Programming design and object-oriented development paradigms of an Android-based distributed social game system

Sebastian Sbîrnă, Liana Simona Sbîrnă

***Abstract*— The present work demonstrates the mechanisms, programming design choices and paradigms of developing an Android-backed online turn-based social game application. Each step of the programming process is detailed, and the underlying implemented concepts and reasonings for the different modules are explained using the Model – View – Controller architectural pattern. Software engineering design choices are compared and exemplified at each stage of the development phases, giving suggestions for improving the logic behind such a distributed service. This paper also shortly presents insights on the usage of an online cloud database (Firebase, from Google) in order to handle player data in real-time using JSON format.**

Keywords — object-oriented programming, Android, Java, MVC, development paradigms, distributed systems, social game

I. Introduction

In today's world, technology is playing a leading role in people's lives. Humans are now used to have synchronous verbal connections between people living at a far distance from each other (sometimes on two opposite sides of the globe), and, with the use of the internet, people can also text or see each other, which gives a possibility to make them feel like they would be together.

Despite these incredible opportunities, many contacts are still weakening or even being lost. In an ideal world, the next step would be that people do not just keep in touch through long conversations, but even through activities that requires only a few minutes of one's precious time.

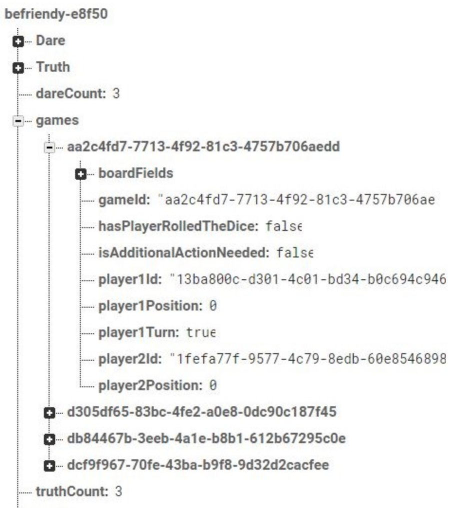
The idea behind our thinking is that "coldened relationships" could be well mitigated with the use of technologies that are most of the time with their users and providing fast user-device interaction [4]. The most operated tools for such purposes are smartphones [5].

The goal of this research paper is to present the ideas and findings behind creating a game that works as an element of interaction between people not being in physical proximity; and a social game which is meant to be an enhancer of human-to-human communications, not a replacer of it.

II. Product description

Our application is meant to combine a board game (e.g. Snakes and Ladders) with socializing games (e.g. "Truth or Dare", "I have never..."), where there would be two players for each game, playing on separate devices and exchanging pictures, videos, and information between each other, as they advance on the board. For that, we employ a server communicating with each of the clients, which is capable of storing and exchanging data whenever necessary.

In order to keep all the information about the users and their games, we have saved them in an online database that would allow retrieving of data in real time, in order to have two users playing a game without big delays in data exchange. There are a lot of possible solutions for such a database, but we decided to use Firebase from Google [1], as it is a cloud based solution with an online platform, which would allow multiple users to connect at the same time, to save or retrieve information [1]. As our application also needs to store photos, the integration of the online database with "Firebase Storage" is a suitable solution for our implementation. A snapshot of our database structure is shown in figure 1.



```
befriendy-e8f50
  Dare
  Truth
  dareCount: 3
  games
    aa2c4fd7-7713-4f92-81c3-4757b706aedd
      boardFields
      gameId: "aa2c4fd7-7713-4f92-81c3-4757b706ae
      hasPlayerRolledTheDice: false
      isAdditionalActionNeeded: false
      player1Id: "13ba800c-d301-4c01-bd34-b0c694c946
      player1Position: 0
      player1Turn: true
      player2Id: "1fefa77f-9577-4c79-8edb-60e8546898
      player2Position: 0
    d305df65-83bc-4fe2-a0e8-0dc90c187f45
    db84467b-3eeb-4a1e-b8b1-612b67295c0e
    dcf9f967-70fe-43ba-b9f8-9d32d2cacfee
  truthCount: 3
```

Figure 1. A snapshot of the Firebase Database web interface, showing the structure and value of variables in a JSON key-value format

III. Feature implementation

This chapter aims to provide a detailed look on the core Android-compatible ideas and methods used within the implementation of the application.

The chapter will be divided into two main parts, corresponding to the two major development stages: the "general features" implementation phase and the "gameplay and testing" implementation phase. The main differences between them lay in the focus of the programming tasks.

Manuscript received June 3, 2018.

S. Sbîrnă is with the Technical Faculty of IT and Design (student at Innovative Communication Technologies and Entrepreneurship - "ICTE"), Aalborg University Copenhagen, 15 A.C. Meyers Vænge Street, Denmark.

L. S. Sbîrnă is with the Faculty of Sciences (Chemistry Department), University of Craiova, 107I Calea Bucuresti Street, Craiova.

A. "General Features" Development Phase

In the following paragraphs, a guided insight on the execution flow of the game will be presented, so that the readers of this paper can have a clear idea about what we are trying to achieve as coding approach. The application's logic flow is supposed to behave in this manner:

The first time a user is starting the application or if the application is deleted from the memory and needs to be restarted, the user is greeted with a login screen, in which one can input its email and password in order to connect online to our database and retrieve all of its data. On the screen, there is one "login", one "create account" and one "forgot password" button. If the user does not yet have an account, he/she can press "create account" to register, case in which he/she will be asked to enter a desired username, a valid email, and a password. The application then goes on and generates a random unique UUID for every newly-created player, to have its own space inside the database.

The user is then redirected to the main screen of the application, also called the "game list" screen. Here, the user can see all the current games that he is playing in, in a detailed and orderly manner. One can always know in which games his turn has arrived by looking at the text on the button next to each game: if it appears as "PLAY", then it is his turn; while if it appears as "VIEW", it is not his turn and he doesn't need to take any action yet. Regardless of the state of the button, if the user clicks it, it will enter the game that he has clicked on, and he can either perform certain actions such as: roll a dice, input a question, etc., or he can see the state of the board (shown in fig. 2): where each player is positioned, the current field card that is in play, and what information is until now inside the card. We will discuss field cards more in the next paragraphs.

There are five types of field cards that have been defined for a game: Truth, Dare, Minigame, Gameplay, and Wildcard. Whenever a player rolls a dice and lands on a field on the board, a Dialog pop-up appears in the form of a "card". Depending on the field, a different card layout will be presented. All the field types allow a player to postpone its action, however the game will be put to a standstill until the player decides to take action. When describing the possible actions on these cards, we will refer to the player whose turn it is as the "current player", and to the player that is not having its turn as the "other player":

- A ***Truth*** card shows to the current player a question taken from the database and allows him to answer to the question or postpone it until a later date. The other player will also see the current question, and the screen will show "waiting for answer", if the current player has not yet answered the question.
- A ***Dare*** card shows to the current player a dare taken from the database, and gives the current player a possibility to send a photo of him/her doing the dare. The photo is saved into the database only until the dare is marked as completed, which makes sure that the users have seen the media.
- A ***Gameplay*** card gives the current player the possibility to roll a 6-sided-dice and a special game boon is activated depending on what the player has rolled, such as advancing fields or switching positions with the opponent. A text appears on the Dialog activity telling the player a description of the special effect that has been activated automatically.
- Each ***Minigame*** that our game offers needs to implement a different layout, due to the differences between different minigames. To give an example, for a "Rock-Paper-Scissors" minigame, both players have three buttons on the screen, corresponding to the three available choices: rock, paper, and scissors. After both players make their decision, the screen shows a text specifying what each has selected, along with if that player has won/lost. The winning player gets different in-game boons, depending on the minigame played.
- A ***Wildcard*** card allows the current player to type in a question that he/she wants answered, along with the type of answer that is wanted: text or photo. In principle, the implementation is similar to the Truth and Dare cards combined together, with the difference that, instead of having a question selected from our database, the player itself writes the question.

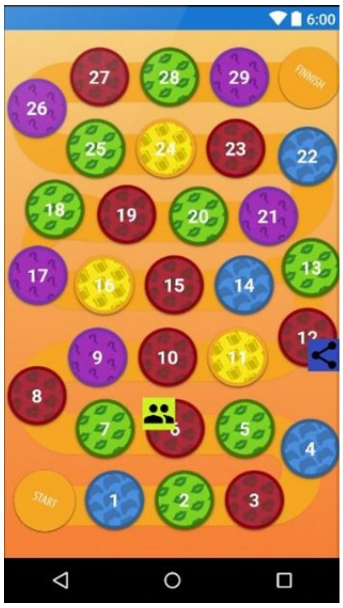


Figure 2. A typical screen of a gameplay session between two players, showing the players' positions and tokens; along with the different fields that are available: Green – Truth; Red – Dare; Blue – Minigame; Yellow – Gameplay; Purple – Wildcard;

The implementation of the gameplay inside the application has a turn-based approach and depends very much on four variables that hold the state of a game within the database: "isPlayer1Turn", "hasPlayerRolledTheDice", "isPlayer1AdditionalActionRequired", along with "isPlayer2AdditionalActionRequired". It is worth mentioning that the turn-based flow of the gameplay has major turning points, which affect what type of content is shown on the screen.

An activity diagram, presented in figure 3, was drawn up to show the various activities involved in a single game turn of our application. For the user, the process of playing a turn follows a straightforward way of progression.

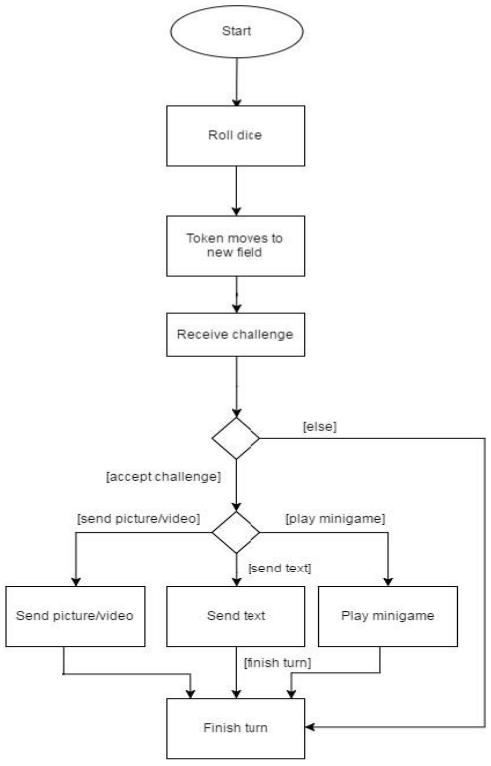


Figure 3. Architecture diagram of the system, based on the Model-View-Controller structuring method

- ***If it is the device owner's turn***, then:
  - ***If the player has not rolled the dice yet***, then the current player is at the beginning of the turn, needing to advance on the board. In that case, a semi-transparent window would appear over the board, having the text "Roll" written in the center of the screen. When the user taps that, a pseudo-random number generator inside the code will generate a pick from 1 to 6, and the result would be shown on the screen in the form of an image with a dice face having as many dots on it as the number that was rolled. If the user taps again on the screen, the player's token will advance on the board to the new position, after which a new corresponding field card will be in play.
  - ***If the player has already rolled the dice***, the current player has already advanced on the board. In that case, the user is at the point where he needs to give some input to the current field card (by answering a question, typing a sentence, pressing a button, etc.), but has not yet done so.
- ***If it is the other player's turn***, then the actions that the current player can take is visualizing the board, along with visualizing the current field card that is in play, what information is so far written "inside" it, and perhaps performing some additional action on the card that is already in play, only if requested.

The gameplay continues in this manner until one of the players arrives at the Finish field, case in which both players receive a pop-up Dialog showing, respectively, "You won, congratulations!" or "You lost this time, but thank you for playing". After both players close this pop-up, the game is removed both from the database and from the two players' game list. Any two players are only allowed to have one single game between themselves in the database at a time.

Continuing, we will describe the components which compose the applications' code, with respect to the MVC (Model-View-Controller) architecture, based on which most Android applications are designed [2]. The architecture diagram in fig. 4 reveals the overall organization and structuring of our system.

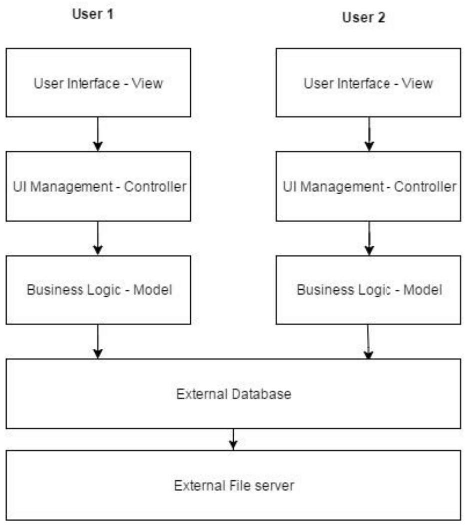


Figure 4. Architecture diagram of the system, based on the Model-View-Controller structuring method

A Model holds the data and "business logic" of the application; View objects construct the User Interfaces of the application; while Controller objects contain the "application logic" and link the view and the model objects together. Controllers are made to respond to various events triggered by the user interfaces and to control the flow of data between model objects and view objects. In that sense, "in Android, a controller class is typically a subclass (extension) of the classes: Activity, Fragment, or Service" [3] Since this is a service-based mobile application (making use of the Firebase database cloud solution), the database has been modeled and deployed on the provider side, making use of the higher storage capacities available; while the client side uses algorithms to process specific data locally and, afterwards, update the data to the database accordingly [7].

1) Model object description

The application development started with implementing model classes for our objects inside the application. These objects formed the basis of our object-oriented programming approach. Some examples are: Board, Truth, Player, Field, etc.

To give an example, the Player class has been designed using the singleton development pattern, meaning that there can only be one instantiated Player for the duration of the whole lifecycle of the application. This is to make sure that player-personal data are globally accessible from any part of the program, but also to prevent having multiple players from being logged on at the same time on the same device. This influenced our choices of interacting with the data of the other players from the database, since, instead of locally storing all the data about another particular player with whom we are playing a game, we are using the database to extract the necessary information and, if needed, to check for updates of its content.

Here we will present a short description of the most important models developed for our application:

- ***Player***: used to keep all the necessary user's data available throughout the whole application. This includes the user UUID, its first name, last name, email address, game list and game list IDs.
- ***Game***: class responsible for the Game data initialization and for the instantiation and initialization of the Board-specific object for each Game. Here we set which positions inside the board are going to be of type Truth, Dare, Minigame, etc. Right now, there is only one possibility for the board layout, since it has been hardcoded which type of fields correspond at which positions on the board. However, a further development would be to have a random field type generator for the board layout, so that every time there would be a different distribution of the field types across the game board.
- ***Board***: keeps track of which fields are present inside a game-specific board, and what position contains what field type.
- ***Field***: abstract class, which is the parent of all the five field types: Truth, Dare, Minigame, Gameplay and Wildcard. It defines the signature for two important methods that are needed inside each field type: getType() and getDrawableId(). getType() returns an ID that is unique to a specific field type, so as to identify that field by the specific ID (for example, a Truth Field is of has the ID of 1, while a Dare has the ID of 2). getDrawableId() stores, based on each child class of Field, the ID from the "R.java" class that links it to its board field image from the "drawable" folder, which will be shown on the screen.

2) View objects and layouts description

The View objects inside an Android application are intrinsically connected to certain layouts that define the user interface of that application [6]. All our layouts are constructed using the XML language that Android is working with [6]. Some of the layouts in our game, already available at the end of the "general features" phase, are:

- ***activity_login*** is the first screen in our user interface, where the user will input its email address and password in order to login to the game database. In case the user does not have an account, he/she also has the option to register to the game by press the button "Create Account", which sends the user to another screen, namely: activity_registration.
- ***activity_registration*** is the screen that users will see when attempting to register in our game. The user needs to input a desired username, email and password, after which he/she will automatically be taken to the game list screen.
- ***activity_game_list*** and ***game_list_recycler_view*** form the main game screen, which consists of a list of all the games that the device owner is currently playing in. Each game inside the list has, as identifiers: the username of the other player that is engaged in the game; a square image with the username's initial centered on it, and filled with a color depending on that initial; and a button that reads either "Play" or "View", depending on whose turn it is inside the game. A floating "+" button provides a way to create a new game, after which the player is taken to a screen which requests inputting either the username or the email address of the friend that will join the game.
- ***board_layout*** is the layout that defines how the game board looks like for all the games of the player. It contains 31 fields (including "Start" and "Finish"), uses different colors and image resources for each of the possible fields, and allows for the player-chosen tokens to be represented on it as part of the gameplay. The device owner's token would always be positioned on the top-left quarter of its particular field on the device owner's screen, while the competing player would be always positioned on the bottom-right quarter of its own field.
- ***navigation_drawer_header*** and ***drawer_menu*** are the parts that make possible to have a hamburger menu in all screens of the application. This menu would allow for quick access to the game settings, game list, friend list, achievement list and game history of the player. The header of the drawer menu would show the full name of the player and an image at choice.

3) Controller object description

Control objects inside our application are the different activities that provide a mechanism to put together data into the views of the application, and, subsequently, populate the user interface with information meaningful to the user.

In most cases, each of these activities is designated to "draw up" a new "screen" or "part of the screen" inside the application. Due to the complexity of the tasks that each activity needs to accomplish, we will describe most of the gameplay activities in the second half of this chapter, when we reach the second phase of development. We will provide here a description of some of our activities already-implemented at the end of phase one, in the order that they show up on the user interface in a general use-case scenario:

- ***LoginActivity*** is the activity bound to the first layout that appears on the screen, activity_login. It instantiates the database objects inside the memory; defines listeners for the various buttons; provides field validation for the email and password fields; validation for internet connection; and starts the service to retrieve information from the database. The activity then checks inside the database if the user already exists, and, if so, redirects him/her to the game list, where his/her personal games information would be further retrieved from our database and displayed on the device screen.

- ***RegistrationActivity*** is launched in case the user decides to register into the system. In that case, the activity provides field validation for the four fields that the user needs to input, and provides a way to save the new user inside the database, if it doesn't already exist. Afterwards, the user is redirected to the application's login screen.
- ***GameListActivity*** defines the current list of games that the user is playing, and provides a way to create a new game via the "+" floating button in the top right corner of the screen. From a technical perspective, this activity only provides the layout in which the game information cards would be put in place, while the next presented activity provides the actual data retrieval and screen update functionality.
- ***GameListRecyclerViewAdapter*** is the activity responsible of populating the game list layout with information. This Activity uses a CardView to show the different games in a card format, giving memory benefits by not reconstructing any more data than the screen can show, unless the user scrolls in search for it; and, also, a custom ViewHolder arrangement.
- ***GameActivity*** is the activity that inflates the board layout on the screen, sets up each of the field views with the corresponding image resource (depending on what field type it is), and handles most of the board update and turn coordination. More on this activity will be detailed in the next subsection of this chapter.
- ***RetrieveInfoFromDatabase***, is a service that is running in the background all the time even if the application is not opened, so it can retrieve data from the database. In particular, it listens for changes in the player's game, so it can send a notification that it is his/her turn.

B. "Gameplay and Testing" Development Phase

After the foundations of our application were implemented, along with establishing a main outline about how the gameplay of the application would be internally represented, we set out to develop the features that turn our application into a "game", namely: the different fields, their behaviors, and the flow of gameplay.

The first step that was taken in this direction has been to create a framework that allows creation and restoration of the unique state of a game using information received from the database, so as for a user to be able to "enter" a specific game with a person and see all the latest changes, while also saving the changes he/she makes inside his/her turn. This was done by using Bundle objects, which consist of key-value mappings that can be transferred from one activity to another by sending them together with an Intent. Every time a game is started, the program checks against some key values that, if present, would return essential information about the reconstruction of the game, and would be saved accordingly inside some local variables.

The GameActivity is the "main" class of a game, here being present the core functions that restore, update the board and establish a turn-based flow inside a game. The method play() inside this class deals with the testing of certain conditions, in order to make the turn-based approach possible. It allows to specify which field is being played right now, appends a visual indicator stating this, checks for dice roll necessity, and, if there is a played field at that moment and it is clicked, allows the application to select the right "Field" type object to be instantiated, out of all the possibilities. All the further gameplay, relating to how a player interacts with the different field types and with their content, is done separately for each Field type, inside their own classes.

After that, we went on to implement each separate field individually, in a logical order: Truth; Dare; Wildcard; Minigame; respectively, Gameplay.

The classes have been chosen to be Fragments, more specifically: extensions of the DialogFragment class, due to the properties that Fragments have upon the application layout. Since a Fragment is attached to an Activity, and becomes a part of that Activity, transitions between different screen layouts is much easier from Activity to Fragment than from Activity to Activity. Also, due to the possibility of listening for the result of other activities that have been called not just by the main activity, but also by any of its attached fragments, this allowed for a way to implement a global function that would deal with Camera return type intents, and what to do after a picture has been taken with the camera and should be uploaded to the storage solution.

The onCreateDialog() method in each of the DialogFragments is where the gameplay mechanics and checks are hidden. If "is[...]AdditionalActionRequired" is true, then it means that the field card that is currently in play needs additional input from the current player. An example could be in the execution of a Wildcard field: The player who received the card would type in a question, after which turn ownership would be switched to the other player, so that he can input an answer. In that case, we can think that "additional action is required for the card before the turn can end", which is why "is[...]AdditionalActionRequired" will be true. This prevents checking for e.g. if the player is at the beginning of a new turn.

"is[...]AdditionalActionRequired" will also be true before the removal of a card from play, when it is needed for one of the players to confirm that he/she has seen the contents from the other. This is our method of making sure internally that both players have seen what has happened during the last moves of the opposite player, and such an approach was made possible through turn-based integration. Therefore, "confirming" a card, or, more precisely, "confirming" that you have seen what the other has done, is usually the last exchange of information within a Field type card, after which it disappears and a new one is played.

Our gameplay is built around fewer, major “turn changes”, which can be known as “player turns”, and many smaller exchanges of information at a Field card’s level, involving sending data back-and-forth to each other’s player, which can be considered as a “player pseudoturns”. An example will be given in order to better illustrate the concept: when both players have just finished exchanging information on a card, then it has been confirmed by both of them, and afterwards has disappeared from the screen (meaning that the placeholder variables in the database have been reset to values neutral to the game), it will be then clear that another “player turn” has begun. A player turn is also generally characterized by being the only time where one can roll a dice in order to advance on the game. However, after a turn has started, there can be many “pseudoturns” from one player to the other, depending on how the specific card gameplay requires. For example, if a Truth field has been rolled, during the first pseudoturn, the user receiving it will answer to the question that is present on the screen, after which, if he clicks “Submit” on the card layout, the database information is updated and the pseudoturn changes to the other player, where he/she will confirm that they have seen the written answer to the given question, while the actual “player turn” doesn’t change. In this respect, inside the code and the database values, one can think of “player turn” as “who of the players received this card first”; while, for pseudoturns, one can speak about “who of the players is requested action to be taken now”. Please note that this is all spoken purely from a coding point-of-view, as it is of no concern to the actual players themselves, since such a distinction is never mentioned during the actual gameplay.

In order to properly adjust to the diverse behavior that each of the Field card types was bringing into the game (especially considering that each of them has different rules as to when a pseudoturn is changed and what happens during each pseudoturn), we created multiple layouts for every DialogFragment, each representing how the contents on the screen should look like, considering a sequence of data checks that marks a change in pseudoturns.

These layouts have been divided into “active” frames, and “passive” frames, depending on the type of action requested on the screen. An “active” frame is defined by having at least one element that will change the state of the game when activated (for example, having an input text field and a pressing a Submit button that will send the data from the input field to the database); while a “passive” frame is defined as having no state-changeable elements and is being shown on the screen only if that player is not requested any action. A case where an “active” frame would be shown would be when a user wants to answer the question inside a Truth field, while a case of “passive” frame usage is when a player is waiting for the other one to send a photo inside a Dare field, but would like to see again the dare text that his friend has received, in order to remember it better. After each turn or pseudoturn of a game, the information for the game must be saved in the database, which then fires an event in the background service that is responsible for retrieving game information for each user.

The number of “active” and “passive” frames that a Field card is required to have depends especially on the nature of the gameplay inside it, on the recyclability of the layout in the context of gameplay, and on the number of pseudoturns that the card is expected to have before it “disappears” from the field.

IV. Conclusion

THE paper has presented many of the programming paradigms behind the implementation of an Android application/game created to make people get to know each other better and share parts of their lives, regardless of the physical distance between them.

The general development was separated in two phases, each with its own specific focus. The underlying code followed the MVC architecture [2], as a way of better organizing and separating tasks between the different classes and layouts.

In the “general features” development phase, we have presented ideas behind the design of core classes that make up the login and registering into the game, listing initiated games, playing and viewing the different game elements on the screen.

In the “gameplay and testing” development phase, we have presented in-depth designs upon the gameplay elements that enable our application to be played in a turn-based manner.

Anyone who would follow a similar implementation pattern, or who is interested in building an application with a similar purpose can benefit from reading the findings of our project, which concluded successfully with a working Android social online game.

V. References

[1] Firebase; App success made simple, 2016. Available at: https://firebase.google.com [Accessed: 05 March 2018].

[2] T. Cornez, R. Cornez, Android Programming Concepts, First edition, Jones & Bartlett Publishers, 2015.

[3] B. Phillips, C. Stewart, B. Hardy and K. Marsicano, Android Programming, The Big Nerd Ranch Guide, Second edition, Pearson, 2015.

[4] S. Sbirna, H. B. Bakalov, K. R. Babos and F. J. Eriksen, ITCOM 15 – Group 2, P3 project, Semester Project, Aalborg University, 2016.

[5] H. Falaki, R. Mahajan, S. Kandula, D. Lymberopoulos, R. Govindan and D. Estrin, Diversity in smartphone usage, in Proceedings of the 8th international conference on Mobile systems, applications, and services: ACM. pp. 179-194, 2010.

[6] Android; Layouts, 2017. Available at: https://developer.android.com/guide/topics/ui/declaring-layout.html [Accessed: 03 March 2018].

[7] H. J. La and S. D. Kim, Balanced MVC Architecture for Developing Service-Based Mobile Applications, 2010 IEEE 7th International Conference on E-Business Engineering, Shanghai, pp. 292-299, 2010.