Eclipse Summit Europe 2006 Position Paper
Test-Driven Development for the Embedded Systems Space
Doug Gaff
4 October 2006

Test Driven Development (TDD), a concept from Extreme Programming, involves engineers writing automated tests prior to writing the code that implements the desired features. By building the test cases first, the developer is forced to consider the exact tasks the code must perform, the possible failure and edge cases, and the design patterns that will generate a concise solution. The development time is typically doubled in this process, but time is saved on the back end of the development cycle since the TDD-designed code has less bugs to be fixed during the testing phase.

TDD has the following cycle as described in *Test-Driven Development by Example* by K. Beck (2003):

1. **Add a test for a feature.** This occurs before the feature is developed.
2. **Run all tests and see the new one fail.** This is just a retest of the existing test harness. It ensures that the rest of the code is functional before writing the new feature.
3. **Write some code.** Write initial code that will pass this test. This code isn't the finished product yet.
4. **Run the automated tests and ensure they succeed.** This verifies the initial code.
5. **Refactor.** The purpose of this step is to clean up the code written in step 3.

Clearly since the developer is writing both the test cases and the code, he or she is considering the API's and the design from the very beginning of the process.

So why should this model be used for embedded development? Other than the standard answer that software processes for enterprise development are almost always applicable to embedded development, there are some important characteristics of embedded development that highlight the need for increased code quality and reliability. Embedded devices are not personal computers. While it's true that some consumer-oriented embedded devices can be restarted if a bug causes the device software to crash, that's not typically a desired workflow for the customer. Furthermore, many embedded devices are mission or safety critical in nature, many are parts of control systems, many are parts of high-availability infrastructures, many are not easily updated for bug fixes, etc. This leads to the generalization that software quality and reliability requirements in the device software space are higher than in many enterprise applications. TDD provides a mechanism for building quality checks into the cycle from the beginning.

How does this relate to Eclipse? There are three important facets of Eclipse that make it well-suited to supporting Test Driven Development:

- *Eclipse as an IDE* provides an excellent infrastructure for creating and running unit tests on code under development.
- *Eclipse as a developer community* has an obsessive focus on good API development as the first step in designing a framework. Good API's and effective test harnesses go hand-in-hand.

- *Eclipse as an ecosystem* provides a framework for software tool companies to deliver pluggable test harness infrastructures to address a wide range of domain-specific testing.

This last facet is essential in a pragmatic way: in order to get developers to adopt a TDD methodology, the tools and harnesses must be present and easily accessible to the engineer. Eclipse as an ecosystem built on common frameworks helps solve this by allowing vendors to deliver specific test harness tooling that targets languages, RTOS's, middleware, etc. These harnesses can be tied into the standard development workflow in Eclipse, thereby making the job of the engineer easier and increasing the likelihood of TDD adoption at the engineer level.

Finally, while Eclipse is well-suited for JUnit testing, embedded development goes well beyond Java. In addition to the plug-in ecosystem, TDD should look to open source projects like TPTP to provide open source test harnesses for embedded.