to achieve better results. These struggles include the following:

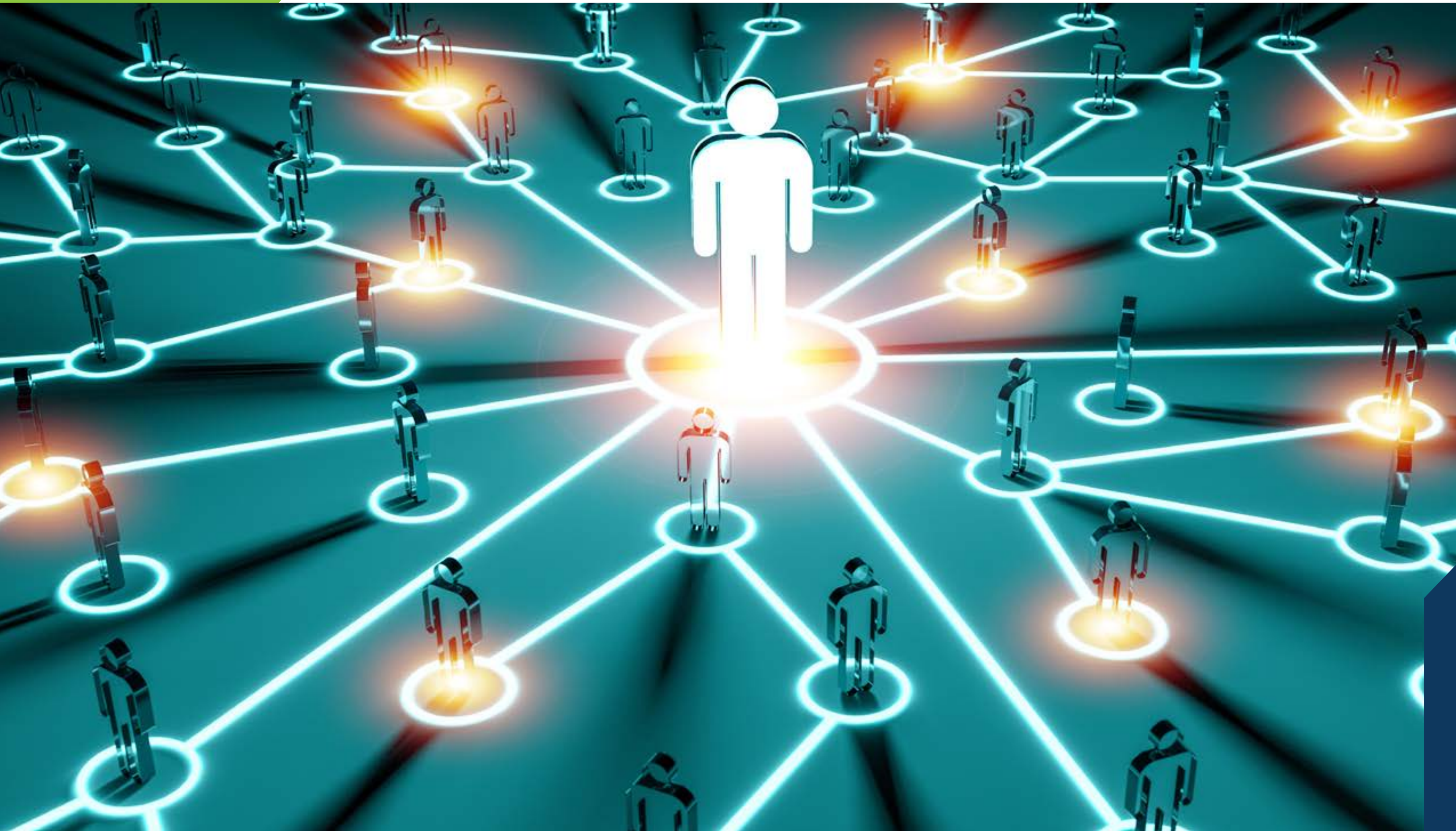
- **Test creation is extra work and often tedious.** Understandably, unit testing is extra work and is often seen as the least desirable aspect of programming. Creating a comprehensive test suite while trying to maintain project goals and deadlines are two competing pressures for development teams.
- **Test maintenance is expensive.** Just like code, unit tests require maintenance. Any code change may introduce changes in associated tests. Not only that, tests may fail due to codebase modifications, but the failures may seem unrelated to the changes, leading to the need to triage the failures. Extra maintenance creates "double the work" in many developers' minds.
- **Mocking and isolating units under test is difficult and time consuming.** It's critical to isolate units under test, but doing so requires mocking of dependencies, which can be a time-consuming process.

Software development teams must address the problems of test creation, isolation, and maintenance if they want to achieve the benefits of thorough unit testing. It requires a lot of development skill and effort, and it takes commitment and time to maintain test suites. A top-down approach that emphasizes unit testing priorities and goals can help the team to implement better processes. This ebook provides guidance on management strategies for improving unit testing processes and understanding the impact for return on investment.

AI-POWERED UNIT TESTING TOOLS

Java unit testing has clear benefits, and although most development teams and managers realize this, many are stymied by the effort of creating and maintaining tests. Using AI-powered unit testing technologies from Parasoft will help Java developers knock down these roadblocks, automating the mundane aspects of unit testing, including creation, mocking, and maintenance. To accelerate adoption, Parasoft Jtest leverages the development team's existing investment in test and mocking frameworks and gives back time to the developer while building quality into the product.

Taking advantage of AI-powered assistive technologies helps make unit testing tasks less tedious. If you can let automation do the simple tedious parts (that computers are good at), it frees your developers up to do the things that require actual human intelligence (which they are good at). The more you take advantage of automation, the less time unit testing will take, and the higher your ROI will be. Parasoft is a key partner for delivering quality at speed for unit testing, enabling development teams to be agile and deliver faster without sacrificing quality, making the business successful.



Strategies for Managers & Leaders

HOW DOES YOUR UNIT TESTING PROCESS STACK UP?

UNIT TESTING MATURITY MODEL

Unit testing is a cornerstone of a successful development testing strategy and should be adopted by any organization seeking to reduce risks and costs over the application development life cycle.

As a primary quality gate, unit testing:

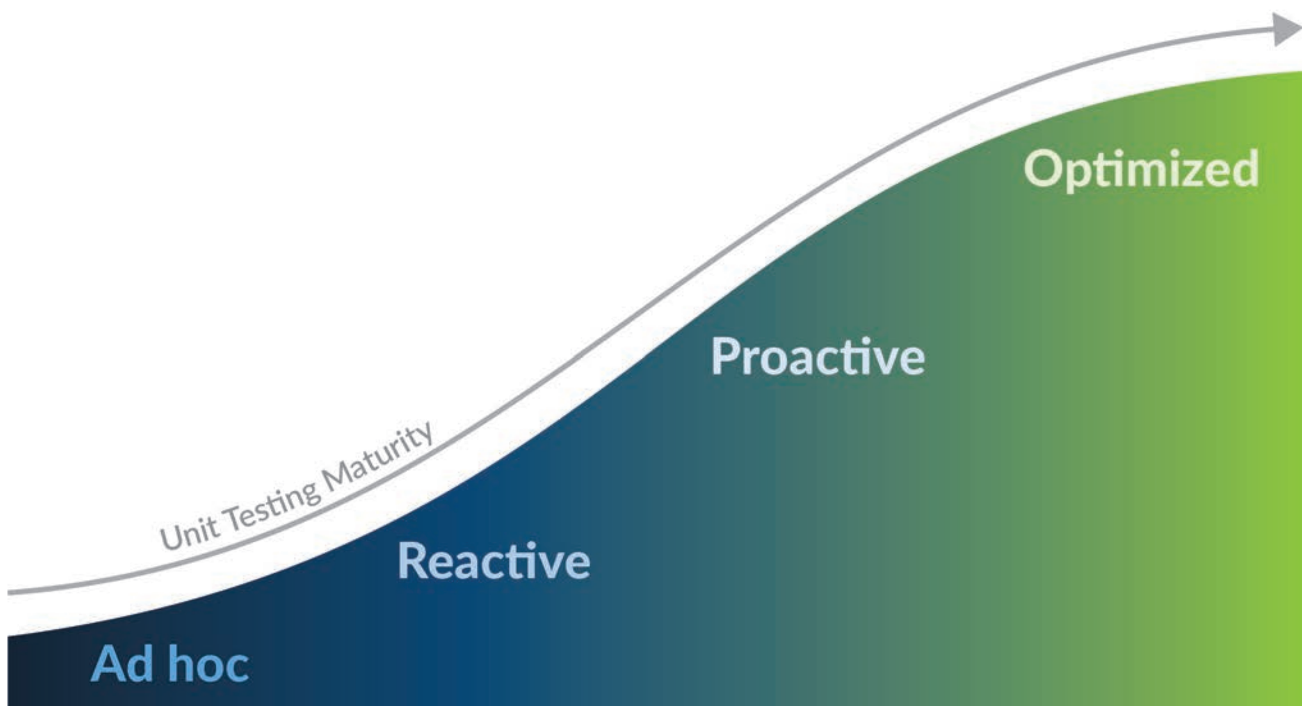
- Prevents the introduction of defects that would consume valuable resources if detected later in the development process.
- Reduces the amount of time required to remediate issues that are found during testing.
- Reduces risks associated with faulty software, which may result in costly litigation, brand erosion, or even loss of life.
- Verifies the functionality of the application at the earliest stage possible and serves as a way to provide artifacts for compliance traceability.
- Fundamentally improves the design and maintainability of the code.

Unit testing process maturity can range from simple, ad hoc work to reactive efforts, and with increasing maturity, to highly optimized efforts where policies are regularly

updated as part of a root-cause analysis effort to prevent defects from hiding until they're discovered late in the testing cycle.

Our maturity model consists of four stages.

1. **Ad hoc.** Test irregularly for basic functionality, based on the initiative of individual developers.
2. **Reactive.** Test intentionally but with inconsistent policies that may be done separately from product development.
3. **Proactive.** Test with consistent policies that are applied during the development process throughout the organization.
4. **Optimized.** Test strategically based on optimized test execution and high visibility into overall quality.



AD HOC UNIT TESTING

In the early stages of unit testing adoption, the so-called "ad hoc" stage, developers independently choose to create and run unit tests on their own initiative while developing functionality. There are no common policies nor is there an established development guideline to perform testing. Ad hoc unit testing characteristics include:

- No common practices, test requirements, or guidelines.
- Developers write unit tests in their own environment, irregularly. Not all developers are writing tests or even using the same tools or process.
- Testing is for basic validation that code works, without consideration for corner cases or covering all important business logic.

- Test reports are non-existent or unverifiable.
- Tests are not maintained and might not be stored in a common source code management (SCM) environment.
- There's no scheduled automation for test execution.
- Lack of established regression testing practice means reintroduction of defects is accepted as "normal" or "unavoidable."

Any pockets of maturity at this point are based on the experience and initiative of individuals. There's no centralization of assets. It's every person for themself.

Tests and test artifacts are typically created as one-off solutions and may or may not be stored on a local machine. Tests are created without consideration of the business or use case.

Signs that it's time to advance from this level include:

- The application is not adequately tested prior to system integration due to "lack of time."
- Business is lost or hard to win due to users' perception of application quality and instability.
- The business is experiencing poor customer feedback, downtime, production outage, and lost time and money, due to "emergency firefighting" at, or immediately after, release. User feedback, bug reports, and technical debt tend to interrupt feature development.

REACTIVE UNIT TESTING

The next level beyond ad hoc unit testing is reactive, where the team realizes software quality needs improvement and they're "reacting" to the business impact that poor software quality has caused them.

There may also be external factors, such as software security or safety standards compliance, that drive the need to adopt a more rigorous approach to unit testing. At this level of maturity, the team and management have committed to unit testing, but implementation is inconsistent across the organization, or even within a specific application's codebase.

The reactive level is characterized by:

- Testing policy and guidelines are in their infancy and are modified on the go. The team is learning "on the job."
- There are no common criteria for how much unit testing needs to be done in terms of code coverage, functionality coverage, or level of regression testing.
- There's limited adoption of best practices. Unit tests are written or modified at a different time than the functionality is implemented, likely after the code is considered done and possibly by a completely different team.
- Unit tests are stored and managed in a central repository with the product code but the maintenance of the test code itself might be lacking.
- Unit tests are run manually and locally during a smoke test or pre-release testing phase.
- Existing tests are rarely executed as code is developed and most likely

not run before code is submitted to a source code management (SCM) tool.

» Simple release criteria are used based on subjective assessment as to when/whether a release can proceed. Unit test failures are usually tolerated in order to satisfy release schedules.
» Generated reports might offer some visibility into testing status and progress but usually the information is insufficient to influence decision-making.
» At this stage of maturity, software teams have no expectations or strategy for validating legacy code with unit tests.

At the reactive level, the value of unit testing is recognized, but inconsistent definitions of measurement diminish the value.

Here are signs that it's time to advance from this level.

» If unit testing adoption is incomplete, then data isn't shared across teams and it's difficult to gauge status or level of quality across teams.
» Defects are still slipping through to later in the SDLC, causing delays in release schedules and/or production outages.
» Tests are only run at a late stage in each cycle. Tests aren't proactively maintained and too many tests are failing during regression testing.
» The increased focus on testing results in too much time spent on cleaning up and maintaining these tests. Test failures are hard to diagnose.
» Despite the new focus on quality, the organization still experiences extended delivery cycles due to lots of end-of-cycle rework for defects.

PROACTIVE UNIT TESTING

Moving beyond reactive testing, organizations realize the need to standardize the practice of unit testing across the organization. A common unit testing policy is clearly documented, and teams recognize unit testing as a part of the development process.

Developers have standardized on a testing framework and regularly create (and extend) unit tests in-sprint. The use of a data-driven approach to decision making is in its early stages. A metrics-driven policy increases visibility as a centrally defined process manages unit testing activities. The development team can easily see testing results in their development environment, and through the test results understand the impact of changes to the application.

Following are characteristics of a proactive maturity level.

- Unit testing is done earlier in each cycle. The question "how to test this user story?" is asked during sprint planning.
- Defined criteria for code and test coverage and defects that apply at release time.
- Tests run continuously and as part of an established development pipeline. Tests aren't created in isolation but rather as part of a consolidated test suite shared within the team. The team applies policies within this continuous process.
- Tests are stable and maintained for reuse. Test coverage includes legacy code. Unit tests for the legacy code are required before changes are made to legacy code.
- Unit test suites become regression tests for future changes.
- The test failure rate decreases over time due to policy changes and regular maintenance. Developers quickly respond to unit test failures and act preemptively with pre-commit workflows.
- Stubbing and mocking use extends coverage and test error conditions/ corner cases.

At this level, organizations start to see real benefits from an organization-wide unit testing policy, usually in terms of a tangible decrease in serious defects. Increased visibility and traceability enable management to make better business decisions. Unit testing is institutionalized as part of the process and is expected and measurable development behavior.

Here are signs that it's time to advance from this level.

- Testing still remains a bottleneck in a continuous CI/CD pipeline.
- Although data is helping make decisions, the team is still unsure what to test and how much testing is "enough."
- Code and test coverage is not increasing as much as expected. Despite best efforts, there isn't enough information to help drive test coverage.
- Defect rates are still too high in later stages of development. Overall quality is up but the team is motivated to do more.

OPTIMIZED UNIT TESTING

At the optimized level of unit testing, there's an organizational focus. Policies are regularly updated as part of a root-cause analysis effort to prevent defects from being discovered in QA. Software teams are testing smarter. They make full use of the data collected during the development pipeline to obtain visibility into overall application coverage, identify any testing gaps, and optimize and focus testing exactly where it's needed the most.

This maturity level is characterized by the following.

- Unit testing is now validating functionality and quality. Defects are detected earlier, fixed sooner, and reduce the downstream workload.
- Developers, testers, or managers kick off a test run based on any combination of technical and business requirements.
- The testing infrastructure automatically appropriates the needed environments, containers, VMs, and tests, then provides results into a centralized repository for analysis.
- Test and code coverage data drives the decision-making process. Increasing coverage is an established goal with it trending upward over time.
- Policies for code coverage for modified legacy code are used to minimize regressions and to optimize in-sprint testing.
- Testing policies are fully "enforced" but the workflows are optimized to make unit testing less of a burden.
- The adoption of test-driven development (TDD) makes unit testing a first-class citizen in the development process.
- Correlated and merged test results are shared with other testing practices (such as API and UI testing) bringing unit testing into a broader quality practice.
- Tests are directly associated with requirements (or user-stories) and individually correlated with source code. This provides bi-directional traceability that clearly shows the business requirements that have been tested and the business impact for test failures.
- The use of test impact analysis to run an optimized set of tests to validate recent application changes, reducing the amount of time before the impact of changes to the application is understood.
- Unit test policy is integrated within a controlled quality framework that includes security, performance, and reliability as key aspects, orchestrated from a centralized management interface.
- Data derived, analyzed, and coalesced from development and testing are available beyond the management team for visibility and business Intelligence integration.

The optimized stage of maturity sees software organizations evolving from a proactive mode of operation into a truly data-driven team that uses both established and fine-tuned policies while leveraging all the capabilities modern test automation tools have to offer. At the optimized stage, teams are analyzing data collected during development to drive better decisions with test impact analysis.

Would you like an assessment of your organization's unit testing maturity level?

Contact Us

BEST PRACTICES FOR UPDATING LEGACY CODE WITH CONFIDENCE

When you're dealing with legacy code, you need a sustainable way to manage change. Working with legacy code can be a barrier to an Agile workflow, but you can conquer the challenge by leveraging appropriate technologies.

WHAT IS LEGACY CODE?

Many people use the term "legacy code" to simply mean old code. But "old" and "legacy" mean different things to different people. Michael Feathers in his book *Working Effectively with Legacy* Code defines legacy code as "code without tests." Here, let's define legacy code as **any existing code that the team has limited knowledge about**.

Knowledge about the code could be incomplete for several reasons. Here are a few.

- The team acquired a project from another part of the organization.
- The original author left the team and took knowledge about the code with him or her.
- The functionality delivered by the code is no longer a business priority and has remained unchanged, resulting in forgotten details about the code.
- The code has too few tests, so there will be an unknown impact when changes are made to the code.

In any case, let's be clear: **legacy code is the rule, not the exception**.

Much of the software infrastructure in the world today runs on legacy code. The question is, then, how do we mitigate the risks associated with legacy code when we need to make a change? We'll give you some solutions for working effectively with legacy code.

LEGACY CODE IS A BARRIER TO AGILE

The problem with legacy code isn't its age—it's that you don't understand how changing it can affect existing functionality. The knowledge gaps associated with legacy code can become a barrier if you are transitioning to a new development methodology, such as Agile.

Agile has become the dominant methodology for creating software because it helps teams quickly iterate and release applications as soon as minimal marketable features are ready. Short and frequent development cycles are the hallmark of iterative development methodologies, but these approaches don't prescribe a solution for mitigating potentially problematic outcomes when you're dealing with legacy code. Rapidly iterating on code you don't understand is likely to introduce new issues.

The reality is that mature unit testing techniques are much easier to apply when starting new projects. For projects that have been around for a while, a set of best practices exists that can be applied to move the project in the right direction and give confidence in the quality of the changes being made.

BOOSTING COVERAGE ON LEGACY CODE

The risk that legacy code in production represents is relatively low. If it's been working for years, chances are it will work for years to come. However, the risk comes when you need to make changes to that legacy code to either fix a defect or introduce new functionality.

When organizations decide that they need to make changes to a legacy codebase that has few tests, they often (rightly) determine that they need to add tests for the codebase before they begin to make changes. The question then becomes, how to do this?

One possibility is to allocate in-house development resources to write tests before commencing with code changes. This can come with a huge cost, both in terms of money as well as time lost that could have been spent on developing new functionality. Parasoft has seen organizations estimate that it would take them weeks or even months of time to write the necessary unit tests for a legacy codebase.

A second possibility is to outsource the creation of unit tests to an external team. This may save on some of the costs, but the downside of this approach is that developers who don't understand the code end up writing the tests. This can lead to tests that are hard to maintain because they don't test the code appropriately or meaningfully enough.

A third possibility is to adopt a unit test generation tool to create the tests. This approach may save time, but it is even more likely to result in unmaintainable and meaningless tests than the second approach. For these last two approaches, a high code coverage number may be achieved but the tests may be of minimal value.

CREATE MEANINGFUL, MAINTAINABLE JAVA TESTS

To be clear, we're not advocating for blindly generating tests. Instead, use a tool that helps you rapidly create meaningful tests to cover your Java legacy code. Parasoft Jtest provides a point-and-click interface that gives developers an automatic test creation process based on the existing code. This capability generates a test suite with meaningful, maintainable, and extensible tests to achieve high code coverage.

UPDATE LEGACY CODE WITH CONFIDENCE

Here is a set of steps to update legacy code with confidence.

1. **Define the scope.**

   Rather than considering the entire legacy codebase, identify the portions of the codebase that need to be changed and focus your quality activities just on those areas of the code. In some cases, this may be the entire codebase.

2. **Automate existing tests and capture current code coverage.**

   If the legacy codebase already contains some existing tests, set up an automated repeatable test run that executes the tests. Configure a code coverage engine to capture the current code coverage achieved by the existing tests. Report the test results and the code coverage to a central location that all stakeholders can easily review.

   Low coverage represents potential risk associated with making changes to the codebase and indicates that additional tests need to be added before making changes. If the scope of the project is smaller than the entire legacy codebase, capture code coverage just for the scope that will be affected in addition to the entire project. Parasoft provides tools and dashboards to help you capture and view test results and code coverage in your projects.

3. **Add missing test coverage.**

   Armed with knowledge about the current state of the codebase, capture the current behavior of the system by creating tests where there are coverage gaps. We previously discussed the cost and challenges with creating a large set of tests for legacy code. To solve those problems, use a tool that helps you rapidly create meaningful tests to cover your legacy code.

   Parasoft Jtest's point-and-click interface enables developers to create a baseline of JUnit tests in bulk that include assertions based on the existing code. Jtest analyzes all possible code paths and generates a set of unit tests with human-like intelligence to cover those code paths, configuring object states and mocks as needed. For cases when the legacy code was not written with testability in mind, Jtest also includes the ability to create tests that directly access private methods. The resulting regression suite is meaningful, maintainable, and extensible.

When the legacy codebase has some existing tests, Parasoft Jtest can also perform a coverage gap analysis that identifies existing tests that can be cloned and mutated to reach untested parts of the code. This approach produces new tests while capturing the "human intelligence" already present in existing tests, increasing the return on your previous test creation investment.

When adding missing tests, strive for the highest level of coverage reasonably possible, but in most cases achieving 100% coverage on the entire codebase is not practical. A common goal is to achieve 80% code coverage. Aiming for more coverage than that decreases return on investment since the last 20% of code coverage is often very difficult and time-consuming to achieve.

4. **Modify code with confidence.**

   When you have good coverage from a functional perspective, start making changes to the codebase with minimal risk of breaking existing functionality. Modify the previously created tests as you go and add new tests for modified and new functionality.

5. **(Optional) Run static analysis tools and address violations.**

   Although only somewhat related to the topic of this whitepaper, another practice that teams can use to modernize a legacy codebase is to run static analysis tools on the code and fix reported violations.

   Static analysis can help find potential or hidden defects that are built into the code. When doing this, don't waste time on style or maintainability checkers. Instead, focus on violations that have a higher risk and are related to serious reliability or security issues. However, you should do this with care, because addressing static analysis violations can introduce unintended errors into the code. Because of this, it's best to perform this step after the code has been covered with tests. Static analysis results should be reported to a central location that all stakeholders can easily review.

6. Monitor quality metrics for the code.

As development proceeds on the codebase, continuously monitor the test results, code coverage, and static analysis results for the focused part of the code being changed. Monitor changes from build-to-build as part of the ongoing process, to ensure that the software quality doesn't take a turn for the worse. This allows teams to identify when best practices are not being followed and adjust behavior to ensure the quality of the changed code.

Parasoft provides a powerful analytics platform for capturing, correlating, and reporting test results, coverage analysis, static analysis violations, and other software quality data. The platform goes beyond static reporting. It also applies additional analysis to help you identify parts of the application affected by change. Leveraging the concept of resource groups, you can identify a specific set of files or directories and report coverage, static analysis violations, and metrics data against the scope of the codebase that is being changed.

Figure 1: Analyze your unit test results and manage code changes with a customizable reporting and analytics platform.

PRO TECHNIQUE: ENSURE COVERAGE ON MODIFIED CODE

The steps outlined above help prevent code changes from negatively impacting existing functionality, but you also need to establish good practices for the legacy codebase moving forward. Maintaining a high level of coverage by writing and updating tests as the code evolves requires buy-in on a cultural level. However, for a large codebase, from sprint to sprint it can be hard to determine whether an appropriate set of tests was created for the updated code. To solve this problem, teams should measure modified code coverage.

Modified code coverage refers to the amount of code that was covered by tests, when only considering the code that was modified between two different builds. By monitoring the coverage of the modified code, the team can focus on the parts of the code that are actively being worked on and have confidence that all changes are tested.

This technique works whether the starting point for the codebase is a high or a low level of coverage. In either case, enforcing a high level of modified code coverage will ensure that all new and modified code is well-tested. A common goal for modified code coverage is 80%.

Parasoft provides advanced technology that can automatically capture the coverage for modified code, notifying you when modified code (new or changed) fails to comply with the coverage policy.

BEST PRACTICES IMPROVE CODE QUALITY

The world's software runs on code that has been passed on from team to team. Dealing with legacy code is an everyday reality. The gaps in knowledge about the code represent potential risks as developers make changes to maintain or extend the functionality. The best practices for processes and technologies shared here should help you gain confidence to take on just about any codebase thrust upon your teams and improve the quality of your legacy code.

THE TWO BIG TRAPS OF CODE COVERAGE

Measurement of code coverage often receives a lot of focus when assessing unit testing practices because it gives important information about the state of the tests. The code needs meaningful coverage that indicates that the tests adequately test the software.

On one hand, many organizations don't know how much of their code is covered during testing, which is really surprising! At the other end of the spectrum, there are organizations for whom the number is so important that the quality and efficacy of the tests has become mostly irrelevant. They mindlessly chase the 100% dragon and believe that if they have that number the software is good. This can be as dangerous as not knowing what you've tested. In fact, perhaps more so since it can give you a false sense of security.

Code coverage can be a good number to assess software quality, but it's important to remember that it's a means, rather than an end.

Coverage is not important for coverage's sake. Coverage is important because it's *supposed* to indicate that a good job was done testing the software. If the tests themselves aren't meaningful, then having more of them certainly doesn't indicate better software. The important goal is to make sure the logic of the code is functionally tested, not just that the lines of code are executed.

Low code coverage can reliably tell teams when they are not testing enough. When there's not enough time and money to fully test everything, it's important to at least make sure that everything important is tested. Conversely, high coverage by itself doesn't necessarily correlate to high code quality. The picture is more complicated than that.

TRAP #1: "WE DON'T KNOW OUR COVERAGE"

Not knowing the code coverage seems unreasonable in today's environment because coverage tools are cheap and plentiful. In some cases, organizations may know their coverage number isn't good, so developers and testers are loath to expose the poor coverage to management. Hopefully, this isn't the usual case.

Another more common way to fall into this trap happens with organizations that have a lot of tests but cannot determine a real coverage number because they haven't found a proper way to aggregate the numbers from different testing practices. If teams are doing a combination of manual testing, functional testing, unit testing, and end-to-end testing, the coverage numbers from each practice cannot simply be added up. Even if each practice achieves 25% coverage, it's unlikely that the combined coverage is 100%. In fact, it's more likely to be closer to 25% than to 100%.

Coverage from multiple testing practices should be measured and combined in a meaningful fashion. A reporting and analytics tool like Parasoft DTP provides a comprehensive, aggregated view of code coverage. Parasoft's application monitors gather coverage data from the application directly while it's being tested and then send that information to Parasoft DTP, which aggregates coverage data across all testing practices, as well as across test teams and test runs.

An interactive dashboard allows teams to navigate the coverage data and make decisions about where to focus testing efforts. If multiple tests have covered the same code, it won't be over counted, while untested parts of the code are quick and easy to see. This shows what parts of the application have been well tested and which ones haven't.

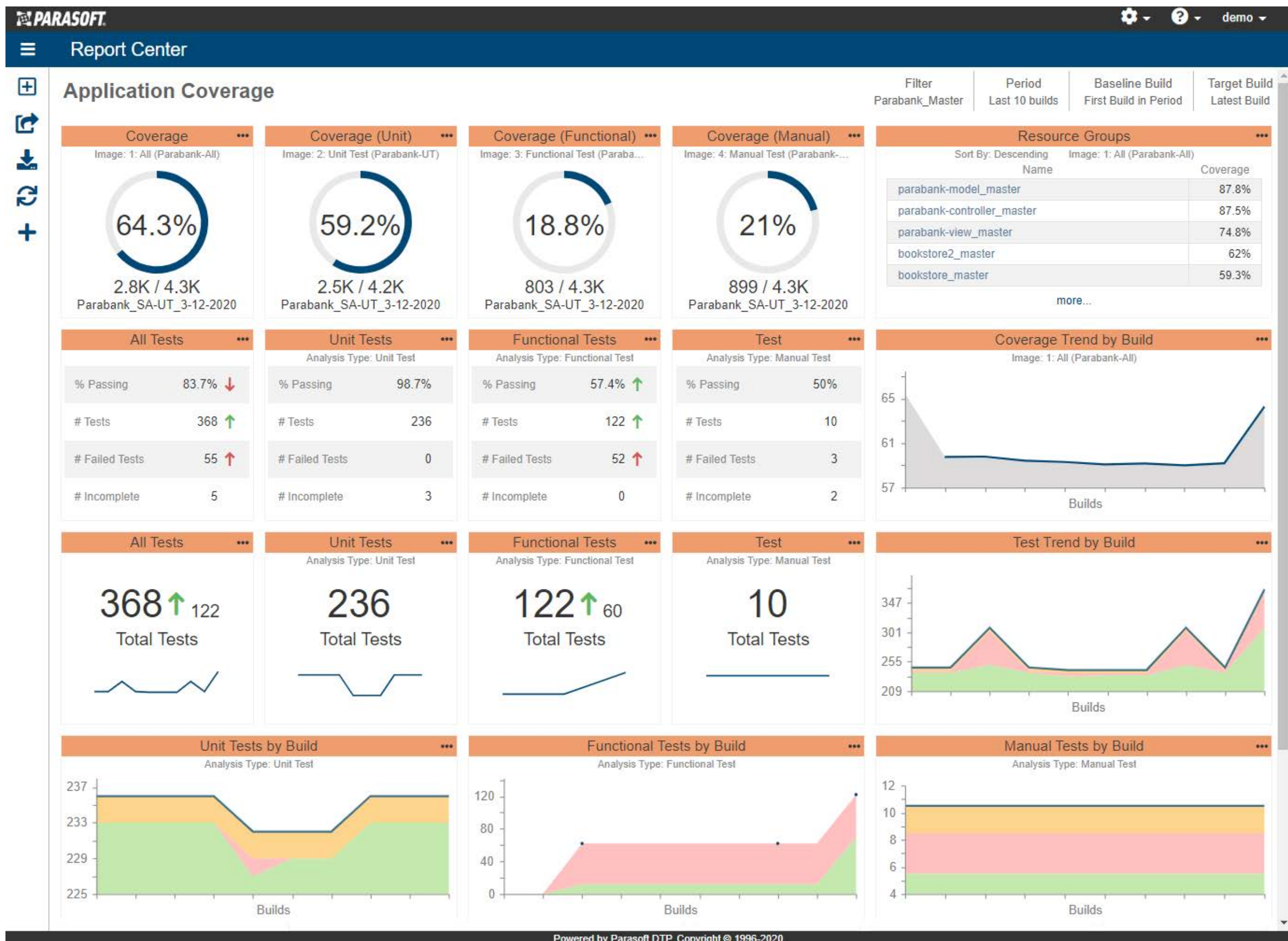


Figure 2: View coverage data that is aggregated across all testing practices.

With Parasoft technology, there are no more excuses for not measuring coverage.

Want to learn more about releasing faster with application coverage?

Get the Whitepaper

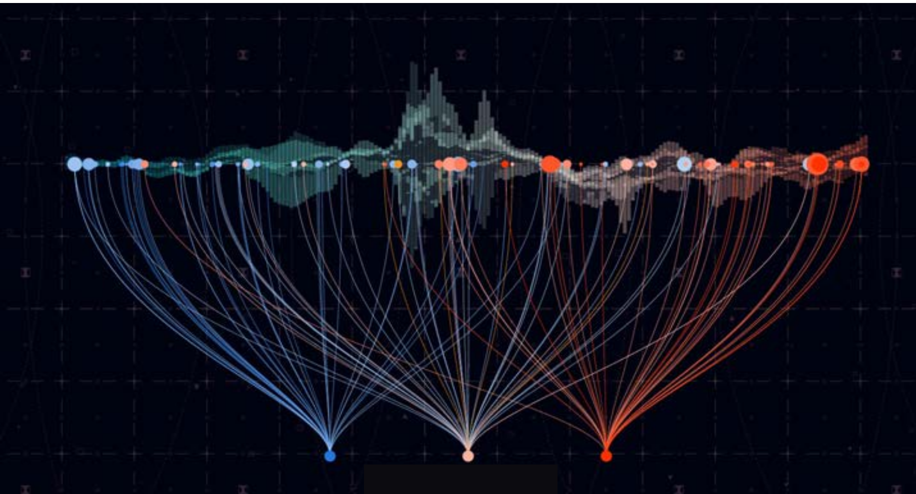
TRAP #2: "COVERAGE IS EVERYTHING!"

It's common for organizations or managers to mistakenly think that coverage is everything. Once coverage is being measured, it's not uncommon for managers to say, "Let's increase that number." Eventually the number itself can become more important than the testing. Perhaps the best analogy comes from Parasoft's founder, Adam Kolawa:

> "It's like asking a pianist to cover 100% of the piano keys rather than hit just the keys that make sense in the context of a given piece of music. When he plays the piece, he gets whatever amount of key coverage makes sense."

Therein lies the problem. Mindless coverage is the same as mindless music. The coverage needs to reflect real, meaningful use of the code, otherwise it's just noise.

And speaking of noise. The cost of coverage goes up as coverage increases. Remember that tests don't only need to be created, but they must be maintained going forward. If the team is not planning on running and maintaining a test, they should probably not waste the time creating it in the first place.



If tests are poorly designed, the amount of noise they produce increases in unexpected ways. Twice as many tests may mean two or even three times as much noise. The poorly designed tests create more noise than good tests because they may fail even when application behavior has not changed. In other words, they're not very maintainable. Poorly designed tests are a real danger.

In certain industries, such as safety-critical industries, the 100% coverage metric is a requirement. But even with this requirement, it's too easy to treat any execution of a line of code as a meaningful test, which is simply not true. There are two basic questions that can be asked to determine if a test is a good test.

1. **What does it mean when the test fails?**
2. **What does it mean when the test passes?**

Ideally, when a test fails, we know something about what went wrong, and if the test is well written, it will point the reviewer in the right direction to fix it. Too often when a test fails, no one knows why, no one can reproduce it, and the test is ignored. Conversely, when a test passes, the team should know what was tested. A passing test should mean that a particular feature or piece of functionality is working properly.

If one of these questions cannot be answered, there's likely a problem with the test. If neither of them can be answered, the test is probably more trouble than it's worth.

The way to escape this trap is first to understand that the coverage percentage itself isn't the goal. The real goal is to create useful, meaningful tests. This takes time and expertise. Writing unit tests for simple code is usually easy, but for complex real-world applications it can mean writing stubs and mocks and using specialized frameworks. This can take a lot of time and for teams that don't do it frequently, the nuances of the APIs involved must be learned and/or reviewed. Even for teams that are serious about testing, the time it takes to create a good test can be more than expected.

The Java development testing tool [Parasoft Jtest] contains technology to address this. Unit tests are generated with stubs and mocks appropriately configured to cover the code in question. Its Unit Test Assistant helps developers take on the tedious tasks of configuring mocks and stubs correctly. It can also expand existing tests in a useful way to increase coverage. Parasoft Jtest helps teams create good unit tests from scratch as well as makes recommendations to improve test coverage and test quality for existing tests.

Coverage is important, and improving coverage is a worthy goal. But chasing a coverage percentage isn't nearly as valuable as writing stable, maintainable, and meaningful tests.

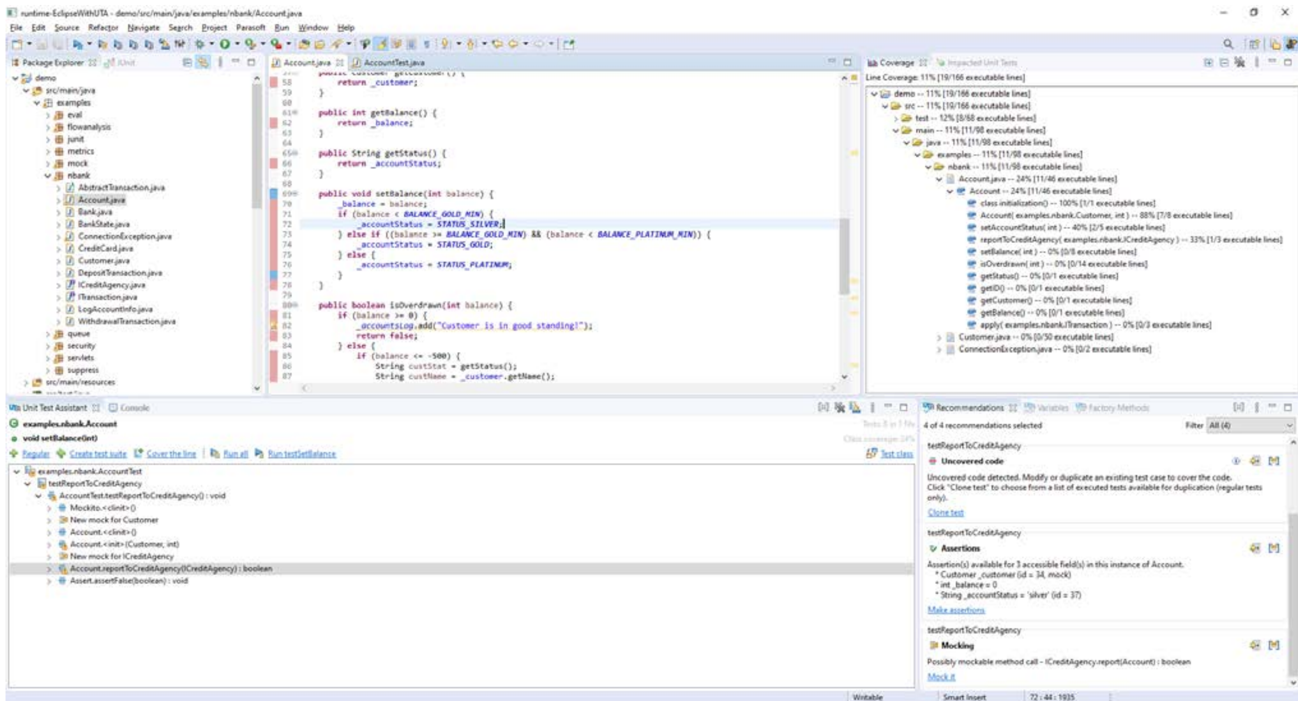


Figure 3: Jtest's Unit Test Assistant creates and executes tests, provides recommendations for improving tests, and provides details on code coverage for your application.

THE SIMPLE WAY TO INCREASE JAVA UNIT TESTING ROI

Capers Jones' body of work is a great asset to software developers in any industry. He has spent his career studying the successes and failure of software projects, much of which culminated in 2011's The Economics of Software Quality. Although enterprise Java developers may feel this collected information doesn't apply to them, they're missing out on some valuable insight.

Here are some sobering statistics that apply as much to developers using Scrum and the latest software tools as they do to any other type of software development. As a whole, for every dollar the software industry spends on development, 50 cents of that dollar is spent on maintenance and finding and fixing bugs. However, most forms of testing only remove about 35% of bugs, leaving a majority of bugs in the software despite a team's best efforts.

Jones was very much an early proponent of shift-left testing, although he may never have used the term. Finding and fixing defects (especially defects in requirements, analysis, and design) as early as possible in the software life cycle is key to higher software quality.

Interestingly, Jones stood out by making a good case against the cost-per-defect metric used by many vendors to calculate tool ROI. If anything, cost-per-defect undersells the ROI of automation tools and the shift-left testing effort.

CURRENT TESTING SUCCESS RATES

We already know that software developers are spending half of their development budget on finding and fixing bugs, and that current test methods are still leaving two thirds of the bugs in the software. Here are some more interesting statistics.

- Even using the latest software development methods, defect removal rate is roughly 85% in the best projects. This means 15% of the defects are yet to be found and make their way into the final product.
- About 6% of test cases have bugs in the test cases themselves.
- In large projects, as many as 20% of regression tests are duplicates, which add to testing costs but do nothing to increase product quality.
- About 7% of bug fixes include new bugs. So, even as bugs are being resolved, there are new ones being introduced.

HOW TEST AUTOMATION & THE PARASOFT JTEST UNIT TEST ASSISTANT CAN HELP

Unit testing is a necessary but a tedious requirement of software development. Test automation helps by removing much of the tedious processes from the developer, but test creation and maintenance continue to be key problems facing Java developers when tackling unit testing of their code. Let's consider the economic benefits realized by automatic unit test creation and the impact it has on testing efforts.

In a survey conducted by Parasoft, we learned that a majority of developers are spending about 40% of their time on unit testing. Considering a two-week development sprint cycle consisting of 10 days, four days are devoted to testing. It's easy to see why testing can become a drag that slows down iterative and agile software development. In addition, the current testing success rate means that this amount of time still isn't enough or, more importantly, a way to decrease this time while improving outcomes is needed.

We've also retrieved data from customers who use Parasoft Jtest for Java testing and it's encouraging. Java development teams see a decrease in their unit testing effort by a minimum of 50%. In other words, they can complete their four days of unit testing in two days using Jtest's Unit Test Assistant.

This type of savings on a per-sprint basis is impressive but becomes more so when this is compounded over many sprints in a typical project. **For example, if a typical project releases every three months, with six sprints of development, Jtest saves the equivalent of 1.2 sprints or 12 days of development effort for every member of the team.**

With these kinds of savings, software teams can increase their productivity without sacrificing quality, resulting in a considerable decrease in delivery times. Better quality and delivered on time—or even early—are serious economic benefits.

THE REAL ROI OF IMPROVED QUALITY

There's more to return on investment for increased quality than the cost to fix a defect. Fixing a bug early in the life cycle is cheaper and doing so saves money. Although this is one metric and even alone it's sufficient to justify the investment in better quality, it actually undersells the ROI.

One of the leading causes of project delays is missed defects and security vulnerabilities that make their way into late stages of a product development cycle. Of course, it's cheaper to find and fix these earlier because the development team still has the code fresh in their minds and hasn't moved on to the next iteration (or project, for that matter).

Using only the cost-per-defect metric and approach to calculating ROI, consider the example above with a team of 20 people working on a project with a loaded labor rate of $100 per hour. This team, using new test automation tools with all the benefits thereof (shift-left the identification of defects), discovers 20 more defects than they did in the previous sprints.

Finding and fixing these bugs early might require three hours per defect for a total of $6,000. Finding and fixing these bugs later in the integration or system testing might triple the effort, for a cost of $18,000.

Simplistically, for this sprint, the ROI is $12,000. Sounds great, right? However, this doesn't factor in the two days of development time savings for the sprint for an additional $32,000 in savings and increased productivity.

Looking at the big picture, we can see that decreasing the development time for the entire release is the real money saver here, not the cost-per-defect. The real payoff for shifting left is achieving or beating project schedules and goals.

Consider the above example again. This time, look at the ROI in terms of the whole development team finishing the release early by 12 days. For this team, that's 12 days of 20 people which amounts to $192,000! Although this simple example seems obvious, it does point out that ROI from tools is realized at the team level, when products are delivered to market faster without sacrificing quality.

INCREASE UNIT TESTING ROI: UNIT TEST EXECUTION

Again, considering the example above, let's assume the development team creates more and more unit tests, ranging from solitary (isolated) unit tests that take seconds to run, to sociable (integration-level) unit tests that take significantly longer to run.

The full unit test suite execution time eventually grows to two hours, which is a lot of time for a software development team to wait before they can get feedback on their code changes. The team follows an Agile methodology, so they build the project multiple times a day to provide frequent feedback, but due to the two-hour unit test execution time, they can only execute three builds a day, corresponding to six hours of overall unit test execution time daily. This can result in inefficiency in the development process since other team members may start depending on code changes before feedback from the automated tests is available.

To reduce test feedback time, the team can optimize unit test execution with Parasoft Jtest. Jtest reduces the time it takes to execute unit tests with AI-enabled test automation, by executing an *optimized* set of unit tests based on the code that has changed, rather than running the full suite of tests.

This optimization happens within the developer's IDE before code is checked in, as well as during the CI builds, while the full test suite continues to execute nightly. By reducing the number of tests that get run to just those needed to validate the changes, the overall build time gets dramatically shortened and enables quicker feedback to the software team. Now the team can get more builds per day, which optimizes their productivity during the active part of the day, resulting in a faster delivery time.

CONCLUSION

Traditional unit testing methods consume a large amount of software development time, and the outcomes from these approaches need improvement. Parasoft Jtest can help reduce the unit testing effort by 50%, which pays off handsomely in terms of increased quality and decreased sprint schedules.

The ROI is significant when you consider how much unit testing impacts the team and the project as a whole. Unlike simple cost-per-defect analysis, finishing projects on time and meeting requirements on goals is the big payoff, and saving time and money while doing it makes it that much better.

Parasoft Jtest helps businesses deliver quality at speed. Using this integrated Java testing tool, developers can reduce late-cycle defects and focus more time on new feature development for the business.

Developers also benefit from immediate feedback. They can rapidly identify whether their code changes are breaking functionality in the application and address it quickly. Parasoft Jtest enables development teams to increase their productivity and deliver faster without sacrificing quality, making the business succeed.

TAKE THE NEXT STEP

Request a demo to learn how Parasoft Jtest can help you reach your technical and business goals.

ABOUT PARASOFT

Parasoft helps organizations continuously deliver quality software with its market-proven, integrated suite of automated software testing tools. Supporting the embedded, enterprise, and IoT markets, Parasoft's technologies reduce the time, effort, and cost of delivering secure, reliable, and compliant software by integrating everything from deep code analysis and unit testing to web UI and API testing, plus service virtualization and complete code coverage, into the delivery pipeline. Bringing all this together, Parasoft's award winning reporting and analytics dashboard delivers a centralized view of quality enabling organizations to deliver with confidence and succeed in today's most strategic ecosystems and development initiatives—security, safety-critical, Agile, DevOps, and continuous testing.