weCharade

Architectural Specification

The weCharade application consist of a main application, run locally on any android device. The goal of this game is to play charades and compete with friends. The charades are recorded as video files and sent to the opponent.

The social nature of the game warrants the use of an online database in order to connect different users to each other. We will specify the schema of this database in a later part of this document. We have chosen to use Parse.com as a database provider. The choice of not setting up a server by ourselves has had the effect that we, as a development team, have been able to focus on the internal structure and features of our game when run natively, which has had an overall positive influence of the final quality of the delivered product. Parse.com also provide a great range of APIs for communication from different platforms, such as iOs, Android, Windows Phone and more. This makes it possible to develop versions of this application to other platforms in the future without having to rewrite the underlying data structure. The use of Parse.com's services has, in short, provided our team with a stable and extensible service; not developing and maintaining our own servers have also made it possible to have a working and releasable application at the end of this project.

Because of the inclusion of video as an element of the application, we have had to make some arrangements in order to transfer these files between different connected users. Currently, we utilize a simple web server for storage, which we communicate with by the use of FTP. We are aware of the complications which might arise form an increase in user count, and will probably have to deploy a different form of storage as the application gets a large user base. This initial solution does, however, provide us with a place to start, and will be a sufficient solution in the initial stage.

We have chosen to use a MVP pattern in the design of our application. In our definition, we made the choice to consider our Android Activities as our view-classes. These classes provide the canvas for displaying content, while interpreting user input and transferring this on to other classes. Each view is associated with a presenter, which further specify how the actions of a user should be controlled. These classes also transform data to formats that can be displayed on the views. Finally, we have the model classes, which contain the data structure of the application. The model is stored both locally and in the online database, which has put requirements on a fairly extensive method of synchronizations in order to keep the two sets of data compliant with each other, as well as reduce the risk of corrupted data due to incomplete transfers.

Our goal when constructing the application has been to minimize the requirements of requests to the database and place as much of the work needed for the application to work in threads running in the background. Because of the nature of mobile phones, which have limited capabilities in terms of performance, as well as the value of data traffic, we felt that this focus was a must in order to create a better user experience. At the start of the development cycle, we implemented much of the work on the main thread, running alongside the UI. This approach soon felt too cumbersome and unresponsive, whereby we made a redesign towards a more concurrent system of presentation, leading to a more satisfactory user experience which feel much more responsive.

Database Schema

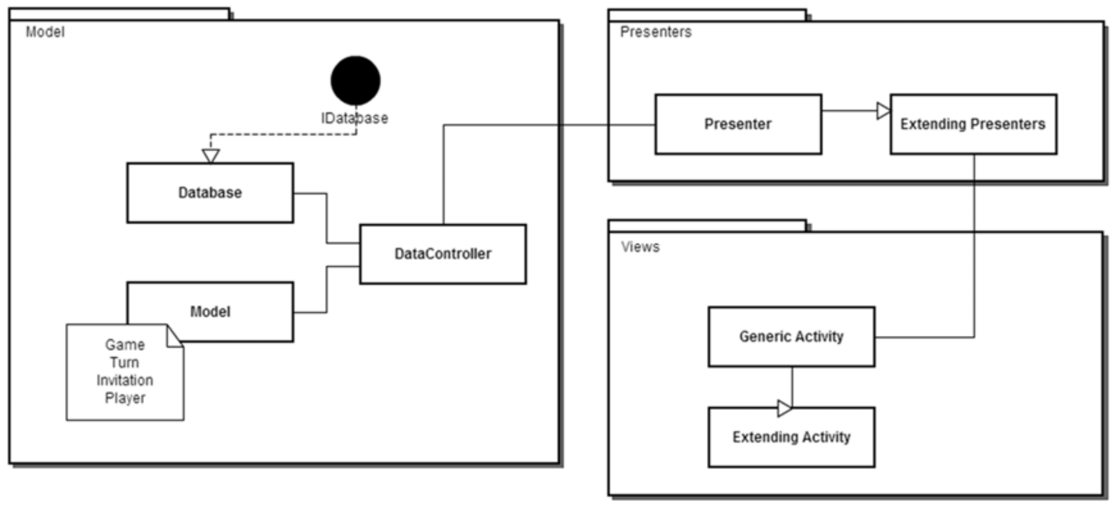
The following database schema has been used in order to store relevant data in the database. Because of how Parse.com is constructed, it is easy to change and extend these classes in the future.

```
Installations(objectId, appName, appVersion, badge, channels, deviceToken,
deviceType, installationId, parseVersion, timeZone, createdAt, updatedAt, ACL)

User(_objectId, password, authData, emailVerified, email,  gamesDraw, gamesList,
gamesPlayed, gamesWon, naturalUsername, createdAt, updatedAt, ACL)

RandomQueue(_objectId, player, createdAt, updatedAt, ACL)
    • player -> User.objectId

Game(_objectId, currentPlayer, finished, player1, player2, turn, createdAt,
updatedAt, ACL)
currentPlayer -> User.objectId
    • player1 -> User.objectId
    • player2 -> User.objectId

Turn(_objectId, ansPlayer, ansPlayerScore, game, recPlayer, recPlayerScore, state,
turn, videoLink, word, createdAt, updatedAt, ACL)
    • ansPlayer -> User.objectId
    • recPlayer -> User.objectId
    • game -> Game.objectId
    • word -> wordList.word

WordList(_obejctId, word, createdAt, updatedAt, ACL)
```

UML Diagram

Bellow, a condensed UML diagram of the general structure of our application. Most classes are not included in this diagram, but it does non-the-less give an overview of about the general design principles.

Each of the model classes, Database, Model and DataController are using the singleton design



principle. This makes these parts of the application persistent throughout the lifetime of the application, which is beneficial in terms of data control. We have also implemented an observer design pattern between the database, DataController and the presenters. By using this pattern, we can use background threads to fetch data and send an appropriate message when fetching is complete.

Future

We have worked towards maintaining good coding practices in order to create a maintainable and extensible code base for further development and operation of the application. We have plans to further develop this application and add more functionality after hand in order to further improve the user experience. We also have the goal to port the application to other mobile platforms in the future.