Test-Driven Development and Unit Testing with Parasoft Concerto

What is Test-Driven Development (TDD)?

Test-Driven Development (TDD) was first introduced as a key part of Extreme Programming. In a nutshell, it involves developers creating tests that verify code requirements before the developers actually write the code to implement that requirement. TDD tests are typically implemented using unit test frameworks: either as "pure" unit tests, or as slightly less granular component tests. This paper will use the term "unit test" to refer to any test that is implemented with a unit testing framework such as JUnit, CppUnit, NUnit, etc..

Once written, each test is added to an automated regression test suite, which runs on a regular basis (e.g., as part of continuous integration). Tests will fail until the code is implemented—providing the developer a constant reminder that additional implementation work is needed. Once the test passes, code can be refactored as needed—without fear that the refactoring might inadvertently break or change the verified functionality.

If a test failure occurs at any point after the code was implemented, this alerts the team that previously-verified functionality has been impacted by a change to the code base.

Test Driven Development and the Agile Manifesto

Test Driven Development is critical for supporting Agile Manifesto principles such as:

- Enabling early and continuous delivery of valuable software.
- Welcoming changing requirements.
- Delivering working software frequently.

Test-Driven Development Obstacles

The key obstacle that development teams face in truly adopting TDD is the time required to build and maintain the required test cases. Without process automation, time constraints allow the TDD practice to become “optional.”

Time constraints stem from two main sources:

- **Setting up good tests can be complex:** The process of writing a test that effectively verifies a requirement involves both creativity and solid coding/testing skills. The complexity of creating effective unit tests adds to the challenge: setting up the proper initial conditions for realistic unit tests can be difficult and time consuming.
- **The test suite has to be kept in sync with the evolving application:** Ideally, as each TDD test is created, it is added to a regression test suite, then run regularly against the evolving code base to alert the team when any code base modifications break or change the already-tested functionality. However, in order for this strategy to deliver the desired results, the team needs to keep the test suite in sync with the evolving application; otherwise, the test results will be so noisy that they will be ignored—or the entire test suite will be abandoned.

Considering the compressed nature of today's development schedules and the constant pressure to do more with less, it is not surprising that such tasks tend to get dropped unless TDD tasks are made a natural and non-negotiable part of the team's day-to-day workflow.

Parasoft addresses these obstacles in a number of ways:

- **To make tests more valuable—**we offer technology that automatically correlates requirements, tasks, code, tests, builds, developers, and artifacts—as well as add coverage tracking, advanced reporting, and runtime error detection during test execution.
- **To ensure that TDD practices are maintained—**we offer an automated infrastructure that notifies the developer if a unit test is not yet associated with a task or requirement.
- **To facilitate unit test development—**we offer technologies that reduce the work required to implement realistic and useful test cases.
- **To facilitate unit test suite maintenance—**we establish an automated infrastructure that automatically assigns each test failure to the responsible developer and distributes it to his IDE to facilitate review and response.
- **To ensure that manual tasks become a continuous and natural part of the workflow**—we help you implement a policy-driven process. Management expectations are set by defining what practices are required as well as when and how to apply them. Related tasks are then seamlessly integrated throughout the SDLC and unobtrusively monitored for compliance.

Unit Test Case Development

The process of writing tests that verify requirements is always a creative one. The developer really needs to think about the code and how to test it effectively. But automation can reduce the work required to implement realistic and useful test cases.

For instance, Parasoft's object repository stores initialized objects, which are very helpful to use when you're trying to set up realistic initial conditions. Parasoft's stub library can be used to "stub out" external references so the unit can be tested in isolation. Also, test case parameterization can be used to feed additional inputs into the test cases— inputs from a data source, or a stream of automatically-generated inputs using corner case conditions.

As tests are being developed, static analysis can be performed to check that the unit tests follow industry-standard unit test guidelines. For instance, rules check the proper implementation of:

- Assertions
- Test methods
- Error message strings
- Fail methods
- Setup and teardown methods
- Constructors
- Test documentation

Once unit tests are developed, they can be executed in Parasoft's industry-leading unit test framework. Using this framework, teams can:

- Centralize execution and reporting for all unit tests.
- Track targeted or cumulative test coverage using multiple coverage metrics.
- Automatically generate additional unit tests using the TDD unit tests as a template.
- Extend the regression test suite with automatically-generated regression test suites that detect regressions in application behavior that your TDD test cases do not cover.
- Track which test cases failed, since when, and who is responsible for fixing each failure.

In addition, runtime error detection can be performed as the TDD and other tests execute. This is key for efficiently identifying defects that manifest themselves only at runtime (for example, file overwrites) and zeroing in on the root causes of the application crashing, running slowly, or behaving unpredictably. Categories of defects detected include race conditions, exceptions, resource & memory leaks, security attack vulnerabilities, null pointers, uninitialized memory, buffer overflows, and more.

Unit Test Suite Management and Maintenance

To ensure that test suite maintenance is as painless as possible, Parasoft establishes a supporting infrastructure and workflow. Each night the infrastructure automatically:

1. Gets the latest code from source control.
2. Runs the entire regression test suite.
3. Determines what assertions failed as a result of the day's modifications.
4. Figures out which developer caused each assertion failure.
5. Distributes this information to the responsible developers.

When the developers arrive at work each morning, they import the results into their desktops or IDEs, review any assertion failures reported for the code they authored, and respond to them. When they respond, they are either addressing functional defects in the code or updating the test to reflect the correct behavior of the code. With this daily process, a little effort each morning goes a long way in terms of extending the test cases' value and life span—and also, of course, in terms of exposing unexpected impacts as they are introduced

Unit Test Adoption: Automated assertion failure assignment and distribution is key

We learned the value of this ourselves, in the Parasoft development team. Years ago—before we standardized our current testing policy and infrastructure—we started examining the unit testing process that we were using at the time. We found that developers would write functional unit tests to verify requirements as code was implemented. However, when test assertions later failed as the application evolved, the failures weren't being addressed promptly.

In each case, someone needed to review the failure and decide if it was an intended change in behavior or an unwanted regression error. Our open source unit test execution tools couldn't tell us what tests were failing from whom, for what, since when. Instead, our nightly often produced a report that said that something like 150 of our 3,000+ tests failed. Each developer could not determine if his own tests failed unless he reviewed the full list of failures—one at a time—to see if it was something that he should fix or if it was a task for someone else. Our system was lacking accountability.

Initially, we tried asking all developers to review all test failures, but that took a lot of time and never became a regular habit. Then, we tried designating one person to review the test failures and distribute the work. However, that person was not fond of the job because it was tedious and the distribution of tasks was not well-received by the others.

Eventually, we built automated error assignment and distribution into our unit testing products, and started using it internally. Now, assertion failures are automatically assigned to the responsible developers based on source control data. If a developer causes one or more assertion failures, he is notified via email, imports only the relevant results into his IDE, and resolves them. As a result, failures are now resolved in days instead of months.

Policy-Driven Development

Parasoft's policy-driven approach converts management's expectations for TDD—as well as other development and testing activities—into actionable, measurable work tasks. An automated infrastructure runs in the background, orchestrating mandated processes (manual and automated), and continuously monitoring policy compliance. Notifications are generated only when actions don't align with policy expectations.

For example, in terms of TDD, the system could be configured to generate notifications if:

- Code for a new requirement is checked in—but no test case is added to validate it.
- Added test cases do not follow test case guidelines.
- Test cases fail.
- Test cases are added, but not properly correlated to a specific requirement.

Test-Driven Development: Supporting Capabilities

The Parasoft Concerto platform not only facilitates the development and execution of tests that verify each requirement, but also provides additional capabilities that help the team more rapidly achieve the ultimate goals of TDD:

- Working software that provides valuable functionality.
- A flexible, easy-to-evolve code base.

Capability	Description
Policy Definition and Management	Parasoft Policy Center converts management expectations into actionable, measurable tasks. This helps the organization ensure process consistency while rapidly adapting to changing market trends, regulatory environments, and customer demands.
Task Definition and Management	Parasoft Concerto interacts seamlessly with the IDE for task definition and management. Users are automatically notified of tasks that are assigned to them or that have been generated based on a "policy."
Correlation of Requirements, Tasks, Codes, Test Cases, Developers, and Artifacts	Parasoft Concerto correlates requirements with automated and manual tests, source code, and development/testing tasks. The current level of verification for each requirement or task (including task pass/fail status and coverage) can be assessed at any time by back tracing to all associated tests. This correlation also enables change-based testing, which identifies exactly which tests are impacted by source code and requirement modifications.
Unit Test Cases *Unit Testing Framework*	Parasoft assists the team to start writing, running, and tracking unit test results before code is written—providing the framework for Test-Driven Development. Parasoft also establishes a continuous regression testing process that instantly alerts the team if code modifications impact existing functionality. This provides a safety net that helps developers rapidly change code with confidence.

Capability	Description
Unit Test Cases *Extended Execution Environment*	Parasoft's extended execution environment centralizes execution and enhances reporting for all of your manually-written and automatically-generated xUnit (JUnit, CppUnit, NUnit, Cactus, HttpUnit, etc.) unit test cases. To promote fast remediation, each test failure or exception detected is prioritized, assigned to the developer who introduced the problem, and distributed to his or her IDE with direct links to the problematic code. This provides developers instant feedback on whether their code changes broke the existing functionality. In addition to this automated error assignment and distribution, the extended execution environment also provides capabilities such as: • Coverage analysis • Debugger integration • Runtime error detection
Unit Test Cases *Test Optimization*	Parasoft provides a variety of technologies to help you extend your manually-defined and automatically-generated test cases, enabling you to increase their value with minimal effort. Optimization technologies include: • Stub Creation / Management • Object Repository • Test Case Parameterization • Data-Driven Test Cases • Graphical Test Case Editor
Unit Test Cases *Automated Test Generation*	To extend the coverage of the TDD tests, Parasoft can automatically generate additional test cases for exception detection and regression testing. For exception detection, automatically-generated test cases use corner conditions to check responses to unexpected inputs, exposing potential reliability problems. For extending the scope of regression testing, automatically-generated behavioral regression test cases capture the code's current behavior—helping you rest assured that your changes don't change or break some application behavior that you did not explicitly test.
End-to-End Test Cases *Functional Testing*	Parasoft's solution isolates and tests individual application components for correct functionality without requiring scripts. Additionally, dynamic data can be stubbed out with constant data to reduce test case noise. End-to-end tests continuously validate all critical aspects of complex transactions which may extend through web interfaces, backend services, ESBs, databases, and everything in between. Parasoft's solution eliminates the need for writing scripts, and instead provides a simple GUI interface to create tests. To provide additional flexibility, tests that are more suited to the control that scripting offers can be generated as JUnit tests and extended using the Java programming language.

Capability	Description
End-to-End Test Cases *Regression Testing*	With Parasoft's solution, you can execute a component-based testing strategy that ultimately allows you to focus on the impact of change. Parasoft's continuous regression tests are applied to the multiple layers throughout your system. These tests will then immediately alert you when modifications impact application behavior—providing a safety net that reduces the risk of change, enabling rapid and agile responses to business demands. You can evolve regression suites for heterogeneous systems rapidly and easily, then leverage the same functional regression test suites—including complex business scenarios and rich message validation assertions—into load tests without scripting.
End-to-End Test Cases *Load & Performance Testing*	Parasoft solution's performance testing not only monitors the server's response rate with the specified number and mixture of simultaneous requests, but also verifies whether the test loads cause functionality problems. We facilitate performance testing by allowing you to load test your existing end-to-end functional test suites (which may span from the web interface, through services, to the database) according to preset or customized load test scenarios. Preset scenarios can be used to verify robustness, scalability and durability. You can easily customize these scenarios to use different test cases, load levels, load distributions, and so on. You can also distribute virtual users across remote server machines to simulate extreme loads and/or test from different locations. Support is also provided for load testing non-Parasoft components such as unit tests or lightweight socket-based components. This provides teams an integrated solution for their various load testing needs.
End-to-End Test Cases *Application Behavior Virtualization*	Evolving test suites for complex applications is a difficult endeavor due to the interdependencies between systems, messages and business processes. This issue is compounded when you are working with an incomplete or evolving system/application. Parasoft offers a wide range of capabilities to assist you in rapidly configuring stubs that meet any data, performance or behavior conditions in real-time. All stubs can be deployed locally or made available as a service so that different teams or business partners can collaborate on evolving their components within a business system.
Automated Peer Code Review	Parasoft automates and manages the peer review workflow. Our Code Review module, which automates preparation, notification, and tracking of peer reviews, addresses the known shortcomings of this very powerful inspection method. It automatically identifies updated code by scanning the source control system (if the organization reviews code after source control check in) or the local file system (if code is reviewed prior to check in), matches the code with designated reviewers, and tracks the progress of each review item until closure. With the Peer Review module, teams can establish a bulletproof review process where all new code gets reviewed and all identified issues are resolved. For teams working with Extreme Programming, this capability can be used to facilitate pair programming.

Capability	Description
End-to-End Test Cases *Manual Testing*	Parasoft's solution allows the developer to establish the "success criteria" as well as define the expected manual steps that need to be tested. This manual test case/suite is then automatically correlated to a requirement, task, code, automatic test case and any other related artifact. If a manual test is impacted by a code change in associated code, that test is immediately flagged for retest.
Change-Based Testing	Parasoft's change-based testing helps you optimize your testing efforts by identifying and executing only the test cases directly related to your most recent source code modifications. Not having to test the entire system after each modification yields tremendous productivity improvements.
Static Code Analysis Pattern-Based	Parasoft's pattern-based code analysis monitors whether code follows industry-standard or customized rules for ensuring that code meets uniform expectations around security, reliability, performance, and maintainability. Over 15 years of research and development have gone into optimizing Parasoft's patented pattern-based analysis engine. Although the power of the pattern-based analysis is the combination of analysis engine technology and the strength and flexibility of the rule library, our engine uses various code analysis techniques to optimize the analysis and reports. Pattern-based static code analysis can also be used to ensure that tests are implemented correctly and completely
Static Code Analysis Data Flow	Parasoft's data flow static analysis provides automated detection of runtime errors without requiring the software to actually be executed. This enables early and effortless detection of critical runtime errors that might otherwise take weeks to find. We statically simulate application execution paths which may cross multiple units, components, and files to identify paths that could trigger runtime errors. To simplify defect analysis, a complete analyzed path trace for each potential defect is reported in the IDE, and automatic cross-links to code help you quickly jump to any point in the highlighted analysis path. This ability to expose these errors without executing code is especially valuable for teams with legacy code bases lacking robust test suites or embedded code, where runtime analysis and detection of such errors is not effective or possible.
Static Code Analysis Code Metrics	Parasoft calculates various metrics for your code to help you assess your code base and monitor changes. Code metrics calculation identifies brittle or overly-complex code that could impede agility or reuse. It also helps you better understand code complexity and assess the potential impacts of an anticipated code change. This enables you to make more informed decisions as to how to modify, refactor, and test it. In addition to reporting calculations for industry-standard metrics such as Inheritance Depth, Lack Of Cohesion, Cyclomatic Complexity, Nested Blocks Depth, Number Of Children, we enable you customize the acceptable thresholds for each metric, then alert you when metrics are outside of the prescribed range. Leveraging this automation, team resources are freed to focus on analyzing and improving the problematic code– tasks that truly require human intelligence.

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." This forces developers to look at requirements from two different angles—implementation and validation—which results in a thorough understanding of each requirement.

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. In the case of TDD, test cases can be written and submitted into the Parasoft Concerto platform, which then waits for code to be generated to meet the demands of the test case.

When the test suite is submitted into the Parasoft Concerto platform:

- **For automated tests—**The test suite continues to fail until code or assertions are successfully modified.
- **For manual tests—**The test suite is flagged for "retest" until an individual verifies that the success criteria are satisfied.

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.

Learning More

To learn more about how Parasoft Concerto can help your team with Test-Driven Development, see http://www.parasoft.com/alm.

About Parasoft

For 21 years, Parasoft has investigated how and why software defects are introduced into applications. Our solutions leverage this research to dramatically improve SDLC productivity and application quality. Through an optimal combination of quality tools, configurable workflow, and automated infrastructure, Parasoft seamlessly integrates into your development environment to drive SDLC tasks to a predictable outcome. Whether you are delivering code, evolving and integrating business systems, or improving business processes—draw on our expertise and award-winning products to ensure that quality software can be delivered consistently and efficiently. For more information, visit http://www.parasoft.com.

Contacting Parasoft

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

Europe
France: Tel: +33 (1) 64 89 26 00
UK: Tel: + 44 (0)208 263 6005
Germany: Tel: +49 731 880309-0
Email: info-europe@parasoft.com

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

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