Group 2 Advanced Software Engineering Reflective Essay

1. Introduction

Using professional software engineering processes the group project involved working as a team to design and create a mobile application and a supporting backend to serve requests to the database. The team was required to change the system in response to changing customer needs, validate these and verify that the system fulfilled the customer's needs.

2. Essay style

This essay bases its reflective topics on the chapters of the book Software Engineering by Ian Sommerville. What follows are the team's thoughts having taking inspiration from the book and applied it to the group project.

3. Software Processes

The team recognised early on that it was important to adopt software processes. One of them was Slack. Slack served as an instant messaging tool and was used in the following ways: firstly, as an early warning system, Slack was coupled with the continuous integration process and so could flag when code was committed to GitHub, show the result of software builds from the two remote machines, and report on the automated deployment of backend code completed with Forge. It was for such a wide range of integration that the app was chosen originally. The second use of Slack was for chat between the team members. It improved the team's communication and kept all team members up to date with their progress. Slack was not so good when it came to debates and discussions. Sometimes the team ended up having conversations that lasted thirty minutes or more, that would have been better suited to a face-to-face discussion that would have been completed within five minutes. Failing that, adoption of Skype or similar tool would have also reduced this problem.

Given the changing requirements and short timescales the group aimed to produce "a minimum viable product" as soon as possible for each deliverable in order to validate the requirements and implement the changes if required.

Given the lack of budget, the choice of appropriate CI servers was limited. The team ended up needing two different servers, one for the Android/Java implementation and one for the server side that could handle PHP. These proved more difficult to configure than the team would have liked in that they required scripting which had a learning curve putting even more pressure on timescales. With more time allocated to the setup of the infrastructure at the beginning of the project it would have been possible to perhaps have made better choices. The

team recognised the benefit of CI servers for automated builds and running of tests.

4. Agile Software Development

a. Agile development techniques

In developing the engineering model and processes further, the team decided to adopt a Git-flow approach. The benefits were clear, namely that each team member knew exactly how to develop a feature and conduct the release process. It was automated, and repeatable.

Further, since the team was already using GitHub for version control and had adopted SourceTree as a user-friendly front end this made practical sense. SourceTree allowed for the reduction in the likelihood of making mistakes with the git command line by using a GUI to simplify commands. These would otherwise have been prone to human error.

SourceTree further simplified matters by having Git-flow built in. This allowed for more complex actions but operating within the safety-net of SourceTree, such as creating, managing, and merging of branches in the respect of feature development, proposed releases and ultimately a "clean" master branch containing only code that had been made available to the customer.

Other approaches were considered such as a fork/pull-request model, and while that might have worked it was realised that it is more suitable for a project with many contributors (often geographically dispersed) and that it required some form of approval process for merges.

It had been stated in the project plan that the team individuals were equal and so the idea of introducing a reviewer hierarchy did not really fit well.

There was also the risk that it might have introduced developer-complacency. A developer knowing that there was someone available to check committed-code might have been less rigorous with their own checks and balances.

Additionally, as a small team operating on very tight timescales finding the time to conduct a series of back-and-forth communications to correct code would have been difficult. Change-reviewers would have been required to interrupt their own work in order to conduct approvals and the overall process would have put an enormous burden onto the reviewer perhaps affecting their own plans and timescales.

That said, the team recognised the advantage of conducting code reviews. With the constraints of the project they just needed to be planned and conducted with the entire project team present to extract the most value.

b. Agile project management

The team decided that working with an information radiator board in the usual agile manner was the most appropriate way to communicate the status of tasks. The team's choice was Trello which offered an appropriate balance of simplicity and functionality. It would have been easy to adopt a more complicated system such as Jira and get bogged down in the intricacies of the product. Trello served as a single source of truth and with team members working independently, often at different times of day worked well to keep the entire team up to date.

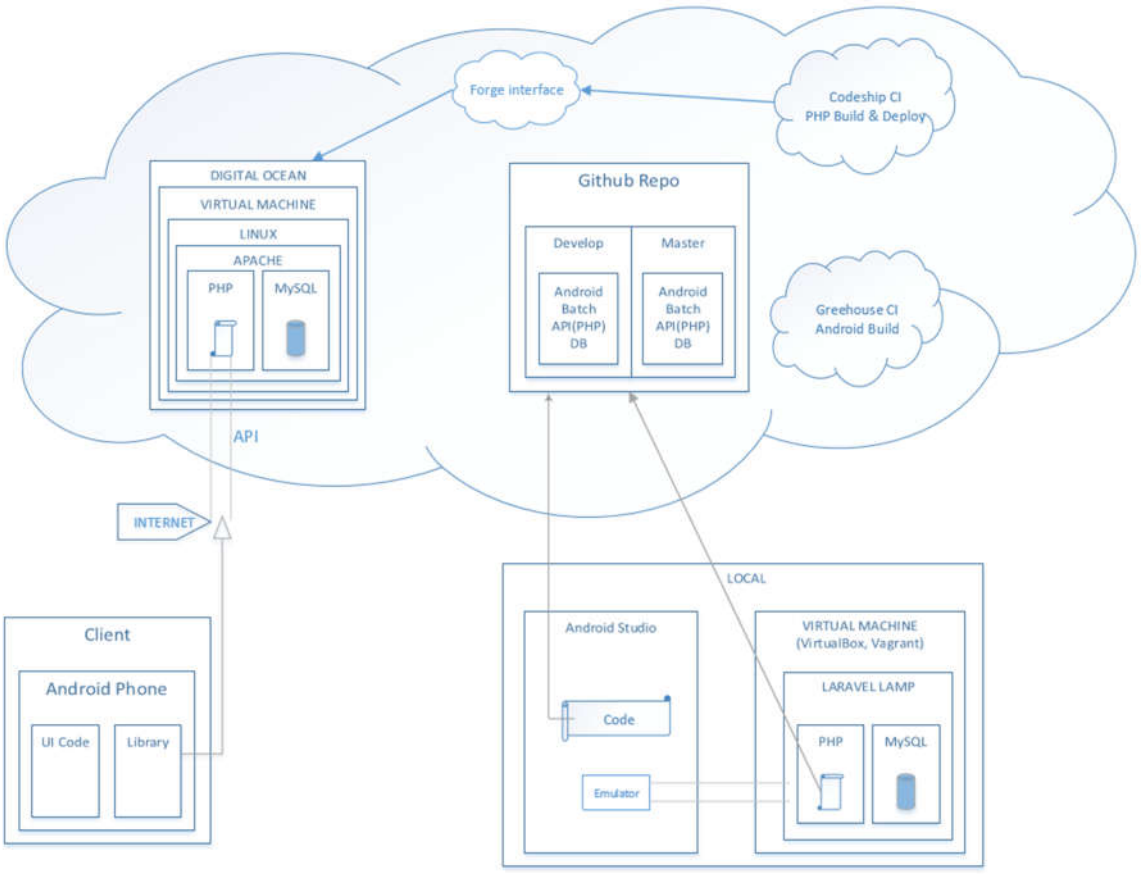
5. Architectural Design

a. Architectural design decisions

Architectural design was one of the tasks that had to be done thoroughly as identifying the main structural components and their relationships would be critical for the development of the system. Changing the architecture of the system after development has started can be costly and time consuming and as such it had to be decided before any development started.

The main components of the system are: The Android client application, the Lumen API hosted in the cloud (Digital Ocean) and Forge, the GUI used to manage the server. Figure 1 shows the detailed architecture.

Figure 1



As seen in the figure above the group used virtual machines for local development which runs the same operating system and has the same versions of programs as the server. Using virtual machines offer various advantages over developing natively on the host machine, a few of these being:

- Isolation from the host operating system.
- Being able to switch to different versions of software on a per project basis.
- Ensuring that all members working on the project is running the same environment.

Choosing the right framework for developing the API was another important task that needed to be done before any development could begin. There are many web development frameworks available for use that can be used to create web applications. The team considered these options and narrowed it down to three possibilities: Lumen (PHP), Flask (Python), and Sinatra (Ruby).

It was decided that Lumen would be used as it's built on the most popular PHP framework Laravel, which has the biggest market share and because of this is better documented and easier to find support for online. Another potential benefit is the ability to migrate the project into a Laravel application, adding more features such as views and middleware without needing to restructure the entire project.

The team decided to use Digital Ocean as its server provider as it offers dedicated computing resources instead of other solutions such as Amazon Web Services which offer a "flexible" service that scales the cost as you use more. Instead of managing the server directly from the bash of the server a Laravel Forge was used. It provides an easy to use GUI which is used to set up the server, the database and Git hooks to various CI servers to deploy the latest builds that passed the tests.

In the Android application it made sense to utilise existing implementations like Google Maps.

For graphing the MPAndroidChart library was used, licensed under the Apache License, Version 2.0

Early on the application was developed using the web version of Google Maps, but later it was switched for the native Android version on the basis that it would be more performant. While that turned out to be true it also became apparent that the two maps were implemented with different functionality - specifically in the way the heatmap clustering of points worked.

With the native Google Map when it is zoomed such that nearby points overlap, the map clusters the points and for each point involved increases the weighting on that location. That means that the larger the number of clustered points at a given location the greater the weighting, which is

reflected visually in the heatmap by making the point red. That was not the behaviour that was desired since the heatmap was meant to show relative property pricing not property density.

To overcome this issue, the team experimented with a backend process that would cast a virtual grid over a given area defined by the map display. If a grid square contained multiple points they were removed and replaced with a single point that was given a weighting based on the average prices of the properties in that square.

This was difficult to get right and it was only later that the team realised that the web version of Google Maps had a setting that allowed for density to be ignored when clustering points. By that time, it was too late to switch the map implementation again, but even if it were possible may have introduced different issues.

b. Architectural patterns

For simplicity and convenience, the team used a restful API design over HTTP. The code makes these calls asynchronously allowing the Android UI to remain responsive.

6. Design and Implementation

a. UML

The team made use of some UML diagrams to confirm and communicate design ideas.

b. Design patterns

The Android implementation was developed using a Model-View-Controller design pattern to separate presentation and interaction from system data.

No design patterns have been followed in the backend as its only purpose is to listen on specific routes and query the database with parameters taken from these routes and perform certain mathematical operations.

7. Software Testing

At various stages the team conducted manual testing. This largely consisted of exploratory testing. Given the size of the project this seemed adequate, especially when performed in conjunction with the automated unit testing.

Having abstracted away the Android dependencies and using dependency injection it allowed for concentrated unit testing on the code written by the team using JUnit.

Use of Mockito further aided the unit test process.

The code coverage was not as good as the team would have liked overall, but as can be seen in Figure 2 for code that had a separation of concerns i.e. without direct references to Android or other 3rd party libraries the coverage is 100%. Classes like DataTask are multi-threaded and therefore inherently difficult to test.

Coverage Summary for Package: org.softshack.trackme

Package	Class, %
org.softshack.trackme	44.6% (25/ 56)

Class	Class, %
DataSetMapperFactory	100% (1/ 1)
LocationProvider	100% (1/ 1)
TrackLocation	100% (1/ 1)
BuildConfig	100% (1/ 1)
MapDataProvider	100% (2/ 2)
MapsActivityView	100% (5/ 5)
GraphsActivityController	100% (3/ 3)
GraphsActivityView	100% (1/ 1)
GraphDataProvider	100% (3/ 3)
JSONFactory	100% (1/ 1)
MapsActivityController	100% (6/ 6)
MyAxisValueFormatter	0% (0/ 1)
DataTask	0% (0/ 1)
R	0% (0/ 17)
Manifest	0% (0/ 2)
MapsActivity	0% (0/ 1)
DataSetGraphMapper	0% (0/ 1)
BaseActivity	0% (0/ 1)
TaskFactory	0% (0/ 1)
GraphsActivityModel	0% (0/ 1)
GraphsActivity	0% (0/ 3)
DataSetMapMapper	0% (0/ 1)
MapsActivityModel	0% (0/ 1)

Figure 2

PHPUnit was used to test the backend, testing valid and invalid routes, and asserting the expected results.

8. Software Evolution

The software evolved with each task that was asked for by the customer. This meant that some code got more sophisticated and the standard improved over time. The team made use of the extensive refactoring capability within Android Studio to help achieve this.

9. Security Engineering

The team recognises that the app was operating over an unsecured network and would consider changing it for any real-world deployment/usage.

10. Resilience Engineering

Currently there is little provision for recovery from failures. The team implemented code assertions and debug logging within the application, and on the server logging of requests and pinging of the server coupled to SMS notifications to in part manage such issues.

11. Software reuse and Component-based Software Engineering

The team implemented some of the application code in separate components. This naturally allowed for decoupling and supports the possibility of reuse.

12. Software management

a. Project management

At the very beginning the team lost two members. The team were given the option to either disband or continue as the smaller team. Choosing to remain as a team of three meant individually taking on more work.
This also meant the less experienced members were under pressure even more. Through various pair programming meetings and code reviews they were able to understand the implementation and offer meaningful feedback.

b. Project planning

Looking back at our project plan, some things went as proposed and anticipated, others did not.

The team first decided to use Eclipse ADT as a development tool, but that proved difficult to install and set up with recent updates to Android, so it was swapped with Android Studio. The limited support for this in the University's laboratories did not aid the development process which meant that on occasion it required individuals to remotely connect to home machines to complete programming tasks.

The team had envisaged that individuals would work on vertical tasks, i.e. complete a task that involved working on the full stack to implement a feature. In fact developers were required to specialise in either the Android or API development because each needed extensive investigation. Even if it had been possible the scope of the project was not really broad enough to accommodate enough vertical tasks.

The team did not have any need to implement its conflict resolution plan.

13. Relating to the real world

The team's project experience was not necessarily that similar to the real world.

Normally team members are allocated to their project on a fulltime basis. Often that means that real-time communication is easy, especially if workers are co-located. The project situation was different. Conflicting timetables, deadlines and personal priorities made meeting difficult and affected individual capacity from week to week. This in turn made it difficult to plan work.

Perhaps it could be argued that in the real-world capacity can fluctuate because of annual leave or sickness. The former however, is generally flagged early in the agile cycle meaning that it can be catered or planned for, or in extreme cases, not approved. Absence due to sickness, while could be prolonged, generally doesn't happen very often.

14. Conclusion

The learning curve associated with new platforms in respect of the short delivery timescales was found to be the most problematic area for the team. Given more time, the team could have self-trained perhaps through more pair programming.

Despite this, the team enjoyed the challenge and believes the project was a success as all tasks have been accomplished and delivered on time. Modern software engineering processes have been followed and the team has learned valuable skills from the experience.

In addition, the project could have been improved; for example, by designing and implementing a better user interface utilising HCI methodologies and, given a budget could have provisioned higher performance in all distributed aspects, providing a more concurrent, faster service.

The team recognises the shortcomings of the current implementation that would prevent release of the application to the public domain.