Project Part 4

Team: Andrew Gordon, Tommy Hoffmann, Connor McGuinness, Daniel Zurawksi

Title: What to Wear Weather

1. Implemented Features/Requirements and Final Class Diagram

Features:

- Fetches user's location using GPS.
- Retrieves weather forecast based on user's location.
- Makes clothing suggestions based on user's gender and weather forecast.
- SQLite drive by extension of SQLiteOpenHelper class
- SQLite database file by using native android database
- Weather API GET request from forecast.io that receives environmental information
- User information – gender that influences your recommendation
- A recommender class that takes in profile information and produces a ClothingItem that can be shown to the user.

Requirements:

- **US-001**: As a user, I want to see daily clothes recommendations when opening the app.
- **FN-001**: On app startup, current GPS coordinates will be retrieved
- **FN-003**: App will use weather API to get weather at current GPS
- **FN-004**: Use a relational database (MYSQLlite) to hold user data (age, gender)
- **NF-001**: Coordinates will use the Android system's fastest "best guess"
- **NF-003**: App will respond quickly and precisely to user touch/interaction

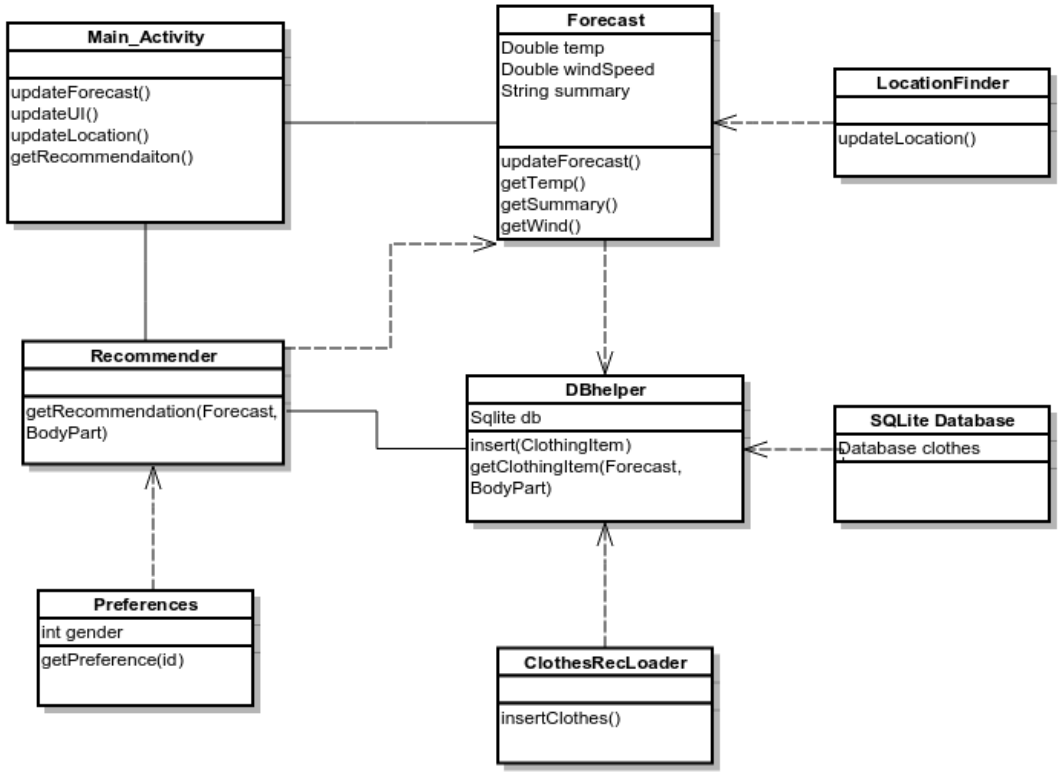

Figure 1. Final Class Diagram

The benefits of designing before coding were obvious here from several perspectives. First, from a team perspective, it was simple to delegate tasks to group members without worrying that they would write code that doesn't make sense or would need to be refactored as there was already a clear design in place. From a design perspective, we already knew exactly what member variables, functions, and classes were required, saving us a lot of time and mental energy. We also noticed that when your design for something is flawed, it is necessary to return to the drawing board and improvise a new solution. We had to do this in the case of our Database helper: we did not fully research how SQLite works on android, which resulted in our original class diagram looking slightly different from our final diagram as we created a solution for DB queries by extension of a preexisting class.

2. Design Patterns

One design pattern that was not utilized but could have been is **Observer**. Most methods and classes in our system talk only directly to each other instead of together. Observer would have been used to update information stored in the database, change the suggested clothing options, and change the forecast day and summary, all by the click of one button. Objects and methods would notify each other when one was changed, thus reducing repeated code and methods. In our system, methods only update each other when specifically called upon. Therefore, the database could be out of sync with a changing forecast. Fortunately, we avoided synchronous problems by making sure that classes and objects are explicitly updated when they need to be. Our system is almost closer to a related pattern **Chain of Command**. Our system begins with the Android MainActivity and makes a call down the line of different components that each are responsible for computing results based on components higher up in the chain. To truly implement this design patter, however, we would need to add a system in which the Chain could be extended and manipulated.

Instead of **Observer**, the group decided to implement **Singleton** in our system. One of the main uses of Singleton was with the DBHelper, a class that is used to intermediate between the Recommender and the SQLite database. This class is generated one time when the Android application loads, and is passed from the MainActivity to the Recommender each time a request is made. This ensures that there will only be one class ever attempting to access the database. Additionally, since our application is only running on the Android operating system, there are only ever single instances of system objects. Therefore, all of our methods access the system objects during runtime and do not need to account for third party systems. This allowed us to focus on implementation exclusively for Android OS. Without having to worry about multiple platforms, the group was able to write more robust code for the system. There wasn't a need for base functionality across platforms, so the system's code remains organized and clear.

3. Comparison of Part 2 Class Diagram and Final Class Diagram

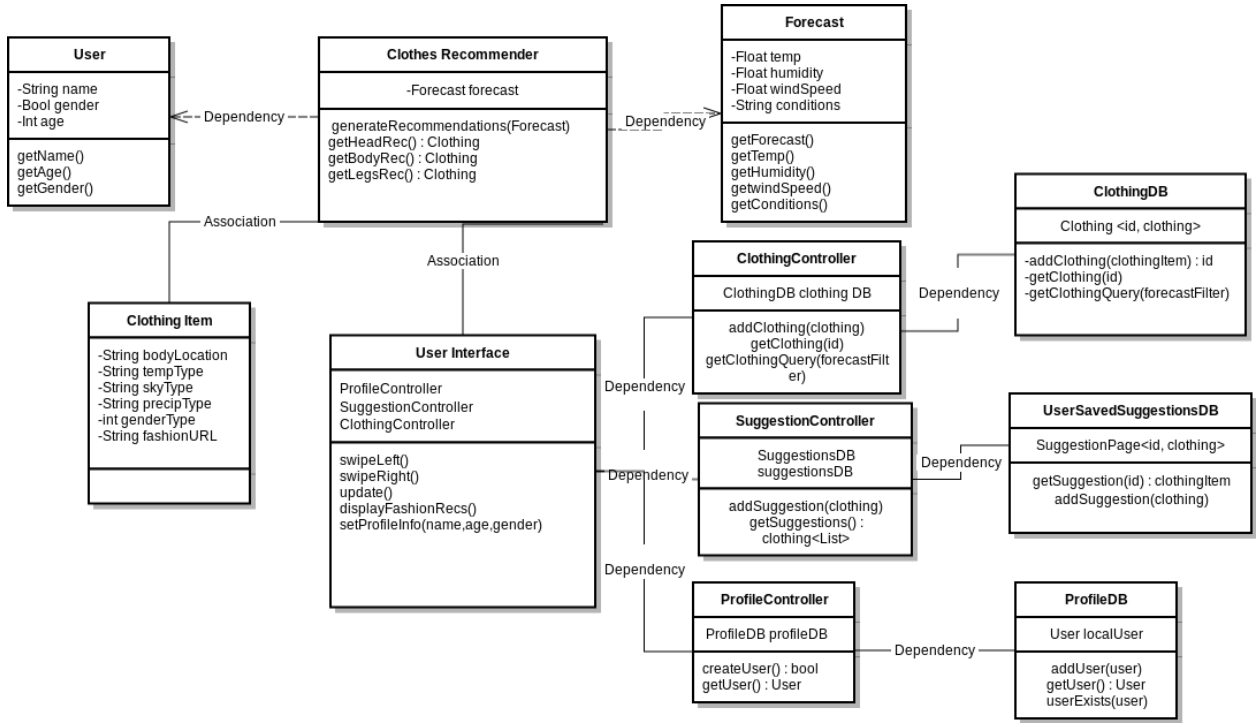

Figure 2. Class diagram from Project Part 2

The primary difference in our class diagram was the implementation of the DatabaseHandler. This was largely due to our lack of discussion on how android handles data persistence. SQLite is native to Android, and there is an implementation of many SQLite functions provided by external libraries. Other differences were the addition of the DB in the class diagram, and the lack of user login.

The class D BHelper extends S QLiteOpenHelper, an class that includes the implementation of several functions native to Android SQLite. With this, we were able to include several functions to manage all the database queries we needed. These were primarily to retrieve a clothing item based on a weather forecast. The DB query inputs parameters based on JSON data from a weather API, and return appropriate Clothing Items. ClothingItem objects were persisted by having each private variable such as temperature or precipitation stored in columns of the database, and instantiating ClothingItem objects from the rows returned from database queries. Our original class diagram simply had a Clothing Controller, which had a function getClothingQuery(id). While this makes sense, there are many SQL related functions that need to be implemented in order to actually make this work. This was achieved by simply extending the SQL helper class.

The final design of the project did not include a user login field or user database. This was because we only ran the project on a single Android device, and this is not where the users database would be located. User verification would be handled on an external server, not on mobile devices, and would be handled using Get or Post requests to see if users existed or needed to be written to Database. For a potential improvement on our final design, we would

like to implement the observer pattern as there are many updates that need to take place once the weather API receives information about the environment, a date changes and the UI needs to be updated, or the database performs its standard query. This would streamline our UI updates greatly.

4. Lessons Learned About Analysis and Design

The process of analysis and design is not something to be rushed through. Although we diligently prepared for implementation, we did not account for timing well. During the design phase of our system, we felt that the aspects we thought would be easy to implement, proved to be difficult.

One such aspect is requesting/sending information to and from the database. The system relied heavily on this functionality yet it was not very clearly defined in our design phase. We tied it into multiple features like saving gender preferences and storing clothing suggestions. As a group we simply thought that database manipulation would be easier than expected to implement. Since we had to spend more time on this functionality, other features had to be delayed. Therefore, one lesson learned is to plan the timing of implementing features and to expect errors. Without accounting for errors, the implementing phase will take longer to finish. By expecting errors during the design phase, implementation will be easier to handle because the group has already figured out what error is likely causing problems.

Another important lesson the group learned about analysis and design is project consistency. By defining key components like SDK's, API's, platforms, coding languages, and databases early in the design phase, it helps to stick to these choices through implementation. A small example can be seen in our UI Mockups, that were created using Xcode. Although, these mockups did provide a good reference for how our application should look, it was not consistent with how Android applications look. Since we had defined it would be an Android application, creating UI Mockups with Android Studio would have been a more consistent choice. By keeping project consistency, implementation is more straightforward and organized. Fortunately, the group remained consistent on most of the design choices during implementation. Therefore, it was easier to stay focused on specific errors and remain organized even when working on different parts of the system at the same time.

Overall, planning ahead for errors and maintaining project consistency throughout all phases of creating the system are important lessons learned by the group. By not accounting for the amount of time debugging a specific error would take, the group had to "brute force" the error until the application would work. Also, by not keeping certain aspects of the design phase consistent with the implementation phase, the group had to create new parts of the project different from the original plan. However, these fallbacks were pushed through in order to create a functional system. As a group, we are going to remember these design pitfalls when working on a new project. Analysis and design is a learned skill that is best developed from experience.