Extend Beyond JUnit's

By Matt Love

What's the first thing that comes to mind when you think about unit testing? If you're a Java developer, it's probably JUnit, since the tool is generally recognized as the de facto standard for Java unit testing.

However, JUnit doesn't necessarily define the testing methodology for which it may be leveraged. The tool can be used to associate a one-to-one mapping between every test class and tested class in a code base, or it may be used to manage a set of end-to-end tests for an entire Java EE application. But if a development group is asked, "How is this software tested?" and they respond simply that it's unit-tested or tested with JUnit, have they really answered the question?

The emergence of Java EE frameworks for everything from server-side business logic to data persistence forces the requirement for specialized tests that fit an application's frameworks. JUnit is becoming a commonality between specialized testing frameworks for the ever-expanding domain of Java EE applications.

Many of these frameworks can be well tested using proper setup and deployment of JUnit tests that have been supplemented with a framework-specific test harness. However, when additional test harnesses are involved, JUnit is no longer sufficient to identify the means and breadth of testing.

This article will teach you several unit-testing techniques that go beyond JUnit as they apply to Java EE applications.

Matt Love is a software development manager with Parasoft. He has been involved in the development of Jtest since 2001.

1. Identify the Goals

The first step to implementing a testing solution is to clearly define the goal and scope of each test. After these have been established, the next step is to investigate which implementations of testing for Java Enterprise frameworks are best suited to achieve that goal.

Regression testing. The adage "If it wasn't tested, it probably doesn't work" is true for any software development project. The most important question that needs to be answered by software testing is "Does this application work?" Any type of test can verify if software is functioning correctly according to the specification.

Typically, when new functionality is added, a manual test is written to ensure that the new code has the desired effect. That manual test will answer the immediate question about behavioral correctness, but the application may not work the next day, after the code base has changed.

Automated tests provide confidence that existing specifications are still satisfied as the code base evolves. Beneficial code optimization, reorganization or new features are often postponed or rejected because of low confidence in the software's functional correctness. Such changes are commonly resisted when sufficient tests aren't available to verify that the changes didn't break previously working functionality.

A good regression suite will not only catch new errors introduced into previously correct functionality, but will provide the confidence needed to make significant modifications faster. Automatic tests are no replacement for manual QA, but they do provide added confidence that certain features are known to be well tested. This allows QA to focus on

Basic Color Palette

Four Steps For Taking Unit Testing To the Next Level For Java EE

testing the higher-risk features.

Unexpected behavior. Sometimes testing is performed to vet unexpected behavior so that bugs can be identified and fixed before they reach the end users. This is a smart goal because leaving such problems for the end user to discover is typically much more expensive—both in terms of the impact on the organization's image and the resources required for a patch release.

The best results in testing for unexpected behavior are achieved by a tester other than the person who wrote the code, since such a tester is more likely to think "outside the box" of the original specification. An automated test-generation tool that has no knowledge of the specification is a prime candidate for performing this type of testing. These tools can help you test for unexpected behavior by designing and executing tests that check how the program handles unexpected stimulus and boundary conditions.

Tests with unexpected inputs or outcomes can be used as regression tests once they've been reviewed and incorporated into the specification. The end result is a large suite of regression tests that are used to identify when a unit's functional behavior changes or to verify that all units are functioning properly.

Test-driven development. Test-driven development (TDD) is another popular term that comes up when discussing unit testing. When taken to the extreme, TDD means writing tests before writing code. This is difficult to do when the tests need to make programmatic calls to tested code that hasn't yet been written.

Granted, this meets the TDD goal of tests that initially fail (because compilation errors are considered failures). However, compilation errors in tests tend to interfere with the rest of the test suite. This is especially true in Java EE systems, where the test suite is compiled to a JAR, EAR or WAR file and deployed in a container. None of the tests can be deployed if there are any compilation errors. This forces test code and tested code to be written at the same time, and thus fails to comply with pure theoretical TDD, which mandates that tests be written before the code.

A more practical approach is to apply TDD to functional tests for problems found in an application's current functionality. TDD procedures fit into the QA cycle very nicely:

- Manually reproduce a problem report
- Reduce the problem to identify the problematic unit(s)
- Write tests to automatically reproduce the problem in the problematic unit(s)
- Verify that the tests fail as a result of the problem
- Write the necessary code to fix the problem
- Verify that the same tests pass as a result of the fix

To truly adhere to TDD, these steps need to be followed for every problem report. The same methodology can be applied to new features in a practical manner, as long as there's a means for the tests to compile and run. If it's not practical to create tests before creating a new feature, the tests should be put in place immediately after the feature is created to serve as regression tests and verify the specification.

The second step attempts to define the term *unit testing* as the smallest unit that exhibits some functional problem. More comprehensive tests might be appealing because they can test all components at once. However, tests must isolate the target problem and have few other points of failure. Otherwise, maintenance for full end-to-end system tests will be overwhelming.

The best results are achieved by a tester other than the person who wrote the code.

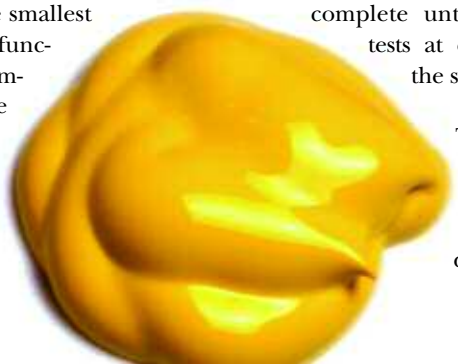
Imagine tests that compare a program screenshot to a saved control. Even the smallest change in presentation will require that all tests be reviewed and updated. Building a suite of tests that operate on the smallest functional units provides the best return for the lowest maintenance overhead. However, these functional units and the associated tests can become very large in Java EE applications when a problem exists in the integration between several components.

2. Define the Scope

Borrowing an analogy from the animated feature film *Shrek*, it's safe to say that unit tests are like onions and ogres: They all have layers. Unit tests may be performed at the method, class, component or integration level. Even some complete end-to-end system tests can be organized using JUnit in such a way that the unit tested is related to a unit of specification that exercises the whole system.

The majority of tests should be for as small a unit as possible to meet the associated goal. Testing small units often exposes problems that aren't obvious when testing larger components or systems. However, an application is likely to fail when there are no tests for the layers where the units of code interact. Integration testing will verify that each unit is not only functioning correctly on its own, but also that the units are connected correctly in the application.

Integration testing for Java EE applications involves testing interactions with third-party systems that are assumed to be correct. Third-party systems may not be easily changeable, so components under development must detect and work around any third-party flaws. A Java EE testing strategy isn't complete until it includes tests at every layer of the system.



Code-level. Testing at the class or method level can be done in most

Java EE applications by using mock objects and stubs. Mock objects use special implementations of popular interfaces for testing purposes. These mock objects can be custom classes written for specific tests, or they may be provided by a testing framework. Object-oriented programming allows for the code under test to execute on mock objects as it does on the live objects that are seen in production.

Stubs allow specific method calls to be replaced, usually by prepending alternate implementations of classes to the Java class path. With this approach, the tested code can be executed against different dependencies without needing to be recompiled. Code-level testing is easily automated, especially when testing with unexpected inputs. It provides great regression value by identifying specific pieces of code that change functionally. However, when the scenario spans several pieces of code, it's difficult to represent a use case or problem report in a code-level test.

Component-level. Tests for a component can usually be associated with part of the specification. A component is a functional unit of related classes. In Java EE applications, each component may integrate with one or more enterprise frameworks. A specialized framework for testing is needed to effectively test that a component integrates correctly with other enterprise frameworks—without having to set up the entire enterprise system. These testing frameworks usually process application configuration files and provide helpful functions to facilitate testing. The best approach to component-level tests depends on which enterprise frameworks are involved.

System-level. An enterprise testing strategy is not complete unless the entire system is started, initialized and tested. This is typically the role of manual QA testing, but many system-level tests can be automated. System-level tests exercise the application at the same access points that end users would, and verify the same results that end users would obtain.

System tests may also verify internal data at several steps through the process to expedite detection of problems. System testing is often slow, difficult to set up and prone to frequent test failure. Most tests should target more specific components or units of code instead of the whole system. The system level still needs to be addressed, although only a small portion of an enterprise application test suite should be implemented as system tests.

3. Select a Framework

Java EE systems employ many frameworks to speed integration with Web page, Web service and database technologies. Enterprise frameworks are used to simplify the raw interfaces provided by Sun's Java EE development kit.

In Java EE applications, each component may integrate with one or more enterprise frameworks.

A common enterprise solution is to move configuration information from Java API to XML files. Although XML configuration files simplify development, they complicate testing because traditional JUnit works only with Java classes. Most enterprise frameworks provide test utilities to be used in conjunction with JUnit so that the development efforts outside individual Java classes can still be tested.

Struts. Apache Struts is an open source framework for building servlet- and JSP-based Web applications. Struts works well with conventional applications as well as with SOAP and AJAX. Apache provides testing frameworks for Struts that mock the Web application server and integrate with the server for testing.

The Struts framework simplifies online forms and actions by using simple APIs with an XML configuration file. Web page actions and form data are directed to the appropriate Java code based on data in the Struts configuration file.

The mock Struts test framework also uses the same configuration file to emulate the Web application server in an ordinary Java Virtual Machine. As a result, testing Struts applications becomes as easy as specifying the configuration file and context directory once for all tests in the setUp() method. Running mock Struts tests is equally easy. Since the frameworks extend JUnit, tests can be run by any JUnit test runner. The framework supplements JUnit with utility methods to programmatically exercise Struts Web pages and assert results.

The same utility methods from the mock framework are available in Apache Cactus Struts Test framework. The difference is that Cactus will deploy tests and run them in a Web application container instead of mocking the container. Tests written using the mock framework can easily be extended to run in the container to test for integration issues.

Spring. The Spring framework also incorporates XML configuration files to facilitate an abstraction layer between plain old Java object (POJO) logic and the Web application container. This allows for many scenarios to be tested with traditional JUnit and mock data access objects (DAO) for the service layer. However, some functionality requires integration testing that JUnit cannot handle on its own.

A Spring test framework provides a way to test the Java code with respect to the configuration files—without requiring deployment in a container. This is achieved using the spring-mock.jar file that ships with Spring. You can also run unit tests for Spring applications in a container using

Cactus. Thus, unit testing is feasible for Spring code at the class level, the mock-container level and in a running application server container. This is ideal when creating regression tests for functionality or applying TDD to problem reports. Test cases for Spring can involve as much or as little of the application and surrounding system as needed.

Data access objects with Hibernate. Hibernate is object relational mapping (ORM) for persisting data access objects (DAO) in databases. It's used by the Spring framework and can manage database transactions and provide an abstraction layer for any SQL or JDBC code. Hibernate uses XML mapping files to relate database elements to Java DAO. Spring framework code that uses the DAO can be tested easily by providing mock objects that implement the DAO interface or override calls that would go to the database.

You can also test that the mapping files and database transactions are working properly, similar to how the Spring test framework verifies Spring configuration files. The org.springframework.orm.hibernate3 package in spring.jar provides classes for the Hibernate configuration, session and template properties to be configured for testing.

This is adequate to detect errors in the mapping XML files, but special care must be taken for the database. Test results may not be repeatable or deterministic when the tests change persisted data in a database. Fortunately, the test harness can be configured to use a volatile database in memory that won't be persisted between runs.

Hypersonic HSQLDB is a good example of an in-memory database. The database can be initialized with a snapshot of data and later examined to verify if the tests manipulated the data correctly. The database in memory provides the benefits of testing that the data written to it through the Hibernate framework can be retrieved using the same framework—without the consequences of permanently altered data or the risks of deleting important data.

System-level testing against the production database that is persisted in the file system is also possible, but it's usually difficult to set up such a database from a snapshot for every test. The risk that another client may access the data simultaneously during testing adds to the difficulties of testing with file system–persisted databases. Any system-level testing should use a dedicated test database that won't interfere with valuable live data.

Use system-level tests sparingly, but a few are essential for ensuring that all the components fit together.

Eclipse plug-in development. Even though Eclipse plug-ins aren't considered to be Java Enterprise applications, Eclipse IDE plug-in development is a good example of using a testing framework that runs units inside a larger application container. Eclipse provides a framework to run JUnit tests as additional plug-ins when launching a graphical workspace.

The plug-in tests can programmatically control the Eclipse IDE in a way that visually displays actions as they happen during testing. For example, a plug-in test can use the Eclipse API to import a new project, refactor source code and check for compilation errors. This is yet another example of a framework that extends JUnit to check that the code under development is integrating correctly with the system in which it's contained.

4. Cover the Entire Spectrum

Techniques for unit testing can be applied to any level of an application. Unit-testing strategies for Java EE applications aren't complete unless every layer is addressed by the tests. Tests for the top layer exercise smaller units at lower levels, but such tests are highly sensitive to changes and fail easily. These system-level tests should be used sparingly, but a few are essential for ensuring that all components fit together.

Component-level tests are often able to tell a story or test a scenario without depending on the entire system. This makes them the best regression tests for verifying code that corrects problem reports or implements feature specification. Code-level tests that focus on one class or method at a time are excellent for pinpointing regression changes, but they're usually difficult to understand because they lack the context that component-level tests provide.

Automated test–generation tools are best suited for creating a code-level test for every class or method because that amount of test creation is tedious when done manually. JUnit itself is not sufficient to test every layer of a Java EE application. Specialized testing frameworks must be used and matched to the Java EE frameworks used to build the application. Mock objects, configuration processors, synthetic databases and live containers all play a part in a complete Java EE testing solution. Having a test suite that covers the entire spectrum allows application development to proceed with confidence and reliability.

REFERENCES

- www.junit.org
- struts.apache.org
- www.springframework.org
- jakarta.apache.org/cactus
- www.hibernate.org
- hsqldb.sourceforge.net
- www.eclipse.org

PARASOFT

Jtest®

DATA SHEET

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

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

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

Capabilities

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

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

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

Key Features

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

Infrastructure Support

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

System Requirements

Operating System

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

Hardware

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

www.parasoft.com

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