Reflection Document - Group 03

Coverage

The first time we used tools for unit tests - and even wrote unit tests at all - was during the spring, while taking the IT programme’s Java project class. We found this to be very helpful while identifying bugs in the code and to prevent behaviors of the software that we didn’t want, together with a greater understanding for the Java programming language and the framework that we used.

In this project, we used Emma to get code coverage percentage and to see what part of the code that was tested. At first we tried to use EclEmma (which is Emma for Eclipse), but we soon found out that it is not compatible with the Android Instrumentation Runner, so we decided to use the command line based tool together with Ant instead.

Setting up Emma in Ant was a bit of a hassle since it’s an extremely fragile process where paths need to be set correctly and issues with mismatching signatures prevented the Emulator to accept the install request from Ant. We got tests running together with Emma using Ant after a long Saturday afternoon though.

After getting an understanding of how Emma works together with Ant we’ve had very much to gain from using it while writing tests. Since we’re all very new at unit testing Java code it’s a great help to get visualized reports on which lines that are covered and which aren’t. It’s easy to assume that all your code is tested, and then when looking at the report you find out that you missed testing parts of it after all.

Team

Developing in teams has the benefits of several minds being able to spot more problems and together reflect over different solutions. It has the disadvantage however when it comes to making quick decisions.

We’ve come across several situations where we’ve had the whole group on hold because two or more group members have not been able to agree upon a solution for a Product Backlog item. Not for too long, and not that seriously. But it’s still a process that takes time from development.

These situations were probably way more thought through though after said discussion than they would have been if the decision had been made quickly by one individual who’d have had no input from someone else before implementing the solution.

Something as simple as coordinating schedules will also end up taking time from development. Even if we’ve all been studying the same courses we all have things to do outside of school, such as work, different circadian rhythms and hobbies. So making sure that everyone has time to do their part of the job while maintaining some kind development team (as in at least parts of the group developing at the same place at the same time) is yet another thing taking time from the actual development.

Team members might have varying level of expertise in different technologies. For example one person might have worked for several years with database technologies or the most efficient way of storing data. Which will benefit the group as a whole with that person being able to share his knowledge and thoughts on how to store the data. And diversity amongst individuals more generally is also something that we feel helps develop both the team members and the application.

Another great benefit of working in a team is motivating each other. Even though we did not hit any low-spot during this project it’s still nice to know that if you do, you have the whole team that will help motivate each other in order to get through.

Overall we had a pleasant time working in a group and we all felt that each and every member contributed to the team.

Docs

Throughout the project we’ve been using Google Drive for writing and storing our documentation. To publish the documentation to our git repository we’ve been downloading new versions of the Google document(s) and committing them before each release/handin.

Google Drive has been a perfect tool for a small development team such as ourselves. If you compare it to a more traditional text editor without internet connection or any kind of live distribution between team members on different workstations it has quite a few very concrete examples where it enhances productivity.
The best example that we’ve thought of is instant feedback from the other members. If one of us writes a paragraph that is not approved of by the other members everyone can in the wink of an eye provide feedback to the author and it can be revised by the whole team.

During the whole project we've always been using this method and tool when we've been in the same room. Even though it would provide us with enough features to manage documentation on separate geographical locations we've felt that a distributed editor combined with real life interactivity and communication is the most flexible and productive way of using Google Drive for documentation.

It also helps keep up the transparency between team members when writing larger documents such as this one when everyone can work simultaneously on very closely related parts of a document without having to worry about writing about the same things and related issues.

Even though we've never had to actually use the feature, Google helps us maintain the version control of each document.

Specifically we used Google Drive Spreadsheet for our Product and Sprint Backlog(s). Mainly for keeping all kinds of documentation in the same place as it helps build an environment where each developer can look for all kind of documentation without having access to the git repository. But it is also a great way to maintain structure throughout the Sprints by having consistent spreadsheets for each backlog with tabs, so each week we have a new backlog with that weeks sprint review, sprint retrospective and time report.

Software Engineering

Scrum

We used Scrum in the development of the product. The greatest benefit in using Scrum is that we can easily avoid over-planning. When using other, less agile methods, it is easy to try to figure everything out in the beginning of a project. As the requirements change over the course of time, the planning we did in the beginning becomes useless, just a waste of time. Things never go as planned, and with Scrum it is easy to change path during the course of the project. Scrum is best suited for solving complex problems, which software engineering is.

Focus on writing code instead of writing documentation, is another great benefit of Scrum. Very often the documentation just becomes obsolete as new features are added or changed. Code is really more accurate as documentation. But, we of course realize, documentation is still important so new developers easily can get involved and start contributing as soon as possible.

As the Scrum Guide says, Scrum is very easy to understand and extremely difficult to master. This is very true. Although some individuals in the team had certain knowledge of Scrum, it wasn't just a walk in the park. Scrum differs from a lot we have learned about working in team, planning and transparency. But now, when we have learned from our mistakes, we find it to be very efficient and look forward to do Scrum in future projects.

One backside choosing Scrum over for example Waterfall when in a group of inexperienced developers however. Is that without a solid knowledge and experience in design patterns such as the MVC pattern, things might quickly develop into code without structure and thought. We've found it hard to know when to make time for refactoring and structure planning which have resulted in a less structured source code than we would've wanted ideally. During an earlier project which had more of a Waterfall approach to it we were strictly requested to follow certain design patterns and had to be able to present these quite early in the project. Applying this on a new project which focuses on very much other new techniques and tools such as the Android API it is easy to forget about the things you (should) already know how to do.

During the beginning of the project, we wrote a list of features we wanted to include in our application. These were turned into User Stories and later into Requirements and Product Backlog Items.

We organized our work by having sprints from Wednesday to Sunday each week. Sprint planning every Wednesday before starting to code and sprint review/retrospective during Monday. This worked well as we could use the weekends to code.

A problem with having such short sprints as we had was that the features we could implement during the sprint had to be quite small. And if you got stuck on a problem for a day, or half a day, it would jeopardize much of the Sprint's planning.

Test-driven Development

We believe that in order to be able to write tests before writing the actual code, you need a lot of experience with the technology/framework that you are working with as well as quite good understanding on how to write tests for applications using these frameworks.

Since programming is often learned through trial and error by playing around with the code, writing tests before the classes while learning a framework would be like letting a five-year old review his retirement plan. If you've used a framework several times, you know what classes that are included and generally how to go about writing the application. Then test-driven development would be a good idea.

This is something we came to conclusion about at the end of this project. And we feel this rectifies why we never tried to execute a test driven development sprint. Instead we wrote the tests closely together with the implementation code and tried to never leave a feature without unit tests to cover it. But with Android being a huge API to get to know we had to make compromises on some features and leave them untested. Sometimes getting a usable product out is worth more than covering it with unit tests.

One example of this was getting ourselves into trouble while writing asynchronous tests for a very small part of our application. This led to a sprint that could've been much more efficient in the end with almost a whole day wasted trying to get it to work.

Purposely left out features

Spotify integration

Spotify offers libSpotify for Android which is nice. There was however two crucial reasons for to leave this out. LibSpotify is written in c which would have required us to use android-ndk and a wrapper to use in our project. No up-to-date wrapper was found and creating one would be very time consuming since none of us have any experience with the c language. The second reason was that Spotify wouldn't respond to our request to receive a developer key.