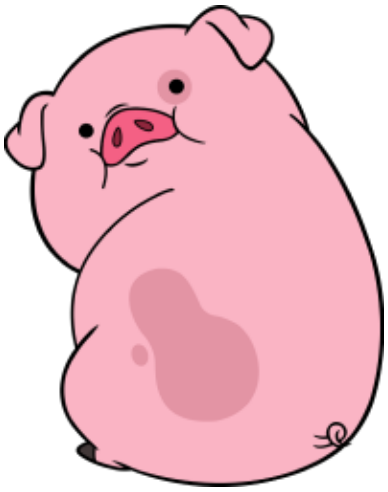
System Design Document
for
FlashPig

Jesper Bergquist, Wendy Pau, Madeleine Xia and Salvija Zelvyte

October 23, 2020



Contents

1 Introduction

FlashPig is an Android study application that is mainly designed for students but will work for any other user that want to learn new information in a quick, effective and fun way.

The System Design Document will describe the implementations of FlashPig including the system architecture, implementation logic, design and many more.

1.1 Definitions, acronyms, and abbreviations

- **Flashcard** - Consists of a two-sided card where each side holds information through either some text and/or images. The user can choose whether to show the frontside or backside of the card and then flip the card. The user can for each card choose what difficulty (easy, medium, hard) they thought about the card. This information will be used to show a "Flashcard Progress" which will be accessible from the dashboard.
- **Pair Up** - Pairs two cards with each other from a deck as the player studies the information on the card.
- **Memory** - Memorisation game that trains the players memory as well as studying the subject of the chosen deck.
- **Java** - A platform independent programming language the application is made of.
- **GUI** - Graphical user interface.
- **UML** - Unified Modeling Language. UML is used for visualising the design of the system, both on high level and low level.
- **MVVM** - Model-View-ViewModel. A software architectural pattern used in FlashPig.
- **User** - Any user of the FlashPig application.
- **Flashcard Progress** - Allows the user to follow the progress for a chosen deck.
- **Deck** - A collection of cards that belongs together.
- **Spaced repetition** - An algorithm to handle the frequency for a specific card to show up based on the users previous experience with the card.
- **UI** - User interface.
- **LiveData** - An observable data holder class. Unlike a regular observable, LiveData is lifecycle-aware, meaning it respects the lifecycle of other app components, such as activities, fragments, or services. This awareness ensures LiveData only updates app component observers that are in an active lifecycle state.

2 System architecture

The application is written in Java in Android Studio. The android application implements the concepts to make the components of the app reusable, extendable and maintainable. For instance, it has a MVVM structure that uses Android architecture components for managing the lifecycles of the UI components. One example is by implementing View Model that stores the UI-related data.

Moreover, LiveData has been implemented in order to handle data persistence by notifying the view when database changes. The code, especially in the view Model classes, uses mutable LiveData which is a wrapper that can hold any type of data and is accessed via a getter method. The View will observe the state of the data in the ViewModel class using a LiveData. Note that a real database has not been implemented, more about how Flashpig handles data persistance in described under the "Life-cycle and Data Management" section.

The app uses activities to represent various interactable screens where every screen is a fragment. Like Activities, a fragment has a lifecycle with its own layout and behaviour. In addition, the app has a set of views containing the user interfaces. Each important component of the application will be described in this document.

2.1 Navigation and high-level application flow

The main activity is the entry point of the application and it represents the different screens the user interacts with. When the activity has been created in the onCreate() method, a splash screen will be shown before displaying a dashboard screen. The user uses the dashboard to navigate to the different parts of the application. Each of the parts in the dashboard has its own activity with its own fragments.

The navigation in the app is mostly a linear process after entering a new activity from the dashboard. However, you can always go from the Flashcard activity back to the dashboard without having to finish a round. The navigation will be described more in detail in the sequence map further down.

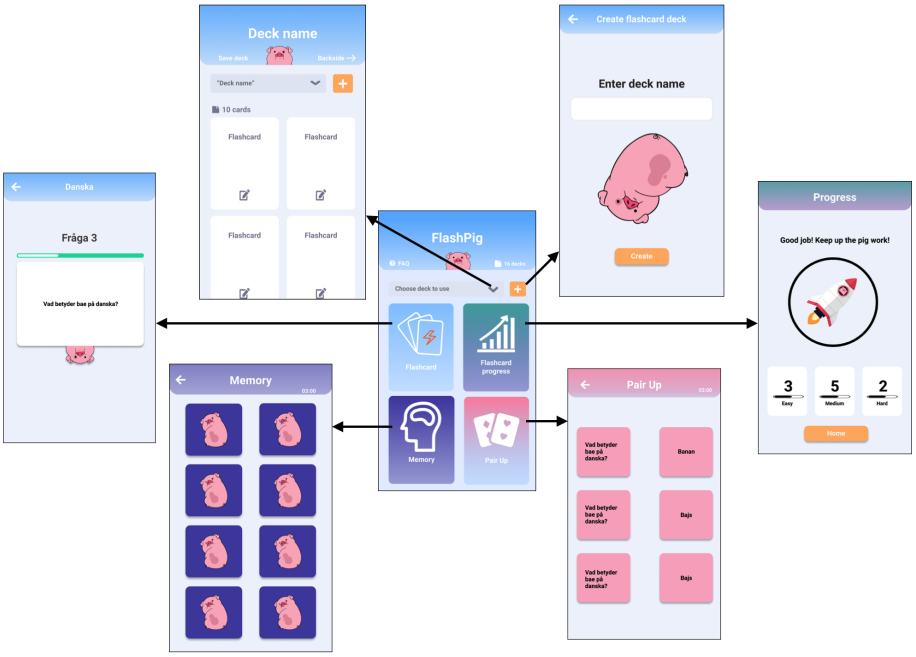


Figure 1: Naviation from the Dashboard to the application's different screens

3 System Design

The following section will discuss the structure of the code and describe the application's components on different levels.

3.1 Software Architectural Pattern

A Model-View-ViewModel structural design pattern is implemented in the application. The pattern has three components: the Model, the View, and the ViewModel. A MVVM structure implies that the View are aware of the ViewModel and the ViewModel are aware of the Model. However, the Model is unaware of the ViewModel and View while the View Model is unaware of the View. In this way, the Model can evolve independently since the View Model isolates the View from the Model.

Using MVVM is more suitable for an application with more than two screens since it breaks down the code into modular, single purpose components as well as adding complexity to the code. The traditional MVC structure set some limitations on the implementation of the application in comparison to the MVVM structure. One disadvantage with MVC is that the Model updates the view but as the amount of views expands the difficulty with knowing which view has been

updated increases.

In addition, the controller updates the Model which updates the View resulting in over-dependence of View on the Controller. This is one example of many why MVC does not make an appropriate architecture for android applications. Therefore, FlashPig was implemented with a MVVM structure instead of MVC. The latter sections will describe the MVVM components responsibilities.

3.1.1 Model

The Model package holds the application data.

3.1.2 View

The View displays an representation of the model and is what the user sees on the screen. It receives the users interaction with the view and forwards it to the viewModel via data binding.

3.1.3 ViewModel

The ViewModel transforms the Model information into values that can be observed by the View. The View Model has no reference to the View.

3.2 Dependency analysis

Dependencies can be found in the appendix.

3.3 Application decomposition

The application is broken down in packages and classes. This section will describe them.

3.3.1 Package Diagram

Flashpig is a standalone application that uses it's own "fake" database, which makes the top level package diagram very small (making the FlashPig application the only component in the package diagram). However, the lower level package is shown in Figure 2. The diagram shows that the Model package does not have any references to the other packages and the View is updated by the View Model that has a reference to the Model. In other words, the View classes has instances of the View Model classes and the View Model classes has

instances of the Model classes and this is the way the View and Model communicates. The DataBase holds a reference to the Model and the View Model has access to the DataBase. In conclusion, the key factor of following the MVVM pattern is to separate the View from the Model, which is shown in the diagram.

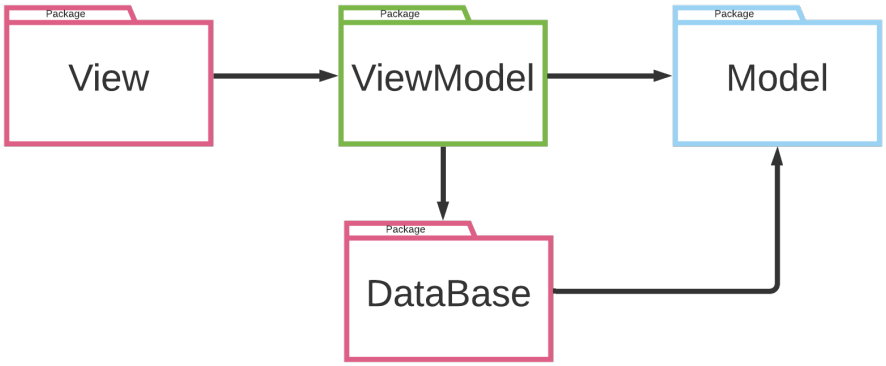


Figure 2: MVVM

The main packages used in this project is listed in Figure 3, more in depth information about the dependencies and content between and in the packages are shown in the appendix. The following list describes the package's responsibilities.

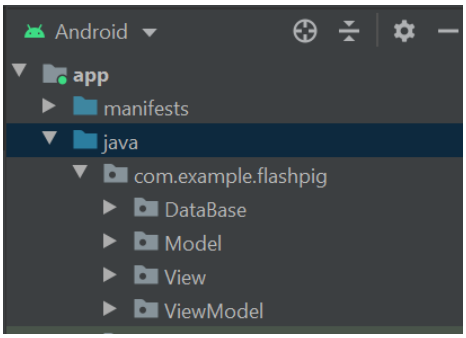


Figure 3: Packages used in the project

- **DataBase** holds all the hard coded decks and cards that are created.
- **Model** consists of all the data and logic, for instance the class Flashpig.
- **View** includes all the fragments and adapters.
- **ViewModel** includes all the ViewModel classes.

The domain and design model of the application differs a little, but are still very similar. As illustrated in Figure 4 the DashBoard used in the Domain

model has the role as the GameLogic class in the Design model. Another disproportion between these diagrams is the FlashCardProgress class. At first the main responsibility for this progress property was to make it accessible for all the mini games but during the development of FlashPig it was noticed that the only required dependency for the FlashcardProgress is with the Deck class. Otherwise, all the other classes and dependencies in these diagrams is exactly the same.

UPDATE MODEL UML IMAGE!

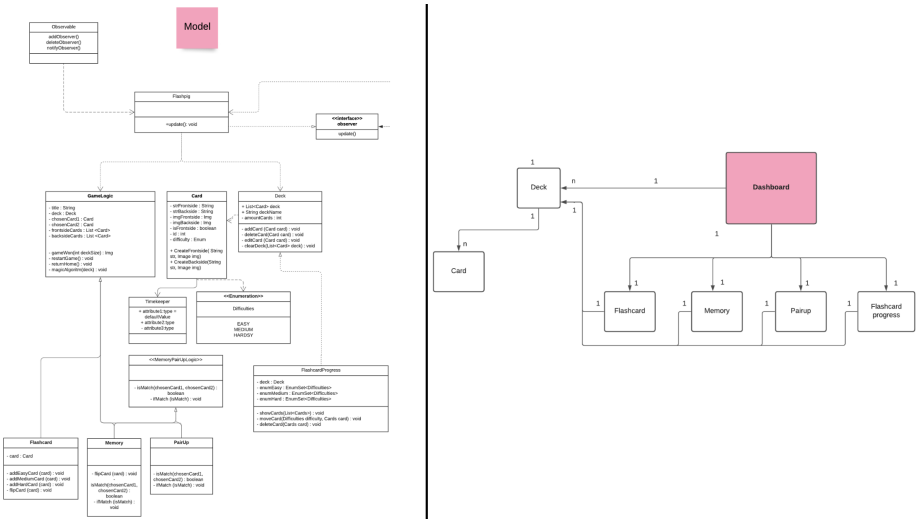


Figure 4: Design versus Domain model

3.3.2 Class Diagram

ska skrivas mer här (Sage)

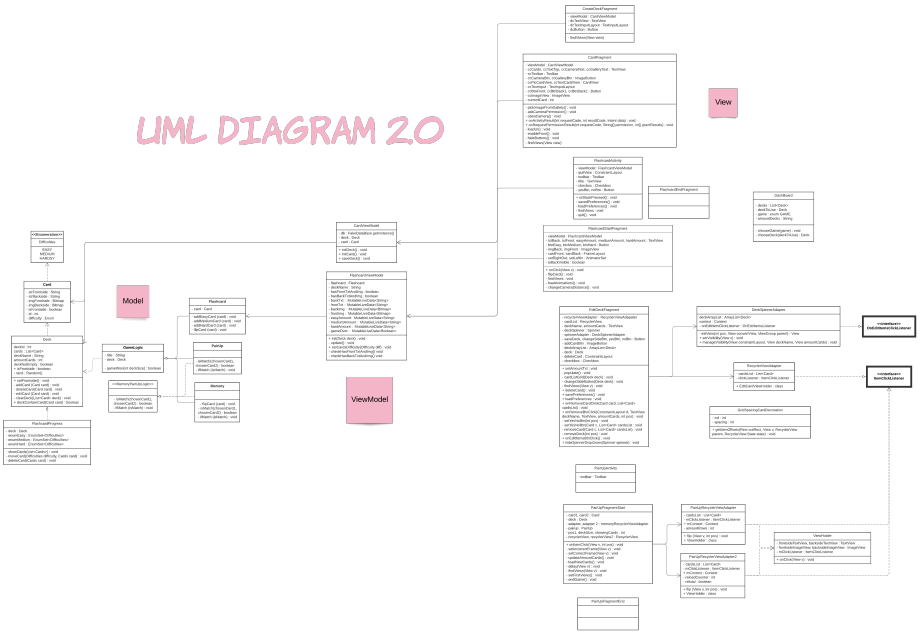


Figure 5: UML Class diagram

3.4 UML sequence diagram

The Figures 5-8 illustrates some use cases in the application in form of sequence diagrams. The chosen use cases are "Play Flashcard", "Play Pair Up" and "Edit a deck".

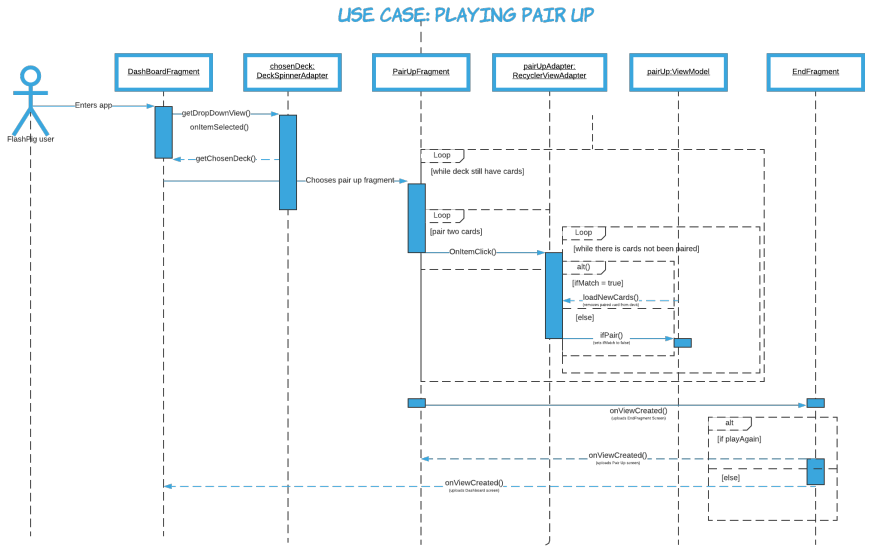


Figure 6: UML Pair Up sequence diagram

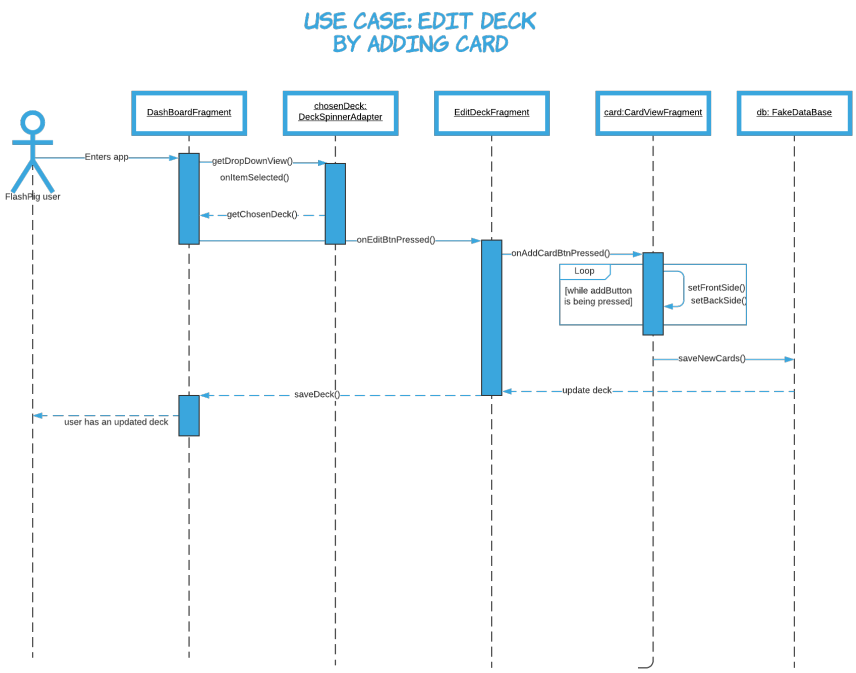


Figure 7: UML EditDeck sequence diagram

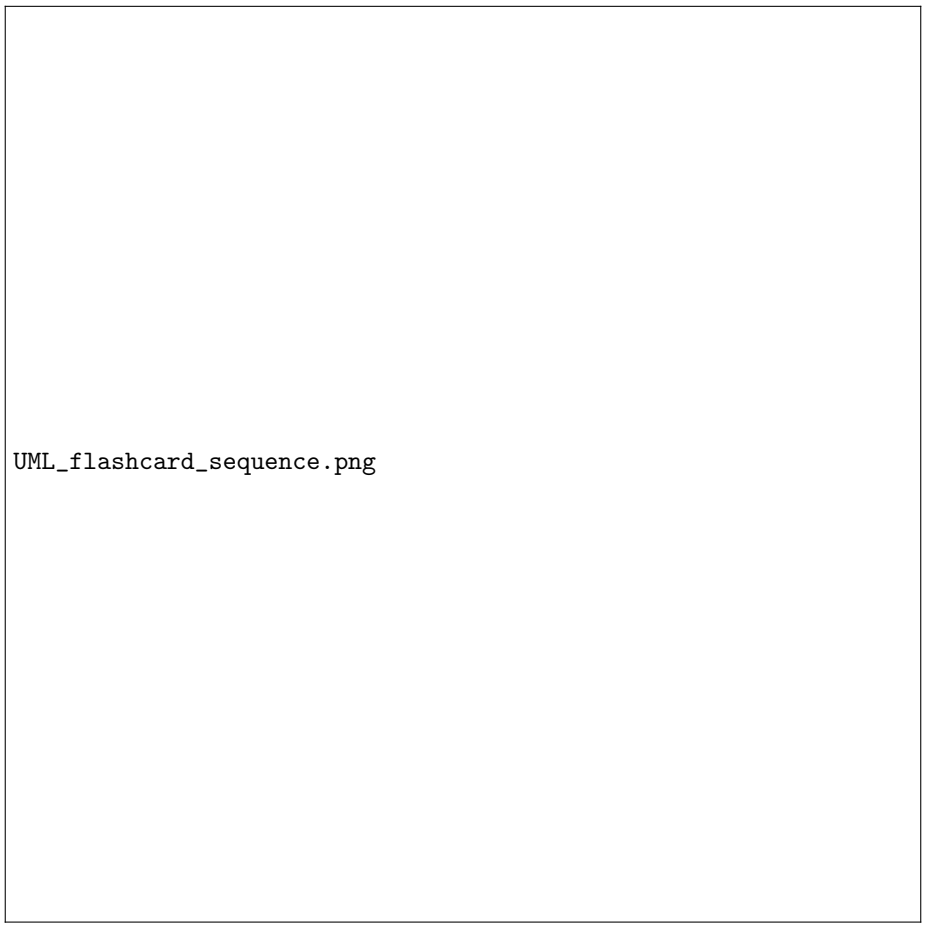


Figure 8: UML Flashcard sequence diagram

3.5 Design Patterns

In order to make the application reusable, maintainable and extendable, relevant design principles and design patters have been implemented. Firstly, the classes in the code follows the Single Responsibility Principle which makes the code more robust and simplifies testing of the code. Following the principle also makes the code more easy to access specific logic since it is gathered together for instance in the same class or method.

In Android Studio all of the logic and behaviours can be easily gathered together in the activates and fragments in theory, however this does not support the Separation of Concern principle. Hence, the code have been implementing the MVVM pattern to make the code better suited for changes in the future.

The code have been developed with other common principles in mind such as High Cohesion, Low Coupling for making the code more flexible, Command-Query Separation to avoid unexpected results and so forth. The principles work together to make the code easier to test, reuse, maintain and extend. The sections down below will discuss the most relevant implemented patterns.

3.5.1 Singleton Pattern

The code includes an implementation of the Singleton Pattern for the Repository class. This is not the optimal solution for FlashPig since the Repository becomes accessible from multiple objects resulting in higher risk for unexpected side effects. The side effects can be challenging to identify and manage, making the Singleton Pattern less safe and the code harder to maintain. Moreover, using the pattern will make testing significantly harder since it is not possible to isolate classes that it is dependent on. Unfortunately, due to the fact that other code bugs when trying to replace the Singleton Pattern by making the repository class parcelable lead to a non-executable code, the Singeton Pattern had to stay. However, it is possible to refactor in the future to improve the code.

3.5.2 Observer Pattern

The pattern comes along with the import of androidx.lifecycle package. The Observer pattern in Android is used to observe LiveData which is an observable data holder. It is the View that observes the changes in the ViewModel through LiveData. Since LiveData is aware of the lifecycles of other app components, it will only automaticaly notify the observers if they belong to an active lifecycle state.

3.5.3 Model-View-ViewModel Pattern

This pattern is described in the "System Design" section under the subsection "Software Architectural Pattern".

3.5.4 Adapter Pattern

The code has a subclass of "RecyclerView.Adapter" which is a container for the UI. The adapter manages and creates the View Holder objects, the View Holder is a list of views. Moreover, the adapter bind the view holder to its respective data by assigning it to a position.

4 Life-cycle and data management

FlashPig implements a fake "database" which holds hard coded data. This is a temporal solution and can be easily replaced by a real database without having to make a huge refactor the code in the future. The data is stored in DataBase class containing a list of decks and cards and method to add and remove decks.

Parcelable is used for persistent data management and is Android's version of Java Serializable. It is a way of communicating data with different system components. Using a binder to facilitate the communication is an effective way of procressing data. The Parcels is a message container in which the Binder communicates with.

Other resources such as icons and images are stored as Drawable resources which is a concept for graphics that is visuals on the screen.

Life-cycles management

TO DO

5 Quality

The following tools listed below are used for facilitating maintenance and development of the code.

5.1 Git

A distributed version-control system to track changes. Moreover, Git simplifies coordinating work among many programmers and supports non-linear workflows. The RunnableCode branch holds the main build of the application. Sprint features from the dev branch will at the end of a sprint be merged into RunnableCode. Finished feature branches created for respective feature will in turn be merged into the dev branch.

5.2 JUnit

For testing the model, it covers tests of the model which are located in a test package.

5.3 Travis

Travis is used for continuous integration. Due to the access Travis has to the Git repository, it will recognize any commits or pull requests and run tests for them. The results will be forwarded via mail, through Github or in a discord channel.

5.4 Javadoc

To make the code more understandable and readable for others, comments on the classes, methods and interfaces have been written.

5.5 PMD

A source code analyzer that identifies unused variables, unnecessary object creations, empty catch blocks, and so forth.

5.6 Stan4j

An Eclipse-based structure analysis tool for Java that helps with understanding the code, finding design flaws and measuring quality by building a visual representation of the code.

5.7 Tests

Describe how you test your application and where to find these tests. If applicable. Run analytical tools on your software and show the results. Use for example: – Dependencies: STAN or similar. – Quality tool reports, like PMD. **NOTE: Each Java, XML, etc. file should have a header comment: Author, responsibility, used by ..., uses ..., etc.**

5.7.1 Known issues

- **xx**
- **xx**
- **xx**
- **xx**
- **xx**

5.8 Access control and security

NA

6 References

Figma Design
`figma.com/file/FVwpn4oMpd4NVaKKtvg2qP/Startpage?node-id=0%3A1`

FlashPig GitHub
`github.com/Wendaaj/FlashPig`

FlashPig UML
app.lucidchart.com/invitations/accept/27bf8038-df7c-486f-af46-f8c23bb2126d

Parceler
github.com/johncarl81/parceler

Git
git-scm.com/

JUnit
junit.org/

Travis
travis-ci.org/

PMD
pmd.github.io/

Design Patterns
refactoring.guru/design-patterns

MVVM
docs.microsoft.com/en-us/xamarin/xamarin-forms/enterprise-application-patterns/mvvm

Stan4j
stan4j.com/

7 Appendix

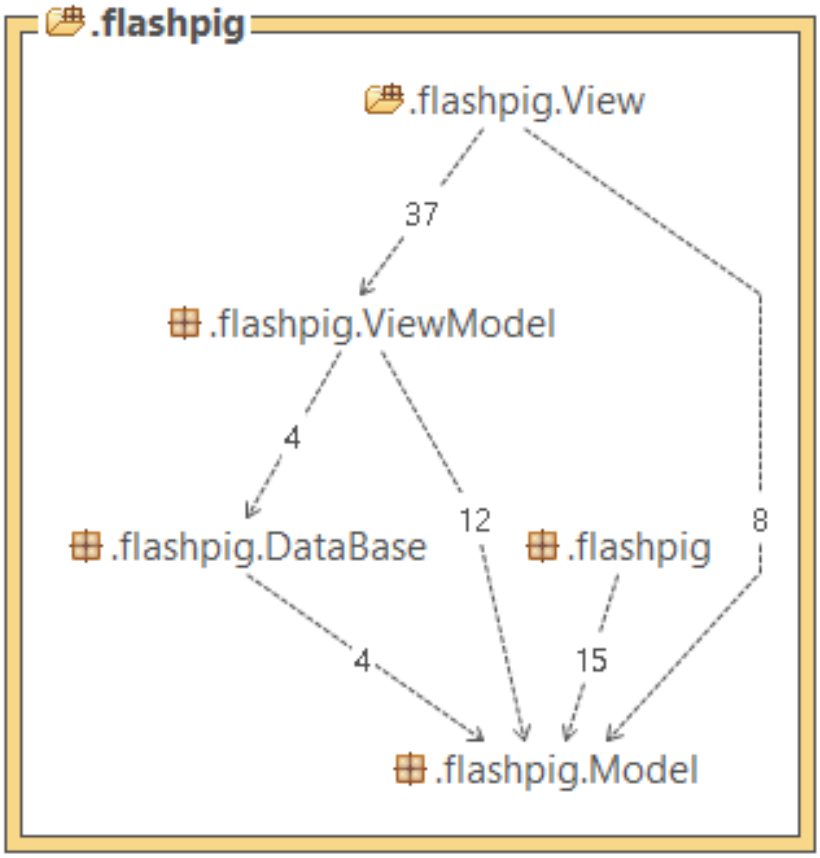


Figure 9: The overall project structure.

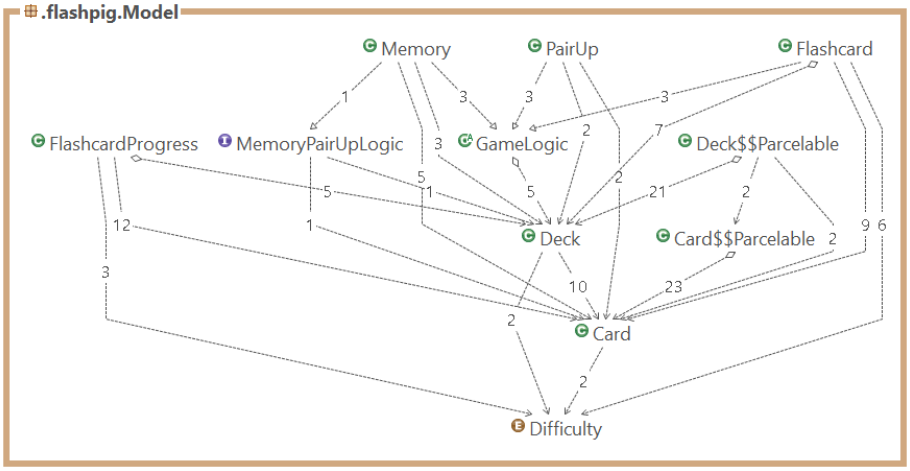


Figure 10: The structure in Model package.

Figure 11: The structure in View package.

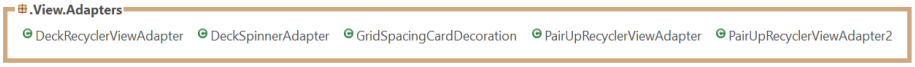


Figure 12: The structure in Adapters package.

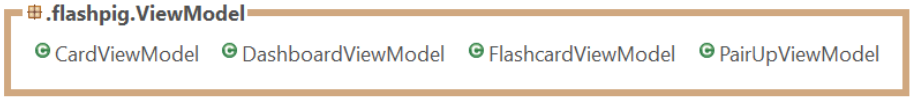


Figure 13: The structure in ViewModel package.