

TEST DRIVEN DESIGN CHALLENGES FOR FASTER PRODUCT DEVELOPMENT

Jason Fraser
Florida Atlantic
University
777 Glades Road
Boca Raton, FL 33431
561-297-3000
Fraser.Jason@gmail.com

Baldev S. Mattu
Florida Atlantic
University
777 Glades Road
Boca Raton, FL 33431
561-297-3000
bsmhs@hotmail.com

Abstract - Test Driven Design is a programming methodology that provides an iterative design cycle with integrated testing at each step, leading to reduced time and cost associated with end of cycle testing in the traditional waterfall method of development. In this paper we implemented a small portion of a PBX system to expose common problems in the implementation of Test Driven Design, and propose solutions to the same.

to overcome them. We also considered less common but serious problems that may arise as a result of real time and concurrency constraints. We found that slight modifications to the pure TDD approach can lead to reduced development complexity and ease the implementation of the TDD methodology.

INTRODUCTION

Test Driven Design (TDD) is part of a larger group of programming methodologies that are often referred to as 'Agile Software Development' or 'Extreme Programming' that encourage a faster iterative development cycle. TDD offers advantages over the typical waterfall development, such as improved black box testing performance, reduced code complexity, greater number of fulfilled customer requirements, and greatly reduced test and debug time near the end of the development cycle[1]. Despite these advantages, the waterfall method of development still prevails among software developers. While this can be partly explained by the resistance to large-scale methodology change among developers, we also believe that the pure TDD methodology as proposed by Beck[2] has several hidden disadvantages to developers that may account for the reluctance to use TDD.

By approaching a project as developers new to the TDD methodology, we attempted to discover some of the typical problems that might be encountered in the development cycle and ways

BACKGROUND

The Test Driven Design methodology consists of unit tests being written from a system requirements document that describes the functionality of the system in terms of use cases and other traditional methods. The first requirement for good TDD is a set of exhaustive use cases that can be broken down into very small units of functionality, which are termed 'user stories.' Taken together, all of these user stories make up the complete set of requirements. The customer at this point can choose those user stories that are essential to the system or prioritize them for time and cost considerations. These user stories should be small enough that they can be fulfilled by a developer in a short amount of time, in most cases much less than one day. Each of these user stories have a unit test written for them before any actual business logic is designed or implemented. After the unit test is verified as meeting the requirements of the user story, the actual functional code to meet that unit test is written and tested. This is repeated for each user story with all unit tests being run as often as possible, at each incremental change of the business logic code. By repeating this test and

code cycle working software can be produced quickly and continuously (even if this software initially only meets a small subset of the full requirements). Refactoring of the code base can be performed to reduce complexity as long as it does not affect the ability of the code to pass all unit tests that it previously was able to meet.

Traditional waterfall development leaves most of the testing until the end of the development cycle, where it is more costly to detect and fix defects in the original design. A poor quality product can result if product deadlines force inadequate testing to be performed, leading to increased maintenance and support costs. In addition, customer requirements can change during the development cycle with significant cost implications if these changes are attempted during the implementation and testing phases of waterfall development. While the costs associated with late stage requirements modifications cannot be eliminated, they can be reduced in the TDD case by allowing a comparison on a user story by user story basis of what changes to existing code those new requirements will entail and through the isolation of functionality that occurs while following TDD.

Another important advantage of TDD is that a certain amount of hardware/software co-development is possible through the use of unit and acceptance testing. This can lead to a much faster development cycle when only a hardware specification is available, as the interfaces to the hardware can be written with hardware simulations being utilized to fulfill the business logic until actual hardware is available. When the hardware is made available, the interfaces and tests for that hardware have already been written and validated. In the waterfall method, software development may not begin until hardware prototypes are developed, and there are few tests to validate whether code already written will work.

METHODS

We attempted to implement a small portion of a private branch exchange (PBX) telephone system in Java. It was assumed that hardware and software for the system was being simultaneously developed, so a level of abstraction was necessary when the actual business logic code was developed. A specifications document was developed that described the desired functionality of the system, and from that a set of use cases was defined. These use cases were then broken

down into smaller user stories and eight of the user stories were chosen for implementation. The general hardware architecture of the system was defined as a generic line card containing a call processor that was able to connect to other line cards for internal calls and to a trunk switch containing a finite number of outside lines for external calls.

With the hardware defined, software development could begin. We used Eclipse as our development environment and JUnit for running the unit tests. Each user story had a unit test written for it and then we began to write the actual code to implement that user story, following the TDD process. First, interfaces were written for the line card, call processor, and later the trunk switch. We then wrote implementations of those interfaces that were a simulation of the hardware that we did not have; because of this, the actual business logic contained in these implementations was not as important to us as the interfaces themselves. The unit test for that user story was run against the code developed for it after each incremental change, if it passed we moved to the next user story, if it failed we repeated the test and code development cycle until it passed. It is important to note that in TDD if a test results in a failure, the only modifications made can be to the code, not to the test. Our code base was small enough that refactoring was not necessary

RESULTS

The first challenge encountered occurred well before any code was written, and involved the definition of a set of use cases. For pure TDD, the only code that gets written is that which fulfills some portion of a use case, so obviously the more exhaustive your set of use cases, the better your final product will be. Obtaining this set of use cases can be quite difficult, especially if the end-user customer is not the one who is contracting for the software. A traditional use case for a PBX system might consist of the following: "Telephone extension can place a local phone call by going off hook and dialing 9 followed by the seven digit phone number, the cheapest line available is chosen from the trunk lines, and the call duration is recorded in a database for administrative purposes." This leads to questions such as how and when the call is to be ended, how the database is to be set up, and who chooses which trunk line to use. As such, it would seem appropriate to define our use cases very narrowly to take into account every question we have for

that one situation. While this is a suitable practice, it can lead to an explosive number of use cases for even the smallest of systems. We instead decided to make our use cases small, and define them quite broadly in order to cover as many requirements as possible without missing important areas of functionality, and worry about the questions raised by these use cases during our user story definitions. The use cases that we focused on were

- Telephone extension can place an internal call by going off-hook and dialing the extension number.
- Telephone extension can place an external call by going off-hook and dialing 9 followed by the telephone number, the availability of an external line is determined, and the call is placed.
- Telephone extension can transfer an ongoing call by flashing and transferring the call.
- Telephone extension can place a three-way call to another extension by flashing and selecting the other extension.

From our use cases, we could define many user stories based not only on the explicit language of the use cases, but also on our domain knowledge of how PBX systems typically operate. For example, none of these use cases explicitly mentions the on-hook state of the extension, although we can infer that we need it. By using domain knowledge to logically infer the existence of certain user stories, such as the inverse of a previously defined user story (busy vs. non-busy, off-hook vs. on-hook) we are able to reduce the number of times we need to go back, define and redefine additional use cases that incorporate these user stories. These types of logical operations could be automatically done to infer the existence of additional user stories not covered in our list through software, and use cases could be defined if needed by taking the intersection of multiple user stories.

A few of the user stories that were generated from our use cases were

- Off-hook generates dial tone
- On-hook resets card to default state
- Extension goes off-hook and places call to non-busy extension
- Extension goes off-hook and places call to busy extension
- Flash, call transfer
- Flash, three way call
- Extension goes off-hook, dials 9, and an outside line is available
- Extension goes off-hook, dials 9, and no outside lines are available

It should be obvious that duplication of user stories is not only likely, but expected when multiple use cases are considered. In our example, the off-hook story was included explicitly in two of our use cases, and implied in the other two. Because the scope of our project was small enough we were able to remove these duplications by hand, however a larger project with thousands or tens of thousands of user stories would need to employ a faster and more robust way of eliminating the duplicates. Again, automatic elimination of duplicate user stories could be implemented in software for larger projects.

Unit tests for the eight user stories were written using JUnit, and they all followed a similar structure to the following, on-hook test:

```
package pbx;
import junit.framework.TestCase;

public class OnHookTest extends TestCase {

    public OnHookTest(String name) {
        super(name);
    }
    public void testOnHook() {
        CallProcessor cp = new CallProcessor();
        SimulatedLineCard slc = new
SimulatedLineCard(cp);
        assertTrue(slc.isDialToneOn() == false);
        slc.simulateOffHook();
        assertTrue(slc.isDialToneOn());
        slc.simulateOnHook();
        assertTrue(slc.isDialToneOn() == false);
    }
}
```

Considering the third and fourth, and the last two user stories, we might want to combine each pair into a single story such as “Extension places a call to an extension” and “Extension goes off-hook, dials 9, and checks whether outside line is available.” According to TDD, you should not combine these tests together as it will complicate your debugging of the code you are writing for that test. It turns out that in fact you can combine first two tests into a single test fairly easily, but not the other one. These four tests were a source of two large mistakes that we made in following the TDD methodology that caused a large headache as development proceeded. Part of our requirements document stated that the system had five outgoing lines, and so we wrote a test that made five outgoing calls, then attempted to make a sixth

call, with the expectation that it would return a busy signal:

```
package pbx;
import junit.framework.TestCase;

public class Dial9Test extends TestCase{

    public Dial9Test (String name) {
        super(name);
    }

    public void test_line (){
        CallProcessor cp = new CallProcessor();
        SimulatedLineCard_2 slc = new
        SimulatedLineCard_2(cp);
        assertTrue(slc.isDialToneOn() == false);
        slc.simulateOffHook();
        assertTrue(slc.isDialToneOn());
        assertTrue(slc.isConnected() == false);
        slc.MakeCall("91344");
        slc.MakeCall("91345");
        slc.MakeCall("91346");
        slc.MakeCall("91347");
        slc.MakeCall("91348");
        assertTrue(slc.isConnected())
        assertTrue(slc.outsidebusy == false);
        slc.MakeCall("91349");
        assertTrue(slc.isOutsidebusy());
    }
}
```

Unfortunately, with TDD you are only as good as the tests that you write, and because this was attempting to validate two separate user stories a simple mistake in not setting the `slc.isConnected()` to true when making a call led us to a unnecessary session of debugging through our busy signal checking code.

The tests dealing with “Extension places a call to an extension” were another source of problems. Part of the code for our line card interface is below:

```
...
public interface LineCard {
    public void dialToneOn();
    public void dialToneOff();
    public void DialExtension(String dialed);
    public void DialBusyExtension(String dialed);
    ...
}
```

As can be seen, instead of having a single `DialExtension` (or as we added later, just `DialNumber`), with the business logic in the simulated hardware checking whether or not the number was busy, we split each test into its own method implementation. It is then trivial to implement logic that will fulfill this test, but is completely useless. Herein lies a very easy trap to fall into when using the TDD methodology: it is exceedingly easy to write unit tests and code which fulfill those tests perfectly, but neither of which have any practical value or actually fulfill any of the requirements.

DISCUSSION

Each step of the TDD process makes assumptions regarding both the quality of requirements and the domain knowledge of the individual developer; on large projects this can lead to production delays or disposal of the TDD process entirely. TDD has several weak points that can be overcome with awareness and possible modification of the design process. Translating requirements to unit tests, and then making sure those tests are fulfilled with code that actually does what the requirements authors intended is problematic. Taking broad use cases to user stories and then having experts or software that can automatically infer missing user stories can be invaluable for delivering an end product that meets all the requirements without an explosive growth of use cases and duplicate user stories. In addition, the order in which user stories are fulfilled can have a great impact on the difficulty in implementing the system as a whole. If we had attempted to implement an “Extension calls extension” test before the “Off-hook” was working, we would not have been able to fulfill the “Extension” test without first getting the off-hook functionality working. In addition, while we did not attempt to implement any concurrent or real time constraints, the lack of system design upfront that is common with TDD could make it impossible to meet the concurrent or real time constraints later without a severe overhaul of the code base. We would propose that if concurrent or real time constraints are known to be a factor in your design, that other model driven methods (such as concurrency modeling or UML diagrams) be used up front to provide an additional guide for developers to use when writing their tests and code using the TDD method.

CONCLUSIONS

Test Driven Design holds great promise for developers who need to decrease the development cycle time and deliver functional code quickly. The tradeoffs and disadvantages to TDD that are perceived can be reduced or eliminated through some of the following methods: An overall architecture design (taking elements from Model Driven Design) before code development with the involvement of domain experts to avoid costly design omissions; the involvement of both the customer and domain experts when breaking use cases into user stories; and careful planning in the ordering of user stories to be implemented so as to maximize customer satisfaction while minimizing problems that may arise due to real time and concurrency requirements. The future of TDD may lie in a combination of both Model and Test Driven Design principles, with a greater reliance on software tools to find implied user stories and automatically generate unit tests for those stories.

ACKNOWLEDGEMENTS

The authors would like to thank the Center for Systems Integration at Florida Atlantic University, specifically the One Pass to Production Project funded by Motorola. The authors would also like to acknowledge the whitepaper written by James Grenning "Extreme Programming and Embedded Software Development" as inspiration for the idea of implementing a PBX, and provided a template for the first two user stories to be implemented. This whitepaper can be located at <http://www.objectmentor.com/resources/articles/EmbeddedXp.pdf>

REFERENCES

- [1] Williams, L., and Maximilien, E.M., *Assessing Test-Driven Development at IBM*, Proceedings of the 25th International Conference on Software Engineering (ICSE 03), IEEE CS Press, 2003, pp. 564-569
- [2] Beck, Kent, *Extreme Programming Explained*, Addison Wesley, 1999