Post Mortem Report

IRC client for Android

2013-10-27

Chalmers University of Technology

Wilhelm Hedman
Alexander Hultnér
Johan Magnusson
Oskar Nyberg
Joel Thorstensson

Introduction

In the undertaking of this project we have tried to simulate enterprise conditions as closely as possible. We have tried our best to employ the best practices in Agile development through the use of Scrum. The emphasis on testing and testability in Agile prompted the use of Test-driven Development and Continuous Integration. With the guidance of Clean Code (Robert C. Martin), Code Complete (Steve McConnell), Agile Estimating and Planning (Mike Cohn) and Agile in a Flash (Tim Ottinger and Jeff Langr) we achieved a streamlined development process and were able to deliver a much better product.

Test-driven Development

We set out on a course to use TDD as it is meant to be used. With the modus operandi "red-green-refactor" there is always the temptation to skip the "red" and the "refactor" and continue writing more code with less coverage in tests. We decided from the beginning that honesty with all three steps is the key to long term success. Minimizing technical debt, as with everything, starts with the small things. As necessary as having clean and working production code, is to have clean and working unit tests. To achieve this, the Spock[1] specification and testing framework, which is a DSL for Groovy[2] was utilized. Groovy is a dynamic language for the JVM, allowing us to combine the development speed of dynamic languages with our existing Java classes. The Spock syntax is based on "given-when-then" - which is the most natural approach to testing, and results in human readable unit tests. Groovy has a feature called power assertions, which provide rich information on an assertion failure. Consider a stacktrace, which contains the call stack up until the program execution halted - a power assertion contains all the data which caused the assertion to fail. These very powerful tools make writing unit tests a breeze, and have proven to make it much easier to create passing production code. On several occasions after large refactoring, we came to the conclusion that this would never ever have been possible without unit tests. Our conclusion from this project is that TDD is a must for any serious software project that wishes to deliver a high quality, low defect product.

[1] http://code.google.com/p/spock/
[2] http://groovy.codehaus.org/

Continuous Integration

To allow the unit tests to reach their full potential, we set up a Jenkins[3] server. With our configuration, every time a push was made to `develop`, Jenkins received a hook from Github. Jenkins then pulled and executed the Maven goal `test`. If the goal failed, an e-mail was sent to those that Jenkins deemed responsible for the failure (i.e. those who had committed code between the last passing revision and the current failing one). This proved to be useful, although the definite majority of test failures were detected when running the unit tests locally before pushing to develop (after a while we got tired of the e-mails, so we stopped pushing bad code!)

In retrospect, we wish that we would have had an integration test suite containing automated UI tests that could be run by Jenkins. Having a CI server is definitely a must in any future project, as it "forced" us to push quality code, lest we be singled out as the person responsible for a failing build.

Minimizing technical debt

By our use of TDD we did everything we could to minimize the technical debt on the unit level. What we should have done in addition to this, was to focus on tests on the integration and system levels. Automated UI tests would have saved us lots of time and naturally we would have been able to catch bugs before they enter production. But, just like in any enterprise environment, the feature-mania caught the better of us. With more time on our hands, we would definitely have used these tests. In addition, we might have used code review for all committed code.

Scrum

Clearly, our mini-Scrum is far from the real deal. However we went into it with the mindset that it was real. In the beginning, we sat down and gathered user stories. Acting as both developer and customer, we could find a feasible solution, but the "customer" did not always

[3] http://jenkins-ci.org/

agree with the “developer”. We used planning poker as described by Mike Cohn in Agile Estimating and Planning, with a scale of (0), 1, 2, 3, 5, 8 and 13, to reach consensus on the weight and size of each user story. Our burndown chart can be seen in Figure 1 below. It shows a very stable velocity up until the last sprint. Hardly surprising, the velocity increased by more than 50% the last sprint before release. In the end, we only had to drop one user story from our product release.

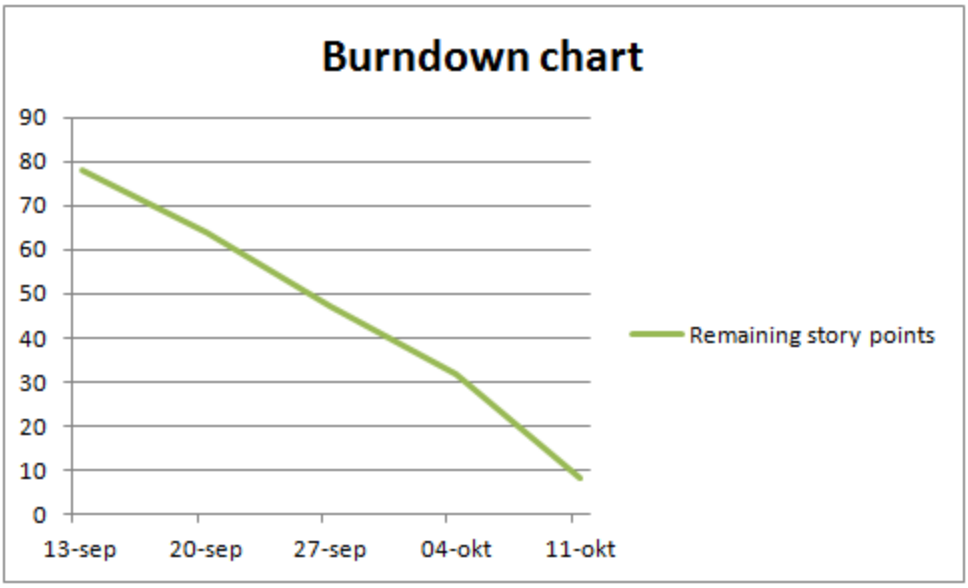

Figure 1. Burndown chart. X-axis: Sprint deadline, Y-axis: Story points.

Our sprints

With our tiny sprints, time was of the essence. We started each sprint with a sprint planning meeting, where we discussed and decided which user stories to implement the coming week. In addition to the sprint planning we had daily standup meetings to update each other on the status of our features. These standup meetings really helped us staying focused and on track since we each day got updated on how much work was left. Each sprint ended with a release on Friday. Which worked out good for us. Not having any scheduled work on the weekend meant that nobody had to feel bad for not contributing on a “day off”. It could have been argued that we could have produced more work if we had more days in the sprint, but in the end, it seems that it paid off to have the weekends off because the work in the weekdays became more focused. As we didn’t meet for business on weekends and a regular week of

work is Monday to Friday it would have been difficult and unnatural to have meetings on a Saturday or Sunday. In the future, where we're not limited by the fact that the course has to have an ending, we would prefer longer iteration lengths.

The IRC protocol

As most of us had been using IRC with various clients for some time before this project we had a basic understanding about how the protocol worked. Therefore we did not think implementing it would take an awful lot of time. After all, sending, receiving and parsing basic strings could not be that hard. We simply assumed that IRC, being the widespread protocol it is, would be well documented. When we later started to implement the protocol and testing it against a real server we realized that the documentation was not much help at all. While all messages and replies you could send and receive were listed, the syntax was overcomplicated and did not always seem to be correct. Because of this we ended up reverse engineering most of the protocol. This was done by sending messages to the server, check what was received and finally writing unit tests based on that reply. Writing code that would work with a real server was then no longer a problem.

Git flow

When developing larger projects in a group, using a version control system is mandatory. We used Git[4] as it simply is the best VCS there is. Git is a distributed VCS with a great branching model. But using Git can cause problems if there is no standardized way to make new branches. Many different branches with weird names can be confusing for other developers. To address this problem we used a branching model called Git flow[5]. Using Git flow worked well for us, it made working on individual features easy. We are all hooked on Git and Git flow, and will continue to use it in the future.

Developer interaction

What we all realize after this project is how incredibly true the diagram in figure 2 is.

[4] http://git-scm.com/
[5] http://nvie.com/posts/a-successful-git-branching-model/

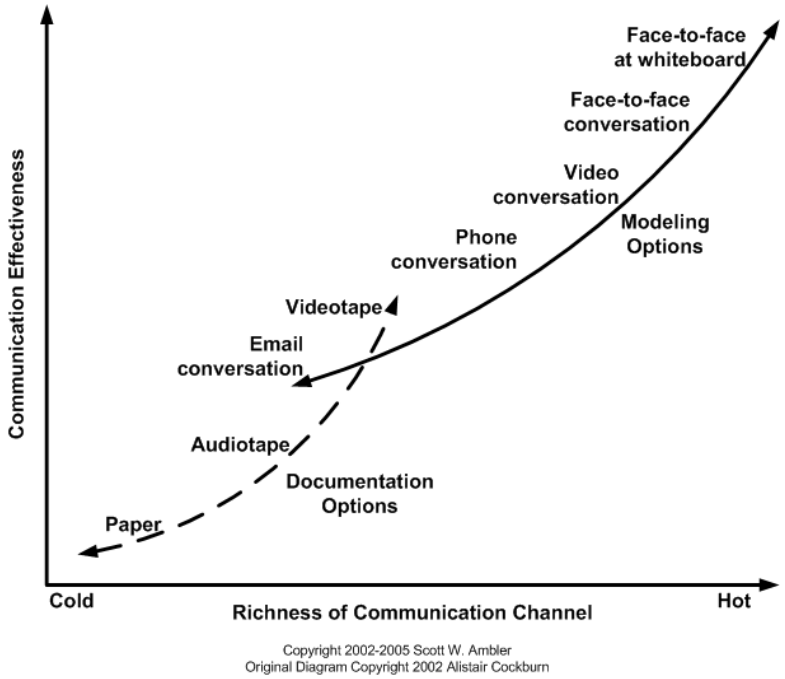

Figure 2. Modes of communication
(http://www.agilemodeling.com/essays/communication.htm)

We could at find ourselves discussing the design, shouting at each other for not understanding what the other person is saying, and then reaching for a whiteboard pen and drawing the problem and the proposed solution realizing that we were saying exactly the same thing. Communication is extremely important, and being thorough in that communication is as important. Some people prefer work from home at times - but that only works when you know exactly what you're supposed to be doing and the task you're performing has no macro perspective. Working together with somebody only works if you're actually together. Our most effective sessions were when we set up our "office" in a room for the whole day and worked together, all five.