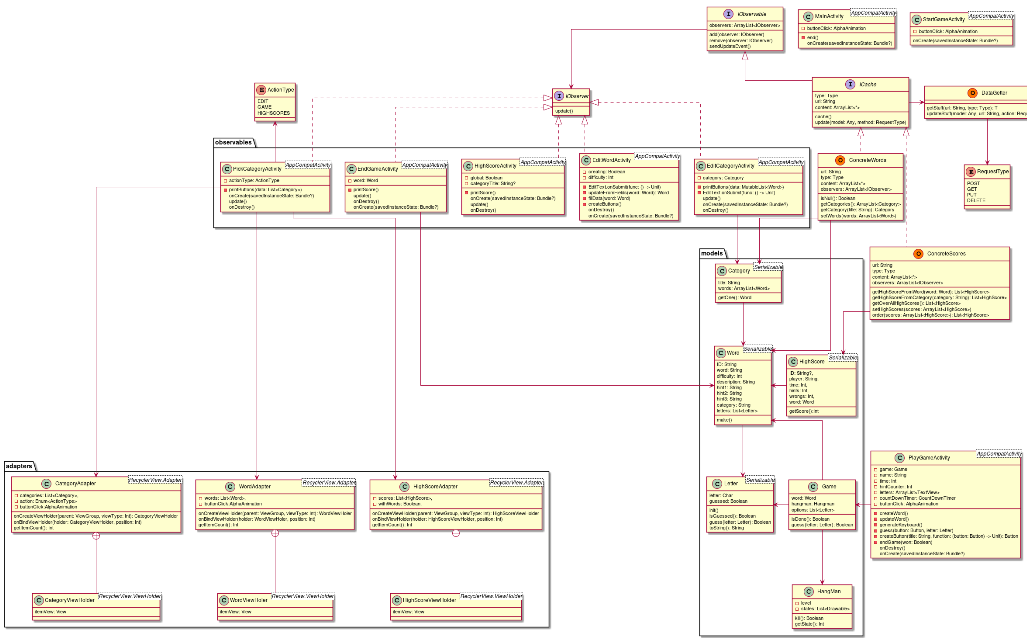
Hangman

Martin Mårtensson

November 9, 2020



Abstract

This is a game where you can pick a category which holds a number of words. The game will shuffle the words and the player will try to guess a hidden word, letter by letter. If the player guesses a letter which does not exist in the word, then a man, will slowly, step by step, be hung by his neck, on the screen.

The app has some flexibility and gives the user a possibility to change, add, and remove words, and categories from a database on which the app is connected to

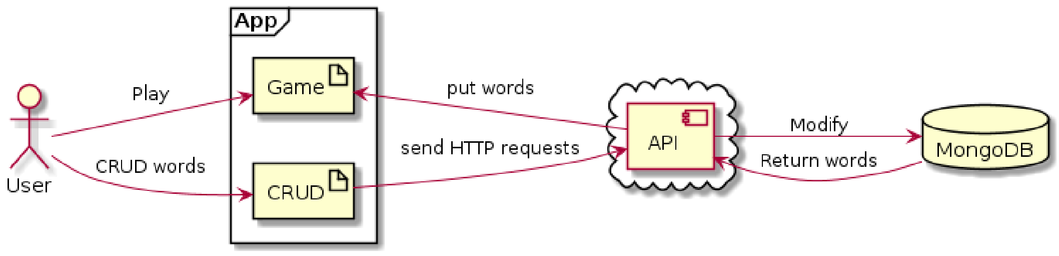


Figure 1: Diagram of the interaction between different components in the system, the app is using.

- PROJECT REPO https://github.com/DAT4/android-galgeleg
- REST API REPO https://github.com/DAT4/android-galgeleg-rest-api

1 Patterns

In the development of the app one key requirement has been to follow software patterns, for me to lean about that.

1.1 MVC

I have chosen to use MVC pattern (Model View Controller), which came naturally to me when I began developing the app, because I worked with this pattern before, and it helped me a lot creating the structure of the game. The MVC pattern is heavily used in the Game Part of the app.

Android is already using MVC in the way that they have Activities which is controlling the Layouts (xml files) So I used that stucture, by implementing my controller in **PlayGameActivity** and then I implemented the models relevant for the game.

NOTE: Some models are reused in other aspects of the App like in data transfer and in the CRUD part

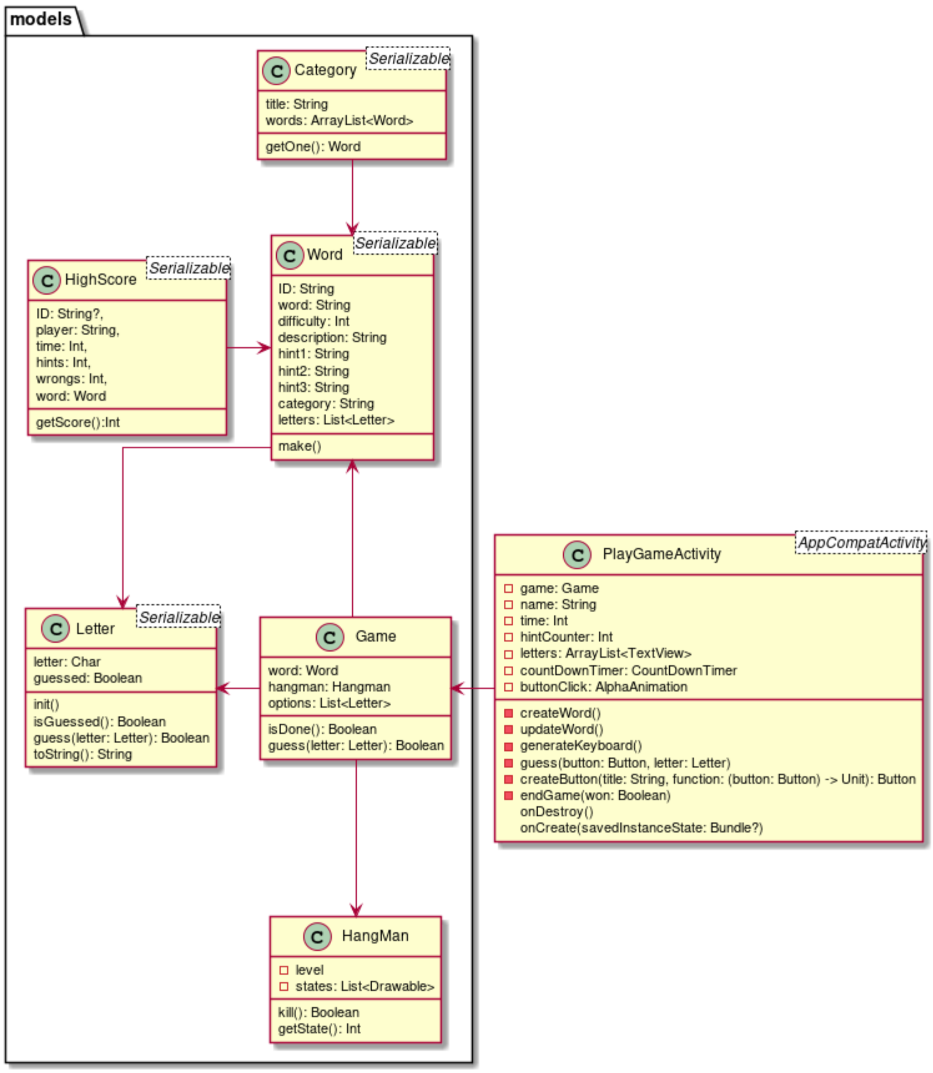
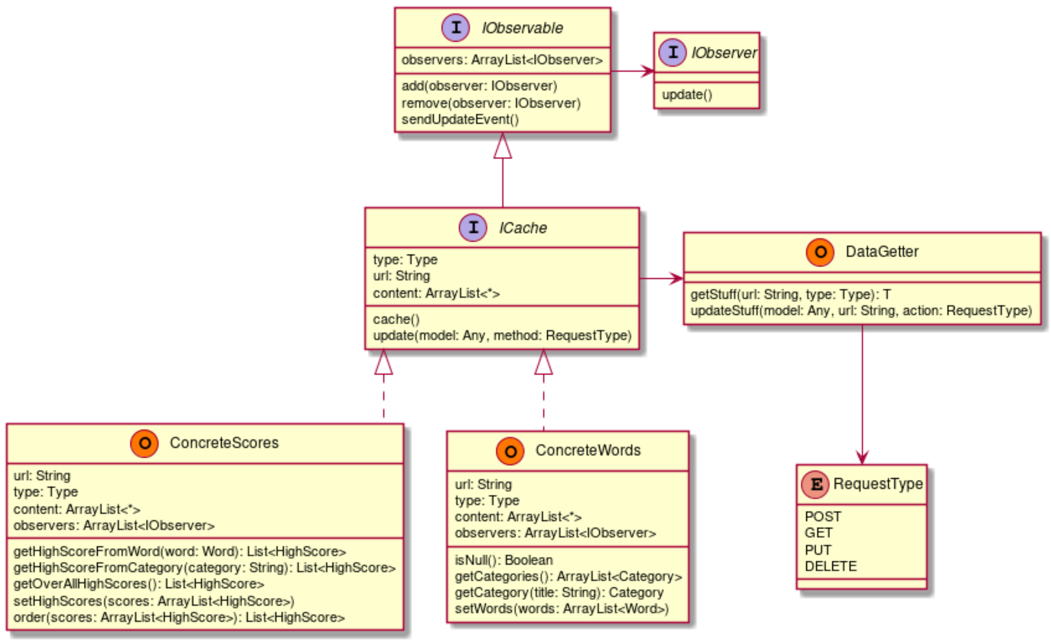


Figure 2: UML diagram of how the MVC pattern is implemented in the app

1.2 Observer Pattern + *Singleton* + *Abstract Factory*

The app is getting data from the internet using REST calls, and this leads to a lot of loading between activities.

A **singleton** class that holds the data and takes responsibility for making the REST calls can solve the problem of sharing the same data between multiple Activities in the app, and the **observer** pattern can provide the functionality, so that all the concerned Activities will be notified. Then they can update their views with the latest data avalible from the cache.



IObservable
observers: ArrayList<IObserver>
add(observer: IObserver)
remove(observer: IObserver)
sendUpdateEvent()
IObserver
update()
ICache
type: Type
url: String
content: ArrayList<*>
cache()
update(model: Any, method: RequestType)
DataGetter
getStuff(url: String, type: Type): T
updateStuff(model: Any, url: String, action: RequestType)
ConcreteScores
url: String
type: Type
content: ArrayList<*>
observers: ArrayList<IObserver>
getHighScoreFromWord(word: Word): List<HighScore>
getHighScoreFromCategory(category: String): List<HighScore>
getOverAllHighScores(): List<HighScore>
setHighScores(scores: ArrayList<HighScore>)
order(scores: ArrayList<HighScore>): List<HighScore>
ConcreteWords
url: String
type: Type
content: ArrayList<*>
observers: ArrayList<IObserver>
isNull(): Boolean
getCategories(): ArrayList<Category>
getCategory(title: String): Category
setWords(words: ArrayList<Word>)
RequestType
POST
GET
PUT
DELETE

Figure 3: UML diagram of how the Observer pattern is implemented in the app

The interface **ICache** implements **IObservable** which has an ArrayList of Observer who are implementing the **IObserver**.

In the app the Observers are the Concerned Activvities, and they subscribe to the relevant Subject in their onCreate method, they also unsubscribe in their onDestroy method.

The **ICache** is using the two static functions from the **DataGetter** utility. These functions are build in an abstract way which makes them capable of dealing with any models given to them.

The Datagetter is using ***RequestType*** *enum to decide which type of request should be used.*

ICache is also responsible for creation of the threads used for internet access.

Concrete singleton classes (or objects as they are called in kotlin) will implement ICache and thereby become an *Observable* and also able to use the DataGetter to create Lists of the models they are responsible of.

1.3 Other patterns

Android development is using a lot of different patterns, like for example the way an Adapter Pattern is used to connect a list of objects to a recycler view, and then do some magic in the background so that the app developer does not have to do it.