Reflective Essay

Introduction

The project is in some ways a very typical software project: we had a set of challenging technical problems to overcome on strict deadlines, the group had varying levels of experience, meaning power dynamics naturally evolved as the project progressed and the group was divided into domain specific tasks.

One of the somewhat typical aspects of this project was that all of the members mainly worked from home on each of their tasks, although remote work has become common usually team members are expected to adhere to the same working hours (accounting for time zones). An issue that arose because of this was if someone was having trouble they may or may not find someone else available to assist. If no one happened to be available this on occasion lead to tasks taking longer than necessary. As everyone also had other deadlines and work to do this often lead to issues being worked on together during our weekly scheduled QA meetings that always occur on Wednesdays or on the Weekend. These QA meetings acted as a Soft deadline for work, this also prevented huge merges from our development branch to master branch, making it easier to find bugs during QAs as less code was introduced at once. QA meetings often took much longer than they should have since several hours would inevitably be spent resolving issues before the actual QA could begin.

Code Review

A Software process we found we benefited from the most was Code Review. As mentioned individuals in the team had varying levels of experience, code review was a means of distributing technical expertise throughout the team alleviating this disparity to some degree. Even in terms of technologies, the team had no experience with things such as React Native people independently learned about them as development progressed and this knowledge would be passed on to the reviewers (this is an atypical expertise dynamic in the code review as usually more experienced devs are the ones doing the reviewing but useful nevertheless.)

Perhaps another practice that made code review so effective is that it was built into our version control and branching strategy (the negatives of this are discussed later). In order to get an issue into the “development” branch, you would need to create a pull request which required approval. This makes code reviewing part of the process rather than something individuals would need to think and remain diligent about.

A significant limitation of our code review was the varying quality, members had a varying level of time and fondness for code reviews and so many were far less rigorous than desired.

Alongside the fact that many PRs introduced over 1000 lines of code (classified by Google as “freakin huge”) the same level of rigour could not be put into these without reviews taking several hours. We were unable to avoid introducing this many lines of change as at the start of a project it is required that a number of lines are added to the code base. To overcome this issue a code review checklist was generated, improving the likelihood that common bugs and quality issues were checked for as well as basic checks such as whether or not the new build even runs.

Code reviews were found to have a real effect at excluding the majority of bugs, although anecdotally, there was an occasion in which one group member thought code had already been reviewed and so they approved it without review, this lead to 3 errors that held up a QA review and reduced the productivity of all group members the evening of that QA. It is perhaps unreasonable to assume that every single code review found all major errors in the code but it goes to show the power of the code review quality gate.

Due to the separationist nature of our branching strategy , we found that code reviews extended accountability of the code, this has obvious benefits namely that more effort is put into finding and fixing bugs, although in our case the level of accountability between the initial feature developer and the reviewer was heterogeneous in terms of the accountability split.

As this is only a project for a single module theoretically we should be spending around 2.5 hours a day working on the module. Therefore, as a good code review could take between 30 - 60 minutes the turnaround time for a code review would be increased as a code review should Ideally be completed in one sitting. This means that around half of a day's work could be just on a code review and this could lead to slowdowns in development.

It should also be noted that code reviews are not a free lunch they take up one of the most precious resources, developer time, although because of the number of benefits we considered it worthwhile. However, there are tools which allow for code reviews to be sped up. To start with we used Github’s Pull Requests but ended up using Reviewable.io. Although Github’s Pull Requests were quite easy to use we found that they were a bit limiting in the end. We looked into other review tools such as phabricator and Upsource which while interesting required the setup of a server and links to the Git repo whereas reviewable simply required the developer to log in with their GitHub account and it displays the status of the pull requests.

Technology Choices

The React Native Frontend

We wanted to develop a multi-platform application however, none of us had iOS development experience or a device inside their ecosystem other than one group member. Although development is possible without iOS devices using an emulator many of us had heard this was a difficult set up to develop with. For the front end we ended up using a platform-agnostic solution, React Native. Specifically, we chose to use the Expo toolchain that's built around React Native.

The choice we made to use Expo was initially great for our needs, but as the project progressed and adapted we eventually found ourselves in a problem. The libraries and frameworks in expo were far scarcer than we expected. This both meant Expo had very limited functionality, but the counterfactual of configuring each platform could have also been arduous, so the true cost is unknown.

Another issue with Expo’s limited functionality is it did not have heatmap library meaning we had to implement this feature from scratch. While the final product works as intended, it is not an optimal solution as the default heatmap library in Google Maps looks prettier and is much faster.

Automation

SonarQube

SonarQube was implemented as it provides the capability to not only automatically show health of an application, but also to highlight newly introduced issues. With a Quality Gate in place, we were able to can fix the problems that were missed in manual code review and therefore improve code quality systematically.

Travis

We chose Travis as our main automation tool. While Travis proved to be extremely helpful when working, configuring it to a working state was quite problematic. When we came to get Travis to run SonarQube we initially tried using Travis’s built-in addon for it, however, this did not function as expected. The initial problem seemed to be that it could only run on compiled code and wouldn't compile it itself. While this wasn’t a problem in itself because we were intending on using Maven to compile the code and run the unit tests, but this was meant to be done for a later deliverable, so this feature got blocked because of this.

When Travis was not functioning as expected it was often hard to diagnose whether it was our maven build or Travis itself.

Pre Commit Script

In the early stages of the project, we implemented a pre-commit git hook formatting code before it was commit. However, we quickly found that it was erroneously formatting React JSX code introducing errors into the code. Later, we used it to format the Java code however it was not making the best decisions when it comes to where to break lines and therefore would require much tweaking to produce the desired result. It was much more appropriate to let developers format code.

Branching Strategy

We opted to use a structured branching and branch naming strategy in our project which goes as follows **[Initials]-[Issue number]-[Issue Name]** e.g. **gm-227-LocationSearchBar** which reaped many benefits, namely that of greater orthogonality and that there was usually clarity in which branches held which code. However, it had an insidious nature in that it contained the initials. We believe that looking back productivity was stifled as initials created an attitude of issue ownership.

Another disadvantageous element of our branching strategy was that because to the individual ownership of issues, peer programming was never conducted. As individuals were assigned to work on their own issues and not actively promoted to collaborate with others we did not in any way utilise peer programming, regarded as a frustration to many but in hindsight a hugely useful learning opportunity for the less experienced group members. This experience divide was one of the most pervasive issues on the project and this could have in part remedied it.

Domain-based Division of Tasks

This involved splitting the group of 6 into 3 teams of 2 each, the teams were as follows: Frontend, Backend and DevOps. However, this was only implemented after the second deliverable. We split the group up as we were struggling to learn Spring Boot and React Native at the same time. This allowed us to speed up development as we only had to learn how to use one new technology. However, this did cause problems later on as the DevOps team started to run out of work to do they were unable to easily switch to the other two teams as they had to learn how to use the new technologies from scratch again. Which reduced productivity towards the end of the project.

Those with prior backend experience went to the backend team, and those with prior DevOps experience went to the DevOps team, the remaining members worked on the front end as no one had experience with React Native, this lead inevitably led to issues down the line with front-end development.

Dependency Management

This involves seeing which issues are dependent on other issues and therefore how to order work and how to assign work to developers. However, there were a number of occasions in which developers were unable to progress with assigned work as there was a necessary antecedent issue that was not yet finished. This problem was identified and solved early in the project by splitting up the teams and “faking” data received in order to progress.

This could be improved further by using smarter issue management tools over Github's Issues such as Jira, TargetProcess or YouTrack. However, these tools require a lot more configuration and set up and typically would have a single person dedicated to managing and distributing the issues. However, our team size was not sufficiently large for of a dedicated issue manager, as our project manager was also a developer.

Conclusion

Overall we believe we have a good base in terms of software engineering processes. Most if not all of the techniques we used were useful transitively contributed to simplifying development and improving software quality. Although there were many ways in which we could improve our software engineering process many would more difficult and/or time-consuming tasks and their value in a 12-week project is questionable. We feel the experience has taught us much about software engineering processes and the tools in which to simplify it.

Word Count: 1948