Advanced Software Engineering Team 3 Report

In this Advanced Software Engineering coursework assignment, our team was assigned a series of tasks that completed in sequence gradually progressed towards an android app that displays the Land Registry Price Paid data for nearby properties on a Google Map fragment. This report chronicles, in our own words, the work that each individual contributed toward the completion of the project in its final state.

On the following pages you will find approximately five hundred words from each team member describing the challenges, triumphs and lessons that each one of us encountered during the course of completing our sections of the work. First off, Grant will describe the development of the Front End in Android Studio. Following this, Dan will describe the development of the Back End in Java Netbeans, before Mehmet describes the Database Systems initially in Java Derby and finally in Heroku and PostGre SQL. To finish everything off, Alex will describe her efforts to test the system.

Front End (Grant):

The project’s front end is an Android application running at API level 24. The application employs Google locations services to fetch and display location coordinates (latitude and longitude) for the user; as well as plot both the user’s location and nearby properties (within the user’s current postcode) on a Google map.

Android programming has proved to be a challenge above and beyond that of basic java programming. Whilst sharing all of the same underlying tools, Android introduces several new concepts to the aspiring programmer, such as the activity lifecycle of applications, requiring the developer to understand what Android is doing behind-the-scenes at any given time. The developer must also be mindful of how the application’s computations interact with the visual XML layout, designed for the user, mapping each element of the display to its background code appropriately.

The following figure is an approximation of the application’s functionality, intended to illustrate the core sequence of events involved in its lifecycle.

**MainActivity**

**Client – Asynchronous task**

App fetches user location (lat/lng) from Google location services.

User’s current location is displayed on a Google map below their lat/lng coordinates.

The Asynchronous Client class is instantiated and executed. User location is passed to Client via constructor.

Application plots properties received from back end on Google map, displaying the postcode and sale price on-click.

Markers are colour coded according to price bracket.

Has the back end responded?

**NO**

**YES**

Opens a network socket and connects to the application’s back end.

Sends user’s postcode location to back end in order to be queried against land registry database.

Receives multiple nearby property sale details from back end.

Price, postcode, address, transaction date.

Stores nearby house sale details in ArrayList that can be used by MainActivity class.

The greatest challenges in developing the front end of this application were thread management, communication between classes and attempting to implement various 3rd party APIs. The application’s main thread (UI thread) performs a great deal of work each time it receives a Google location update. This computational requirement occasionally caused the application to crash unexpectedly and frequently made the interactive map unresponsive whilst it was ‘thinking’. Calling methods between classes also posed its own issues. Due to the inherent nature of Android programming it is impossible (or at least poor practice) to create an instance of the MainActivity class, making it became very difficult to implement MainActivity methods in the package’s other classes.

It was our hope to reduce the computation requirements of the application by moving all geocoding and reverse geocoding to our back end virtual machine. Unfortunately, time and group experience prohibited us from achieving this in time for the task 5 deadline. This improvement should have removed a significant amount of workload from the phone’s processor, allowing for other upgrades such as additional map marker placement and more frequent location updates, boosting the application’s performance. Given more time, this would be worth re-attempting.

Back End (Dan):

When it came to building the backend for our app, the primary matter of concern was to have an initial file system in place to receive and store data from the phone. Even before setting up the connection protocol to transmit and receive the data, there needed to be somewhere for it to go, as well as a framework in place to dictate what information the phone actually needed to send to the backend.

To begin with, when the backend was connected to by a phone it would look for a folder in its directory named according to the MAC address of the device that initiated the connection, and create a new one if one did not exist. In this folder it would create a .txt file named according to the System.currentTimeMillis() at which the connection was opened. The Latitude and Longitude provided by the phone’s GPS was then saved into this file.

Once the file system was in place, it was time to build the port and socket network interface so that users’ devices could actually interact with the back end. Sadly, what should have been a simple affair turned into a nightmare. Our socket implementation simply could not be coaxed into functioning properly. Ultimately it took the entire team working together for two weeks longer than planned for us to get through all of the firewall, network security and virtual machine routing issues until we got consistent connectivity. The end result of this was a catastrophic delay that we were ultimately unable to make up for as the project progressed, leading to a chronic struggle to keep up with advancing deadlines.

Once the Socket implementation problem had finally been fixed, it was time to build a file reader that would input the Land Registry Price Paid data into the program’s file system so that it could be distributed to users when their devices connected to the back end. To begin with a file reader converts each line of the Land Registry Price Paid .txt file into an Array List of tokens. The tokens that correspond to the information relevant to the app are then stored in another Array List, while those tokens that are of no use to us are discarded. The tokens that have been stored in the second Array List are then stored in the program’s database using SQL, and the process begins anew and is repeated until every transaction record in the file has been saved.

Now that the data was in place thanks to the File Reader it was time to build the Data Querying method that would allow the data to be accessed by and sent to user’s devices. When the phone connects to the server it geocodes its GPS coordinates into a UK Postcode, which it sends across the socket to the backend. The Back End then plugs the first half of that postcode into an SQL statement that selects all transaction records whose postcodes share that first half. The server then transmits those records across the socket where the Front End takes over processing that output.

Originally the database was built and queried by the same program, when it was run it would do a check to see if the database was populated or not. If it was not, the program would run the read file method and populate it, otherwise it would proceed straight on to listening for connections. In order to improve efficiency, the program was split into two towards the end of development. There is now a database building program and a database querying program, so that administrators have finer control over which function they wish to invoke.

Databases (Mehmet):

**Backend Migration**

We used a compute engine on Google Appengine to run our backend Java project, Postgres on the Heroku to manage our database on the cloud. We can show this graph to make it more clear:

Appengine

Heroku

Android

Firstly we created a database on the Heroku. We chose Heroku because it is free and has enough speed for our application. And then we created a virtual machine on amazon but it was very slow because of low RAM and that’s why we moved our virtual machine to the Appengine. We provided connection between backend and database with using JDBC. Android side is connecting to the Appengine with TCP with using 23456 port.

The application finds location in android side with Geocoder and we get Longitude and Latitude by using this method. And then we convert these information to get postcode in android side. Then we send these data to backend and backend send the related postcode to database. Database returns to backend an arraylist which contains id, address, price, postcode and date. Then the backend send this data to android side to display it.

While we were implementing this project, at beginning part of the implementation I claimed that using database and http request would be better for next implementations. So, Dan and Grant continued implementing with file systems at first and I tried to rewrite all the backend by using new technologies. But due to lack of experience on this new technologies I couldn’t manage this clearly. In this part of the project I realized that java implementation and android implementation are different even though they seem very similar.

The most important issue during the project development was network communication. We couldn’t communicate frontend and backend for a long time. But after moving the project appengine we didn’t face this problem again due to our previous experience.

We used PgAdmin4 to connect and design our database on Heroku due to its simple interface and strong structure. But in the next steps in the project, Dan and Grant used Derby database for a while, because of performance issue we decided to move the database Postgres again.

I think that my most important benefit in our project is that when our team members faced with any problem regards to database or Java, I most probably could solve it, expect for the network issue and I tried to make code review during the project, especially on the backend and database implementation.

Testing (Alex):

Writing unit tests for the front-end of the application proved to be a significant challenge, due to the nature of the android application framework. The majority of methods contained in MainActivity utilised dependencies on different libraries and objects that could not simply be instantiated in unit tests; instead, mock versions of these objects had to be created in order to successfully run tests on the system. This was achieved through a combination of Junit4 and Mockito, which is a mocking framework that allows you to create mock versions of objects and define their behaviour within tests.

For example, the GeoCoder object, which is used to convert between postcodes and latitude/longitude notation, was defined as a mock object using the @Mock annotation. This could then be injected into an instance of MainActivity using the annotation @InjectMocks. Whilst initially setting up the tests, I came up against the issue of how to pass mock objects that methods that would usually create instances of the actual versions of those objects within their body, rather than receiving them through their respective constructors. Ideally, tests should be created with as little refactoring of the code simply to facilitate the tests themselves without any use or benefit to the implementation itself as possible. The @InjectMocks annotation solves this problem. When the injection is performed, Mockito looks for a private field that matches the type and name of the variable to inject the mock object (this can also be inserted via a constructor that takes the object as a parameter or setter method, though I opted to inject directly into the private field to avoid unnecessary refactoring). The first test, named testGetLatLon(), asserts whether the results returned by the methods getLatitudeFromPostcode and getLongitudeFromPostcode are as expected by the specification. An additional mock object that is a dummy version of Address is created within this test method and used as part of the functionality of the GeoCoder. The behaviour of these mock objects is defined through the when(objectName.methodName(params)).thenReturn(x) statement. In this way, the return values of the functions called and depended upon in the method under test are set by the programmer, and as such ensure that the test is deterministic. Once the dependencies of the test are set up, a number of assertions are tested through the assert(x) statements. This includes asserting that the values returned by the mock objects are as expected by their definition, as well as testing the return values of the getLatitudeFromPostcode and getLongitudeFromPostcode are as expected when different parameters are passed to them.

The second test, testGeoCoderExceptionThrown, asserts that an IOException is thrown successfully by the GeoCoder, and subsequently the method getLatitudeFromPostcode, when an invalid postcode is given as a parameter to getFromLocationName. The IOException is forced through the when.thenThrow notation. The test is instructed to pass when the exception is thrown with the (expected = IOException.class) notation.

Because of the substantial usage of and reliance upon other libraries in the code, as well as the complexity of the methods under test, it was difficult to find opportunities to isolate sections of code suitable for unit testing. It was therefore necessary to refactor parts of the codebase in order to make unit testing a smoother process. Additionally, there is no straightforward methodology for testing the usage of Google maps and map markers, which formed the main part of the application under test. They cannot simply be mocked through the Mockito framerwork, as GoogleMap is a final class, which presents a significant issue when trying to ensure sufficient test coverage. With further time and development, the integration tests and tests of the UI could be performed in order to improve the quality of the application.