Università di Bologna

Campus di Cesena

Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

Progetto finale del corso di
Sviluppo di Sistemi Software (S3)
a.a. 2016/2017

Repository:
https://github.com/simoneapp/S3-16-simone

Michele Sapignoli
matr. 0000819208 michele.sapignoli3@studio.unibo.it
Giacomo Zanotti
matr. 0000793982 giacomo.zanotti2@studio.unibo.it
Giacomo Bartoli
matr. 0000795161 giacomo.bartoli3@studio.unibo.it
Nicola Giancecchi
matr. 0900057545 nicola.giancecchi@studio.unibo.it

▸ 01
Development process adopted

According to a scrum-based approach, we developed Simone splitting the entire development process in 4 sprint, each one of 25 hours.
Giacomo Bartoli was elected as Product Owner, who is responsible for the project management.
At the beginning we had a sprint planning where all the team decided the goal of a specific sprint.
The product backlog table was regularly updated and, at the end of each sprint, we wrote a sprint backlog too. The sprint backlog contains information about a specific sprint, while the product backlog keeps track of each user story that was later deepened by creating many items in Trello, (https://trello.com), a taskboard for agile teams.
Trello Taskboard: https://trello.com/b/LCQgiZ4j/simone-app
Moreover, we had a weekly meeting to sum up the issues faced during the development.
All the documentation generated by the scrum process was tracked as the entire project: https://github.com/simoneapp/S3-16-simone/tree/develop/process

Simone app was developed following a standard GitHub workflow: a central repository was created (https://github.com/simoneapp/S3-16-simone) and each member of the team forked a copy into its own repository. After each local feature was finally developed, completed and tested, it was later integrated by creating a pull request. Following this, the central repository contains the latest version of Simone app.

A list of tools, languages, frameworks we used:

- *git* as VCS.
- *Android Studio* as IDE.
- *Kotlin* and *Java* as programming languages.
- *Gradle* as dependency automation tool.
- *Firebase* as cloud solution and central database.
- *Firebase Cloud Functions* as server side elaboration of data.
- *GitHub* as repository (handling commits, branches and pull requests).
- *Travis* as CI tool.
- *Fabric* as beta distribution tool.
- *Crashlytics by Fabric* and *Firebase Crash Reports* for crash reporting.
- *Trello* as agile scrumboard.
- *Slack* as teamwork chat.

▸ 02
Requirements

Original idea

The basic idea of this project was to develop an Android app replicating the game **"Simon"** (aka *Simon Says*), an electronic game invented by Ralph H. Baer and Howard J. Morrison in 1978 very popular during the 80s, mostly in the US.
The electronic game was produced by *Hasbro* and is composed by four buttons - blue, red, green and yellow. The game generates a sequence of tones and lights that should be repeated correctly by the player by pressing the corresponding colored buttons. The sequence becomes longer and more complex as the game continues.
There are unofficial apps currently available on the App Store and Play Store that emulates the look-and-feel of Simon, but there's not a well designed and more complex version of the game with multiplayer as a key feature.
We decided to start from Simon basics and to build an Android app with new interesting features.

Figure 1: Ralph H. Baer with Simon

Types of requirements

Business requirements

Build an app that mimic the game "Simon", plus adding harder levels and new features to play with friends who have installed Simone.

To be more specific, a general user should be able to play Simone in single mode, both classic and hard mode. The same user should be able to play a multiplayer game of Simone using its Facebook credentials. The multiplayer part consists of two possibility:

- Instant play: play a Simone match and send the same sequence to another friend.
- Nearby play: four players join the same game and try to replicate Simon's sequence of color, but each of them has a unique color.

Additionally, the user should be able to enable/disable music and push notification and, obviously, see the scoreboard of its own matches.

Functional requirements

- **Player vs. CPU**
  - *Classic level*
    - player should repeat the sequence generated by the system
    - sequence is displayed through a blink of the button to tap
    - system should add a tone/color to the sequence after every turn
  - *Hard level*
    - same basic behavior of the Classic level, plus: everytime the player completes the sequence, the four colors change their position, making the gameplay harder
- **Multiplayer**
  - *Classic Multiplayer*
    - asynchronous multiplayer
    - player can face another player to challenge a sequence
    - the highest score wins
    - players face each other on the same sequence
    - history and invites available
  - *Nearby Multiplayer*
    - multiplayer with four friends
    - the four players should be nearby, otherwise they could not play together
    - every player's smartphone will be colored by one and only color (blue or red or green or yellow)
    - sequence starts on the four smartphones: one display at a time blinks
    - player taps only his smartphone in his color turn
    - the one who taps his smartphone in a turn that is not correct (other player's turn) loses
- **Scoreboard**
  - possibility to see friends' highscores both for *Classic* and *Hard* mode
  - possibility to unlock achievements due to ability or number of games played
- **Settings**
  - the user should be able to turn on/off the background music
  - the user should be able to turn on/off remote notifications

Non-functional requirements

- When using Nearby Multiplayer
  - it may happen players experience momentary connection loss, and that can slow the game, creating confusion. Few solutions to solve the issue are the following:

        - A View update informing users that one of the player is having internet connections troubles, by creating a custom widget or by implementing a specific loading bar.
        - timeout time permitted at player's turn of no more than 5s -otherwise player loses
        - server-side mechanisms to re-order players' taps in case of low connectivity through timestamps
- Other players should be notified by using remote notifications.

User requirements

- Players should be able to play a single player version of the game.
- Players should be able to play an harder level of the single player version.
- Players should be able to challenge friends asking them to complete a sequence.
- When four Simone players are near, they should be able to play together, miming each other a colour of the Simon board.
- Players should be able to start a multiplayer match even if they don't have the app open in foreground.
- Players should be able to enable/disable music and notifications.

▸ 03
Architectural design

Distribution

The original idea of the arcade game has been expanded with multiplayer, having a *client-server* architecture, where the server handles the behaviour in the game, holding a shared center of storage and being a communication middleware.
The architecture of the system can be represented by the following *distributed* solution diagram, where the central element acts as a coordinator, being integrated with server-side code.
The server is in charge of:

- storing all the data
- notifying users' devices about variable changes
- having an active role in coordinating users in multiplayer mode

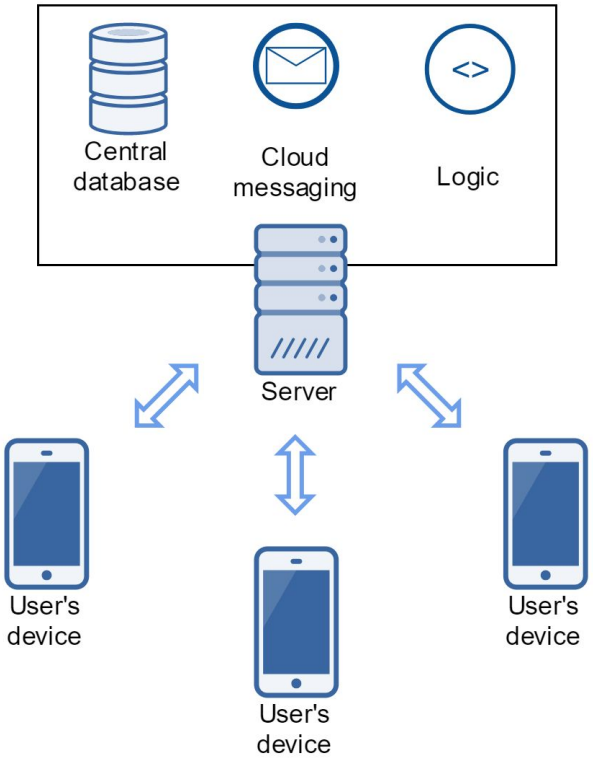


Figure 2: Distributed architecture diagram

App Architecture - MVP with Actors

In android software development, *activity* classes are used to handle User Interface inputs and widgets, making call to database, updating the model and so on. When the application becomes more and more complex, those classes become long and hard to read, creating non stop bugs.

Figure 3: Simple MVP

The *model view presenter* pattern is used to separate logic and view updates from the activity (*View*) by means of a *Presenter* class.
The *Presenter* stands right in the middle between the *Model* and the *View*, working similarly to *MVC*'s controller. The difference is that it *presents* data to the *View*, and the *View* only has to show it in the way the *Presenter* tells to, becoming a passive entity.

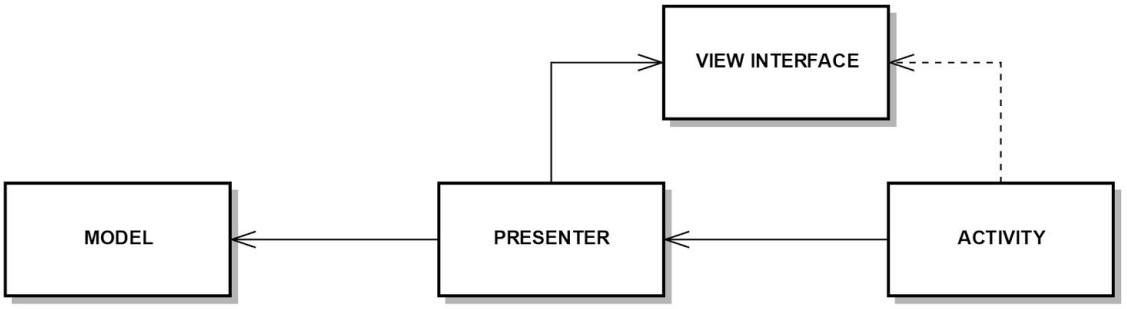


Figure 4: MVP with view interface

Model View Presenter is implemented by using a *View* interface of the *activity*, called *contract*, which defines the communication between the *Presenter* and the *activity* itself.

The piece of view dedicated to the interaction with the user was the key point to analyze, due to its importance given by the role of being the intermediary between the player and the arcade logic. The *message passing* technique was implemented in order to handle the concurrency generated by the user's tap and the computer's round, generating the desidered turn-based behaviour. The *actor model* acts perfectly for this purpose.

Actors

Actors represents a programming paradigm that takes origin from the physics.
The main idea behind is that an actor, which is basically an entity, can interact with other actors only by message passing. When a message is received, if it matches a specific pattern, a computation is triggered. Actors may modify internal state, but can only affect each other through message passing avoiding the need of locks. This is why the actor model can be

considered as the best alternative paradigm to concurrent programming and this is why this model was chosen for this project.

The actor model can be mapped as:

- **Actor System**: is where actors live
- **Actor**: reactive entities
- **Mailbox**: each actor contains a mailbox where messages are stored (FIFO policy).

The main difference that distinguishes actor programming from object oriented programming is that, in OOP, objects are passive (due to information hiding principle), while actors are reactive entities: they do something only when a message is received.
On the other hand, this brings to some difficulty while design their behaviour and logic: what if an actor receives a message while it is still computing its triggered handler? The message is stored into the mailbox and it is processed only when previous handlers are done.
This property is called *macro-step semantic* and it leads to adopt certain kind of rules while programming our actors:

- handlers (meaning the computation triggered after a message is received) should avoid loops.
- handlers should not be synchronous.
- handler computation must not be data intensive.
- data intensive handlers must be splitted into short handlers assigned to different actors.

The *MVP diagram* of the previous page becomes as follows:

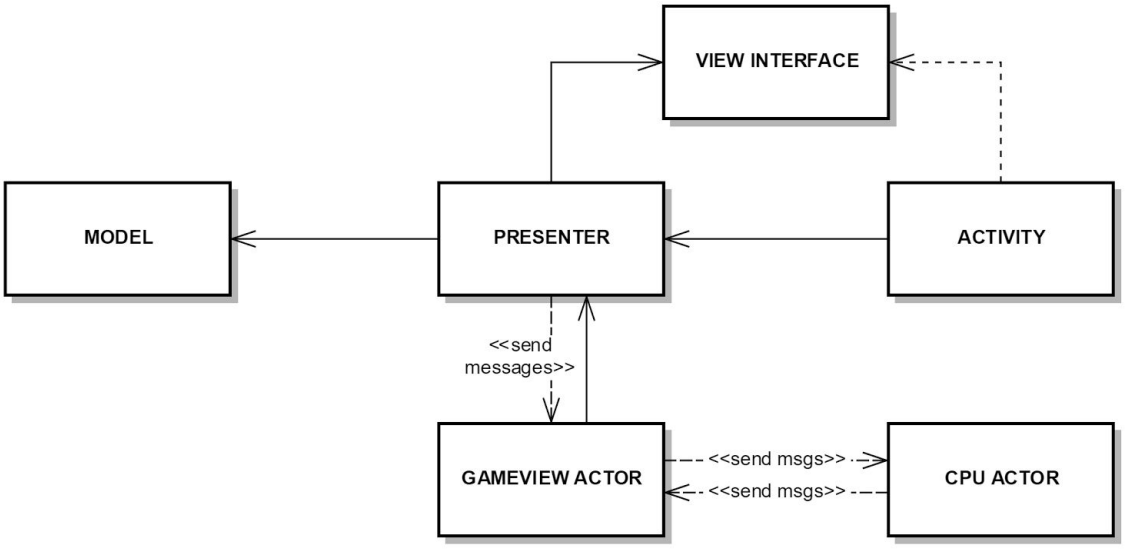


Figure 5: MVP with Actors

▶ 04
Design in detail

Chosen Technologies

Firebase

After trying out other frameworks, like PubNub or Realm Mobile Platform, we chose Firebase because it fits system following requirements:

- data-storage
- sharing common data (match data)
- keeping common data updated
- communication attitude

Firebase is a Backend-as-a-Service cloud platform for development of web and mobile apps, founded in 2011 and acquired by Google in 2014.

Firebase provides tools that help developers implementing the most common features available for web and mobile apps, like:

- Analytics
- **Cloud Messaging + Notifications**
- Authentication
- **Realtime Database**
- **Cloud Functions**
- Storage
- Web Hosting
- **Test Lab for Android**
- **Crash Reporting**
- App Indexing
- Dynamic Links
- App Invites
- Remote Config
- Adwords & Admob

Firebase has been chosen for this project for the following reasons:

- The app is developed from scratch and we needed a RealTimeDatabase technology

- The app needs to be fast, easy to use and doesn't perform heavy data processing needs or complex user authentication requirements.
- Message passing is a key point and Firebase is made to communicate easily and effectively.

The features offered by Firebase used for *Simone* are:

- *Cloud Messaging*: it's the only native library for Android currently supported by Google, after the deprecation of Google Cloud Messaging. Other frameworks like OneSignal are built on top of FCM, which already provides full access about handling notifications.
    - We use it while inviting other people to join a Multiplayer Match.
- *Realtime Database*: provides APIs to persist and synchronize data between devices. Data is stored in JSON format into a NoSQL database, which is accessible by listening on *nodes*. By asking data from a particular node, Firebase returns all the data inside the tree of that node. APIs provide methods to listen to remote changes on the database, which are delivered to all the attached devices by using the Server-Sent Events protocol.
    - We use it for Multiplayer as a base platform to exchange and persist data between devices.
- *Cloud Functions:* with CF you can write server-side JavaScript events that can be publicly exposed through endpoints or attached to Realtime Database events (add, update, delete). Since RTDB is based on NoSQL database and Firebase recommends to flatten data structures, Cloud Functions are helpful to keep updated references to an object. In this way, we can reduce the weight of calls from mobile and web, in addition to a larger control on functions.
    - We use it to detect when all users accepted the invites, check CPU and player sequences in a multiplayer match, and send invites via FCM when a new match is created. Functions are written in JavaScript and are available here: https://github.com/simoneapp/simone-cloud-functions
- *Crash Reporting:* Crashlytics by Fabric and Firebase Crash Reporting intercept crashes of the app and automatically send a report with the complete stacktrace to Fabric/Firebase, in addition to data like smartphone model, battery level, operating system version, etc.

Facebook

In order implement a distributed system, we were searching for a way to identify users connected with the game. Since Facebook is widely used and provides good APIs for interfacing with the Open Graph, it was decided to use them for inviting players in the multiplayer mode. This means that, if the user is already logged into the Facebook App, he's not prompted to fill a login form, making the login procedure easier and faster. Together with Login, *Facebook APIs* were used to download the user's friends list - used in order to let the player choose the opponents to challenge.

Scores and Google Play Games

Google Play Games Services provides APIs that let the user retain and match players for games on Android, C++, and the web. *Simone* project was created -as well as on *Android Studio IDE* - in the *Google Play Console*, in order to gain all the components related with *Google Play Games API*. The feature *Simone* uses are:

- leaderboards: friends-related and global boards with high scores of *Classic* and *Hard* mode;
- achievements: possibility to unlock achievements related with ability or number of games played;
- test: some testers were given the access to check the correct running of leaderboards and achievements during the development.

With this elements the actual distributed diagram becomes as follow.

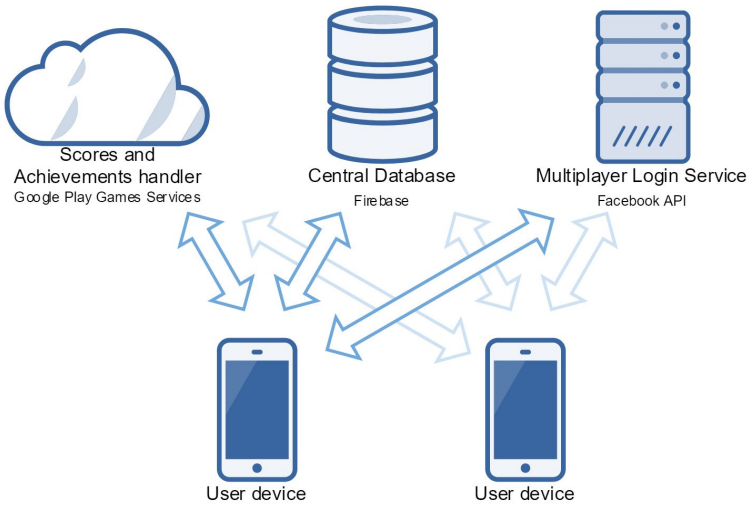


Figure 6: Distributed architecture diagram

Kotlin

We chose to insert some **Kotlin** classes inside the project because we wanted to try this new programming language, which was recently selected by Google as the main language for future Android apps and it will be natively supported in Android Studio 3.0.

Kotlin provides useful features, like null safety and optional chaining (e.g. calling functions like `this.item?.price?.euro`), smart cast, default arguments, data classes, etc.

We found Kotlin really interesting for its syntax, similar to Swift, less verbose than Java, and for its interoperability with the Java Virtual Machine - in fact, Kotlin is written on top of the JVM. It's a good compromise between the verbosity of Java and the concise syntax of Scala.

Kotlin was used in the project by Giacomo Zanotti, Nicola Giancecchi and Giacomo Bartoli.

These packages are (almost) fully written in Kotlin:

- `app.simone.shared.firebase`
- `app.simone.multiplayer.controller`
- `app.simone.multiplayer.view.*`

Arcade

It is worth to spend a few words on the original *Simon Game* before going inside the design and development of the app, in order to understand what was done and for what purpose.

The original *Simon* device was made of 4 different colors, each one reproducing a particular sound when blinked or pressed. A round of the game consists in the random lighting up of the sequence (1 color in the 1st round, 2 in the 2nd round, and so on...) and the identical sequence repeated step by step by the player in the same order. It is a memory game, played and loved by everyone.
Having that gameplay clear in the mind, we came up with the first activity diagram describing the simple arcade match:

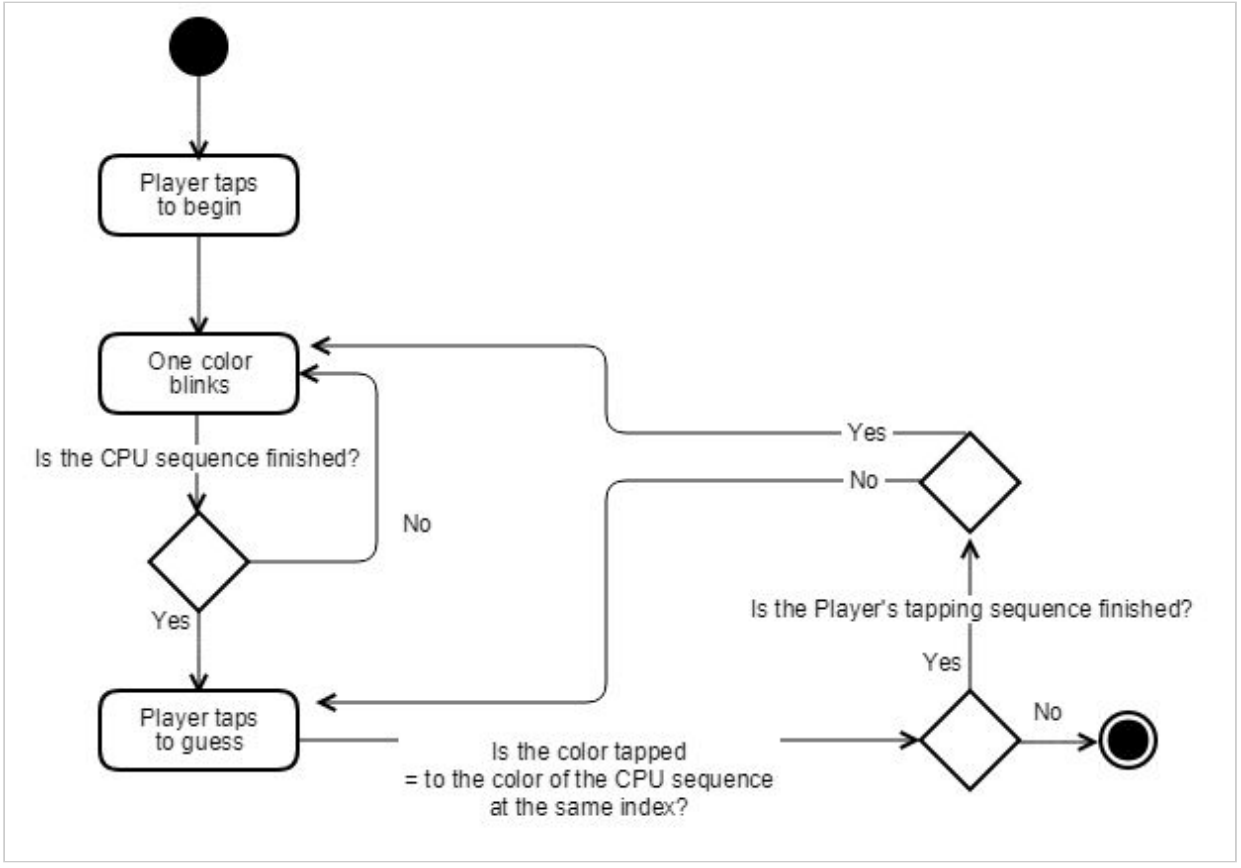


Figure 7: UML Activity Diagram - Arcade match

The initial steps of the arcade game were intended to "make *Simone* blink", so the focus was given to the very early part of the diagram above.
The design pointed out the role of two main actors:

- *CPU Actor,* the one that computes a single color in each turn. It must hold the color sequence, refresh it, and send it to the *GameView actor*;
- *GameView Actor,* the one handling the interaction with the Android activity and the player. It holds a reference to the view and communicates with it through its handler. It also provides mechanisms to handle the turn-based logic and to check the fairness of player's sequence.

We modeled our system as you can see in the following scheme: the view - precisely the view *presenter* - has a method to return its handler, which is the only "public interface" that the `GameActivityPresenter` holds, so its only way to communicate with the actors, simulating therefore an actor behaviour with messages exchange.

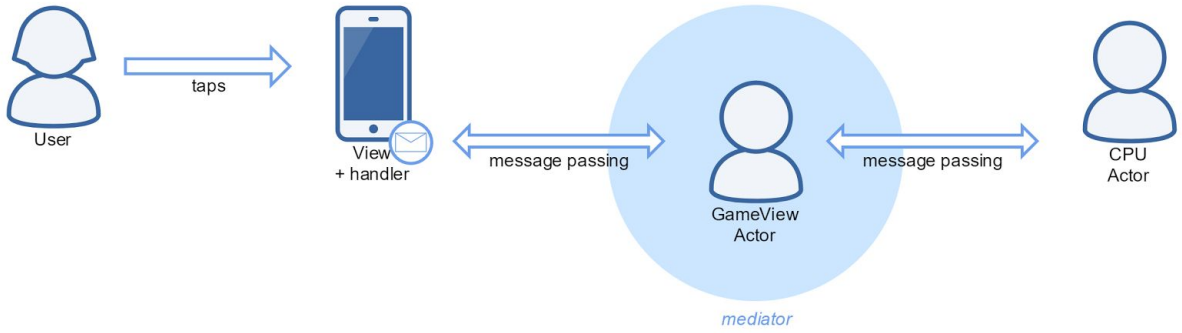


Figure 8: Communication diagram

This scheme was then converted into its easy UML Class representation.

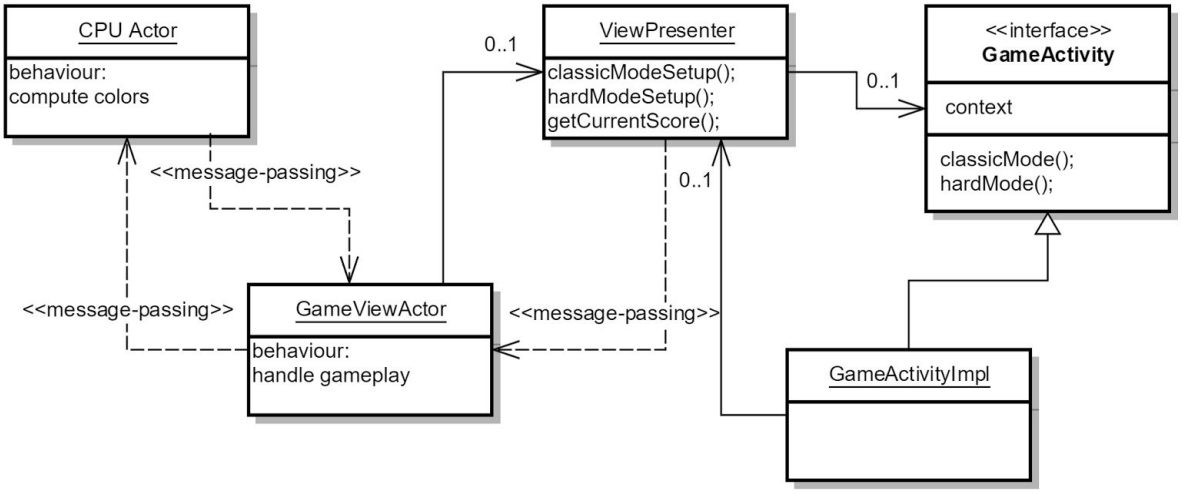


Figure 9: Easy UML Class diagram

User's behaviour can be more easily resumed in the following diagram, which pointed out 3 key states:

- **Seeing (CPU's turn)**, the initial action of Player: when *Simone*'s blinking he has to memorize the sequence, in order to reproduce it later on ;
- **Tapping (Player's turn)**, the moment Player has to tap.
- **KO (Game ended)**, player has tapped a wrong color.

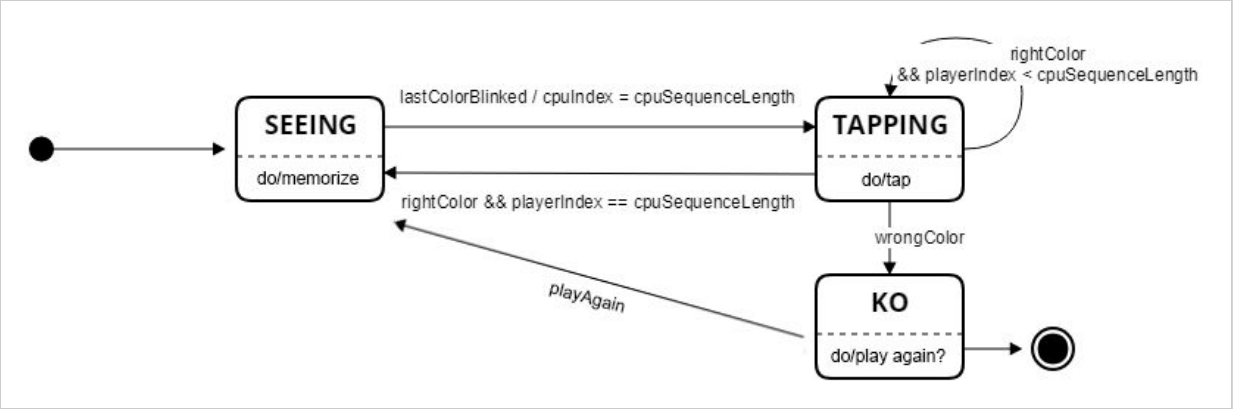


Figure 10: UML States diagram for Player - Arcade match

The sequence diagram proposed in the following page explains the communication between the actors and the player during an arcade match, using the messages of the UML diagram below.

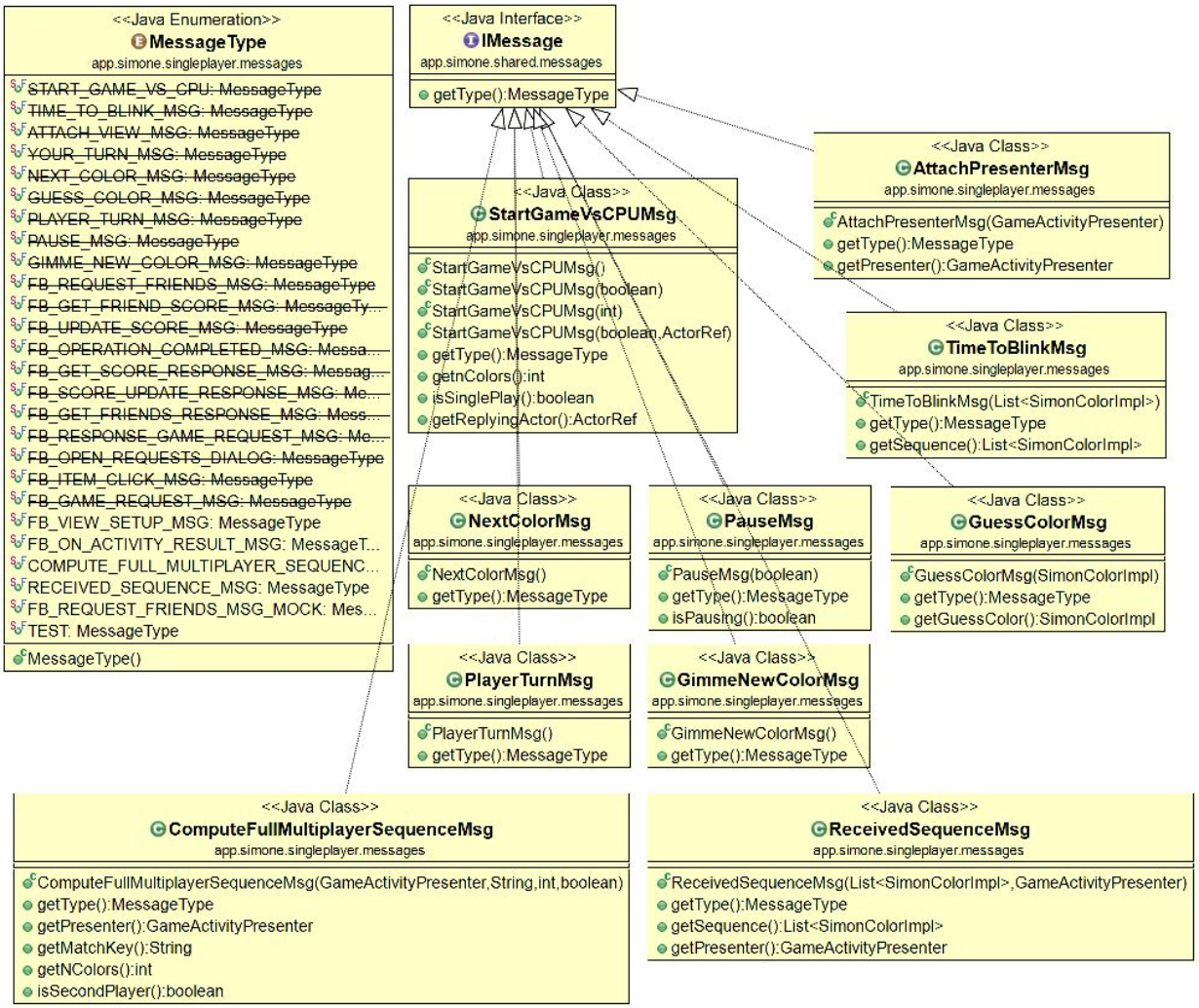


Figure 11: UML Class Diagram - Messages exchanged between Actors

The player is, the one that triggers the chain. With its tap, a message containing a self-reference is sent from the view to the `GameViewActor`: this way the actor can send messages to the view handler. As soon as the `GameViewActor` receives the starting message, it notifies the `CPUActor`

to compute a color. This actor initializes the sequence and add a new color to the list. The `CPUActor` immediately replies the sequence to the sender, which tells the view to blink that specific color. Now, the activity doesn't know if the CPU sequence is over or not, so it asks the `GameViewActor` if it is: the *alt loop* in the sequence diagram describes this situation, so while the CPU sequence is fully blinked it's *Simone*'s turn and the view has to light the 4 buttons one after another. On the contrary, it's `Player`'s turn - so that he has the possibility to tap and guess the sequence - when the *GameView* actor says the CPU sequence has ended.
The game goes on until the `Player` taps a wrong color.

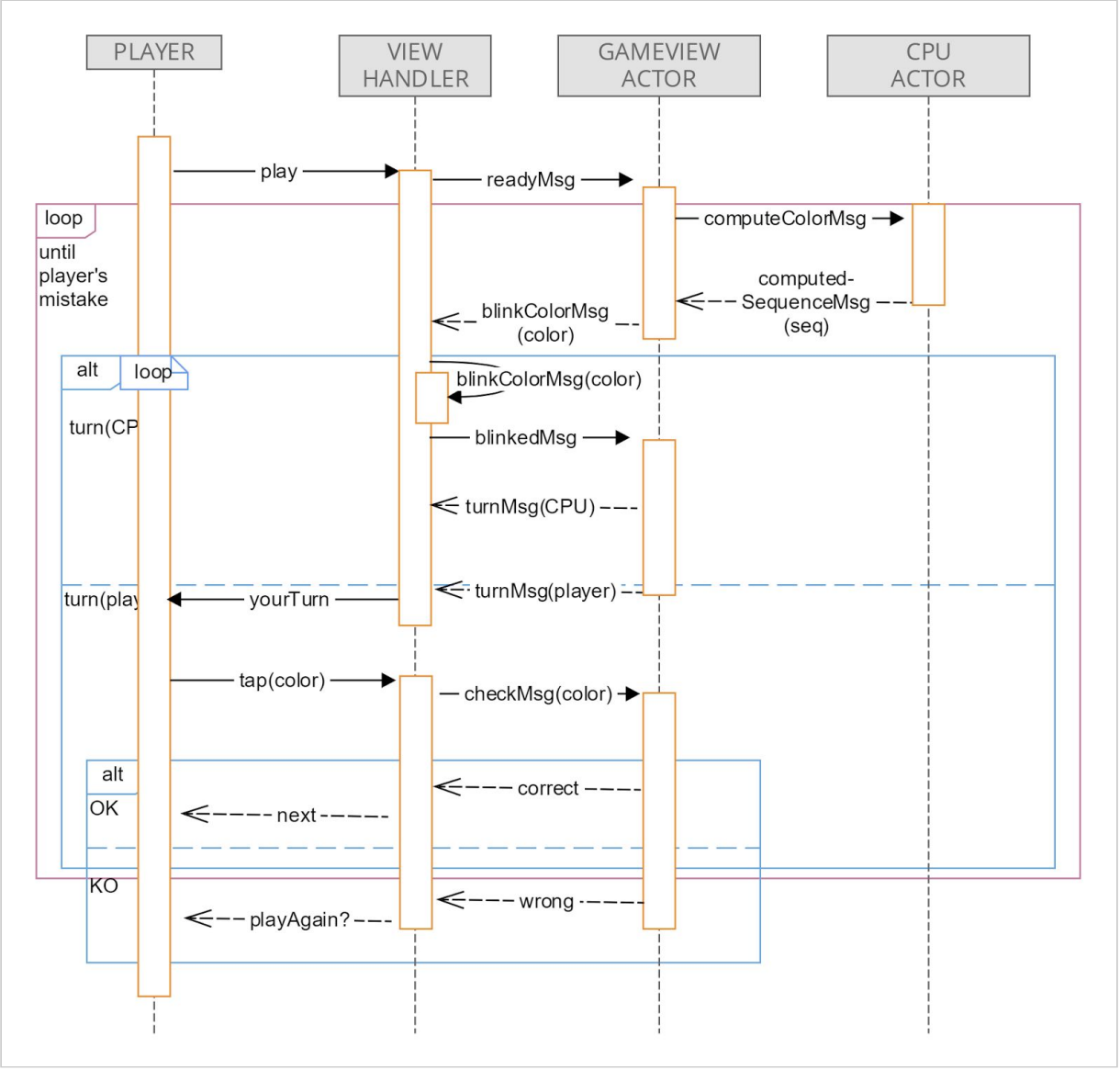


Figure 12: UML Sequence diagram - Arcade match

The view was rendered following the next UML class diagram:

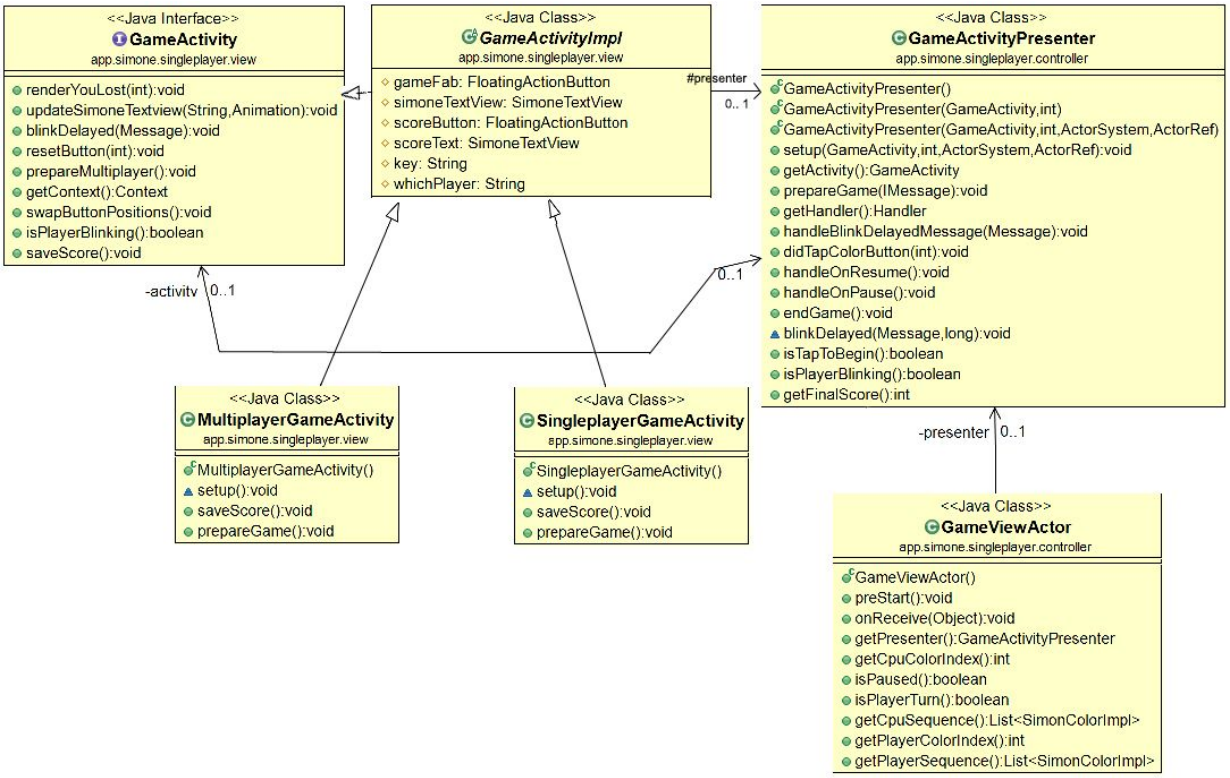


Figure 13: UML Class Diagram - Arcade match activities

Multiplayer

How can an 80s arcade game be converted to a multiplayer game?

Classic

Classic multiplayer mode is a 1 vs 1 match.
What's the difference between this mode and playing an arcade match? Both players face each other on the same sequence of colors. The play is *asynchronous*, as the first player, after Facebook sign-in, selects a friend from the list and starts playing the sequence computed. The same sequence is immediately sent to the opponent through a Firebase cloud message and then the second player can play whenever he desires. The match of the 2nd player can start by tapping directly on the notification or by opening *Simone* later on, in classic multiplayer mode, where he can find all the invites and the pending matches. At the end of each play, the score is stored into the realtime database, which notifies the updated data to both players. The sequence is described in the diagram below, in which the *"play"* self-arrow contains the whole sequence diagram of the previous page, that describes the gameplay. Please note that the case described below is the success case of Facebook sign-in: if a user cannot log-in on Facebook, it cannot select an opponent and so cannot create a multiplayer match.

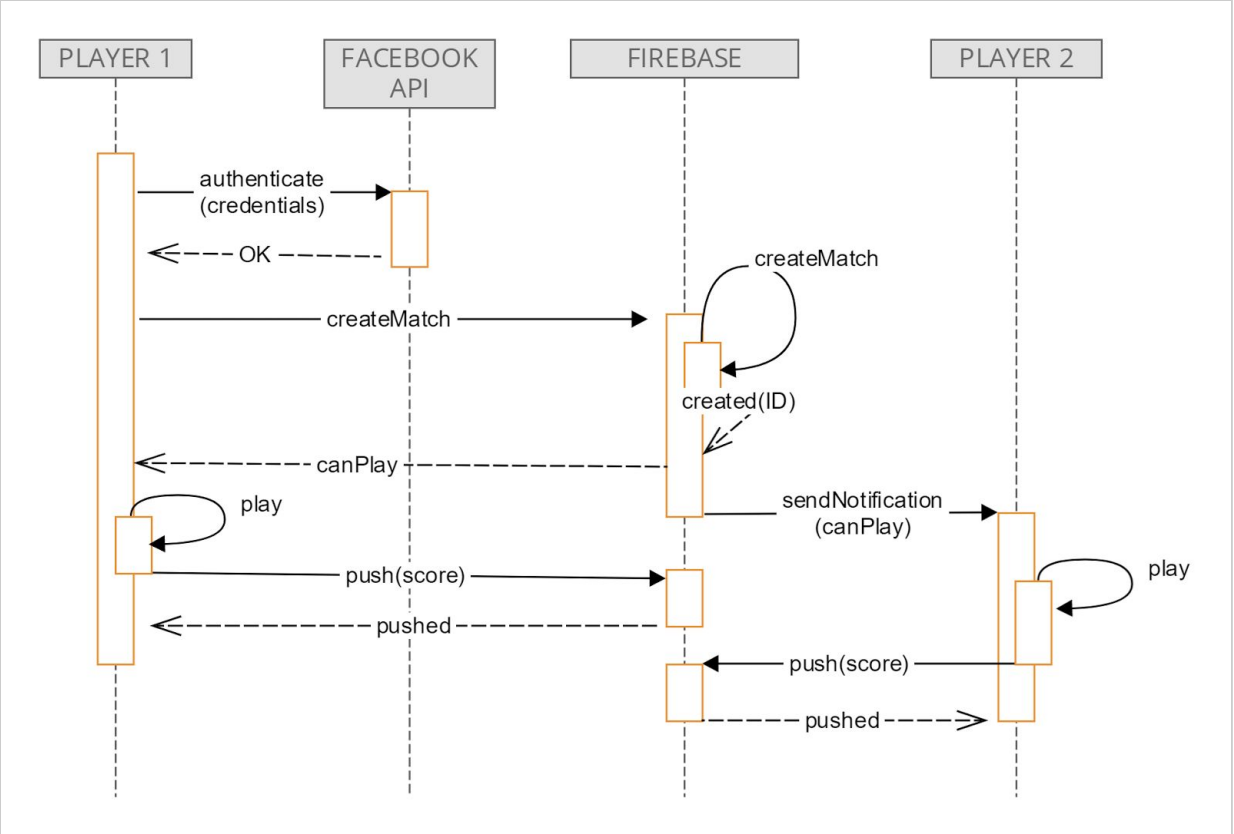


Figure 14:UML Sequence diagram - Classic multiplayer match

The logic used is the same as the arcade match.
`GameViewActor` and `CPUActor` were in fact designed to fit the classic multiplayer mode. These and some of the other components have obviously been reworked and integrated implementing the features described in the implementation part.

An `OnlineMatch` is the combination of:

- First Player (the one who started the game and send the challenge)
- Second Player (the one who receives the request)

Each player has a score, which is stored when the game is ended, and it is modeled through a FacebookUser class, which is instantiated after a successful Facebook login.
Each OnlineMatch is identified by a random key, generated automatically by Firebase when pushed to the database.
Then all the matches are downloaded from Firebase, stored in an array which is taken as input from the class which is in charge of showing all the matches.
The GUI part is divided into two fragments:

- first fragment shows all the Facebook friends who have installed Simone app
- second fragment shows all the previous/on going matches.

`AbstractGraphRequestWrapper` is a class used for testing and mocking the request coming from the server.

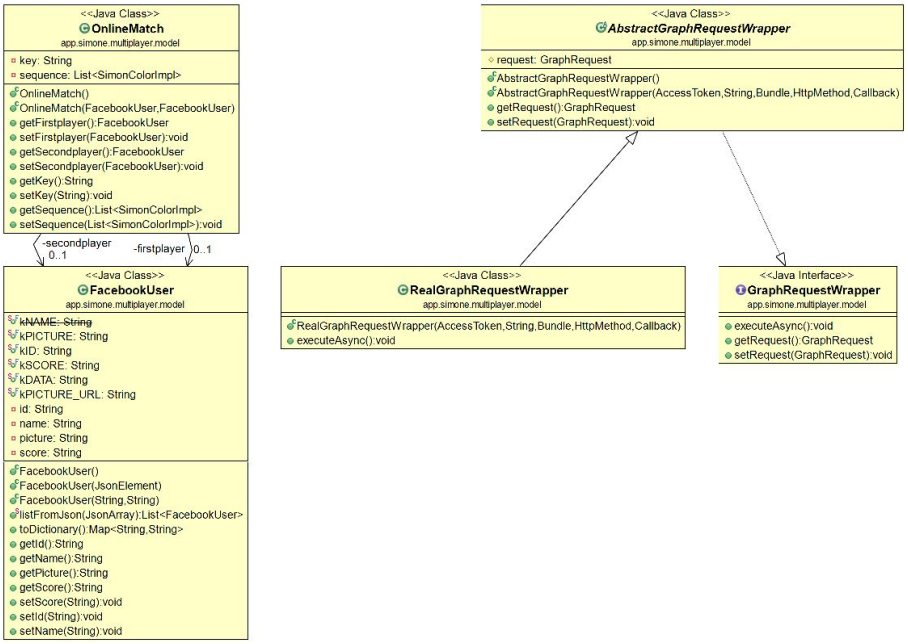


Figure 15:UML Class Diagram - Classic multiplayer match

Concerning the GUI, it has been divided into two fragments:

- the first fragment is where the user logs in to Facebook and select its opponent.
- the second fragment contains the list of previous/pending matches

Nearby

This second multiplayer game type consists in a *real-time* match with 4 people, where everybody is assigned with one specific color.
Considering the state diagram of single player, for the multiplayer version we have to consider the waiting state, as this is a *synchronous* play with turn-based mechanisms. Before the match can actually start, user must wait for other three players to accept the invites, but the game logic should remain the same: each turn a color is added and if one of the players taps the wrong color, the game is over. Below the state diagram of the *Arcade* section is expanded, and a new activity diagram is designed for this particular type of game.

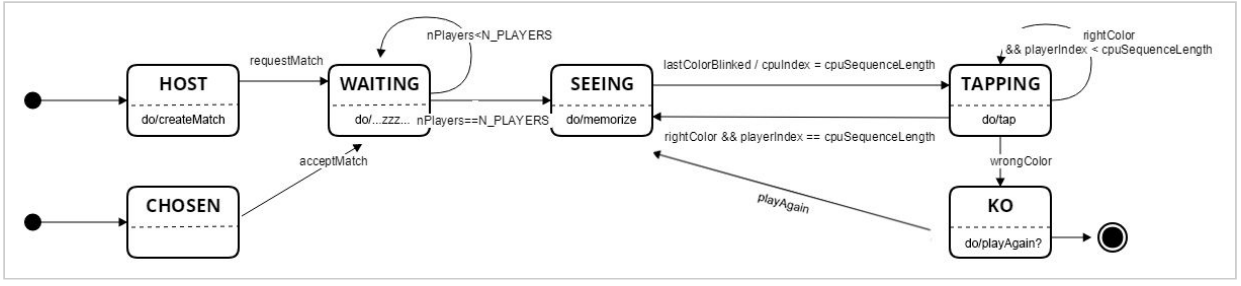


Figure 16: UML States diagram for player - Nearby

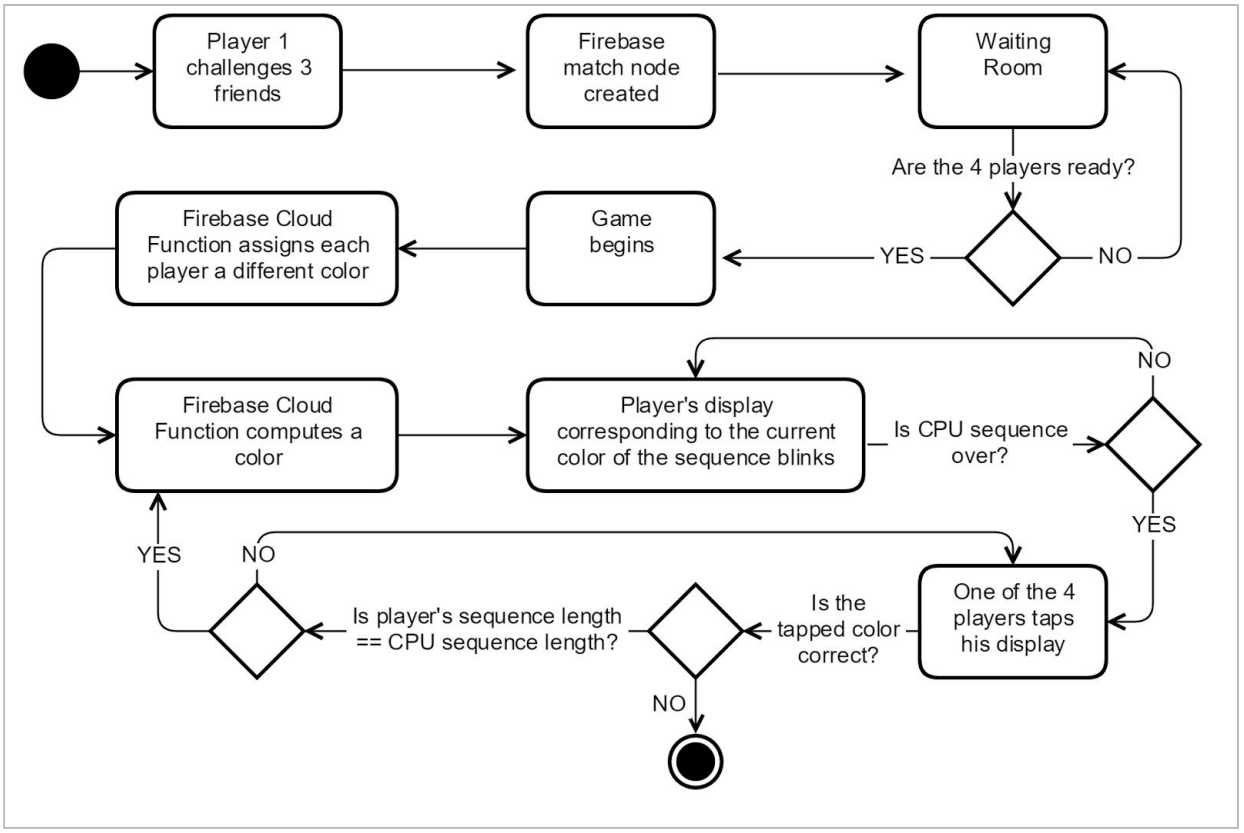
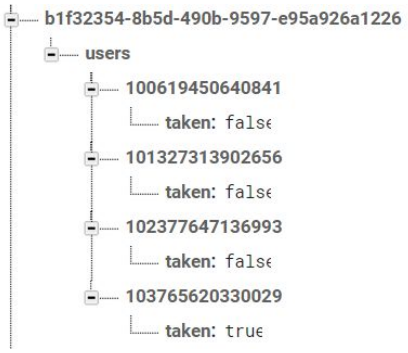


Figure 17: UML Activity diagram for player - Nearby multiplayer match

Even though the same concept of the single player game is used in multiplayer, implementation is very different. When the invite is sent, the node for the correspondent match is created, with the name of the match and four sub-nodes, one for each user player.



"Taken" subnode is generated for the players: when a player accepts the invite, " taken" changes value from false to true. *FirebaseCloudFunctions* trigger whenever data at node "taken" changes, counting how many " *taken*" are true: if the number is four, the game will finally starts. The following image will show how database is set up.
CloudFunctions will generate new nodes in the database: cpu sequence contains the colors, but the player's button is listening to "blink node". Blink node is the one telling the player which is the color that must blink. "Index" is used to understand if the color blinking is actually the last one of sequence: if it is the last one, it updates database "*status"* node from Simon's turn to Player's Turn. Players can now freely push their buttons in order to match the cpu sequence, and Cloud Functions check if the color added is correct.
If the color is not correct the game is over, correctly updating players views.

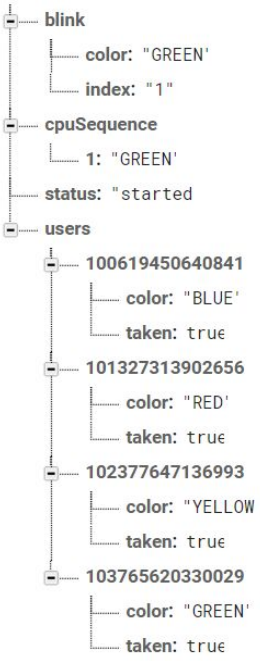


Figure 18 and 19: Firebase realtime database nodes

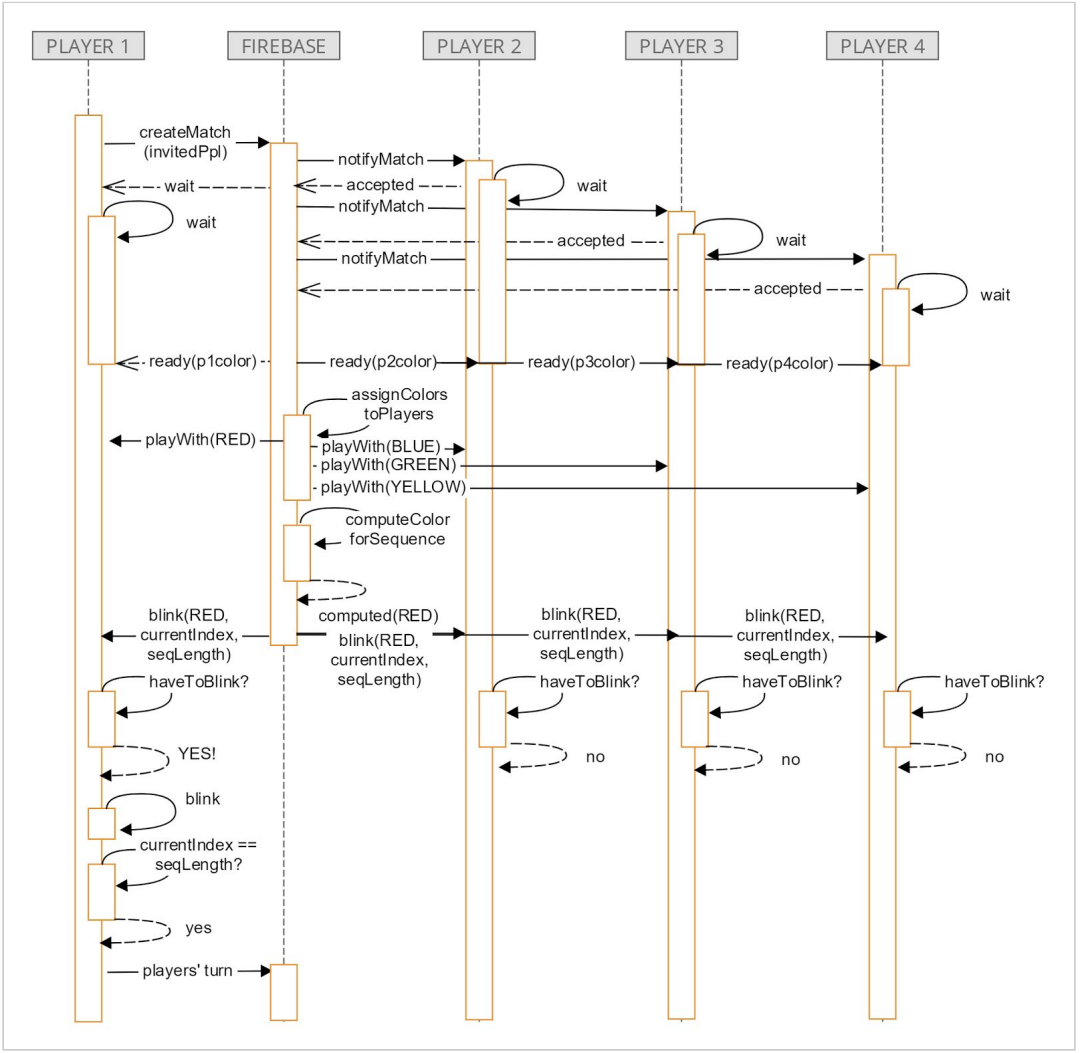


Figure 20:UML Sequence diagram - Nearby multiplayer play

Finally, this diagram describes the discussed above play logic of an example sequence starting with `RED` color. The proposed diagram is interrupted at players' turn.

Design patterns

- Single Player
  - *MVP - Model View Presenter*: presenter of the `singleplayer.view.--GameActivityImpl` in class `singleplayer.controller.--GameActivityPresenter. Handlers and message passing logic moved inside presenter.`
  - *Mediator*: interaction pattern between objects that exchange messages: `singleplayer.controller.GameViewActor acts as mediator for singleplayer.controller.CPUActor` and communication handler of `GameActivityImpl` (inside `singleplayer.controller.GameActivity-Presenter).`
- Classic Multiplayer:
  - *Singleton* pattern used to access the database [see **class**: `multiplayer.controller.DataManager.kt`]
- Nearby Multiplayer:
  - *MVP*-model view presenter for `multiplayer.view.NearbyGameActivity` and `multiplayer.view.WaitingRoomActivity`. Each activity passively updates their widgets by means of their presenters `multiplayer.view.NearbyViewPresenter` and `multiplayer.view.WaitingRoomActivity` respectively.
  - *Decorator*: decorator pattern for `multiplayer.view.NearbyViewPresenter`, a new class `multiplayer.view.EnhancedNearbyViewPresenter` adds new functionality and overrides the *interface method onCreate*. The basic presenter is "decorated" with mainly two new features, the players turn and the game over message. Adding this pattern can allow the programmer to add new functionalities with ease.
- Settings:
  - *Singleton* pattern used to store user preferences, enable/disable the sounds. [see **class**: `shared.utils.AudioManager.kt`]
- Others:
  - 2 appliances of *Template Method* for the view:
    - to handle the content view and the back transitions of different views. [see class shared.main.FullscrenBaseGameActivity and subclasses]
    - to setup the correct configuration for arcade play or multiplayer depending on the chosen play [see `package singleplayer.view` and `classes GameActivityImpl, SingleplayerGameActivity, MultiplayerGameActivity`]

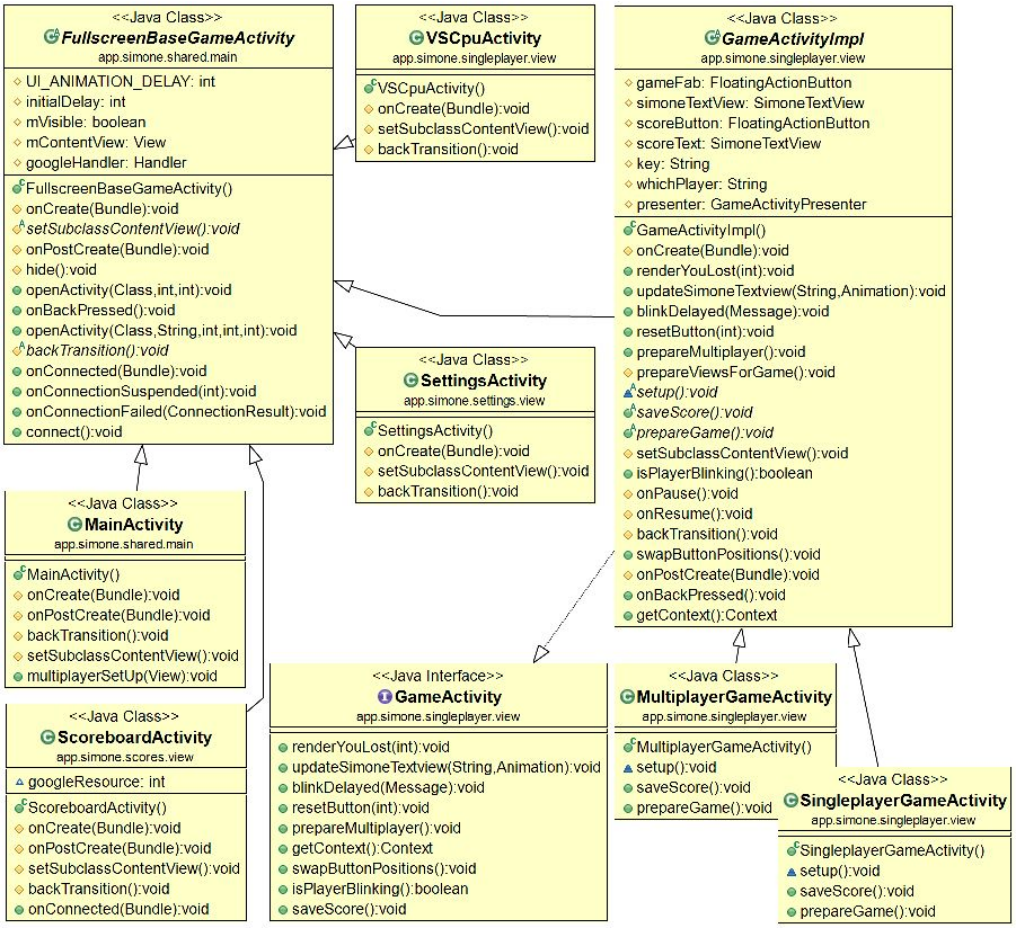


Figure 21: UML Class Diagram - View

- *Singleton* pattern used to make `GoogleAPIClient` unique in the Application

- Tests
  - *Singleton*: `TestableAudioManager` is a singleton class which inherits from `AudioManager`. It forces the `IS_DEBUG` flag to false, so it's possible to write effective instrumented tests on it, and makes available the `AudioPlayer` object inside the unit testing target.
  - *Strategy*: since the `GraphRequest` class included in the Facebook SDK is declared `final`, is not possible to mock it using Mockito, so we created the `AbstractGraphRequestWrapper`. An object that implements the `MockingStrategy` interface is passed to the `MockGraphRequestWrapper` to provide a `CorrectMockingStrategy` or `WrongMockingStrategy` for test purposes.

Tests

Tests are made by using the JUnit 4 library. For the instrumented tests, AndroidJUnit4 is used. Unit tests are often written with the support of *Mockito*, an Android library for creating mock objects. Mockito and mock objects were really helpful to focus on code instead of worrying about libraries, async requests and unavailable classes due to `final` accessor. Furthermore, unit tests are significantly faster than integration tests, since they doesn't need an emulator to perform actions. Writing unit tests also required some refactoring, including the actor classes

(CPU and GameView Actors for Singleplayer, Facebook Manager Actor for Multiplayer), to split logic and graphic code.

We tried to write mainly unit tests; where not possible or too difficult to implement, integration tests are provided. Actually, unit tests cover 16% of classes of this project, 13% of lines of code. Instrumented tests cover tests about settings, music and nearby multiplayer view, bringing the total amount of tested classes approximately to 20%. We choose to not write tests about the visual code (we could use Espresso more intensively for this in the future), the messages classes and the scoreboard.

These are the main tests we wrote:

- **Unit**
  - **`AppTest`**: creation of real actors, creation of a fake actor
  - **`FacebookManagerTest`**: testing of mocked successful and failing calls to Facebook Graph. A strategy pattern is used to provide a mocked calls (`WrongMockingStrategy` and `CorrectMockingStrategy`).
  - **`FacebookUserTest`**: testing of common scenarios for `FacebookUser` objects, including successful and failing initialization from parameters, from JSON object and from a list of JSON objects.
  - **`SinglePlayerTest`**: tests the functions implemented in the CPU and Game View Actors by sending messages to the test actors through Akka's TestKit.
  - **`DataManagerTest`**: tests if the app is able to connect, write and persist data to a mocked `DatabaseReference` object.
- **Integration**
  - **`SettingsInstrumentedTest`**: testing persistency on `SharedPreferences` of settings about music and notifications, handled by the `SettingsManager` object.
  - **`AudioManagerTest`**: tests the audio player of the background music, including start, stop and checking the behaviour according to the settings.
  - **`NearbyViewIntegratedTest:`** checking if button is correctly displayed and the user updates the players sequence in the correct Firebase node.

What went well and what went wrong with tests

We tried to write instrumentation tests at the first stages of development, after implementing a running workflow for the Facebook Login feature, in order to try Espresso.

We came back to write tests in the last stages of development, while the project was under completion. We admit that leaving the writing of unit tests at the end was a bad choice, because it could speed up development and testing of some modules.

On the other hand, we think that writing tests in a pure TDD approach - in this case - was difficult because of the changing of requirements *in itinere*. A mixed approach of writing tests after/while developing small pieces of code could be adopted.

Understanding the role and the utility of mock objects was not immediate. We also found difficult to test Facebook and Firebase classes, since some of them are marked `final` and not

mockable. This problem also affects Kotlin classes, that are considered `final` by default - until you don't declare them as open.

Akka's TestKit framework helped a lot on testing actors, since the `TestActorRef` type provides a `getUnderlyingActor()` method that exhibits the original actor object.

Travis CI helped to automate builds for each commit; it also has a Slack integration that posts a notification when a build is complete. We tried to use it for testing, but without success. We found some troubles on declaring the correct emulator to use in the `travis.yml` file, so we step back to provide only build checking.

▶ 05
Implementation

Michele Sapignoli

Features implemented, listed by pull requests:

- **#PR6** - *Added a compatible version of the Akka library*: configured *Akka* actors framework properly (v. 2.3.16). First tries with actors and gameplay.
  Problems encountered: time spent to find the version of *Akka* and the configuration that perfectly suits *Android* - latest versions of *Akka* need Java 8 dependencies.
- **#PR7** - *Feature singleplayer*: first gameplay classes and methods added and first actors implementation (`CPUActor`, `GameViewActor`) having clear the UML scheme in figure 22. Template method implemented for `FullscreenActivity` and subclasses (`MainActivity`, `GameActivity`, `VsCPUActivity`) to set the *contentView* of the activity. Passed from the simple registering of the activity and first blink by the `CPUActor` to a simple game played - with console message in case of loss. Introduction of two handlers in `GameActivity`:
  - one that the actors gan get, used to communicate with them;
  - one totally private and used to handle delayed messages for button blinking.

  Added interface `IMessage`, enum `MessageType` and all messages classes used to communicate between Actors (other messages added during the whole development).
  Problems encountered: Akka error handling does not write the Exception on the Android monitor but re-instantiate the actor itself. Found out after some tries and debugging.

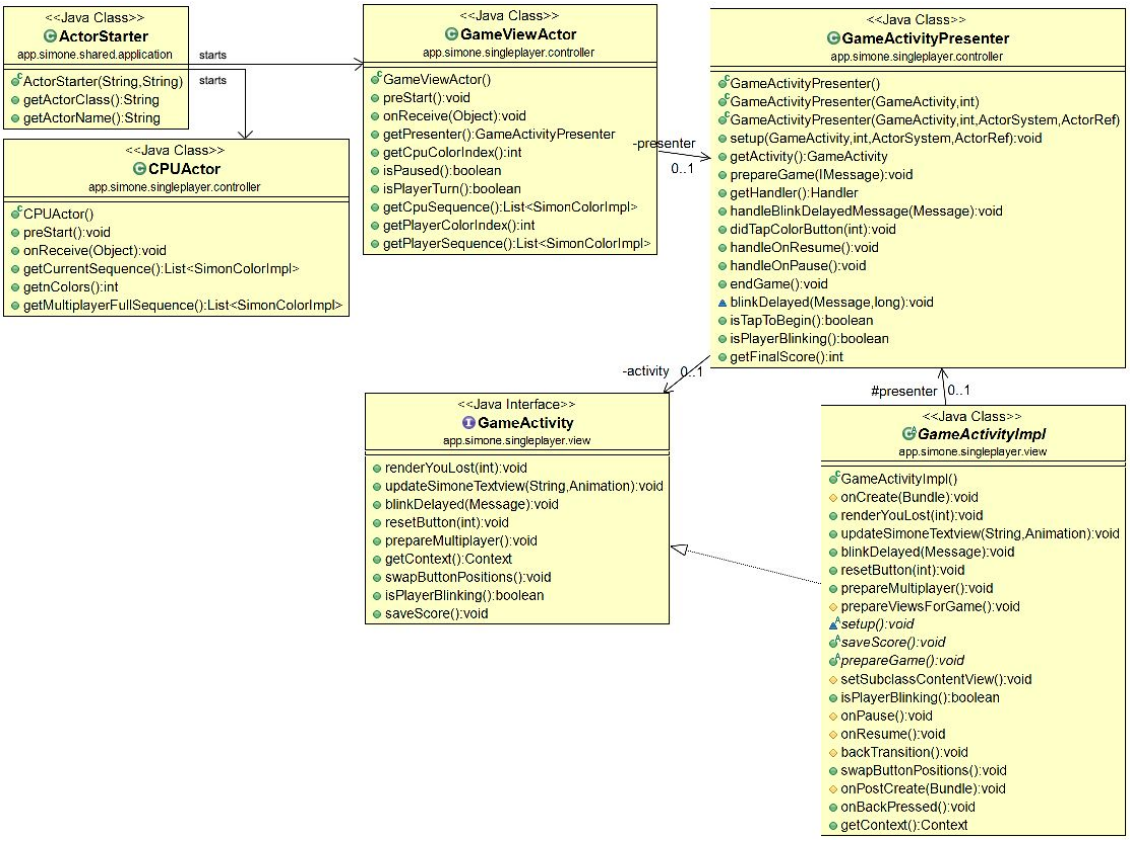


Figure 22: UML Class Diagram - Actors and GameActivityPresenter

- **#PR8** - *Merge branch 'develop'*: better handling of the game: re-organisation of `CPUActor` code and correct check of the Player's sequence by `GameViewActor`. *Fabric* configured, used to distribute and test the app thanks to the *Crashlytics* tool.
  Problems encountered: still have "fast tap" problems, so when a user taps super fast and gets a wrong color, the view freezes.
- **#PR11** - *Merge branch 'develop'*: better view handling: `GameActivity` improved with dynamic button initialization. Changes to enum `SimonColor`: buttonId connected to the enum and method to convert id into enum itself. Introduction of Android animations and custom view elements. Rework of the entire view following material design terms.
- **#PR13** - *Feature single play harder*: introduction of the "pause" function, so the message the view sends to the `GameViewActor` to suspend the game (used when the app is put in background directly from the `GameActivity` during a game or a call arrives). Arcade classic play fixed with correction of the "fast tap" anomaly. First hard mode introduced, with animations and button swap at every player's turn. Approach to the *Google Games API*.
  Problems encountered: handling of the button swap took a while: passed from switching X and Y position to changing of the view child.
- **#PR15** - *Feature-single-player-harder*: Introduction of `ScoreboardActivity` and `ScoreHelper` + `AchievementHelper` to make leaderboard and achievements effectively and correctly work. Re-work of enum *SimonColor*: now groups color, button id, game soundwith methods.s.
  Problems encountered: tries of *GoogleGames* for turn-based game and tries of usage of the pre-set *GoogleGamesUI* for multiplayer selection (not so suitable for our idea, so discarded in favor of *Facebook + Firebase*).

- **#PR18, #PR19, #PR20** - Team work with Giacomo Bartoli on actors and scores, to make components already made suitable for multiplayer.
  Made changes to the following components in order to fit multiplayer behaviour:
  - `ComputeFullSequenceMultiplayerMsg`: introduced message to indicate the multiplayer game, created by the "1st player";
  - `ReceivedSequenceMsg`: introduced message to indicate the reception by the 2nd player of the sequence computed by the 1st player;
  - `CPUActor`: most significant changes were made to this actor. A method to compute a full sequence made of 100 colors was implemented, as well as a new List containing the sequence itself [1st player behaviour]. The actor behaviour intended to receive and process the previous listed messages has been implemented. The logic of the play did not change: when the multiplayer game begins, the CPUActor - who already owns a completed sequence - passes just the first color to the GameViewActor, so the cycle of the activity diagram in the *Design in detail - Arcade* section begins.
  - `GameActivity`: added mechanisms to differentiate single from multi play:
    - *1st player behaviour:* added code to handle events generated by `ComputeFullSequenceMultiplayerMsg`;
    - *2nd player behaviour*: Firebase call to get the multiplayer sequence generated by the 1st player.

  Further changes have led to the creation of a `GameActivity` interface and its `GameActivityImpl` implementation (abstract class), which holds two *template methods* that calls abstract methods implemented in the subclasses `SingleplayerGameActivity` and `MultiplayerGameActivity`, made to setup the correct listeners on the main button of the view and to save the score differently: both scores, if best, are pushed to Google Games but multiplayer match score is always saved on Firebase match node too.
  Correct implementation of `GoogleAPIClient`, following Android best practises, with scores and achievements online. Better handling of game end and withdrawal. Made 4 leaderboards (then reduced to 2) to save high scores of the different modes.*Problems encountered:* not so easy implementation of `GoogleAPIClient` connection. Spent time to learn how it works and how to fit in our app.
- **#PR23** - *Merge branch 'develop':* Handling of no connectivity for high score - stored locally - then pushed online when possible.
- **#PR41** - *Merge branch 'develop'*: Scores and achievements perfectly working. Changes in `GameActivityImpl` to improve efficiency and gameplay + template method with subclasses `SingleplayerGameActivity` and `MultiplayerGameActivity`.
- **#PR47** - *Mediator pattern*: improved message passing technique using `GameViewActor` as a mediator between `CPUActor` and `GameActivityPresenter`.

Giacomo Zanotti

- #**PR8**-*Feature DB Access and views*: in this feature *RealmDatabase* classes and a ListView were developed allowing users to see their score in single and multiplayer. This part was later discarded because it was decided to switch to Google Games API, implementing a *LeaderBoard* that synchronizes way better all the scores.
- #**PR30**-*Feature Distributed Simon*: in this feature distributed game logic and coordination was implemented at client and server side. At server side, Firebase was used to design *CloudFunctions* that trigger whenever a data node is changed, taking care of adding new colors and checking if players ' sequence's colors corresponds to Simone's ones. At client side, *ValueEventListeners* were used to update the view as the data was changing.
- #**PR35**-*Feature Distributed Simon*: further work on distributed simon. The first tentative (**#PR30**) was working poorly. A new logic was implemented, and Firebase data nodes were updated with new designed *CloudFunctions*. In this part blinking rendering logic was not yet ready.
- **#PR43**-*Feature Distributed Simon*: in this pull request, client logic was further enhanced after merging with Nicola Giancecchi waiting room and other facebook classes. Bugs were found and later resolved. `NearbyGameActivity` was refactored with a *ModelViewPresenter* architectural pattern. The aim was to make `NearbyGameActivity` a passive class, with no logic involved. `NearbyViewPresenter` is responsible of making the activity's widgets to change and handling the firebase logic.
- **#PR45**-*Feature Distributed Simon*: final graphic updates, adding players and Simone's turn. Refactoring of `WaitingRoomActivity` according to MVP pattern.

Before starting to implement distributed simon feature, in a side project I started to make experiments with message passing library PubNub, but it made the game sloppy and hard to implement. We did not think about Firebase back then, and I (with my colleagues) lost a lot of time trying to make it work.
An important part was code refactoring using MVP pattern strategy, suited for android development. Even though the game was working, there was a single activity class, `NearbyGameActivity.kt` handling all the client side logic of the game. Decoupling was needed in order to enhance reusability and new features addition for future development, and MVP was chosen as the the perfect way to do that.

Giacomo Bartoli

Features implemented:

- **#PR31 Instant Multiplayer**: sending a game request to another friend using data provided by Facebook API (name, surname, ID).
- **#PR24, 31 Instant Multiplayer**: two players can play the same sequence of colors.
- **#PR39 Settings Activity** for user preferences.
- **#PR37 Code refactoring** following SOLID principles and clean coding rules.
- **Push Notification**, triggered through a Cloud Function everytime there is a new match.

Features which I worked on but not implemented:

- During the first sprint I spent many hours of coding trying to understand how remote actors, provided by Akka, can possibly work on mobile devices. Assuming a network configuration with a dynamic IP and NAT this was not possible.

- As we understood that remote actors were not a good choice we decided upon a message passing framework for Android: PubNub. I developed an entire P2P multigame using PubNub. Despite this, PubNub was not a good solution yet:
  - no central server for handling the game requests.
  - each devices was creating, updating a local database. What if the app was uninstalled?

Relevant elements:

- the algorithm used to download all the data from the database and filter them, showing only data about the player who is currently logged in, has been isolated into a single class. Firebase stores value using a random key generated by itself. So, inspecting the database this is what we get:

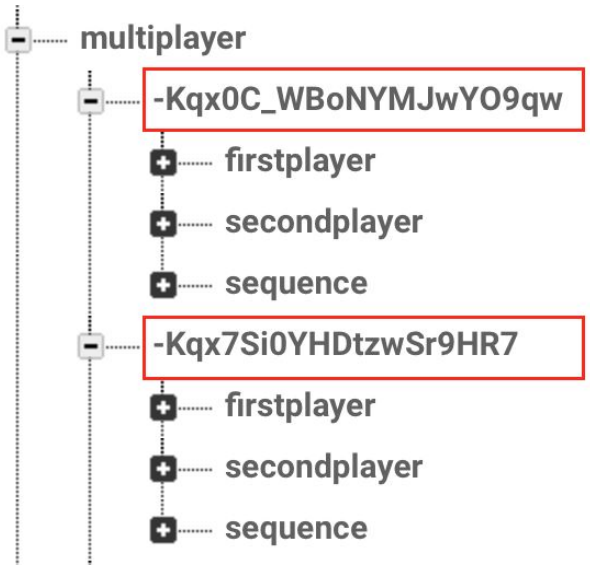


Figure 23: Firebase realtime database nodes

Red labels are the random keys generated by Firebase.
Right now, the algorithm downloads all matches and then it looks inside each one of them if it belongs to the current player or not. If it is, then the match is saved into an array, otherwise it is discarded. Computationally speaking, this is not the optimal solution. Thus, we decided to create a single class, meaning isolating the algorithm that implements an interface, so changing the algorithm can be easily done just working on that specific class. Moreover, it is known this can be provided by default in functional programming: we don't need a class plus an interface, we can pass directly the function, but, in this case, the function is quite complex, cause is made by a listener and a datasnapshot, so we thought that, following code cleaning principles, it would be a better idea keep things separated. [see **package**: multiplayer.view.invites **classes**: InvitesFragment.kt, Strategy.kt, StrategyImpl.kt]

Nicola Giancecchi

Features implemented, listed by pull requests:

- **PR #1: Added backlog document**. Added the Markdown template for backlog document into the repository. First approach to PRs.
- **PR #3: Travis CI Integration**. `travis.yml` file was correctly configured with Travis CI and synced with the GitHub Repository.
- **PR #10: Implementation of the Facebook Android SDK.** Implementation of Facebook Login into the app. This includes the fetch of personal info (name, surname, Facebook ID, profile picture) and the list of friends using the app. Added the "Invite friends" button, which could not be implemented natively: the SDK handles this feature by itself opening a web view. Implemented the management of the score via the Facebook SDK, then removed in future implementations.
- **PR #12: Facebook Implementation Refactored Using Akka Actors.** Complete refactor of the Facebook classes by using Akka Actors, following the same architectural choices made by Michele Sapignoli for the single player feature.
- **PR #17: General Code Refactoring.** Code cleaning, packages reordering, unified similar classes. Implemented the background music.
- **PR #26: Firebase Installation.** Preliminary commit for Firebase implementation.
- **PR #29: Nearby Multiplayer - Handshake.** Implemented remote notifications and token handling via Firebase Cloud Messaging (FCM). Added functions to interact with Firebase, e.g. for creating a new match or accepting invites. Refactored the Multiplayer view by implementing a ViewPager with two fragments (*new game* and *invites*). Creation of the match. Enhanced login with Facebook. Minor improvements.
- **PR #36: Nearby Multiplayer - Waiting Room.** Implemented the "Waiting Room", an intermediate activity that shows the users invited to a match and displays the current status of the invite (accepted or not). When the four users are ready, the game begins.
- **PR #38: Nearby Multiplayer - Integration Between Handshake and Game Logic.** Integration of the Handshake (match creation + waiting room) with the Game Logic implemented by Giacomo Zanotti. Minor changes to the game logic architecture.
- **PR #40: General Code Refactoring.** Minor refactoring, syntax check, conversion of hardcoded strings into constants.
- **PR #42 #48: Testing.** Writing of unit and integration tests; trying to integrate Travis CI with unit/instrumented testing but without success. Bug fixing.

One of the most challenging parts of my work was refactoring.
The first consistent refactoring made at PR#17 helped the team to work in a uniform way by splitting code into packages with an homogeneous structure. The packages are now organized in features, the same visible in the main activity:

- `app.simone.singleplayer`
- `app.simone.multiplayer`
- `app.simone.scores`
- `app.simone.settings`
- `app.simone.shared`

Each package, except the `shared` package, has three subpackages - `model`, `view` and `controller`, where applicable.
In the next refactoring phases, I tried make the code more clean and understandable with some tricks, like:

- making the methods most atomic possible (e.g. actors' on`Receive` methods)
- harmonizing the access to resources (e.g. actors references)
- limiting the length of the lines to 100 characters

- writing the javadoc

The changes affected not only my part, but the whole project.

Another challenging job was writing tests.
At the beginning, it wasn't easy to understand which tests to write (unit or instrumented?). I started writing some instrumentation tests, but after reading some online literature - like this from Zendesk - I understood that it was better to write unit tests for small pieces of code, then integrating them with instrumentation.
I finally understood the utility of mock objects by using the *Mockito* library. Mock objects helped me to refactor code by leaving the problem of dependencies outside the boundaries of the tested method. Instead of calling global or static references, even outside the current package, these references are now passed to the method or the class via dependency injection. An example is the use of `ActorSystem` objects, which are defined differently in the debug and tests environment. Instead of calling `App.getInstance().getActorSystem()`, the reference to the actor system is now passed in the constructor, like in the `GameActivityPresenter`.

▶ 06
Retrospective

A brief sum up about the development process can easily identify good and bad points of the project:

Sprints	Description
1st	SinglePlayer almost completed, implemented through Actors (akka). Classic MultiPlayer not completed yet, but we were able to communicate amongst devices using a message passing library (Pubnub).
2nd	SinglePlayer completed, available both in classic and hard mode. Multiplayer is completed but it is built on a p2p architecture: devices handle a local database, which is conitnuosly synchronized. This can lead to some mistakes into the application like a lack of data in case a user delete the app.
3rd	We moved to a centralized solution (Firebase). Now the Classic Multilplayer is perfectly working. High scores online using Google Games API. Nearby Game modality create, but it still needs integration with facebook invites
4th	Simone app is completed! - Single Player mode - Instant Multiplayer mode - NearBy Multiplayer mode - Settings - Highscores and leaderboards online

Some statistics provided by GitHub:

- 296 commits
- 68% of Java code
- 32% of Kotlin code
- 47 pull requests closed

Concerning the distributed part, we kept insisting on using Pubnub even if that was not the best solution. Pubnub is a library for message passing and each device was creating, updating a local database. This led us to some problems like:

- What if a user deletes the app? Then all data get lost.
- What if a user logs out from Facebook and log in with another account? The previous data, which belongs to the other Facebook user, are still on the device even though not visible.

Introducing Firebase, as a centralized solution, let us solve all these problems at once: all the data (meaning matches, user data, scores, timestamp..) are stored in a central server. Moreover, Firebase supports server side computation, which should not be done on local devices.
In conclusion, Firebase substitutes Pubnub (for message passing) and Realm (for storing data).

With few more work at server and client side, we could also implement 1 vs many classic match and more and entertaining feature for the nearby play.

What we like most about our *Simone* app is that it is fun, easy to learn and realistic, as this project is not just for academic and self-training purpose, but has a “scalable” aim: to be installed on every modern *Android* phone and to let people have fun with it.