System Design Document for Qiwi Quiz

Carl Bergman, Erik Blomberg, Henrik Johansson,
Sara Persson, Louise Tranborg

October 23, 2020
Version 2

1 Introduction

Qiwi Quiz is a quiz application for Android phones. It is directed primarily towards Swedish children enrolled in elementary school. The questions are collected from the different subjects that they study, for example: mathematics and history. All questions use multiple alternatives and there is always one, and only one, choice that is correct.

While developing the application the most important goal was to plan ahead to ease development and make sure it is "future proof". Implementing additional features took a back seat to ensuring that the application remained extensible, maintainable and reusable, in other words "future proof". As long as the application follows these criteria, there will be no trouble to implement more than a thousand features and it will be possible to develop and maintain the application for all eternity.

This document describes the functionality and displays relations between different parts of the application. Furthermore, it reports how the quality of the code was ensured, and also how the application can be improved in the future.

1.1 Definitions, acronyms, and abbreviations

- **Activity** — Our application is built upon Activities, which is a hybrid between a View and a Controller. Every new section of the application is a new Activity, it presents a designated task for the user.
- **Android** — An operating system developed by Google for mobiles.
- **Design Pattern** — Solutions for common programming design problems.
- **GUI** — Graphical User Interface.
- **MVC** — Divide the program into three separate parts, Model, View and Controller. Used to avoid potentially dangerous dependencies.
- **Service** — General term for a modular package which is responsible for reading and storing data, works without the rest of the model knowing how data is retrieved/stored.
- **UML** — Unified Modeling Language.

2 System Architecture

2.1 Overview

Subjects are used to categorize the questions, they are used as a primary key to request questions from a service. Questions in the application are created by a uniform Question Interface, which integrates the ability to get a question text and a possibly unlimited number of alternatives. Questions store their alternatives in a static form, thus patterns can easily be found in the question alternatives. To combat this, the order that the questions appear, and their respective alternatives, are randomized before they are presented on the screen. The game logic, and how the questions are presented, differ depending on the chosen game mode. By contrast, the base gameplay—choosing the correct alternative in response to a question—does not depend on the current game mode.

2.2 Program Flow

The program launches into a title screen where the user can choose between three options, where two of them are: *High Score* and *Settings*. The *High Score* menu is used to view previous quiz results.The *Settings* menu is used to customize certain features to the players liking. The third option is *Play*. Upon selecting this option, the application will advance to another screen presenting a list of categories with subcategories, such as the category “Maths” with the subcategories “Addition” and “Subtraction”. Selecting a subcategory will result in the game starting.

During the game, the user is presented with a question and four alternatives to choose from. The player must now choose the correct alternative, which can be several or just one. After an alternative has been chosen, the alternatives will reveal whether they are correct or wrong. Depending on game mode, the player might, for example, be awarded points upon choosing the correct answer or loose time when selecting an incorrect answer. The process is now repeated until all questions have been answered, a result view is then displayed, where the user can view a summary of the amount of correct answers given. The results are stored and the player can now choose to play again or return to the title screen.

3 System Design

3.1 Overview

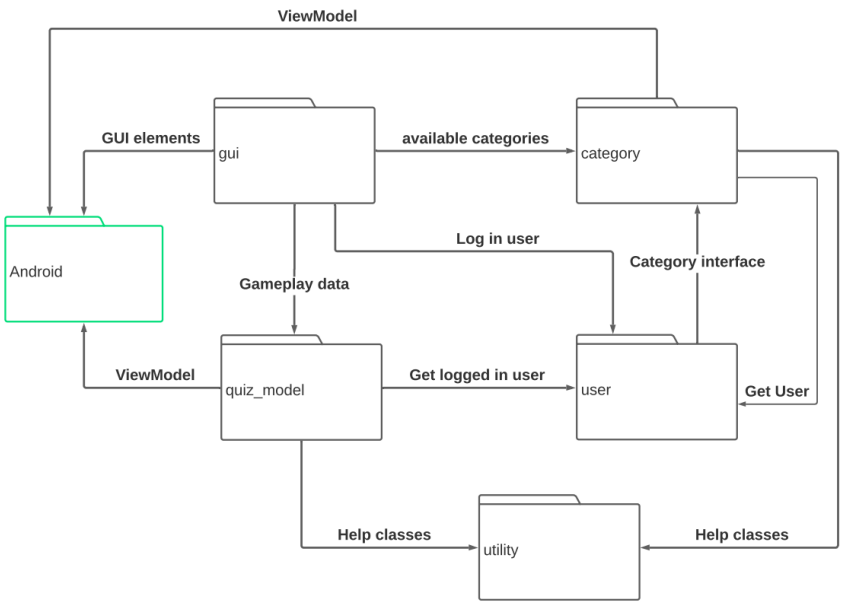

Figure 1: The top-level diagram, identifying the core packages

3.2 MVC Architecture

The program utilizes Android's recommended architecture, which is similar to standard MVC[1]. In Android, the *View* and *Controller* part is handled by `Activity` classes. These classes bind an existing XML layout and can then respond accordingly to user interactions. The *Model* part is handled by so called `ViewModel` classes, which can be used by `Activity` classes. The benefit for using this approach is for example when the user rotates the screen (ultimately destroying the `Activity`) all data is still intact, because `ViewModels` survive configuration changes. In this case the **gui**-package has all visual components such as `Activity` classes, **quiz_model**-package and **category**-package provide `ViewModel` classes.

3.3 Packages

Below are all packages shown in more detail, except for `gui` and `Android`. Subpackages and classes colored green implement components from the `Android` namespace. For a more comprehensive look, view the following diagrams: GitHub

3.3.1 Category

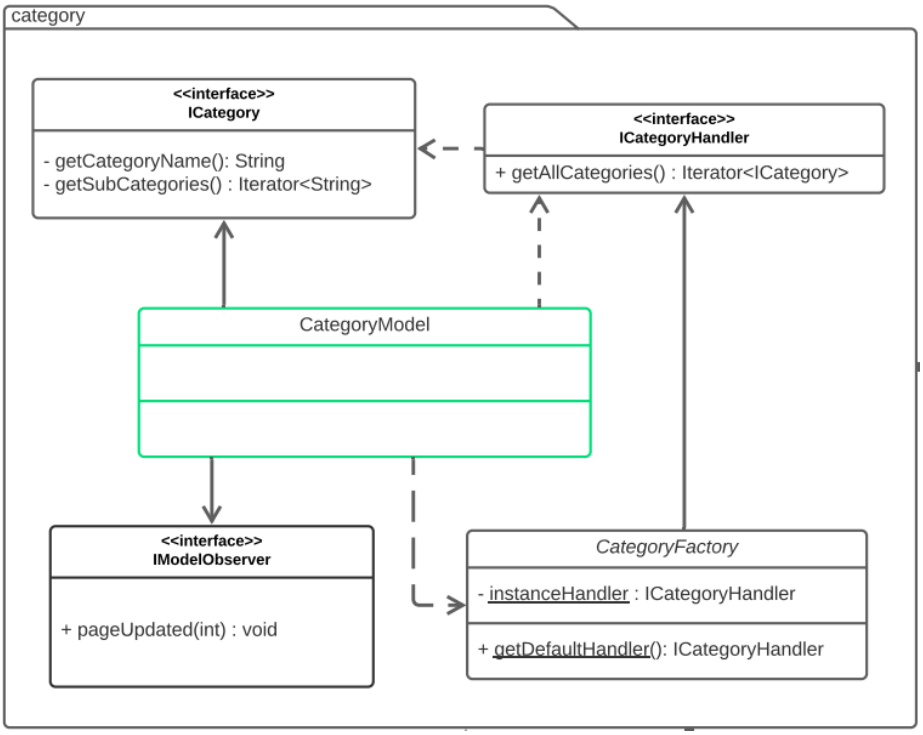

Figure 2: UML diagram showcasing the category package

The essential interface is `ICategory`, which represents a category and it's respetive subcategories. By combining this with the `CategoryModel`, which inherits the `ViewModel` class, the GUI can now showcase categories from multiple sources in a standardized way. The `CategoryHandler` interface acts as a uniform way to access a collection of categories no matter how they are gathered or stored.

3.3.2 User

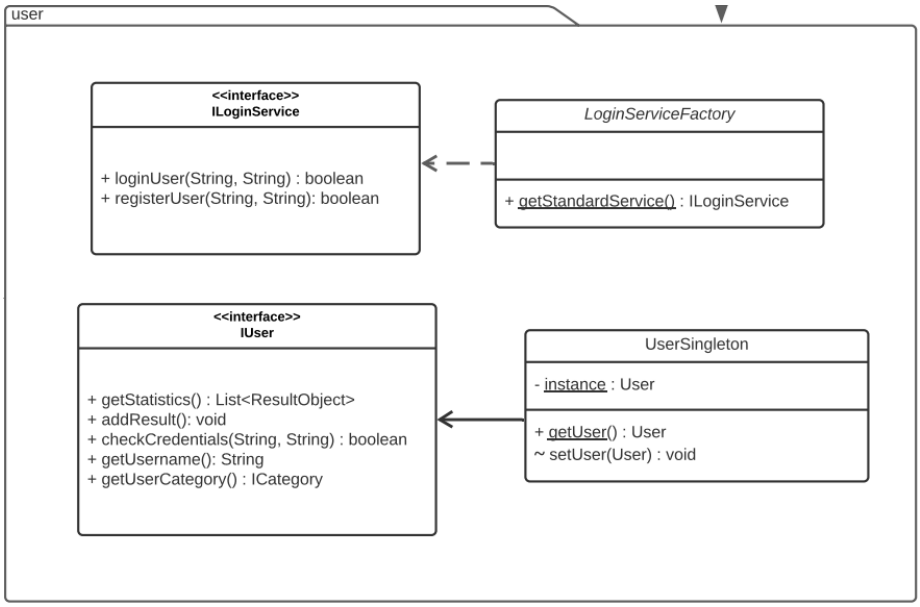

Figure 3: UML diagram showcasing the user package

The `ILoginService` interface is used to log in or register user, how the actually authentication process is conducted is hidden. After a successful authentication process the user is represented as an `IUser`, which enables statistics or custom categories as `ICategory` (see **Category**) to be stored. To make sure only one user is logged in and an easy access point for the rest of the application, an `UserSingleton` is implemented.

3.3.3 Utility

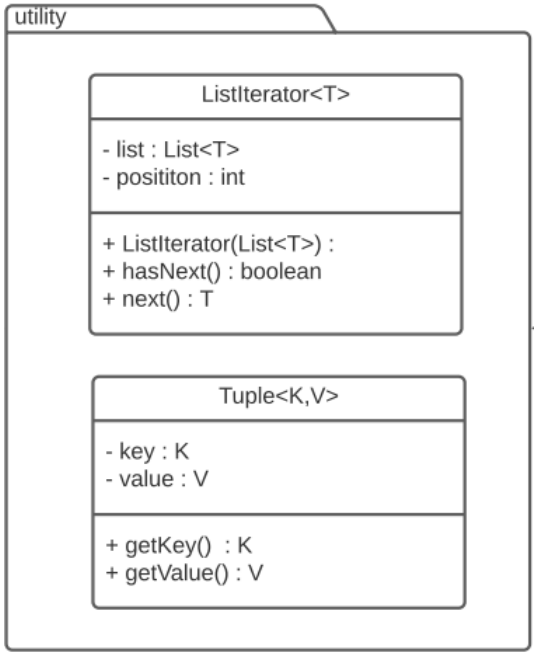

Figure 4: UML diagram showcasing the utility package

Only two classes in this package, however they act as general solution used throughout the model. The `Tuple` class is for example used to store question alternatives, a text component and Boolean to determine it's validity. The `ListIterator` class inherits the `Iterator` interface[2] and is similar to the iterators supplied by the `List` interface[3], however it has one crucial difference. The underlying list can change while another thread iterates over the collection without causing an run-time error[4].

3.3.4 Quiz Model

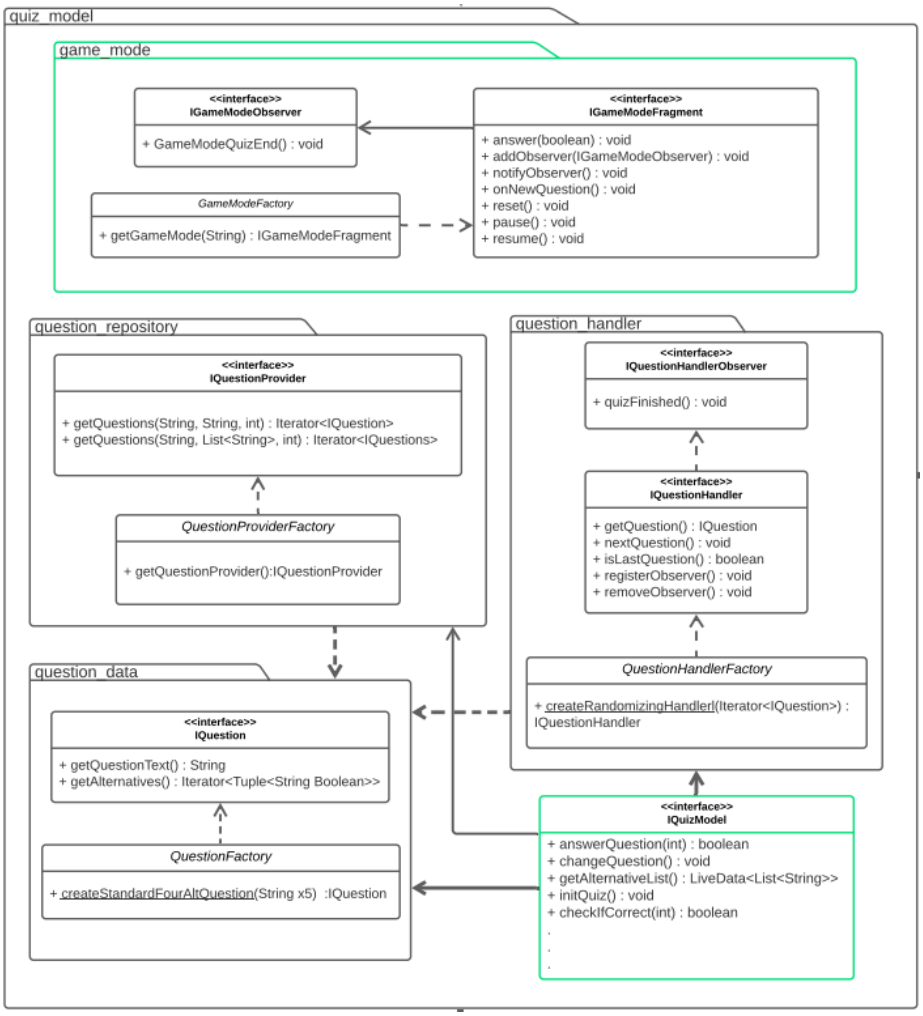

Figure 5: UML diagram showcasing the quiz model package

The core is the `IQuizModel` interface, which ties together all components. Subtypes of `IQuizModel` implements the `ViewModel` class, making it accessible by the GUI. All questions are defined by the `IQuestion` interface. The `IQuestionProvider` interface is utilized to fetch questions for the `IQuizModel`, by providing a category, subcategory(s) and the number of questions. `IQuestionProvider` was purposely designed to hide the internal details of how questions are stored and retrieved, to enable future expansions. Lastly, the `IQuestionHandler` interface is responsible to store and keep track of the questions during the game, handing out for example questions with

randomized alternatives.

3.4 Design Patterns

- **Iterator Pattern** — Used to future proof the application, if a data-structure changes, then the rest of the application is not affected. Can be seen in the file `ICategoryHandler` in **Category** and `IQuestionProvider` in **Quiz Model**.
- **Factory Pattern** — This pattern was used to promote loose coupled design, hiding the instantiating part of concrete classes. Can be seen in the **User** package and the **Category** package, specifically `LoginServiceFactory` and `CategoryFactory` respectively.
- **Facade Pattern** — Used to simplify and hide internal complexity of a module. Used together with factory pattern, to create services which can be seen in **User**, `ILoginService`, but also in **Quiz Model**, `IQuestionProvider`.
- **Observer Pattern** — A pattern utilized to notify 'listeners' about changes. Used with `IModelObserver` in **Category** package, `IQuestionHandlerObserver` in **Quiz Model** package, among other occasions in the application.
- **Singleton Pattern** — Utilized to make sure that only one instance can be created. Used in **User** package and only with the file `UserSingleton`.

3.5 Domain and design model

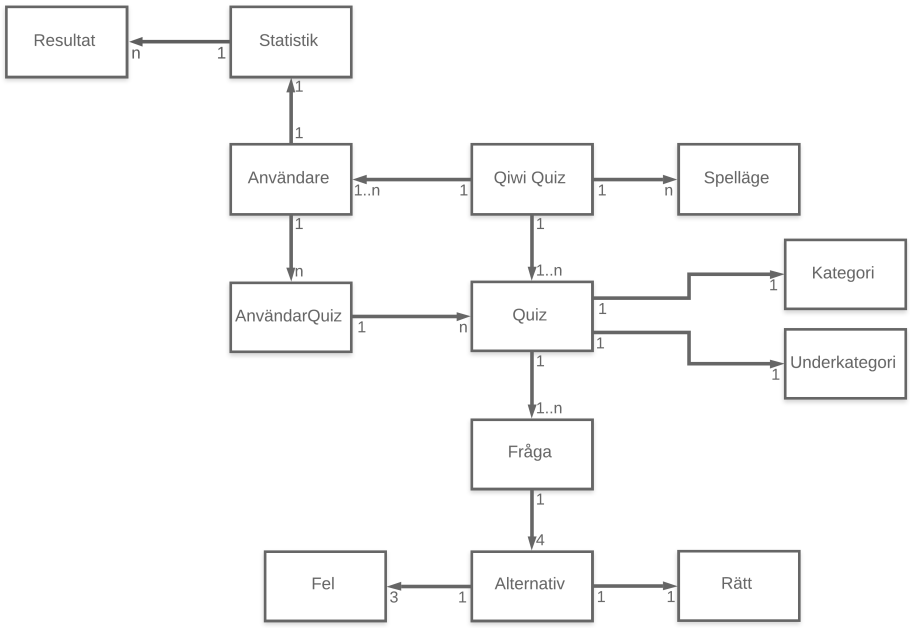

Figure 6: Domain model of the application

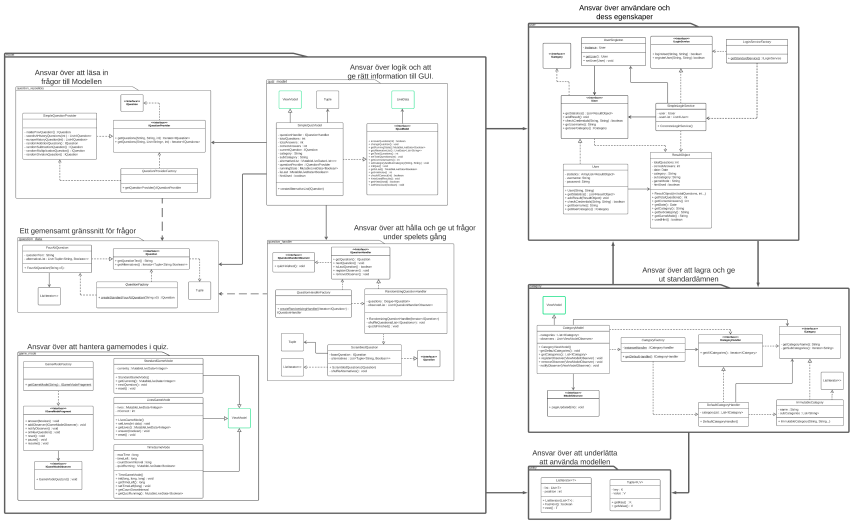

Figure 7: Design model of the application

The relation between the domain model and the design model of the application is clearly visible by looking at "Qiwi Quiz" in the domain model and the `SimpleQuizModel` class in the design model. We can see that "Qiwi Quiz" has at least one Quiz. These quizzes are represented by `RandomizingQuestionHandler` in the design model, which has a Deque with questions. These questions, in turn, contain a list with four alternatives, where one is correct and the rest are incorrect — directly corresponding to the domain model.

Additionally we can see that "Quiz" in the domain model has a "Category" and a "Subcategory", these are represented by 2 strings within the `SimpleQuizModel` class. "Qiwi Quiz" also has a user, represented by a User object accessible via a service. These users has statistics, consisting of a list of Result objects. In the domain model there are "n" amount of Results, which matches with the list that can be empty or filled with n-amount of results. The "User" also has "UserQuiz"s, these quizzes work the same way as the regular quizzes explained above.

Lastly, in the domain model there is also GameMode, represented by the `game_mode` package in the design model. Although not directly working with `SimpleQuizModel`, it is used by the Activities that present the quiz to the user.

4 Persistent data management

The application makes some use of persistent data, there are however occasions where data is not saved in a way that makes it remain on application restart. This is not a conscious choice but it is due to time constraints while developing the app. As there was not enough time to implement persistent data saving completely, the service-pattern is used whenever necessary to accommodate for databases and other ways to save relevant data.

In the current version of the application there is no way to save user-profiles. To keep the same user while the application is active, the active user instance is kept in a class called `UserSingleton`. This makes it possible to reach the user wherever necessary in the application. Note that it is *not* the actual user that is the singleton, as it should be easy to swap out one user for another.

For the images and strings that are used throughout the application, they are stored in the native android directory `src/main/res`. In this directory the android application can reach saved strings in `strings.xml`, images in the `drawable` subdirectory, and many more types of data in other xml files and subdirectories.

To save data throughout the different activities, there are two methods used primarily. Method one is the usage of viewmodels, the relationship between viewmodel and activity is further described above, but the most important part is that an android viewmodel is conscious of the activity lifecycle. That means that even if an activity is destroyed it will remain until the activity calls the built in method `finish()`[5]. The other method is used when there is a need to use data between activities, in that case the built in `SharedPreferences`[6] is used. This saves data with a key and is accessible wherever there is a context, i.e. Activities and Fragments.

5 Quality

When creating the application the aim was to have 100 % of the model tested, and the tests cover close to that. These unit tests can be found in the directory `src/test/java/com/down_to_earth_rats/quiz_game` from project root, and the easiest way to run the tests are by launching Android Studio, right clicking on `src/main/java/com/down_to_earth_rats/quiz_game/quiz_model`, and selecting the option to run the tests with coverage. This will run all the tests for the model and display how much of it that is covered by the tests.

STAN[7] was not able to run properly on any of the computers that it was tested on, and thus there cannot be any generated dependency graph included.

There are no found issues with the application during run-time, but there are things that could be improved or changed, further on this in 5.2 below.

5.1 Access control and security

There is a login-feature in the application and it is required for a user to be logged in for the application to function. At the moment it is just a temporary login and the only user that the application remembers is one that has its credentials written directly onto the code. There are therefor no differentiation between different users, and all logged in users have the same authority.

The login feature is very simple because of time constraints during development, it is however treated as a service, so it is easy to swap out the current login-implementation with something more complex.

As described in the section above, the user is kept within a `UserSingleton` that all relevant parts of the program can reach. There is only support for one user to be logged on at all times, the user logs in on startup, and is logged out upon application finish.

5.2 Improvements

Below are some improvements in the design and code of the application that would have been done if there were more time available:

- Move the `game_mode` package outside the `quiz` package. This because while it does handle parts of the quiz, it is not connected to anything else in that package.
- `UserSingleton` doesn't actually behave as a singleton. At the moment it has static functions and attributes and is never instantiated. The improvement here would be to implement it as an actual singleton, or rename it to something that better describes what it is.
- Removal of the `utility` package. This due to that the generic classes in it are only used in specialised cases so there is no need for them to be generic. As well as that the created `ListIterator` can be replaced with a built in iterator.
- The complexity of the viewmodel for `QuizActivity` is quite high. The viewmodel should be derived into a model part as well as a thinner viewmodel part, where the model handles more logic.
- Removal of the functionality that displays whether a new question or the result will be showed next during a quiz answer. This is already removed from the UI because the game modes make it clear if there are questions left. The removal of this functionality from the code would reduce complexity in many parts of the quiz model and it is currently unnecessary.
- Reducing the amount of attributes in `QuizActivity`. Changing attributes to local variables and moving some to the viewmodel would reduce complexity and make data persist even if Android decides to shut down the activity. Some of the previous improvements would already reduce them somewhat but there are several that are not covered by previous examples.

- Towards the end of development there was a circle dependency found between the `user` and the `category` package. Additionally, the problem `CategoryHandler` is trying to solve, a uniform way to access a collection of categories, is virtually non-existent and a refactoring is much needed in this part. This should be corrected.
- The use of Observer patterns is questionable in many cases. Polling techniques could be used instead, like in `QuestionHandlerObserver`. Another issue is that each Subject implement their own observer version, a more general interface would satisfy both DIP and OCP. This also should be corrected.

References

[1] Google Developers. (2020). Guide to app architecture, [Online]. Available: https://developer.android.com/jetpack/guide#overview (visited on 2020-10-21).

[2] Oracle. (2020). Iterator interface, [Online]. Available: https://docs.oracle.com/javase/10/docs/api/java/util/Iterator.html (visited on 2020-10-21).

[3] Oracle. (2020). List interface, [Online]. Available: https://docs.oracle.com/javase/10/docs/api/java/util/List.html#iterator() (visited on 2020-10-21).

[4] Oracle. (2020). Concurrentmodificationexception, [Online]. Available: https://docs.oracle.com/javase/10/docs/api/java/util/ConcurrentModificationException.html (visited on 2020-10-21).

[5] Google Developers. (Aug. 2020). Viewmodel overview, [Online]. Available: https://developer.android.com/topic/libraries/architecture/viewmodel (visited on 2020-10-21).

[6] Google Developers. (Sep. 2020). Documentation: Sharedpreferences, [Online]. Available: https://developer.android.com/reference/android/content/SharedPreferences (visited on 2020-10-21).

[7] Bugan IT Consulting UG. (2017). STAN — Structure Analysis for Java, [Online]. Available: http://stan4j.com (visited on 2020-10-21).