Beyond JUnit: Unit Testing Java Enterprise Applications

Many Java developers associate unit testing with JUnit, which has become recognized as the industry standard tool for Java unit testing. JUnit, however, does not necessarily define the testing methodology for which it may be leveraged. JUnit may 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 complete end-to-end tests for an entire Java EE application.

The terms “Unit testing” and “JUnit” have become overused, empty buzzwords. Development groups are often asked how their software tested. If the response is simply that it is unit tested or tested with JUnit, then they really haven’t answered the question.

The emergence of Enterprise Java 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, the name “JUnit” is no longer sufficient to identify the means and breadth of testing. This article will teach you how to leverage the benefits of several unit testing techniques that go beyond JUnit as they apply to Java EE applications.

Identify the Goals

The first step to implementing a testing solution is to clearly define the goal and scope of each test. After this has 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 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.

Automatically-executed 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 are not available to verify that the changes did not break previously-correct functionality. A good regression suite will not only catch new errors introduced into previously-correct functionality, but it will also provide the confidence needed to make significant modifications faster.

Automatic tests are not a replacement for manual QA, but they do provide added

confidence that certain features are known to be well-tested— which allows QA to focus on testing the higher-risk features. If it was not tested, it probably does not work. That adage is true for any software development project.

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 will be 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, which 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 have been reviewed and incorporated into the specification.

As in the previous section, the end result is a large suite of regression tests that is 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 buzzword that comes up when discussing unit testing. When taken to the extreme, it means writing tests before writing code. This is very difficult to do when the tests need to make programmatic calls to tested code that has not 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 are 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 fits into the QA cycle very nicely:

1. Manually reproduce a problem report
2. Reduce the problem to identify the problematic unit(s)
3. Write tests to automatically reproduce the problem in the problematic unit(s)
4. Verify that the tests fail as a result of the problem
5. Write the necessary code to fix the problem
6. Verify that the same tests pass as a result of the fix

In order 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 is a means for the tests to compile and run. If it is not practical to create tests before creating a new feature, then the tests should be put in place immediately after the feature is created to serve as regression tests and verify the specification.

Step 2 attempts to define the term “unit testing” as the smallest unit that exhibits some functional problem. More comprehensive tests might be appealing because they could test all components at once. However, it is important that the tests isolate the target problem and have few other points of failure. Otherwise, maintenance for full end-to-end system tests will be overwhelming. Imagine tests that compare a program screen shot 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.

Define the Scope

Borrowing an analogy from DreamWorks’ Shrek (2001), it is 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 which 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 are not 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 will involve 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 is not 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 the same way while operating on special test objects as it would when operating on the live objects that would be 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 is very 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 very slow, difficult to set up, and prone to frequent test failure. Most tests should target more specific components or units of code instead of testing 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.

Select a Framework

Java EE systems employ many frameworks to speed integration with web page, web service, and database technologies. The Java EE development kit has a history of being cumbersome, enigmatic, and laborious. Enterprise frameworks are used to simplify the raw interfaces provided by Sun. 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 only works with Java classes. Most enterprise frameworks will 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-based and JSP-based web applications. Struts works well with conventional applications as well as with new technologies such as SOAP and AJAX.

Apache provides testing frameworks for Struts that mock the web application server and also integrate with the server for testing. The Struts framework simplifies online forms and actions by using simple application programming interfaces 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 configuration XML 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. It is also possible to run unit tests for Spring applications in a container using Cactus. Thus, unit testing is feasible for Spring code at the class level, the mocked 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 is used by the Spring framework, and it 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. There is also a way to 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 will not 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 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 is 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 will not interfere with valuable live data.

Eclipse Plug-in Development

Even though Eclipse plug-ins are not 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, re-factor 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 is contained.

Conclusion

Techniques for unit testing can actually be applied to any level of an application. In fact, unit testing strategies for Java EE applications are not complete unless every layer is addressed by the tests.

Tests for the top layer end up exercising smaller units at lower levels, but such tests are highly sensitive to changes and will 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 are usually difficult to understand because they lack the context that component levels 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 very tedious when done manually.

Ordinary JUnit 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 have 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.

Automating Java Unit Testing with Parasoft

Parasoft Jtest provides a complete framework to help developers efficiently create, execute, and manage unit tests for Java EE.

We help you start verifying reliability and functionality before the complete system is ready— enabling you to identify problems when they are least difficult, costly, and time-consuming to fix. This, in turn, reduces the pain of debugging.

Parasoft can also establish 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.

Test Creation

To facilitate standardized testing and quality processes, our patented technologies automatically generate extensible, reusable test cases. These test cases not only expose potential reliability problems, but also capture the code's current behavior to establish a baseline for regression testing. Automated test creation supports JUnit, Cactus, Spring, Struts, Eclipse plugins, and more.

In addition, Parasoft's Tracer technology provides a fast and easy way to create the realistic test cases required for functional testing. Simply use the application's UI or Parasoft's SOA/Web solution to execute the use cases you want to verify. Tracer automatically designs unit test cases with real data that represents the paths taken through the application. No coding or scripting is required. If the functionality associated with your "traced" use cases later breaks, you will be alerted by the failure of the related test cases.

Collectively, these test cases establish a safety net that alerts the team when modifications impact application behavior. Since all tests are written at the unit level, the test suite can be run independent of the complete system to isolate code behavior changes, reduce setup complexities, and make it practical to execute on a daily basis.

Test Optimization

Parasoft provides a variety of technologies to help you extend the automatically-generated test cases to verify specific functionality requirements and/or increase coverage. We provide a number of ways to help you increase the value of the existing test suite with minimal effort; for example:

- Manually extending the generated unit test classes.
- Using an object repository and flexible stubs framework to make tests more realistic.
- Parameterizing test cases for data-driven testing.
- Reusing a test with different sets of data (automatically-generated or from a data source).

Extended Execution Environment

Parasoft's extended execution environment centralizes execution and enhances reporting for all of your automatically-generated and manually-written 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:

- Runtime error detection
- Coverage analysis
- Debugger integration
- Exception validation

Contacting Parasoft

USA

Toll Free: (888) 305-0041
Tel: (626) 305-0041
Email: info@parasoft.com

URL: www.parasoft.com

Worldwide

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

About Parasoft

For over 25 years, Parasoft has researched and developed software solutions that help organizations deliver defect-free software efficiently. By integrating Development Testing, API/cloud/SOA/composite app testing, and service virtualization, we reduce the time, effort, and cost of delivering secure, reliable, and compliant software. Parasoft's enterprise and embedded development solutions are the industry's most comprehensive—including static analysis, unit testing with requirements traceability, functional & load testing, dev/test environment management, and more. The majority of Fortune 500 companies rely on Parasoft in order to produce top-quality software consistently and efficiently.