



MicroDoc Computersysteme GmbH

Author: George Mesesan

Date: Oct. 01, 2007

Position statement

Test-Driven Development with J2ME and OSGi

Topic outline

The project, with which we introduced TDD in our company, is a small project, with a 3 person core team, and 3 or 4 other team members that at one point or another helped us in specific areas. The target platform for the project was an ARM Processor, running a custom Linux Debian, on top of which we had J2ME and OSGi running. (OSGi, written O,S,G, small i, means Open Services Gateway initiative, but this is now an obsolete name. The core part of the specification is a framework that defines an application life cycle model and a service registry).

This given platform posed some problems of its own, especially the constraints introduced by OSGi.

We were lucky enough to have convinced our customer right from the beginning that tests are important, and that we can only deliver the required quality by investing time into test writing. The tests were thus included in the project plan and in the project budget.

At the start of the project, we identified two main directions, in which we wanted to introduce TDD:

- acceptance tests - for testing our requirement presumptions (building the right software)
- unit tests - for testing the implemented functionality (building the software right)

Acceptance Tests

The acceptance tests must be written by the client, otherwise their very existence is not justified. These tests represent the TDA: Test-Driven Analysis, seen by us as part of the Test-Driven Development process. In our project, we used Fitnesse, a software testing tool with a wiki interface that is designed specifically for acceptance testing. Non-programmers can generate test cases by entering specially formatted input that is interpreted and converted into runnable test cases.

We've made some modifications to Fitnesse, mostly required by the constraints of our target platform: J2ME and OSGi. These modifications allowed us to deploy and run the Fitnesse tests directly on the target platform.

Although encouraged at the beginning of the project, nothing came out of TDA, the only Fitnesse tests that were written, we've written ourselves. The project analysis and design phases were realized using the classical development process: design documents, change requests documents, etc. We ended up integrating the JUnit module tests into Fitnesse, promoting them in this way to acceptance tests. Through this, we gained the Fitnesse graphical user interface for our JUnit tests running under OSGi, as we had none up to this point.

Unit Tests

Unit tests were written naturally with JUnit, as Eclipse has an excellent integration with JUnit. In addition we use the EasyMock and JMock libraries for generating mocks and stubs. They are a great help in testing module interfaces and protocols.

As we developed using the OSGi Framework, we were encouraged, even constrained, by the framework to design the application into clearly-separated and standalone modules (in OSGi they're called bundles). Our unit tests naturally followed this separation, so that each module was created as an Eclipse project that also contained the corresponding test cases.

We strove to write the unit tests in such a manner that we test our classes through their interfaces, to ensure that the implementation fulfills the interface contract. We tried to keep clear of specific implementations, so that we don't have to change our test cases when changing the implementation. This approach proved itself wise, when midway through the project, we had to do some major refactorings. The number of changes required to the test cases was kept rather low, in comparison to the code changes.

The OSGi Framework also brought along some problems for testing. As OSGi is a framework that takes over the lifecycle of bundles, the registering/unregistering of services and event dispatching responsibilities, it soon became clear that the test environment also needed to run under OSGi in most of the cases. A mock/stub implementation of the basic OSGi functionality helped, by making the modules (bundles) testable in isolation, but the majority of test cases had to run under OSGi. Luckily for us, as the Eclipse itself runs as an OSGi application, the JUnit bundle from Eclipse could be easily integrated, and we were thus able to run our JUnit tests under OSGi too.

Continuous Integration

Continuous Integration is an important part of TDD. For this we used CruiseControl running on a Knoppix Linux.

Our CruiseControl machine runs the following build script for each module/bundle: checkout from the SVN repository, compile the sources and the test cases, run the test cases, build deliverables (jars). A full build usually takes about 15 minutes, 30 minutes at the most. The deliverables (debian packages, jars and scripts) are made available through the CruiseControl JSP interface. We plan to write simple tools that help us to automatically download and deploy the latest built version at the click of a button.

What we've learned

1. TDD requires a change in the programmer's approach to writing and reading code.

In every project, the following question comes up at some point: Where is the truth? By truth, we understand what the software/piece of code is meant to do, not what it is really doing. A hidden bug is thus a lie in the code.

a) Old/classical approach: In the ideal case the truth is written in plain language in the design documents, and/or as code comments, but more often than not is the code itself. Writing the tests after writing the code leads only to mirroring the existing code into tests.

b) With the TDD approach, the tests are the truth, the code being only the reflection of the tests. In theory, code without a foundation in tests should not exist. In practice, we will not achieve 100% code coverage, but the main advantage versus the classical approach remains: we can always check the existing code functionally against the specification, by simply running the tests. Furthermore, we can automate this process, so that it takes place continuously. With the old approach, checking the correctness meant reading the written specification and the code comments and looking in the code to see if it fulfils the requirements, a error prone and time consuming process. Of course, one can argue that with the classical approach, one can still write test cases and run them, in practice however the programmer that writes the test cases will use the knowledge gained by writing the code, the test case becoming just a reflection of the code, and not the other way around, as with TDD.

This rather philosophical difference in approaching code writing is what we found to be the most challenging to grasp.

2. TDD helps with the module design.

By changing the focus from design and code to test and code, TDD enabled us to think more profoundly about the module interfaces. By thinking in terms of tests, we defined more clearly how the interfaces were meant to be used and what the responsibilities of the implementers were. Furthermore, the modules were testable from the very beginning, a quality that is often lost in classical approaches, making it a challenge to even write test cases.

3. Writing tests first is just one part of TDD.

TDD is not just about writing the tests first. It has two other, equally important phases: refactoring and continuous testing.

Refactoring is the only way of keeping the quality of the code high. Writing code that fulfils test cases can often lead to bloated code. This needs to be refactored. By having the test cases, we can refactor aggressively, as long as none of the tests fails.

Continuous testing is also important. Each change that we make to existing code, be it new functionality, or refactoring, must not break the test cases.

4. TDD is more challenging than classic approaches.

It is a challenge to write good, clean code. TDD brings in a further challenge: write good test cases. Test cases are not born equal, so writing the good test cases and bypass the bad ones is very important.

Here are some of the qualities that a good TDD programmer must have. He (or she) should be able to:

- write test cases that don't overlap: testing the same functionality twice is a waste of time.

- identify the most effective test data: data boundaries, common data, exceptional data.

- identify the next test case to be written

- write test cases that actually test something: trivial code, like getters and setters, simple arithmetic, don't need to be tested

- discipline himself/herself to keep doing TDD, especially if he comes from a code-first development background. This discipline is particularly hard to maintain in the face of deadlines: with TDD the deliverable code is produced at the end of the development cycle, not at the beginning, therefore one has to fight with the urge to write the code first, to try to get it out of the way.

- finally, write test cases that are easy to maintain. Code refactorings should not require equally big changes in the tests. One solution we found was to test the classes through their interfaces, testing the contract fulfilment and the interface protocols.

In conclusion, you cannot replace a good programmer with a bad programmer doing TDD. The results will be worse. Good programmers are needed to write good code, better programmers are needed to write it with TDD.

5. TDD produces better code in less time.

In the end, we discovered that by sticking to TDD, we actually found ourselves at the deadline with code of higher quality produced in less time than originally planned.

I would like to present this empirical formula:

$$\text{time}(\text{test first}) + \text{time}(\text{code}) < \text{time}(\text{code first}) + \text{time}(\text{test}) + \text{time}(\text{debug \& fix})$$

Needless to say, this resonated well with our customer.

6. TDD is an agile method, less than a full process (like Scrum), but more than a programming technique.

In addition to TDD, we also adopted some agile method, without sticking however to one of the established agile processes, like Scrum or XP. We built releases often (once every 2 weeks), so that we can have quick feedback from our customer. In this aspect, TDD helped a lot, as we had from the start code that we could trust, was tested and deliverable. We were able to use the development intervals of 2 weeks to the fullest, even having code changes on the day of the build. Build times decreased during the course of the project, as our build scripts improved and automated more and more of the build steps. The testing component of the delivery, which in other non-TDD projects, kept us late at work and ruined our evenings, was reduced in this project to the minimum: deploying the application on the target platform, running a smoke test (power up and watch out if smoke comes out) and running the not-yet-automated tests.

In true agile fashion, during the development we integrated and implemented unplanned customer requests, deviated from our original design document, where it turned out that starting presumptions were wrong and had a major refactoring halfway through the project.

Of the 16 Releases during the course of the project, 15 were exactly on time, 1 slipped for several days. I think the customer was happy.

Personal conclusions

I wish I had discovered TDD earlier, so that I wouldn't have to unlearn my approach to writing software. I was sceptical at the beginning, but I was won over after the first practical tries. The success of the project proved to me that TDD is a step in the right direction.

References

OSGi - <http://www.osgi.org/>

Fitnessse - <http://www.fitnessse.org/>

JMock - <http://www.jmock.org/>

EasyMock - <http://www.easymock.org/>

<http://www.agilemanifesto.org/>

Authors

George Mesesan.

He is a software engineer at MicroDoc GmbH in Munich and was involved in the introduction of TDD within the MicroDoc development process. He has experience in the embedded as well as the enterprise domains.

gme@microdoc.de

Christine Mitterbauer

She is a senior software engineer and authorized signatory at MicroDoc GmbH in Munich, leads several OSGi-based customer projects especially in the embedded space. MicroDoc introduced Test-Driven Development within these projects and gained quite some experience in TDD. The mostly positive results lead to the adoption of TDD for all development processes within MicroDoc.