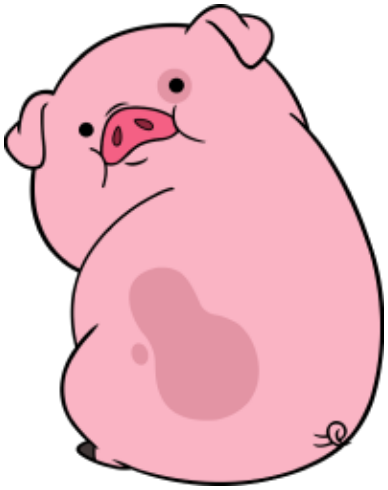
Final report

Jesper Bergquist, Wendy Pau, Madeleine Xia and Salvija Zelvyte

October 23, 2020



Contents

Section 1 - 5 concerns RAD

1 Introduction for RAD

FlashPig is an application that provides the opportunity to create flashcards and use them to study and learn in three different ways.

The first way is the game "Flashcard" which consists of two-sided cards, where one side holds a question while the other holds the answer. Both sides can be appeared as an image or text. After each card the user has to decide how difficult the card was (easy, medium or hard). This makes the learning progress visible and accessible for the user. Furthermore, depending on what difficulty the user chose the cards will be appear in the game in different intervals of time which is based on an algorithm, giving this study technique a spaced repetition.

Moreover, there are two other games named "Memory" and "Pair up". "Memory" works exactly like the classic memory game where in this case you have to combine the front- and backside. On the other hand, "Pair up" displays all the cards information and gives the user multiple possible answers to match the associated card. Therefore, the front sides (questions) are located on the left column and backsides (answers) on the right.

The application 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.

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 front- 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.

- **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.

2 Requirements

2.1 Implemented User Stories

The following list is the User Stories for the application which has been implemented.

2.1.1 Functional Requirements

1. **As a user, I want to be able to create new decks of cards to gather information that I want to learn.**

   Estimated time: 2 days.

   (a) Design a GUI to create new decks.
   (b) Be able to add how many cards the user wants.
   (c) Give each deck an id and let the deck know how many cards it have.

   Acceptance criteria:

   - When the user can create a new deck and name it.

2. **As a user, I would like to play Flashcard so that I can study in an effective way.**

   Estimated time: 4 days.

   (a) Design an GUI for the Flashcard game.
   (b) Connect a deck to the game.
   (c) Ability to choose each card's difficulty (easy, medium, hard).
   (d) Be able to return to the game where the user left it.
   (e) Create a popup message to check if the user wants to leave the game.
   (f) Create a progress page where the user can see its performance in Flashcard. (Unimplemented)

Acceptance criteria:

- Can iterate through a deck.
- Can flip the card to show the backside.
- When the game saves after one round (even after unfinished round).

3. **As a user, I would like to edit my decks that I have already created.**

   Estimated time: 3 days.

   (a) Design a GUI to be able to edit decks.
   (b) Be able to delete decks.
   (c) Be able to change the cards text.
   (d) Be able to add/delete pictures in the cards.
   (e) Be able to create new cards to a deck.
   (f) Be able to change a decks name.

   Acceptance criteria:

   - When a deck is editable after its creation.

4. **As a user, I would like to play Pair Up to learn information in a playful way.**

   Estimated time: 4 days.

   (a) Design a GUI for the Pair up game.
   (b) Connect a cards back/frontside with each other.
   (c) Define when the game is over.

   Acceptance criteria:

   - The user can see when they clicked wrong or correct.
   - The user can see when the game is over.
   - The user can pair 2 cards together.

2.1.2 Non-functional requirements

1. **As a user, I would like the application to be secure to use so that I can feel secure while using the application.**

   Estimated time: NA.

   (a) Be able to choose if a deck should be public or private,

(b) **Ask about access from the user when the application access to the users camera, gallery etc. before use.**

Acceptance criteria:

- The application ask for permission to access to the users information beforehand.
- When the application is optimised for security.

2. **As a user, I would like the app to be user friendly to avoid spending time on learning how to navigate in the application.**

   Estimated time: NA.

   (a) Implement help system designs or add "first time"-tutorials.
   (b) Make the design intuitive to use with help of icons, images etc.
   (c) Implement frequently used design patterns.

   Acceptance criteria:

   - Implements help systems.
   - Implements relevant design patterns.
   - The application acts as expected.

3. **As a user, I would like the application to be beautiful and therefore give a better user experience.**

   Estimated time: NA.

   (a) Use suitable font, colours and shapes in the GUI.
   (b) Use uniform colours in the application.
   (c) Create FlashPig sprites.

   Acceptance criteria:

   - The colours are balanced.
   - The applications appearance are in line with the theory about how a good interface are.

2.2 Unimplemented User Stories

The following list is the user stories for the application that are yet to be implemented. The concerning user stories are extended functionality for the application that improves the user experience.

2.2.1 Extra functional requirements

1. **As a user, I would like to play Memory so that I can train my memory.**

   Estimated time: 4 days.

   (a) Design an GUI for the Memory game.

   (b) Connect a cards back/front-side with each other.

   (c) Create a logic where only 8 cards are shown in a time.

   Acceptance criteria:

   - The user can flip the card.
   - The user can see how many pairs that‘s left under game.
   - The user can see when the game is over.
   - The user can see how much time it took for them to finish the game.

2. **As a user, I would like to be able to share my deck with other people.**

   Estimated time: 4 days.

   (a) The user shall be able to choose if the want to create public or private cards.

   (b) Create a downloadable link to a deck.

   Acceptance criteria:

   - The user can share its deck with other people.

3. **As a user, I would like to have access to others public decks.**

   Estimated time: 4 days.

   (a) Be able to see others public decks.

   (b) Be able to create a copy and save others decks as my own.

   Acceptance criteria:

   - The user have access to others public decks.
   - The user can copy/save others decks as their own.

4. **As a user, It would be fun to learn my flashcards through a mini quiz game.**

   Estimated time: 4 days.

(a) Create flashpig sprite.

(b) Create the game world.

(c) Create the quiz logic.

Acceptance criteria:

- The player can play the game.
- The player can choose which deck to use.

2.3 Definition of Done

- The code should be tested and thoroughly checked.
- The code should be runnable without bugs that ruins the users user experience.
- The application should have tested all the user stories.
- All functions should work as the user expects it to be.

2.4 User interface

The design for FlashPig is made in Figma.

Start screen

Upon launching the screen, the screen will first show a splash screen before taking the the user to the start screen/Dashboard see Figure 1. Here, the user can see how many decks have been created, edit a deck and access all the mini-games. To be able to continue, the user needs to first choose a deck.

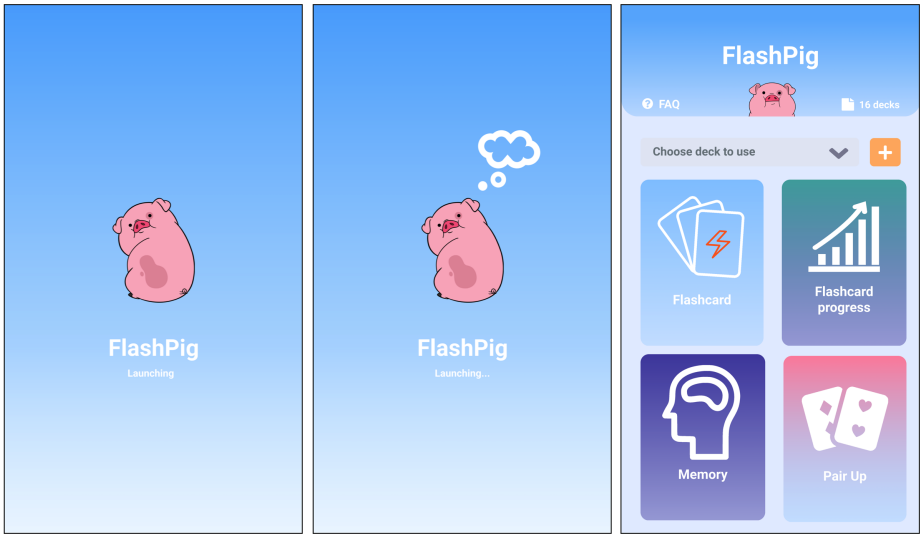


Figure 1: Splash screen and Dashboard

Create a new deck

To create a new deck the user needs to click on the distinct yellow plus button, where the user will then be taken to a new view. The user needs to first name the deck and thereafter create cards by choosing a picture and/or a text for the front and backside and later save the deck. See Figure 2.

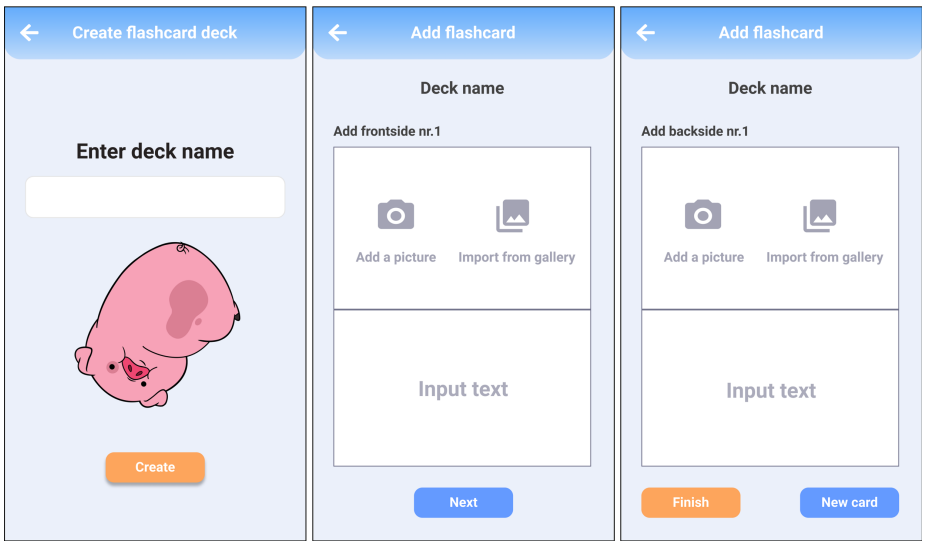


Figure 2: Views for creating a card

Edit deck

By clicking on the drop-down arrow in the combo box, further options are displayed for the user to use. By clicking on the edit-button the user are taken to the page for editing a deck and its cards. The prominent plus-button is now used to create a new card and each created card can be edited from this page. The user can also change which deck to edit from this page. See Figure 3.

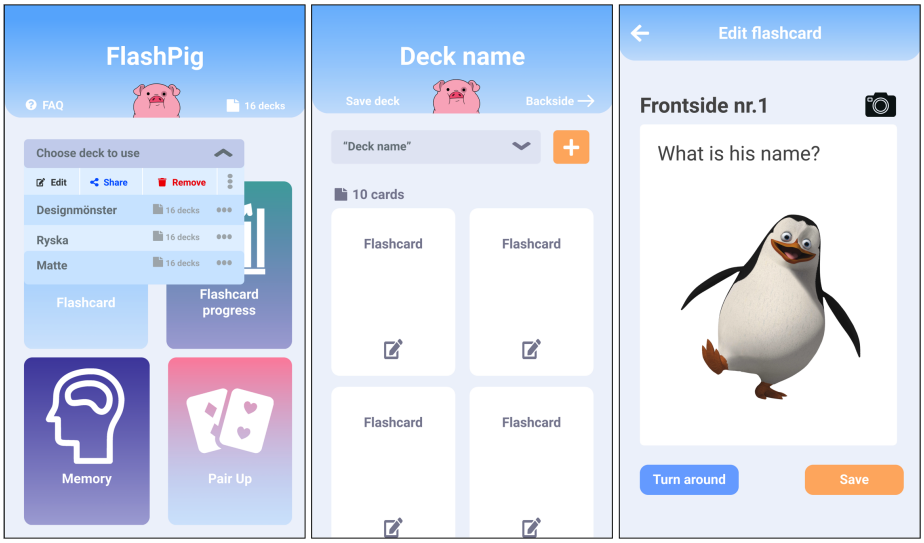


Figure 3: Edit a card

Mini-games

By clicking on one of the mini-games the game will start and show the chosen deck's cards. From there the user can either continue playing the game or return to the dashboard through the back-button.

When playing Flashcard, the user is shown a card's front side and upon clicking on the card, the card will flip and show the content on the back. The user can then choose the difficulty of the card and further move on to the next card. If there are no more cards the user thinks are hard, the game round will end and the user can then either restart or go to the main screen. See figure 4.

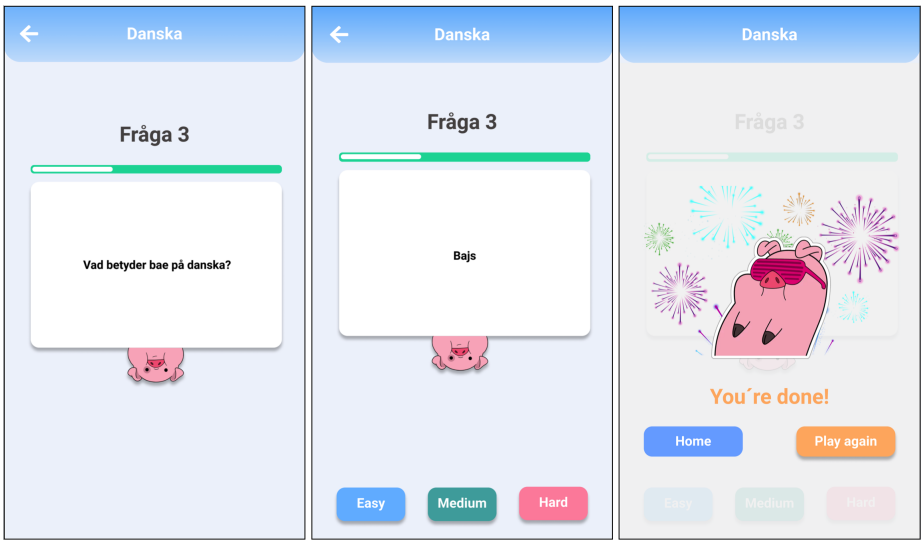


Figure 4: Flashcard game

When playing Memory, the user flips cards by clicking on them and then if the two cards don't match, they will turn over but if it is the other way, the cards will stay as they are. A game is over when all of the cards have been matched and the user can choose to either play again or return to homepage. See Figure 5.

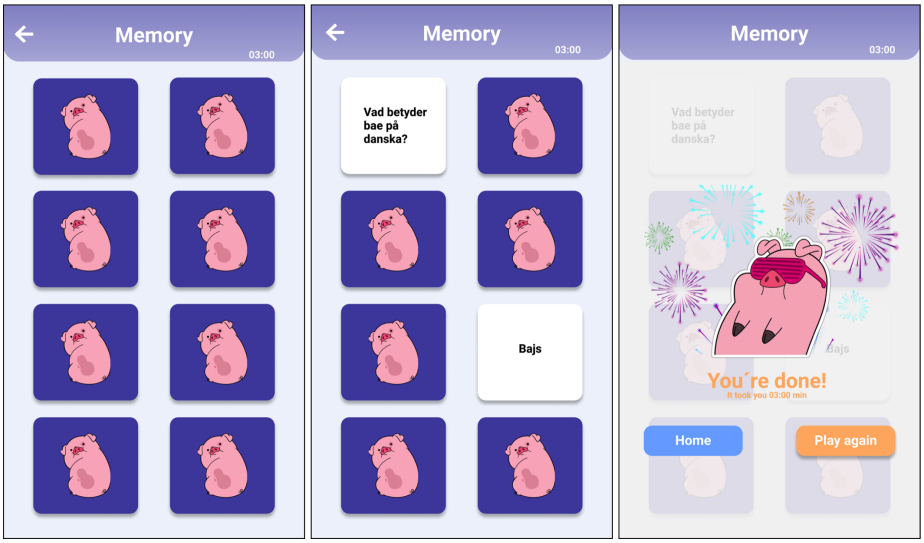


Figure 5: Memory game

When playing Pair Up, the user has to pair two cards on the board. If they

are a match, a green frame will appear. Otherwise, a red frame will appear and indicate that it was wrong. When all of the cards have been paired up, the game round is over and the user can choose to start again or return to homepage. See Figure 6.

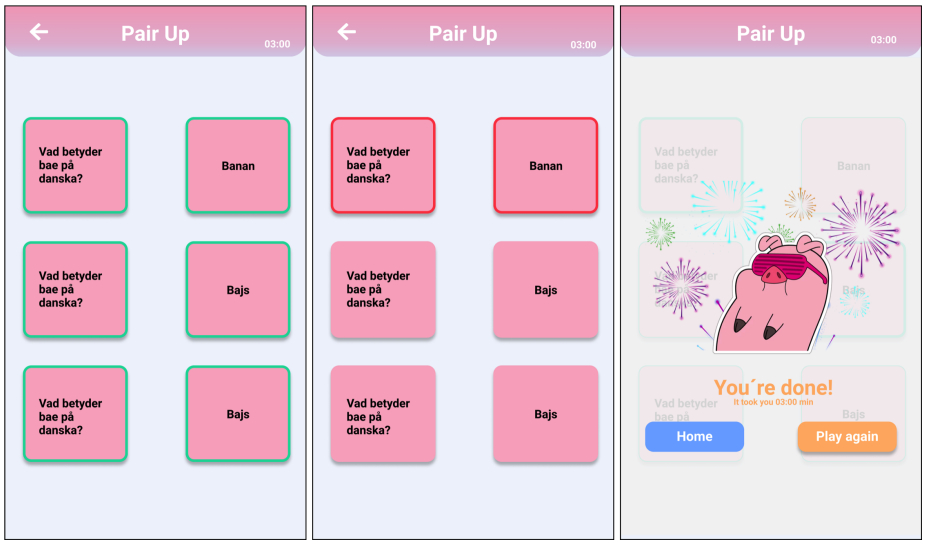


Figure 6: Pair Up game

3 Domain model

The following figures are the UML of the Domain model diagram in the first and last iteration.

3.1 Iteration 1

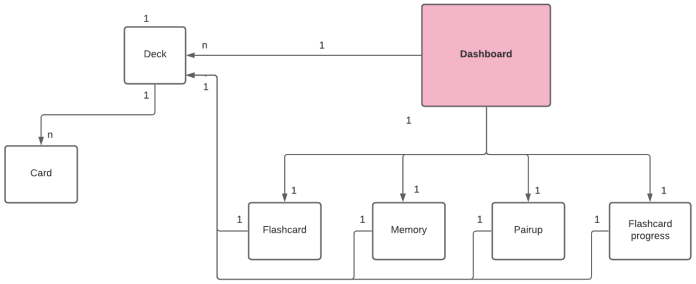


Figure 7: Domain Model iteration 1

3.2 Iteration 2

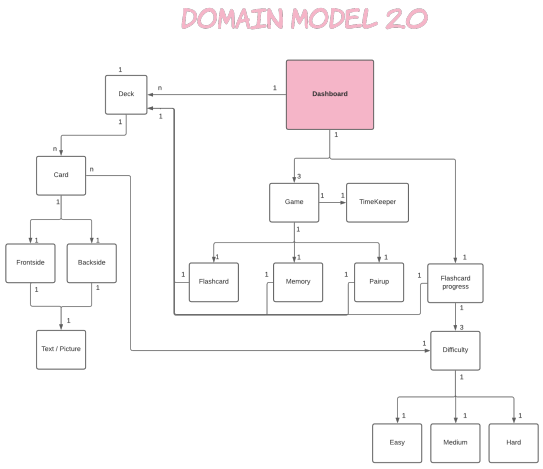


Figure 8: Domain Model iteration 2

4 Class responsibilities

4.1 Card

Card is the model class that holds information that belongs to a card, which is an id, a text and picture for the front and backside. There is also a difficulty for the card, which it can get from the Difficulty class.

4.2 Difficulty

Difficulty is a Enum-class that holds different difficulties for the cards: EASY, MEDIUM, HARD and NOTHING.

4.3 CardViewModel

ViewModel class for creating decks and cards. The purpose is to work as a bridge between the Card and Deck (Model classes) and CardFragment and CreateDeck-Fragment (View classes). This class gets data from the users interaction with the views and updates the model classes with them.

4.4 Deck

Deck is a class that holds information that belongs to deck. It has a int for the id, a list of cards, a string for the deck name, an int to hold the amount of cards in the deck and a boolean to check if the deck is empty or not. The class is also made parceble and holds a Random to generate a random id for the deck.

4.5 EditDeckFragment

Holds the logic for all the clickable objects in edit deck. The class includes instances of the adapters: DeckRecyclerViewAdapter and DeckSpinnerAdapter. The DeckRecyclerViewAdapter is used to access all the required information about the card objects that populates the RecyclerView. Same goes for the DeckSpinnerAdapter, the instance gives all the needed access for the deck objects that populates the spinner.

4.6 DeckSpinnerAdapter

This adapter is used to inflate the spinner drop-down view and views with decks, amount of cards in each deck and buttons for removing or sharing a deck in the spinner. It also makes sure that the drop down view implements the row striping pattern by alternating colours on each row. The class includes also an instance of the interface onEditItemsClickListener which also is located inside

the adapter. The interface is implemented by the two fragments EditDeckFragment and DaashboardFragment, it makes it possible for these classes to override methods that are used for the common spinner.

4.7 DeckRecyclerViewAdapter

This adapter is used to inflate the chosen decks' cards in the edit deck screen. Once a deck is selected in the spinner the cards included in the deck will appear in the screen.

4.8 GridSpacingCardDecoration

This class has the main purpose to locate the cards in desired amount of columns and spacing between the objects.

4.9 FakeDatabase

The FakeDatabase holds hard-coded decks with cards in them that are used throughout the application.

4.10 Repository

Repository handles the data from the FakeDataBase and works as a wrapper. The wrapped data will be sent to whoever calls on it.

4.11 DashboardViewModel

ViewModel class for the DashboardFragment and EditDeckFragment. This class observes the data from the Dashboard- and EditDeck screen and updates the screens as the data changes.

4.12 GameLogic

Gamelogic is an abstract class that holds some values and methods that are needed commonly for the minigames.

4.13 Pairup

The model class for the Pair Up game. The class holds the logic for comparing two cards from a deck and checks if a game is done.

4.14 PairUpViewModel

The ViewModel class for PairUp game. The class gets input from user and checks if the cards are an pair, if the cards are the last cards that are shown or if the user has paired all the cards in the deck.

4.15 PairUpStartFragment

The view for the Pair Up mini-game that the user can interact with. The class shows six cards for the user to pair and will show the user if its a pair or not. If the game is over, the class will navigate the user to PairUpFragmentEnd.

4.16 MainActivityPairUp

The class holds the logic for initialising the Pair Up activity by defining the UI and retrieving the widgets in the UI.

4.17 Flashcard

The model class for the Flashcard game. This class is also responsible for creating a game deck which is a copy of the selected deck and setting the cards difficulty.

4.18 FlashcardViewModel

The ViewModel class consist of LiveDatas of the Flashcard class. The class is responsible for iterating through the selected deck and updating the Flashcard class of the users selection of difficulty. For example, if the user chooses easy then the card will temporally be removed from the game deck for some time before being added in the deck again.

4.19 MainActivityFlashcard

The class is responsible for setting the title for the toolbar and verifies if the user wants to quit the game. The class is also responsible for navigating through the flashcards fragments.

4.20 FlashcardFragmentStart

The view that observes the changes of the Flashcard model by the FlashcardViewModels' livedata and updates it views by the changes. The class also provides the FlashCardViewModel with the users selection of difficulty on a card.

4.21 FlashcardFragmentEnd and PairUpFragmentEnd

The view that turns up when the user has completed the game. Navigates the user depending on if the user wants to play again or return to the home page.

5 References

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

FlashPig: GitHub
`github.com/Wendaaj/FlashPig`

Section 6 - 12 concerns SDD

6 Introduction for SDD

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.

6.1 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.

7 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 factors for managing the lifecycles of the UI components. One example is by implementing ViewModel that stores the UI-related data.

Moreover, LiveData has been implemented in order to handle data persistence by notifying the view when the database changes. The code, especially in the view Model classes, uses MutableLiveData 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 the data is described under the "Life cycles and Data Management" section.

The app uses activities to represent various interactable screens where every screen is a fragment. Like Activities, a fragment has its own 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.

7.1 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 different parts of the application. Each of the parts in the code 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 below in the document.

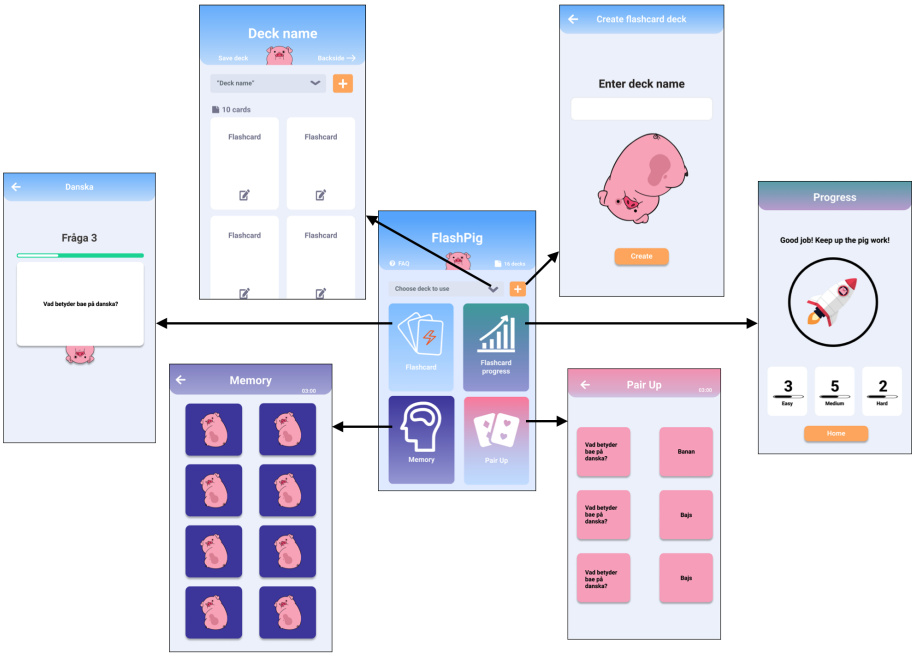


Figure 9: Naviation from the Dashboard to the application's different screens

8 System Design

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

8.1 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 is aware of the ViewModel and the ViewModel is aware of the Model. However, the Model is unaware of the View Model and the View Model is unaware of the View. In this way, the Model can evolve independently since the ViewModel 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.

8.1.1 Model

The Model package holds the application data.

8.1.2 View

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

8.1.3 ViewModel

The ViewModel transforms the Model information into values that can be displayed on a view. The View Model has no reference to the View.

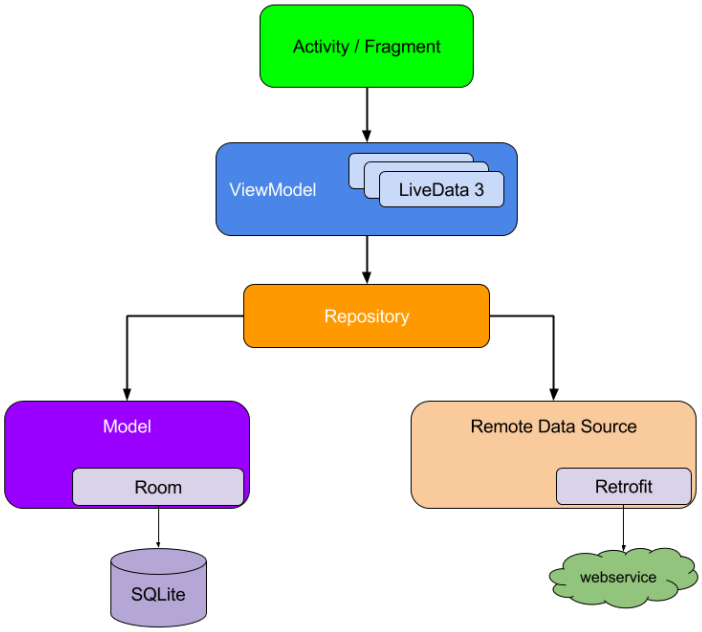


Figure 10: MVVM image taken from Android Studios' website

8.2 Dependency analysis

Dependencies can be found in the appendix.

8.3 Application decomposition

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

8.3.1 Package Diagram

Flashpig is a standalone application that uses its 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 10. The diagram shows that the Model package does not have any references to the other packages and the View is updated by the ViewModel that has a reference to the Model. In other words, the View classes have instances of the ViewModel classes and the View Model classes have instances of the Model classes and this is the way the View and Model communicate. 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.

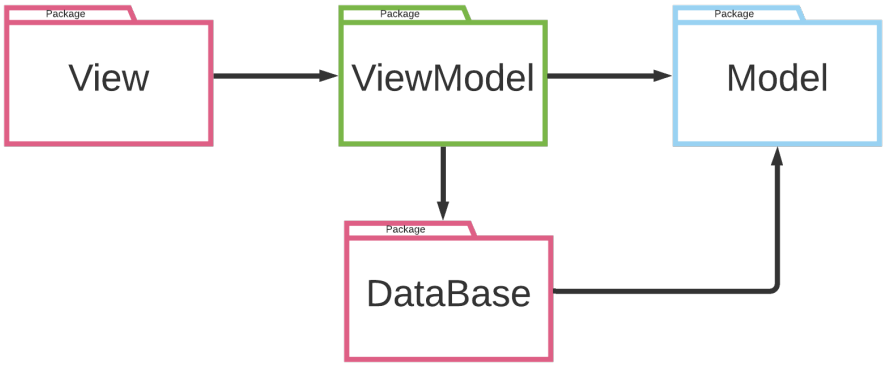


Figure 11: MVVM

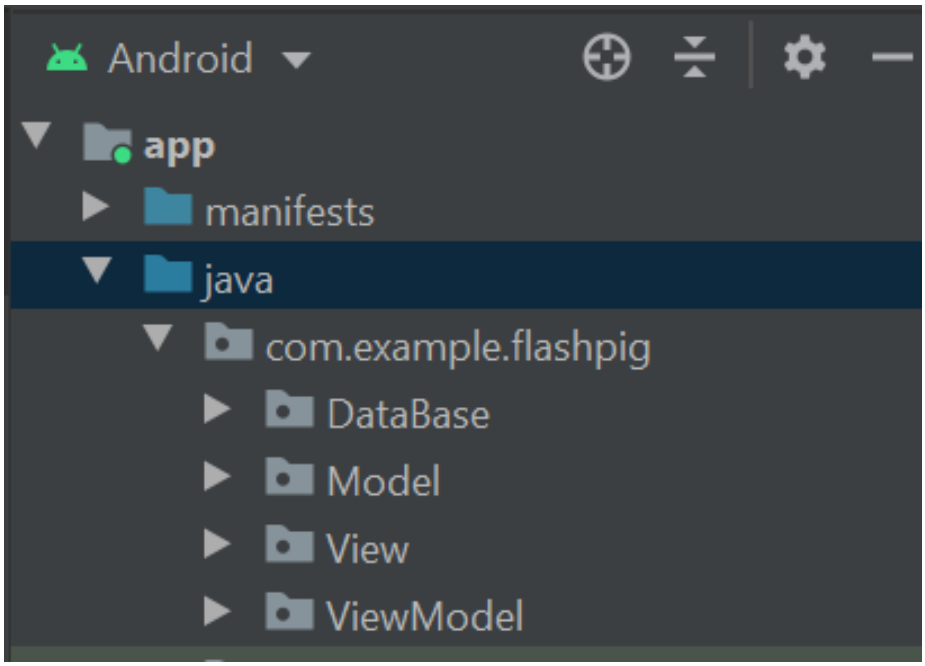


Figure 12: Packages used in the project

The main packages used in this project are listed in Figure 11, 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.

- **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 12 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.

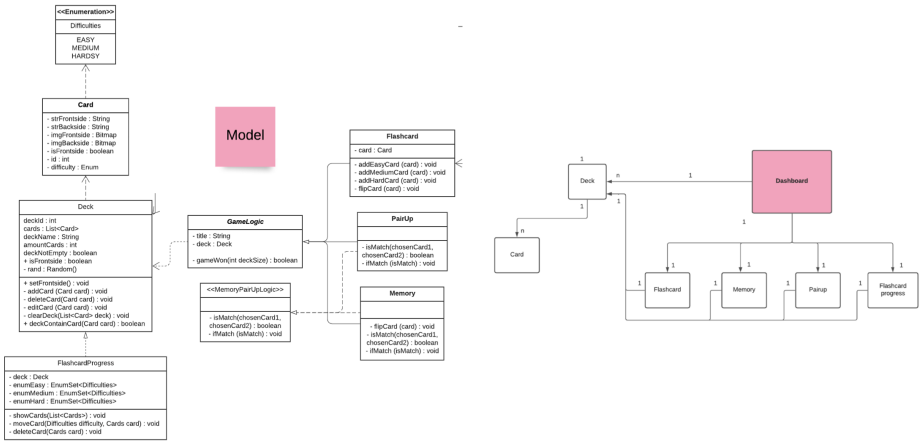


Figure 13: Design versus Domain model

8.3.2 Class Diagram

The Class diagram with its dependencies is shown in Figure 13.

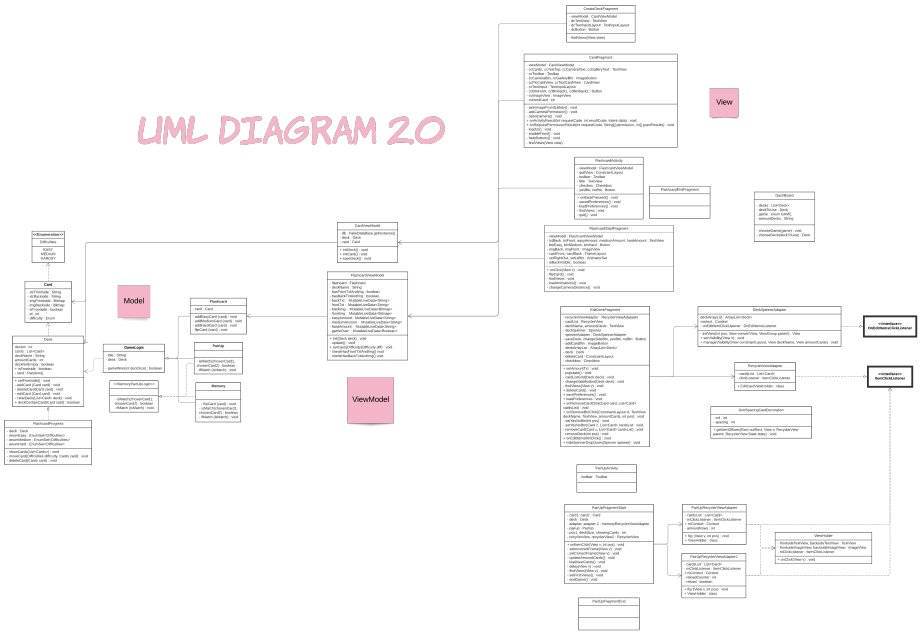


Figure 14: UML Class diagram

8.4 UML sequence diagram

The Figures 14-16 illustrates some use cases in the application in form of sequence diagrams. The chosen use cases are "Play Pair Up", "Edit a deck" and "Play Flashcard".

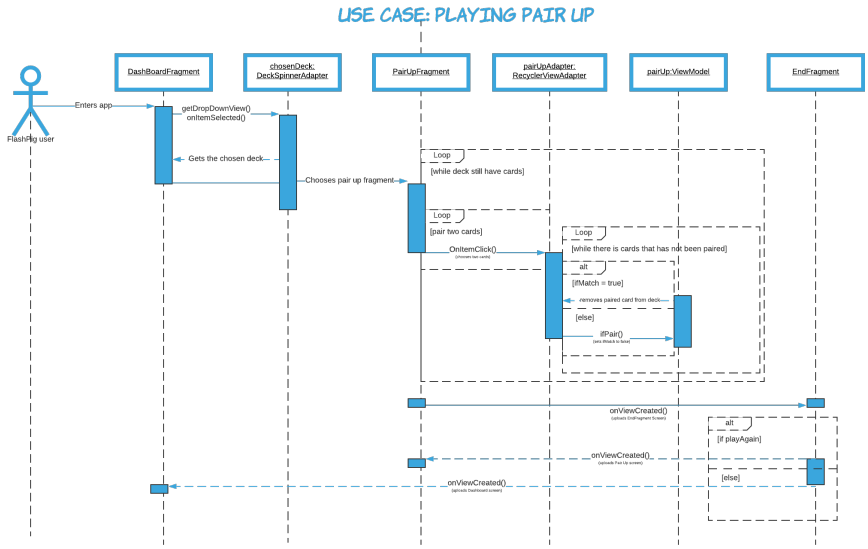


Figure 15: UML Pair Up Sequence Diagram

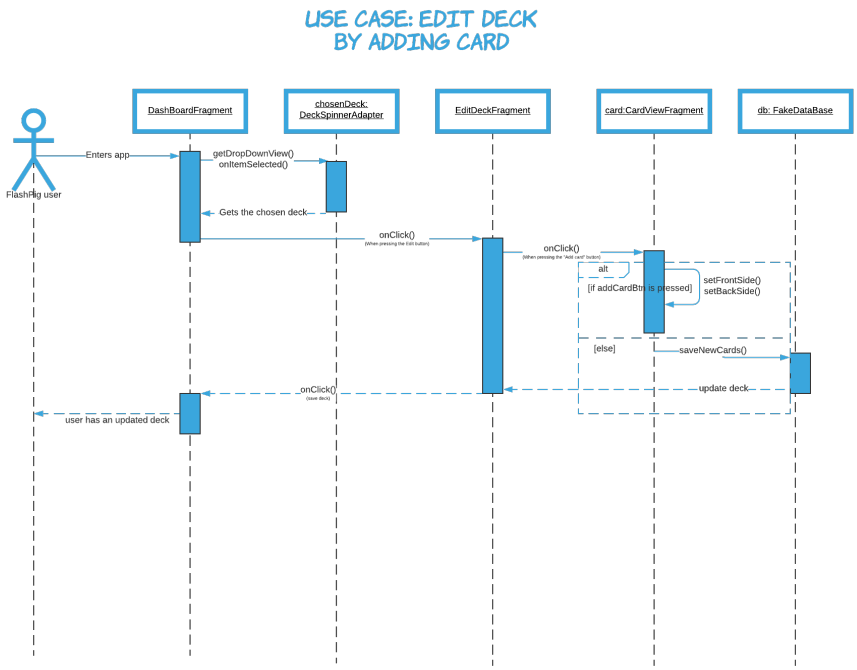


Figure 16: UML Edit Deck Sequence Diagram

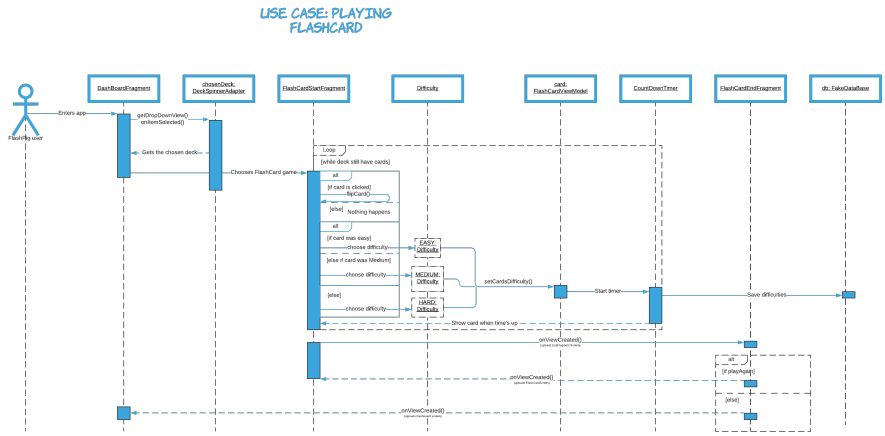


Figure 17: UML Flashcard Sequence Diagram

8.5 Design Patterns

In order to make the application reusable, easy to maintain and extendable, relevant design principles and design patterns 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 activities 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.

8.5.1 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, when trying to replace the Singleton Pattern by making the repository class parcelable lead to a non-executable code, the Singleton Pattern had to stay. However, it is possible to refactor in the future to improve the code.

8.5.2 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 notify the observers if they belong to an active lifecycle state.

8.5.3 Model-View-ViewModel Pattern

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

8.5.4 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.

9 Life Cycles and data management

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

Parcelable is used for 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 processing 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 visual on the screen.

9.1 Life Cycles management

Lifecycle is a class that is responsible for holding information about lifecycles, more specific the current state of a component such as an activity or a fragment. Lifecycles makes it possible for objects to observe the current state. Each and every activity and fragment has a lifecycle.

A class can observe a component's lifecycle by adding annotations to the methods. An observer can be added by calling the addObserver() method in the Lifecycle class and passing the instance of the observer.

10 Quality

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

10.1 PMD

A source code analyzer that identifies unused variables, unnecessary object creations, empty catch blocks, and so forth. The result from PMD is illustrated in Figure 17.

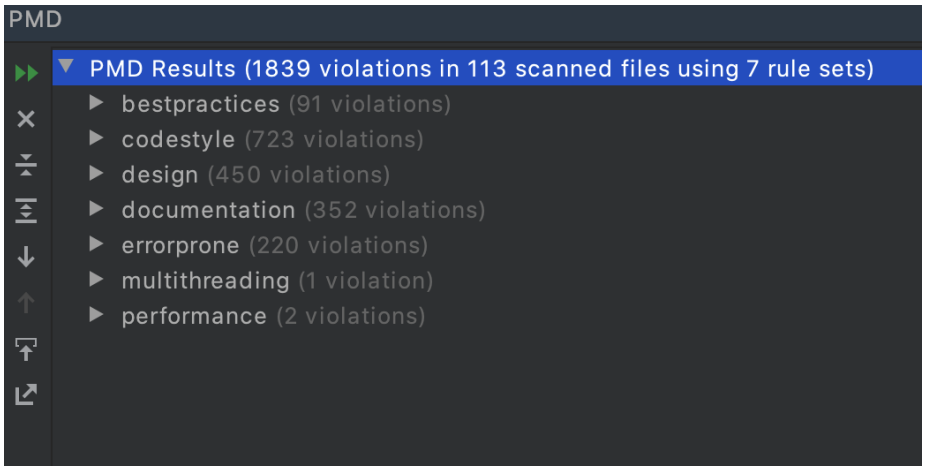


Figure 18: Results from PMD

10.2 Git

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

10.3 Travis

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

10.4 Javadoc

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

10.5 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.

10.6 JUnit

For testing the model. JUnit covers tests of the model which are located in a test package. The model and the ViewModel have been tested.

10.7 Tests

The Model and the View Model have been tested in the application with JUnit.

10.7.1 Known issues

The Pair up game can be improved in some ways. For instance, to make the game harder, a shuffle method for the current game cards can be implemented. As it stands, the game only works with a deck that contains multiplicities of three cards. This could be improved by making it possible for every deck regardless of the size. Moreover, the play again button is not yet implemented.

Another issue is found in EditDeckFragment class, where the cards moves to the right every time a different deck is selected in the spinner. After a while the cards will gradually disappear from the view.

The method for flashcard regarding spaced repetition could be improved by following the devices clock instead of being a countdown as it currently is.

Upon removing all the decks in the spinner in the DashboardFragment, the view with the buttons are visible until the user clicks outside the spinner.

10.8 Access control and security

NA

11 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/

12 Appendix

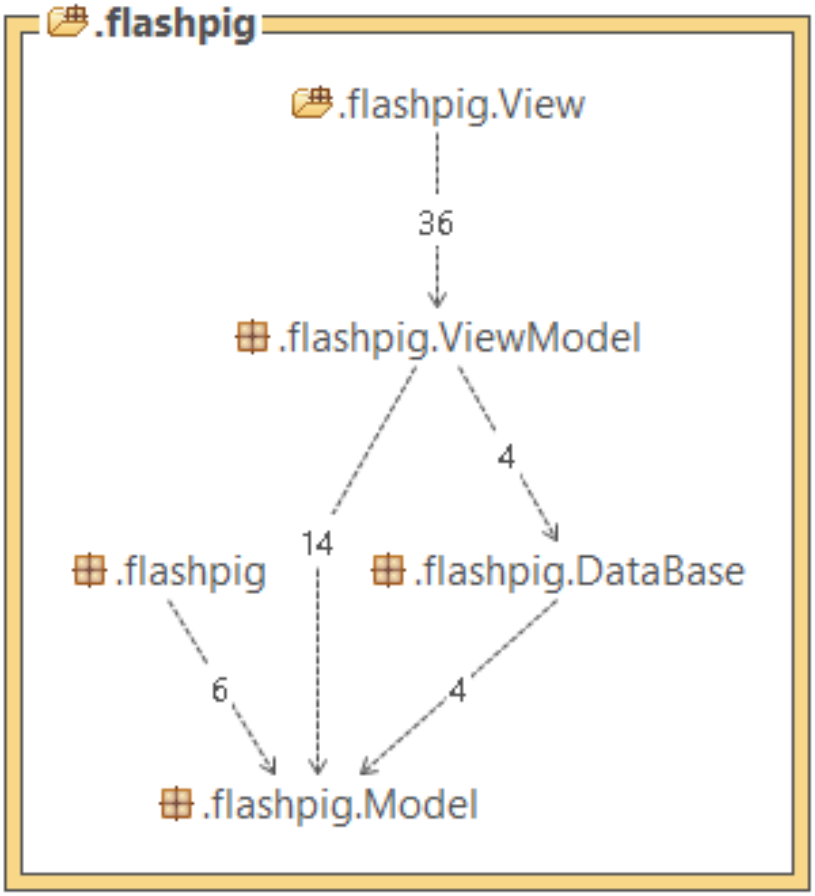


Figure 19: The overall project structure.

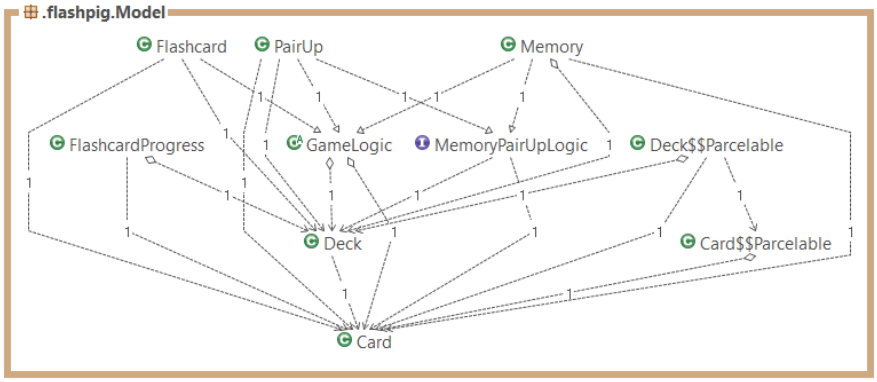


Figure 20: The structure in Model package.

Figure 21: The structure in View package.

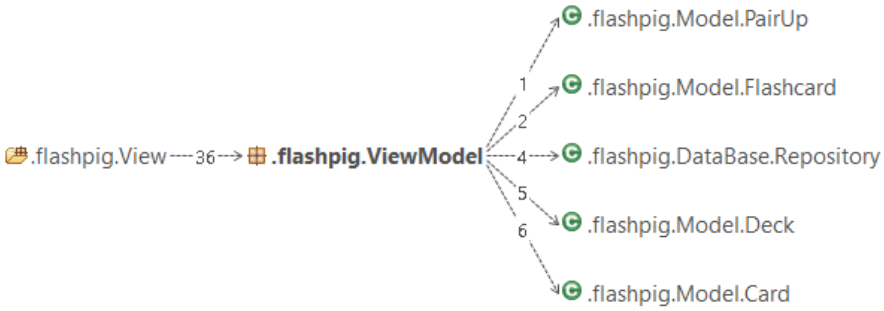


Figure 22: The structure in ViewModel package.

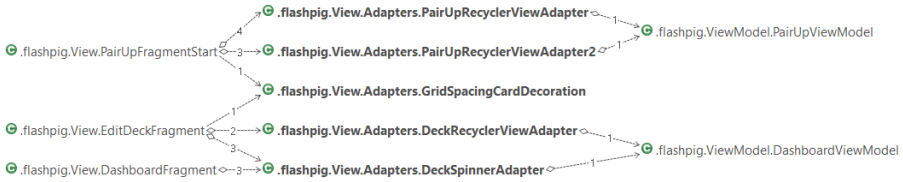


Figure 23: The structure in Adapter package.

13 Peer Review

The code is incomplete and not much has been done yet. The code was hard to understand as it is incomplete and lacks many java documentations and Junit-tests. However, we tried our best to review the code either way.

The System Design Document (SDD) for Locali mentions that the code is organised in a "M-VC" structure, where the controller basically also handles the view. This automatically makes the code difficult to reuse, maintain and extend functionality as the view is not easily replaceable and the controller is too heavy. This violates the open/close principle and single responsibility principle. Additionally, some classes which belong to the model manage some view logic, which does not fall in line with the MVC structure. These violations can be found in the classes iIconSelector, iPane and LogInHandler.

The dependencies overall in the code is good because the viewcontroller and the model have few dependencies between them. However, the view and the controller are too dependent on each other and therefore it violates the High Cohesion, Low Coupling principle because the classes inside the ViewController are not independent. Therefore, should it be broken into smaller modules. In addition, it also violates the Single Responsibility Principle.

The code implements the following design patterns: Factory method, Observer pattern and Singleton pattern. The implementation of the Observer pattern is incomplete and therefore no comments can be made. The Factory method is good since it follows the conventions of implementing the pattern. However, the use of Singleton is questionable. The code implemented singleton on the class User, which results in that only one instance of the user can be created at a time. This limits the possibility for the application to handle more than one user which is a serious performance issue. Since the application does not seem to have a system for saving an user or its information for future references, the user and its data will disappear when the application is closed.

Their code seems to be intended to follow the Dependency Inversion principle via the intended Observer pattern and the abstract classes. It has an overall

good structure setup for the pattern and to abstract the code, but there are some classes that violate this idea. One example is the abstract class loginHandler that seems to have methods that have to do with the view. This breaks the initial intent of implementing the observer pattern and abstracting the code since the model should be free from view logic.

The code has many abstract classes and interfaces but the usage can be refactored. For example, in the method onLoginClicked() in the class LogInPageController, the abstract class LogInHandler, is instantiated and the login() method is overridden (which is unnecessary) to later be called in the method. The interface iSavingsRegister is declared but never instantiated in the SavingGoal class which leads to null.

The naming in the code uses the universal naming convention for Java in its classes, packages and methods in the majority of the code. However, there are some exceptions where the method names differ from the convention. One example is in the class User where the method UserObserver() starts with an uppercase opening letter. Another example is the class AccountDataHandler that handles the sign-up process primarily but the name gives other meaning.

Some methods are too long, and should be more abstracted. For instance the method changeAccountTextfields() includes a lot of if statements, which could be separated into smaller methods. This method doesn't follow the Single Responsibility principle, since it behaves like a void method (by changing states) and it also returns a value. Which furthermore leads to violating the Command Query Segregation principle. Another method worth mentioning is initPane(), which includes a lot of lines where a chart is being populated by different objects. These lines could also be implemented in smaller methods to follow the single responsibility principle better.

13.1 Improvement and solutions

Considering that the code is incomplete, so the recommendations of improvements and solutions only be of the parts that are somehow completed.

One solution for the code is to implement the Model-View-Controller structure correctly so that it is possible to replace views and therefore make the code reusable, easier to maintain and extendable. At the moment, the model has some dependency on the view which breaks the fundamentals of the MVC structure.

Write tests to check that there are no null pointers and that the code works as thought. Also document the code so that it becomes easier to follow and understand what the classes' and methods' responsibility is.

A lot of the methods seem to have no purpose yet, why have a method called editBudgetButton that has only a lot of getters and returns nothing. This

method does nothing for the application as is. An alternative would be to implement more methods that have a clear purpose.

Some variables in classes have missing access modifiers declared, which sets them to the default access modifier (protected) to the variables. This could be improved by writing it down to make the code more easy to read.

Another improvement in terms of better performance, is to remove the implementation of singleton pattern on the user and connect the user with saving goals, budget etc. As of now, the code is unclear on how an user's budget or saving goal is connected to one user and the reason why the code only can handle one user at a time.

Because of the length restrictions, can no more recommendations be given, however we have compiled the most essentials recommendations above.