Post Mortem

SleepFighter

Table of Contents

toxbee

Introduction

We have developed an android alarm application called SleepFighter for Android. The main twist is that the user has finish a challenge, which can be a small game like Memory. But the main focus of this document is not the application, but rather the development process behind it. First we will outline the processes that was used for the project, and their advantages and disadvantages. Thereafter we will go into some detail on the main practices that were used over the course of the development process.

Processes and Practices

The main process of the project centered around the agile, iterative and incremental Scrum process. Our choice of process influenced the organisation of the group in that there is a Scrum master, the person responsible for administration such as booking rooms for the group and taking care of the daily standups, and a product owner, who is mainly responsible for overseeing the product backlog and its user stories, representing a hypothetical customer. These two persons were also part of the main development team, the party responsible for developing the app itself.

Scrum Master and Product Owner

The advantage of having a scrum master and a product owner is that these two have an eye on the big picture and can coordinate the development effort. Otherwise there is a risk that the individuals in the group would work all by themselves and according to individual plans instead of cooperating towards a common goal. The problem here is though that we made this process seem a bigger hassle than it really was.

After a while, we skipped some daily standups because not everyone was satisfied of how they worked. Most of us just wanted to sit and program instead of following the regular steps of the process such as standups, pair programming and retrospectives. As a result, there was not much work for the scrum master to do. But the product owner on the other hand did come to use a couple of times. Whenever there was a hard decision, the product owner always had the final word.

Daily Scrum Standup Meeting

Daily standups were not used very frequently. On our supervisor’s recommendation, we tried doing it every day for one week, but aside from this, it was barely used.

Part of the reason for this is that it was hard to assemble the entire group for a meeting, due the size of the group and the fact that we all are university students with differing schedules. In addition to this, several members found it to be a waste of time compared to

what the practices were perceived to achieve, and instead wanted to work on the code.

On the other hand, the few times we did perform the meetings, we did notice that there are many good things about it; in a mere five minutes, the group was notified what all other the members were doing. The meeting also became a focal point for our work, a point in the continuous process to anticipate or use as a miniature milestone. By performing the meeting we thus became more focused on the tasks at hand.

And finally, we noticed that it is of great importance that all the participants are actually standing up when performing a daily standup. If it is done while everyone is sitting in front of their computer screens, the participants quickly get distracted by other issues, like the current codebase, and this results in the meeting dragging on for ages.

While understanding its value in another project or context, all members did not agree on the value of those things in the context of our project. One reason is that we usually sat together in the same room discussing while programming. Another reason is that the `git log` command/Github commit history page quickly tells everyone reading it what has happened just by reading the commit messages.

Retrospectives

Retrospectives were also barely performed at all. To be precise, we did it exactly one time. This owed much to the fact that the group was generally unavailable at the end and beginning of each week, which led to there not being any good opportunities to perform retrospectives. Although, we did try to encourage individual reflection over the project and its development process.

During the one retrospective, we reflected over a certain developer not using branches in Git at all. This developer kept adding several features and libraries, and in the process completely broke the build. The same developer was at the time also quite difficult to contact, so the group had to spend some time fixing the build on their own.

The one retrospective we did perform yielded good conclusions, such as that we should improve our usage of branches. To ensure that no one worked on the main development branch and broke the build, it was decided that all new features would have to be developed on separate branches.

Pair Programming

The technique was not used seriously more than a couple of times by some of the developers. When it is actually used, it helps to quickly discover bugs and problems with the design. Since two developers are looking at the code at the same time, it is harder for bugs to slip in. And as for writing difficult algorithms and hard design problems, these two

developers can brainstorm and come up with a good solution. The pair can also learn from each other, and improve the quality of their code.

On the other hand, some members were very negative towards pair programming, and did not do it or want to do it at all. The main reason was that they thought pair programming would slow them down. Working too fast, on the other hand, may often result in creation of bugs.

An argument against pair programming is that pair programming takes up more resources. Two programmers working on their own can often achieve more than a pair of programmers working on the same thing would. And it can be a waste of time to make two developers work on trivial problems.

Although pair programming, as it is formally defined, was not fully and always implemented, we did at times sit with each other to discuss and inspect specific issues in front of a single computer. Sometimes we coordinated this effort into partitioning ourselves into groups of two. Some people might call this “light” pair-programming.

Sprint Planning Meeting

At the beginning of every sprint, a sprint planning meeting was held, where the user stories of the week were decided, and then assigned to the members of the group. Thanks to this, in a mere hour the goal of the current sprint become crystal clear to the members of the group.

Although the meetings were held, they were rather informal. These “meetings” were basically performed like this: The developers cherry picked fun features from the top of the backlog, and then assigned them to themselves. So the developers ended up implementing on the basis of being fun to implement rather than their actually being useful to the end-user, though this was not necessarily a bad thing, since it never resulted in any issues.

For the next time, these meetings should be turned into actual meetings. So there should be chairman holding the meeting, and a secretary making a protocol, in order to make the meetings more organized. Or maybe just a stricter product owner, who does not allow the developers to pick as freely or without the say of the product owner, which did happen quite some times. That also includes properly and clearly delegating the user stories and tasks to specific developer(s).

Backlog grooming

This technique was largely necessary for the development to proceed, as it includes breaking down large and cumbersome user stories, “epics”, into smaller ones that are

easier for one or two developers to further break down into tasks and implement. It also allowed us to reprioritize user stories/features when we met complications (e.g. license conflict in using Spotify as a source for ringtones) and add new ones to the product backlog when necessary (mainly developer user stories and challenges).

While it was helpful to us, it also came with complications. The main reason was that we at times added challenge user stories because we thought that a certain challenge or a certain feature in a challenge be fun to include. And after a while we realized the use of breaking down epics into proper user stories, there were times when epics were handled as user stories themselves. In other words, developers started implementing parts of the epic without formally dividing it into user stories, and they either stayed epics or were broken down into user stories to fit the development retroactively, rather than the other way around.

The issue of not being broken down properly before implementation also plagued regular user stories. This was partially due to the now usual theme of being perceived as an administrative task not directly related to actual implementation, but also due to it not always being clear what the tasks would be.

Feature Branching on Git

Feature branches were used during the later parts of the project. At the very start of the project, though, it was a bit problematic to use, since we felt that the core functionality of the application needed to be in place before expanding into new features.

The advantages of the practice is that it results in less merge conflicts and the members do not have to make sure the new features and components work fully all the time in order to not be a hindrance for others. A feature that requires major changes might break parts of the application until it is fully developed, but that is not a big problem if it is only broken for a time in its own feature branch.

A couple of problems though is that the merges will get larger (possibly more conflicts to fix when merging back), and a feature might partly rely on and interact with another feature that is being developed on in another branch. To prevent the first problem, development can regularly be merged into the feature branch. The latter problem can be handled by waiting for the other feature branch to be merged into development, and then merging development into the feature branch, which on the other hand obviously is not very efficient as it creates bottlenecks.

In the beginning of the project, the developers all worked on the same branch, and this created huge problems when a developer accidentally broke the build. Either the developer in question had to be contacted and asked to solve the problem, or if the developer could not be contacted, the rest of the group had to somehow manage solving

the problem on their own due to lack of communication. This often turned out to be a huge waste of time that could have been avoided if branches had been used (or if more than one developer was involved in every part of the app). In the latter parts of the project, we started using branches for even the smallest of features, and as a result the productivity increased. At the end of the process, the number of feature branches had blossomed to over 15 branches.

Issue Tracking on Github

The issue tracking functionality available on GitHub was primarily used towards the end of the project, from sprint four and onward. It was found to be a good way to make everyone aware of problems, since you can attach detailed descriptions and things like stack traces.

You can also choose which developers should work on solving the bug. This ensures that several developers are not separately working on solving the same bug, saving time. So it can also be used to delegate work to developers and streamline the development/bugfixing process.

It is very easy that you forget about the bugs in the application. Thanks to issue tracking, the bugs can clearly be listed and organized, so no bugs are forgotten about.

But using it for every single bug would probably be overkill in our opinion. When for example two people have cooperated and quickly fixed a bug, there is really no need to file a bug report after the bug has been solved.

Another disadvantage of issue tracking is that every time something is done with issues, adding a comment to an issue, making a new issue, closing an issue and the like, an email is sent to all the members. This very effectively spams the mailbox better than most real spammers.

What worked well

Despite the aforementioned issues with developers sometimes breaking the build, the work with the project proceeded at a constant rate for the entire duration of the project. The major reason for this is probably the backlog. Thanks to it, we could clearly see what features remained to be implemented and what we had time to implement in the current sprint, and from this we could easily plan the current sprint, with little to no bottlenecks.

At the very beginning of the project, when we were creating the backlog, some members felt that the backlog was just useless bureaucracy, and a waste of time, but this opinion completely waned in the coming weeks.

Because every developer was working on their individually assigned tasks, it rarely happened that the entire group as a whole got stuck. The few times we did get stuck was because a developer had accidentally broken the build, and the entire group had to sit and fix it as a result. But once we started using feature branches, we did not have that problem anymore.

Another reason we practically never got stuck, was because we were for the majority of the time working in the same room. So issues with the code could easily be resolved by just talking to each other.

What did not work well

Although we did work always work in the same room, there were still problems with the communication. Certain parts of the codebase can be very hard to understand, and since the developers who wrote these parts despised pair programming from the bottom their hearts, it was practically only those who understood these parts well. Had the developers pair programmed, they could have communicated with each other, and the knowledge relating to these parts could have more easily been shared with the other members of the group. Due to how hard these parts were to read, it was often very difficult to understand, let alone reuse the code for other purposes, so lots of code had to be written from scratch instead of reusing the aforementioned developers code. And it goes without saying that it was difficult and time consuming to add new features to this code. To summarize, the cost of not doing pair programming were high for us.

Had we actually performed retrospectives, we could have talked to the developer in question at the retrospectives, and to try to find a good compromise, such as deciding upon feature branching much earlier, or limiting scope of every individual task or user story that were implemented during a sprint.

Non Process-Specific Decisions

Choosing to make an alarm app involves some interesting problems. We as developers need to be thinking about the life-cycle of our app, since it is extremely important for the product that the Alarm is scheduled and goes off as it should, and that might be several days from when the user last used our app. Luckily there is functionality built into Android that makes this fairly easy. Although, there are some things about the scheduling that need to be handled manually, such as making sure an alarm is rescheduled after the device has rebooted.

We chose to keep our app backwards compatible to API level 9, which has meant that much extra code has been written to handle the compatibility so that it still works on lower API levels, but enhancements only for higher API levels will still be available to the user.

This mostly involves the layout, which drastically changed with Android 3.0 and 4.0. Our app will still function well on 2.3 - 3.0 versions, but without things like the new Holo style and Action Bar.

Cooperation

Much of the time we worked together in the same room, which made it easy for us to have discussions and sort out differences within the group. Right after any member discovered an issue or felt something needed to be discussed, it could often be shared with the group right away. There was often no need for having formal meetings.

Except for sprint planning, daily standups, retrospectives and all other actual process related questions, which on the other hand were hard to discuss constructively, because despite being one of the core subjects of the course, the Scrum process was met with antipathy, as were other issues that were not directly related to creating the product.

For most of the members in the group, working in a software project in a group of six people was a new experience. Keeping track of each other and staying in sync is more difficult than if you were fewer. Even practical issues, such as planning of meetings, and making decisions in the group, naturally takes longer.

What to do Differently

In the future, we will probably be less negative towards new things. All the Scrum techniques like retrospectives, pair programming and the like, although barely used, did bring good results every time we performed them. Thanks to the only retrospective we ever performed, we discovered the worth of branching. Every time we did standups, it become obvious to everyone what the group was doing.

What might be worth trying out using pull requests (GitHub feature) where feature branches can be reviewed and commented on by other members before they are merged. The author(s) could also give a description of what has been done and why. At least having some "review" policy of other member's code would also mean having more people knowing about that particular part of the application.

Another very valuable feature we very rarely used was the GitHub issue tracker. Our supervisor very often endorsed it, but despite that, we completely ignored it this until the end of the project. In this period, when a majority of the time was dedicated to solving bugs, we started using it to organize and assign all the bugs (and their fixing). And it was at this time we saw how extremely useful this feature was. We could have gained so much time had we only used it from the start.

And finally, in the future we will put more focus on cooperation and communication. We thought that just working in the same room was enough communication. But seeing how there are many parts of the codebase that only one developer understands, this was clearly wrong. So we need to work even more closely in the future.

Time Estimate

Below is a table with approximate amount of hours that was spent on different parts of the project, by the group's different members.

	Danny Lam	Eric Arnebäck	Johan Hasselqvist	Laszlo Sall Vesselenyi	Mazdak Farrokhzad	Niklas Helmertz
Requirement Analysis and Research (External & Internal)	20	20	20	20	40	25
Programming (incl. Javadoc & Design)	65	105	110	70	280	115
Project Documentation (excl. javadoc)	20	5	15	25	2	15
Testing	1	10	5	3	0	5
Scrum Standup	1	1	1	1	1	1
Retrospectives	1	1	1	1	0	1
Sprint Planning	8	8	8	10	8	8
Backlog Grooming	10	0	5	10	3	5
Total	126	160	165	130	334	175

Total time spent: 1090 hours.