



White Paper

Applying a Java Testing Framework

Most software developers have heard about the benefits of Java unit testing frameworks, and may have implemented unit testing at a previous job; but tight schedules leave little time for such initiatives; and isn't it Quality Assurance's job to identify bugs?

In fact, identifying and correcting bugs as early as possible in the development cycle reduces overall bugs, freeing developers for more creative work. It also improves team cohesion and minimizes the frustration and anxiety common with an endless cycle of bug fixing.

The Cost of Bad Code

From a global business perspective, software bugs are epidemic:

- According to the Washington D.C. National Institute of Standards and Testing (NIST), software errors cost the U.S. economy \$60 billion per year.¹
- The Standish Group estimates that 30 percent of all software projects are cancelled, almost 50 percent are over budget, and 60 percent are considered failures due to software coding errors, bugs and failure to meet deadlines.²
- The relative cost of maintaining software and managing its evolution now represents more than 90 percent of its total cost.³

What does this mean to Java developers? On a day-to-day level, this data typically manifests itself in quality of life issues such as:

- Frustration caused by poorly documented code
- Bugs pouring in from unhappy customers
- Marketing setting unrealistic release schedules

- Management demanding better code faster

Bad code often means poor or no documentation, making it difficult to fix and enhance, forcing other developers to spend a great deal of time trying to decipher it. A change to one area of code can easily damage another, resulting in cascading bugs, even worse than before.

It's a well-known fact that the cost of fixing a bug significantly increases the further along in the release cycle it is detected. Doesn't it make sense, then, to implement development methodologies that prevent bugs from being created in the first place? But what are these methodologies, how can they help, and how can they be most successfully applied?

Software Development Trends

Gone are the days when a team of lone developers only gathered at code or design reviews, or at product review meetings. Developers are working in teams now; and a shortage of skilled developers means that any team will have a mix of different skill and experience levels.

Within any development team there are a variety of skill sets, with the least skilled often taking a great deal of everyone else's valuable time.

What's worse, more junior developers can potentially delay or halt the entire project by contributing buggy or poorly tested code.

Many Java teams are turning to agile software development techniques, an iterative and incremental approach to software development that is performed in a highly collaborative manner. One agile software development practice is to test early and often, with the focus on test-driven development, where teams work in short cycles writing tests for a piece of code first, then writing the code to be tested. The key is to write tests before the code, with the objective of getting as close as possible to 100 percent test coverage.⁴

Unit Testing: What Good Is It?

But why is testing so important during actual code development? Part of the reason goes back to team effort. If all developers practiced unit testing, not only would it improve the entire code set, but it would also simplify maintenance and significantly reduce debug time for the entire team.

However, in real life, almost the only significant metric that is commonly applied to the software development process is whether or not a team is hitting its deadlines. Under pressure to deliver, most developers will focus on getting their code written, and let the Quality Assurance (QA) team uncover any bugs. And there will be bugs. Yet, the purpose of QA is not really intended to be finding bugs. They are responsible for:

- Functional tests – assures that business requirements have been met
- Integration tests – confirms that components work well together

- Acceptance tests – confirms that an application does what the customer expects it to do

Developers should be responsible for testing their code *before* it goes to QA; which is where unit testing comes in—discovering problems as they are created. A unit test is highly localized. A unit test class only tests within a single package. It does not test the interaction between one package and another. That is the job of functional testing.

Most developers have heard about unit testing and agree that it should be done; but claim that they can never find the time, and that management doesn't consider it a priority. Some developers, faced with tight deadlines, can't comprehend the possibility of starting to introduce unit tests to an existing body of code. Along with documentation, it is the least important item on their list. Testing code is boring, especially when it's someone else's code.

So why should developers change the way they work? Why should they take the time to unit test? For years, unit testing was relegated to the "I know I should be doing it" category; but now it has become an important part of a top-notch Java developer's toolkit. Unit testing gives developers the confidence that their code works and that they can change it at a moments notice, without fearing that it will break.

With the trend toward agile software development, being the best programmer on the team is not enough. Agile software development is not just a fad. Its precepts of teamwork and unit testing have been shown to significantly improve code quality, affording developers more time to write code; and decreasing the time they have to spend

struggling to fix or enhance bad code. The bottom line benefit is cost reduction.

Applying a Process Change Template

But how do organizations apply a testing framework to their code base, especially if that code base is extensive? What best practices are available to help them?

The following steps, that provide a template for a quality-driven development process, can be applied to unit testing to provide a manageable path to implementation:

1. Goals and standards: Identify specific unit testing goals and/or standards
2. Tools: Facilitate standardizations and data collection with tools
3. Automation/repeatability: Eliminate the human factor
4. Measurement: Collect key data from systems/tools
5. Analytics: Analyze and interpret data to form a feedback loop
6. Trend Analysis: Tracking trends over time and looking to the future

Organizations may find themselves at any one of these stages without having passed through the previous stage. In many cases, they might have to take a step back in order to progress further.

First, organizations need to identify where they are. Are they using a unit-testing framework? If so, what percentage of code is covered by unit tests, and what is their goal?

To facilitate trend tracking, the highest level of this template approach, each step must be built upon the last. It is not necessary, however, to halt all development in order to implement this template. Organizations with a large installed code base, for example, may want to ensure that all new work is unit tested, so that over time, code coverage steadily increases.

Standards – Level 1

The fundamental rules of unit testing are:

- Test every line of code
- When possible, test code in isolation
- Software must be written with tests in mind

Consequently, the most important standard or goal for unit testing is the code coverage percentage—the percentage of code covered by unit tests relative to the code base's total lines of code. Is it 50 percent, 80 percent or 100 percent? Each development organization needs to establish their goal for this metric.

Another key metric related to unit testing is the pass rate—the unit test pass/fail ratio. Is it important that all unit tests pass? What if 100 percent of unit tests pass, yet these tests only cover 20 percent of the code base? Moving toward 100 percent for both metrics would be a better goal.

Tools – Level 2

Once an organization has defined their code coverage goals, they can implement a testing framework; or they can identify, if they already have such tools in place, their status relative to their goals.

The primary reason for using a testing framework is to make developers' lives easier, by identifying bugs early in the development cycle, when they cost less to fix. These tools do not necessarily have to be standardized across development teams, because some developers may have experience in and prefer using certain products.

Unit Testing Alternatives

There are a few alternatives to unit testing such as inserting statements into the code and using debuggers. But neither of these methods is automated, nor do they test all lines of code.

Inserting debug statements into code is a low-tech debugging method. It usually requires that you scan output manually every time you run the program to ensure that the code is doing what you expect.

Debuggers are used to step through code and inspect that the variables along the way contain the expected values. This is also a manual process that requires tedious visual inspections. Every time you change your program, you must repeat this manual process.

JUnit

The most popular unit-testing framework is JUnit, written by Erich Gamma and Kent Beck. JUnit is Open Source Software, released under the Common Public License Version 1.0 and hosted on SourceForge. JUnit lets you write and run repeatable tests.

More than any other testing framework, JUnit has helped developers understand the benefits gained from consistent unit testing.

JUnit features include:

- Assertions for testing expected results
- Test fixtures for sharing common test data
- Test suites for easily organizing and running tests
- Graphical and textual test runners

JUnit is designed to work with many small tests. It executes each test within a separate instance of the test class, reporting success or failure on each test. JUnit is not intended to report on multiple failures per test. If it does, you need to reevaluate your test. It may be that you have written a functional/ acceptance/customer test, or that your test is too inclusive.

The intention of a testing framework is to write tests before the code. This is called test-first programming, whereby new code is only written when an automated test fails. Good tests will show how to best design the system for its intended use.

For Best Results

For best results, JUnit should test all lines of code that could potentially contain bugs. If defects aren't being reported and the design responds well to change, then your unit testing is probably sufficient. If your team is still spending too much time fixing bugs, and the design is difficult to grow, they need to write more tests.

If your code is difficult to test, this presents an opportunity to improve your design.

If you improve your design so that it is easier to test, a better design will usually result.

Run all tests as often as possible, ideally every time you change your code. All unit tests should run at 100 percent pass rates. Frequent testing will give you confidence that your changes didn't break anything and has the additional benefit of lowering your overall stress level.

The output from these tools is the beginning of increased visibility into the development process, providing metrics that measure improved quality practices.

Automation/repeatability – Level 3

A unit test, by definition, is fully automated, non-interactive and binary—it either passes or fails. In the case of Java, a unit is usually a single class.

Automated unit testing lets you run tests repeatedly (regression testing), and collect tests into suites as you develop your code, promoting the evolution of your design. The key to attaining this step is that your team must have successfully implemented the first two levels.

Measurement – Level 4

While the previous levels provide a good foundation for quality practices, with resulting productivity improvements, quality can only be realized when output from these processes (or a combination of them) can be measured. This is the level, therefore, where organizations can begin to concern themselves with quality *results*.

To introduce new development practices, metrics must be used to identify where resource should best

be deployed. For example, with unit testing, pass rate is a key measurement. Is it enough that all 5,000 unit tests have passed? If 100 percent of those tests pass, this may be a sufficient level of quality. However, if the 5,000 unit tests only cover 20 percent of the code base, the code quality level is less certain.

Only by measuring and combining results, can organizations obtain a clear picture of unit testing coverage, and ultimately code quality.

Analytics – Level 5

After launching a unit testing initiative, organizations can start collecting metrics from all developers, and then combine the results by team. Analyzing data that has been measured allows managers to identify behavior issues or the lack of sufficient process, when it comes to unit testing. This will allow them to track individual behaviors, so they can focus on training and resource allocation. Pulling this data together into a single user interface will provide the team with instant visibility into developer behaviors.

Trend Analysis – Level 6

Tracking unit testing trends over time allows organizations to promote best practices through training, mentoring or other assistance, as may be required.

If an organization is combining data from different sources, analyzing trends, and then using the results to adjust and streamline unit testing, they are to be congratulated. At this stage, they have optimized the efficiency and quality of projects and deployed applications, with a resulting decrease in customer complaints.

Java testing frameworks such as JUnit, however, work at the individual developer level and provide metrics

World Headquarters

Energy Software
900 Cummings Center
Suite 326T
Beverly, MA 01915
United States
Sales: 866-598-9876
Phone: 978-232-0145
Fax: 978-232-0146
ContactUs@enerjy.com

in snapshot format. JUnit does not consolidate data from each individual client copy; or show who is responsible for untested code across an entire code set.

What is needed is a tool that not only consolidates code coverage information from all developers, but also tracks metrics over time. For example, a measured activity of code coverage may show that 70 percent of the code base is covered by unit tests. At first glance, this may seem like a good figure; but if last week's figure was 80 percent or the code base has halved in size from the previous week, how does this figure compare? These measurements must be taken in context, by individuals and by the entire team.

Refining the quality process is an ongoing task, especially with external factors enforcing the change process (such as Sarbanes-Oxley). However, if an organization is at this point in the process, these changes will be easier to implement, with less financial impact, because a unit testing framework is already in place for all developers, allowing organizations to take positive action.

Energy CQ2

Energy CQ2 provides visibility into the development process by consolidating data from key development processes and tool.

- Unit testing results
- Code coverage percentages
- Best practices and compliance to coding standards
- Overall activity
- Bug detection

Using these metrics, development managers can optimize the performance of their teams, thus minimizing time wasted on avoidable rework, tracking down bugs, or in lengthy or ineffective code reviews. Development teams can quantify and improve application quality at the beginning of the development process where it is easier and more cost-effective to address problems.

Energy CQ2 provides a turnkey solution including: Discovery, Recommendation, Installation, Implementation and Training.

Who Is Energy Software?

Energy develops Java™ software integrity solutions for development managers to continuously improve the quality of their application development projects. Energy CQ2 technology provides visibility into a quality-driven development process to aid development teams in delivering projects to specification with reduced coding errors by the required target date.

Development managers can now optimize the performance of their development team, thus minimizing time wasted on avoidable rework, on tracking down bugs, and in lengthy or ineffective code reviews.

Energy Software is a division of Teamstudio, Inc. With headquarters in Beverly, Massachusetts, Teamstudio also has offices in the UK, France, and Japan. To learn more about software integrity solutions designed to improve application quality and enhance developer productivity, please visit Energy's Web site at: <http://www.enerjy.com>.

(References)

¹ National Institute of Standards and Testing (NIST), 2002. (<http://www.nist.gov>)

² Standish Group, Chaos Report 2003.

³ "Leveraging Legacy System Dollars for E-business," Erlikh, L. (2000), (IEEE) IT Pro, May/June 2000, 17-23

⁴ "Google: Ten Golden Rules," Eric Schmidt and Hal Varian, Newsweek, December 2, 2005

⁵ http://junit.sourceforge.net/doc/faq/faq.htm#best_1