Utilizing *Fast Testing* to Transform Java Development into an Agile, Quick Release, Low Risk Process

Introduction

System tests, often called slow tests, play a crucial role in nearly every Java development effort by making sure all the different parts of the system work together as expected to meet the requirements of the end user. But the complexity and long run time of system tests mean that they can usually only be run on a limited number of cases and infrequently during the development cycle. On the other hand, unit tests, often called fast tests, are short and specific tests that exercise specific building blocks of the code on various inputs to be sure they return the right results. The speed and simplicity of fast tests mean they can be run often, typically many times a day, by developers in order to ensure that the code actually does what it is supposed to do. Fast tests increase the speed and reduce the risk of the development process by preventing errors from being built into the code that might slip through system testing and, even if they are caught in system testing, might be more time consuming to identify. Despite these benefits, many development teams perform little or no fast testing because of time required to develop the tests, which typically require three times as many lines of code as the application itself to provide 80% coverage. A new generation of JUnit generators providing a thorough level of unit-level regression tests (i.e., fast tests) with 80% or better code coverage at the push of a button, makes it possible to transform Java development into an agile, quick release, low risk process by identifying and fixing regressions when it's easiest to do so.

The Roles of Slow and Fast Testing

Slow tests and fast tests actually differ more in degree rather than in their basic nature, with the fundamental difference being that slow tests operate at a higher level of abstraction. Slow tests generally are designed to exercise the system as it is seen by the different parties that interact with the software such as users and other systems. Slow tests verify that the right functionality is being delivered to the user and tests are often created based on specific use cases. For this reason, slow tests generally require external resources and/or a running application to be present in order to run. They may be executed manually through a user interface, with an automated system, or even with a unit testing framework; but they run slowly enough that developers probably don't run these tests as they are writing or changing their code. Most teams primarily run slow tests on their software and they run them (manually or using automation) at the end of each build.

The basic premise behind slow testing is that if the entire system works, then everything in it must work as well. But in actuality slow tests usually only test a very limited number of inputs and only rarely address edge cases. When the code is released, it likely will be exposed to a much wider range of inputs and use cases and there is a good chance it will break. Another limitation of slow tests is that when the test fails it is often very difficult to determine which aspect of the code is responsible for the failure. The failure is often caused by a number of different interacting components and identifying which component or components are responsible and drilling into the components to find the problem can consume enormous amounts of time and resources. These difficulties are compounded by the fact that, because of the amount of time they require, slow tests are generally run relatively infrequently and are typically run by a quality assurance team that may have only limited familiarity with the code they are testing. By the time the code is tested, system tests may uncover a rabbit's warren of interacting errors that can be difficult to unsnarl.

Fast testing provides an additional level of assurance that the overall system is working the way it is supposed to. Fast tests do not depend on an external resource or application environment to run. They rely only on the source code and classpath of the project. For Java, these tests most popularly employ the JUnit library and usually a mocking library such as EasyMock. Fast tests test individual units in isolation from the rest of the system, using stubs to return controlled values for everything else. The speed of fast tests, which typically take only seconds to run, means that they are usually performed many times a day by the same people who are writing the code. The frequency of testing, its isolated nature, and the fact that testing is performed by the person who knows the code best, means that when a fast

test identifies a bug it's usually pretty obvious what caused it and how to fix it. Fast testing also substantially reduces development time by catching problems immediately after each code change when they can be quickly and inexpensively corrected. Fast testing is potentially much more thorough than slow testing because testing at the unit level makes it possible to fully test the unit, including functionality that is not used in current use cases but may be used in the future as well as edge cases. Fast testing thus provides extra assurance that the system works and that it can be changed in the future without causing adverse side effects.

Overcoming the Roadblock to Implementing Fast Testing

There is a major roadblock to instituting fast tests as an integral part of the development process. An enormous amount of code is required to achieve thorough coverage on even a single class. Writing unit tests is dreary work for which few engineers volunteer. There are some tools out there to help, such as by churning out a non-working skeleton but it still takes considerable time to complete the writing of all of those tests by hand. On top of that, hand-authored tests only test the scenarios that developers can conceive of. They tend to miss unexpected exceptions, unexpected values and many border cases that have the potential to create future regression bugs.

AgitarOne JUnit Generator overcomes this roadblock by automatically creating high-coverage tests at the push of a button. The JUnit Generator analyzes the entire Java project and then generates tests that capture and preserve the code's behavior. AgitarOne JUnit Generator routinely achieves 80% or better code coverage and, depending on server configuration, can generate 250,000 lines of JUnit per hour. These tests leverage mocking technology that resolves dependencies on components such as databases and generates tests for code that would otherwise be untestable.

In cases where the engine does not have enough information to get full coverage — such as when some domain-specific information or set-up is needed — developers can write test helper methods. The typical test helper method is just a few lines of Java code. Once it's amplified by the test generation engine, it will produce thousands of lines of high-quality JUnit tests. This approach can generate a suite of unit tests that characterize the current behavior of the code in a matter of hours.

Implementing Fast Testing for Legacy Code

AgitarOne JUnit Generator can generate JUnit tests for legacy code without spending developer time. It will create fully working, passing fast tests that developers can run as they make code changes to ensure they are not introducing regressions. The process will look something like this start to finish:

1. Select an individual or small team to work on installation, build integration and test generation
2. Install AgitarOne Server
3. Install Eclipse Client Plugin on developer platforms
4. Select project in Eclipse and click the "Generate JUnit Tests" button

At that point the AgitarOne Server will process each class and return fully working JUnit test classes with over 80% coverage. These are characterization tests as defined by Michael Feathers which should be used to address legacy code problems.

5. Install AgitarOne software on the build machine
6. Export an ant build script or Maven POM
7. Add the ant script or POM to the current build in the Continuous Integration (CI) environment. Jenkins (jenkins-ci.org/) or Hudson (http://hudson-ci.org) are free, easy to set up and are the most popular modern CI environments.

8. Provide continuous visibility to the AgitarOne Management Dashboard. This is the metrics report that is automatically generated with each build. It shows the coverage, test points, test failures and risk. It is an HTML report that is included as a build artifact in the CI environment and can be emailed as a link to whomever it concerns. Set up a monitor that the project team can see at all times and one that auto-refreshes the dashboard report or uses some similar mechanism so everyone in the team can see if the project is successful or not at all times.
9. Set expectations of developers using Management Dashboard Targets. These targets track coverage, test points and risk and test failures. If any of the targets are not met, it means either:
    a. A code change was checked in without tests being run by the developer first
    b. New code was checked in without tests, or
    c. The current tests are insufficient due to increased code complexity

Implementing Fast Testing for New Code Development

For new code development, AgitarOne Agitator can serve as an auxiliary unit testing tool that is easy to use and catches things that hand-authored tests miss. To really unit test code, every line, every branch, and every outcome must be tested. That's a daunting problem. It's not practical to create such exhaustive tests manually. AgitarOne Agitator provides a unique interactive understanding of code behavior as a developer writes or modifies Java classes or methods. Agitator helps developers validate the expected behavior of their code and discover unexpected behaviors. It automatically creates dynamic test cases, synthesizes sets of input data, and analyzes the results. When objects depend on other objects, Agitator automatically constructs those objects, enabling full analysis of the possible behaviors of the code, even if it depends on third-party libraries.
Implementing the Agitator will look something like this:

1. Select an individual to install AgitarOne and complete build integration
2. Install AgitarOne Server
3. Install Eclipse Client Plugin on developer platforms
4. Install AgitarOne software on the build machine
5. Export an ant build script or Maven POM
6. Add the ant script or POM to the current build in the CI environment
7. Provide continuous visibility to the AgitarOne Management Dashboard.
8. Set expectations of developers using Management Dashboard Targets
9. Train developers how to use the Agitator

The above procedure will provide an environment that not only provides a valuable developer testing tool but also provides visibility about the current testing status. From there developers can start using Agitator to show the complete possible behavior of the code they are writing at a glance and suggest observations that can be easily converted to unit tests with a single mouse click. Continuous visibility will help keep testing standards up as development progresses without being bogged down by slow tests.

Creating the Fast Testing Team

The fast testing team should consist of software developers who have a good working knowledge of the code base and are enthusiastic about developer testing. This team must be given the authority to change processes and enforce goals on the existing development process. Their job should be to create the initial unit tests to cover any legacy code, to educate the development team about how to use the unit tests after they are created, and to make sure that these tests are added to the CI builds. In addition, they should take steps to create continuous visibility of the unit test metrics. The fast testing team should train the development team to create its own unit tests and prevent regressing into a legacy code situation again.

The makeup of the fast testing team would ideally be:

- A testing tools expert. This person is the point of contact for any AgitarOne related support questions and is responsible for AgitarOne Server administration.
- A processes expert who has the authority to change processes, builds, and testing targets and influence releases based on the testing status.
- Unit testers with working knowledge of the code under test who will create the tests using AgitarOne JUnit Generator. Areas of the project can be assigned to these developers through developer assignments in the management dashboard.

The makeup of the developer testing team would ideally be:

- AgitarOne trainers drawn from the unit testing team who are savvy enough by now to teach everyone how to use AgitarOne JUnit Generator and Agitator. Training the development team to create their own tests from that point will prevent the legacy code problem from recurring.
- Everyone else should be running existing fast tests after each code change and creating new tests to characterize each piece of new code and each code change.

Enjoying the Benefits of Fast Testing

If everything above has been implemented successfully, the result will be a smooth, agile release cycle. Each code change will be immediately tested by the developer who will run the fast tests, review test failures and regenerate the fast tests after all test failures have been validated as not regressions. When the code is checked in, a CI build will be triggered. The Management Dashboard report will be generated and displayed visibly to the team and QA can immediately start running any manual, functional and acceptance tests that are part of their process.

Any new code created will be tested using Test Driven Development (TDD) if that is part of your process. Agitation will complement any manually written tests to ensure the code behavior matches the intended behavior and will increase test coverage. If the new code is checked in without meeting the required test percentage and test coverage targets, the build will fail. This will enforce fast testing and naturally keep future builds green with passing tests.

The ability to generate a thorough suite of unit-level regression tests with 80% or better code coverage at the push of a button will make it possible to run tests with every build and detect regressions soon after they are introduced, resulting in a substantial reduction in maintenance costs. Fast testing will also ensure that your code is built for change and provide feedback that will bring significant improvements in team productivity and the quality of code under development.

For more information about AgitarOne, please visit us at www.agitar.com.