System Design Document for Debtify

Oscar Sanner, Olof Sjögren, Alex Phu, Yenan Wang

2020-10-22
version 2.3

Contents

1 Introduction

This report aims to describe the software system design structure of the android application Debtify. This will be done through UML-diagrams and accompanying descriptions.

1.1 Design goals

The goal of the design was to have a loosely coupled and testable application. The overall design aims to establish few dependencies and low coupling in order to be extensible and reusable.

1.2 Definitions, acronyms, and abbreviations

- MVVM - Model-View-ViewModel program architecture, usually found in Android applications.
- Debitify - The name of our application.
- Entity - Identifiable object used in the application, such as users, groups, debts and payments.
- To inflate - In Android framework, the XML files are inflated by fragments. It means to parse an XML file and create all view components defined in the XML along with all of their corresponding attributes specified within.
- JCL - Java Class Library
- JSON - Javascript Object Notation. A file format with a large parsing support in several different programming languages.

2 System architecture

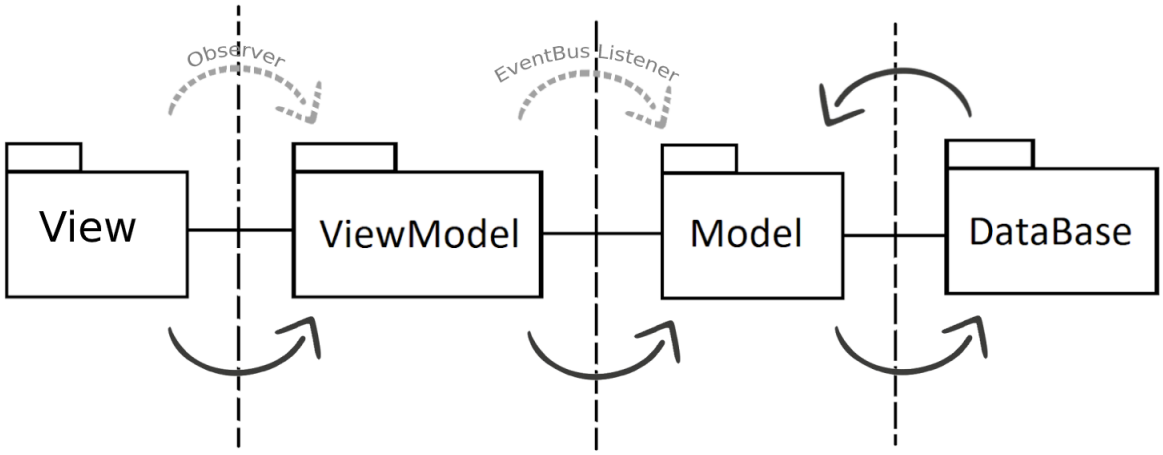

Figure 2.1: Components of the application.

The application is divided into four parts:

- The **View** package which acts as the presentation layer, handling both display and user interface.
- The **ViewModel** which acts as the mitigator between the Model and the View package, handling both Model updates and data requests from the View package.
- The **Model** which is the domain and contains the business logic of the application.
- Lastly the **Database** which is accessed through an interface and injected during the creation of the Model.

2.1 Application flow

The application is as previously mentioned built on the Android Framework. This means that there is no apparent main method as in conventional Java applications, but rather a chosen Activity that is specified as the launching activity in the AndroidManifest. Activities are Android's way of structuring the UI, which is a single screen with accompanying user interface elements. In our case, the LoginActivity is declared as the launcher and will thus be the first screen that the user sees on startup. The Model is instantiated the first time a View calls upon a ViewModel and the same Model is then used for all future ViewModel requests.

Generally speaking, the View package is the user interface of the app which is responsible for presenting relevant data as well as handling user input. The View package communicates with the Model through the ViewModel to get the desired data to display or to delegate user inputs to the Model. The Model is dependent on the database which is instantiated, along with the Model, through dependency injection at the beginning of the application's life cycle. The Model is structured to first attempt to update server side and after a successful

database call, also update the Model client side. At the end of this process an event is published to an EventBus where appropriate ViewModel subscribers are notified. The ViewModels then update themselves with relevant data from the Model which in turn notifies the observers (Views) of that ViewModel.

There are two levels to exiting the application. Simply exiting the application and returning to the Android OS environment will not fully exit the application but simply automatically log out the user. However the application will idle in the background and the Model will not be terminated. This means that, should the offline mock database be used, the session's information will not be lost. However, if the application is terminated completely, then all data will be lost if the real online database is not used.

3 System design

The application mainly follows Android's preferred architectural pattern, MVVM, which stands for Model-View-ViewModel.

3.1 MVVM

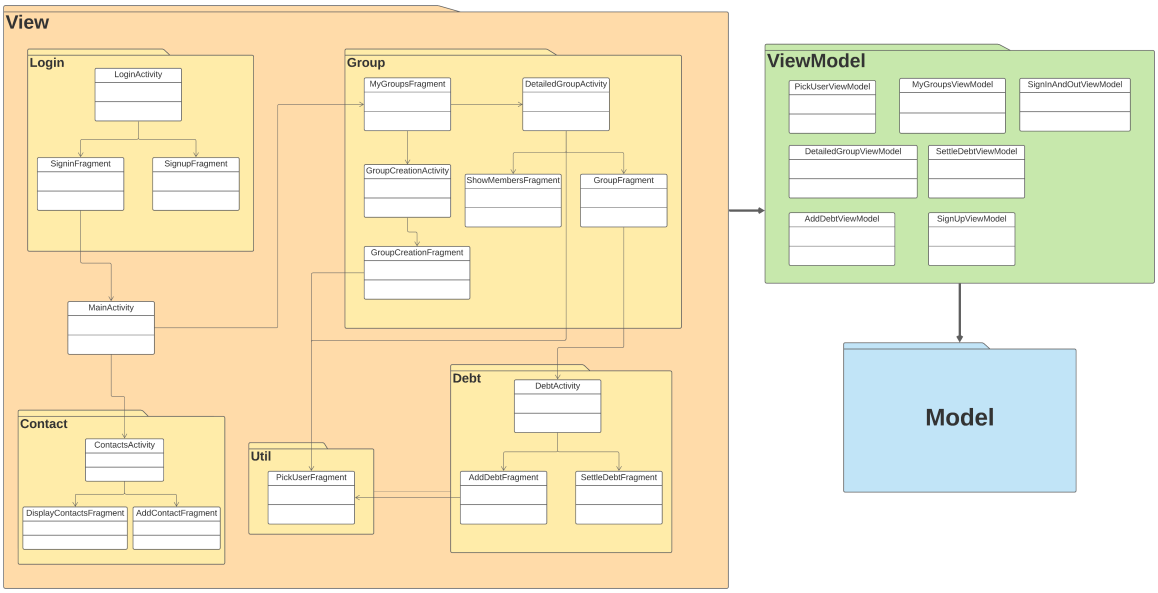

Figure 3.1: MVVM structure of Debtify

The application architecture was created predominantly with the Separation of Concern in mind as the MVVM pattern emphasises on keeping each Activity and Fragment class as focused as possible. The Views, which consist of XML files and UI components allow only the usages of UI handling logic that react to interactions from the users. Each View class may have a dedicated ViewModel, however, a ViewModel never becomes aware of the dependent View. The ViewModels communicate with the Model for data and holds relevant data for the Views.

3.1.1 Why MVVM over MVC

The fundamental concepts of MVVM and MVC are the same. Both design patterns strive to separate logic from the Views and attempt to abstract core functions of the Model package. In Android, MVC considers the XML files to be the Views and the Activities to be controllers, as opposed to MVVM which regards both Activity and XML files as the Views, and ViewModel as the mediator between the View and the Model. Thus separating the business logic from the UI which is the reason why MVVM was chosen for the application.

Another merit with using ViewModels is the simplification of the Android application's internal life cycle management. An Android app usually consists of one or more Activity classes. Each of the Activities can be perceived as its own process that may be terminated any moment due to how Android's life cycle works. By using ViewModels, an inbuilt class

from the Android API, the obfuscation from managing Activities' life cycle becomes negligible since the unpredictable termination of an activity no longer stays relevant. The Views can simply retrieve the very same data from the same ViewModels each time they are recreated. As for how this works will be discussed in a later section.

3.1.2 Package relations

As aforementioned, each View class owns a dedicated ViewModel. There are no extra layers between these two packages as that would be redundant and defies the purpose of using ViewModels. Between the ViewModel and Model packages however, there exists a facade class with the intent of simplifying the Models' functionalities. This makes sure that the ViewModels have minimum business logic for data retrieval and naturally not become directly dependent on the Model package internal structure. This in turn allows modification to the Model's internal structure without directly affecting any dependencies.

There are a few Model interfaces which are globally accessible. These immutable interfaces create a type-safe and secure environment for data retrieval from the Model and avoids alias problems. This, just like the facade class, ensures there are no dependencies on the Model's core functionality, but rather on the abstractions.

3.2 Model

The Debtify Model package is a platform independent package. As such it can be driven by any other view or controller, for any system. In Debtify however, it is driven by the ViewModel package in an MVVM architecture. The model holds all instances of entities used in the application. It is also responsible for conducting all the business logic operations in between these entities.

3.2.1 Internal architecture

The internal architecture of the model package relies heavily on delegation through composition. This leads to a very modular design which lets developers single out and modify, replace and build on each class individually, given that the promises are adhered to. This makes the model maintainable and follow separation of concern as tasks are handled in classes with corresponding areas of responsibility.

The highest level module of the package is the Session class. This class is supposed to always represent the session of the logged in user. The Session acts as an aggregate of different entities and delegates to these when necessary. These classes are in turn often composed of several objects and the delegation occurs in these objects as well, from high to low module

classes.

3.2.2 Database

The database resides inside of the Model during run-time. Once the Model is instantiated it's passed an instance of a database via constructor injection. This usage of dependency injection allows the Model to take in different dynamic types of the database, under the static type IDatabase. IDatabase is an interface promising to store data persistently. The commands and queries for an IDatabase are simple methods for either storing or retrieving data out of a persistent data storage. Commands will take in parameters for the data that has to be set, and queries will return data in the JSON format.

3.2.3 Type-safety assurance with JSON strings

The Model holds an instance of a database responsible for communicating with the server. Queries to this object will return JSON-strings of data and measures have been taken to make the handling of these strings more type-safe when the strings are passed in the Model.

The first measure touches the abstract class JsonString. This class only holds a single final String object and has a getter for this. Classes inheriting from here will add nothing and simply hold their own strings. The purpose of this is to add a layer of type safety between the database and the class responsible for parsing the Strings, so that when working with the Model, the wrong JSON string can not be passed to the wrong JSON parser method. Each class inheriting from JsonString contains a well documented description of how the specific JSON should be formatted. This will also further alleviate the responsibility of correctly formatting the strings to the person working with the Model. For instance, a developer developing a new database is forced to have the database return specific JSON String types and this developer is also responsible for making sure that the Strings are formatted according to the documentation in the JsonString class.

The second measure taken is that JSON strings are never created in the model. Instead of taking in JSON strings on commands, the database will take in various arguments describing the data to be stored.

Another measure is that strings are rarely handled and only for initialization or reload invocations. This means that the Model does not follow the database command up with a query to reload the affected entity. Instead the Model sends a command to the database and then updates itself with the same information. This leads to decreased database usages and fewer string handling methods.

3.2.4 Exceptions thrown in the Model

The Model will throw exceptions to indicate that an operation went wrong. To further specify the exception, the Model has specific exceptions related to the application, such as the GroupNotFoundException. A lot of the exceptions mitigates improper input from the client to different methods and are as such propagated back to be handled by the client using the Model. These are checked exceptions. Other exceptions however are unchecked and represent an incorrect usage of the Model. An example of this is the UserNotLoggedInException, thrown when methods in the Model requiring a logged in user are called before the user is logged in. The purpose of these are to be more specific and more immediate than the more abstract run-time exceptions that would eventually be thrown if these methods were called.

When a checked exception is thrown in the Model, a message is always sent with the exception. The same unhandled exception can be thrown in different parts of the Model but with two different messages attached to it, making it easier to pinpoint bugs when the client is using the Model improperly.

3.2.5 EventBus over Observers

To notify clients of internal changes in the Model, the Model utilises an enum based EventBus [2]. Events in the EventBus are represented by enums, with one example being: EventBus.EVENT.GROUP_EVENT. The client wishing to subscribe to an event will simply run the method register(...) with the event type and itself as arguments. The precondition is that the class implements IEventHandler and thus implements the method onModelEvent(), which is in term run by the EventBus once an event occurs.

The EventBus is chosen over the traditional observer pattern designed by "Gang of four" [3] as well as the PropertyChangeListener in the JCL. The requirements of each observer were such that no data has to be sent with each notification, as the Model provides the methods necessary for fetching data when an observer is notified. This results in all the observers being able to share a common interface, without breaking type-safety. Furthermore, if the application was to use a traditional observer pattern, the facade class, ModelEngine, would have to facilitate three different methods for each type of observable. For instance, registerGroupObserver(...), removeGroupObserver(...) and so forth. This would make for a very stiff and bloated facade.

The reason for not utilizing PropertyChangeListeners, found in the JCL, is that it would require the Model to be restructured in a way that would be unnatural. It would for example require the implementation of the JavaBeans convention on observable classes, putting hard

requirements on these classes to be serializable and having no-argument constructors [4].

3.2.6 Outgoing dependencies from the Model

The dependencies in the Model have been limited to a great degree, to make the Model redeployable to other platforms regardless of which libraries are available. There is however one outgoing dependency from the Model. Namely to Google's Gson library. Gson is used to deserialize JSON strings received by the database. It is always required when redeploying the Model. Google's Gson library is however platform independent and should not restrict the re-usability of the Model.

3.2.7 The facade class

The Model package is accessed through a single public facade class, known as the ModelEngine. This facade represents the capabilities of the entire package. The model is thus driven in its entirety through this class. The ModelEngine class contains no business logic and uses delegation to other other objects to fulfill commands and queries.

The ModelEngine has to remain the sole public class of the Model package to restrict coupling and to preserve the integrity of the package. Queries to the Model thus return objects of various interface types. This serves two purposes, to hide the concrete implementation of the various Model classes required by client packages, and to keep the objects immutable to the clients using the Model.

3.2.8 Domain model vs Design model

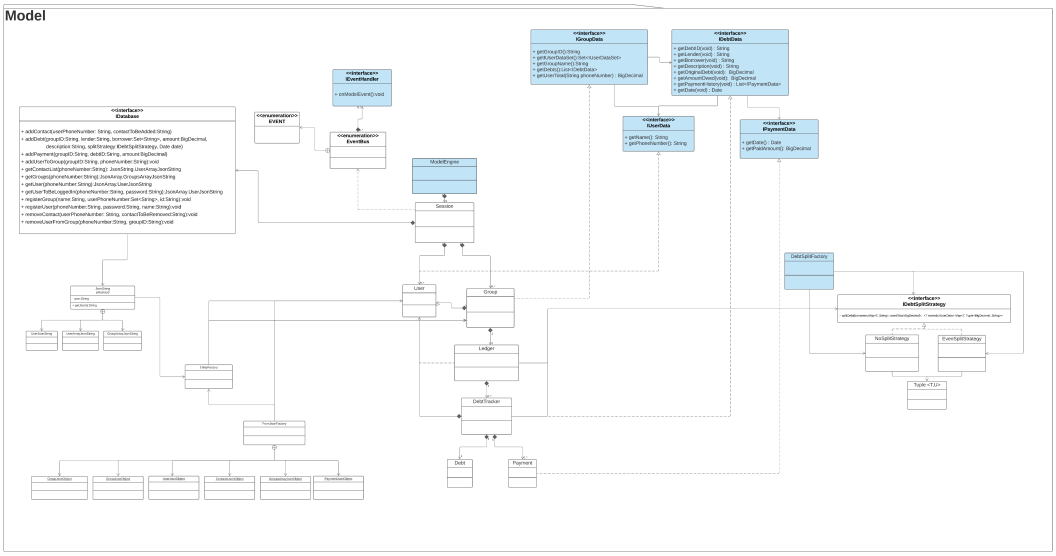

Figure 3.2: The UML diagram of the design model.

The design model is strongly related to the domain model. Each class in the domain model is represented by an actual class in the design model. The relation between the classes in the domain model remains the same in the design model as well. Key differences between the design model and the domain model is that different interfaces are implemented by the various classes holding data required by the client code using the Model. The addition of the ModelEngine class is also important to note as this serves as an access point to the package.

3.3 View

The View is the user interface of the application, which consists of Activity and Fragment classes. No business logic exists in the View as the Views are only allowed to handle the navigation and user interaction of the application. Whenever the View has to communicate with the Model, it goes through a ViewModel instead. The ViewModel is an abstraction for the View classes. Its purpose is to retrieve the required data from the Model and expose such data to the View to display. The ViewModel sends the data through LiveData which is an observable class. The View can thus subscribe to the data and update its view whenever the LiveData is updated in the ViewModel. How the LiveData is updated in the ViewModel will be discussed later.

3.3.1 Activity and Fragment

To quote from the official Android documentation for Activity - "An activity is a single, focused thing that the user can do" [1]. Following this principle we have divided our activities into following classes.

- *ContactActivity*
- *DebtActivity*
- *DetailedGroupActivity*
- *GroupCreationActivity*
- *LoginActivity*
- *MyGroupsActivity*

Each of those activities serves its own purpose and only performs the duty assigned to them. For instance, the GroupCreationActivity launches whenever a user desires to create a new group and once the group creation progress is finished, the activity will immediately finish its process and return the user to wherever they were before.

An activity alone is not enough to build a fully fledged Android application. A fitting metaphor to describe the role of activities would be "a window that holds other visible components" whereas the visible components are inflated by the Fragment class. A fragment represents an interactable component of the user interface in an activity. Namely, an activity

may hold one or more fragments and collectively they create the user interface for this specific Activity class.

3.3.2 XML files

XML files are scripts created to describe data. When it comes to Android development, the XML files are used to convey data of a view's appearance. By specifying view components such as view-groups, widgets and their respective attributes, the XML files can consequently be inflated by a fragment and rendered on the screen [5].

In the application, the XML files can be seen as the presentation layer, whereas the fragments control the behaviours of view components specified in the XML file and thus separating the logic from the view. To give an example, a Button defined in an XML file will receive its behaviour from the Fragment class that inflates this specific XML file. For this reason, the fragments can also be referred to as the UI-controllers.

3.4 ViewModel

The ViewModel fills the role of acting as a connection between the View and Model. Any View that needs to in some way contact the Model, whether it be commands or queries, has a dedicated ViewModel object. The connection between Model, ViewModel and Views utilizes different variations of the Observer pattern such as EventBus or LiveDatas to abstract dependencies and thus allow for a looser coupling.

3.4.1 Updating the ViewModels

A ViewModel will upon initialization add itself as an EventHandler to a specific Event published by the Model. Which event will be handled depends on the ViewModel's area of responsibility. For instance, GroupViewModel will listen to the Model's updates regarding the logged-in user's groups. This allows the ViewModels to always be notified whenever the Model is updated.

This is especially important due to the fact that different ViewModels that are interested in the same data have no relation to each other. One ViewModel can update the model without another one knowing. With EventBus, this problem can be solved by simply subscribing a ViewModel to an Event. Hence, the ViewModel is always kept up to date with the Model.

3.4.2 The Views and the ViewModels

In the View and ViewModel's structural relationship the ViewModel acts as the producer and the View as the consumer. The consumer is aware and dependent on the producer while

the producer has no concrete knowledge of the consumers.

Since View objects can be destroyed randomly, it is strictly forbidden to refer to any View class in a ViewModel while a View class can hold multiple instances of ViewModels. This allows for many-to-one relationships between View and ViewModel packages.

3.4.3 LiveData and the Views

The ViewModels themselves contain LiveData variables. The LiveData class represents a group of persistent data that can be observed for mutations by another object. In other words, the LiveData is intended to be observed by the Views so that the Views may update themselves whenever the LiveData is updated.

In practical applications, a View observes one or more LiveData objects by retrieving them from a ViewModel and presents the received data to users. If a command that updates the Model is executed then the ViewModel will receive a notification of that action and update the LiveDatas correspondingly. Subsequently, this will notify the Views which will update the displayed data.

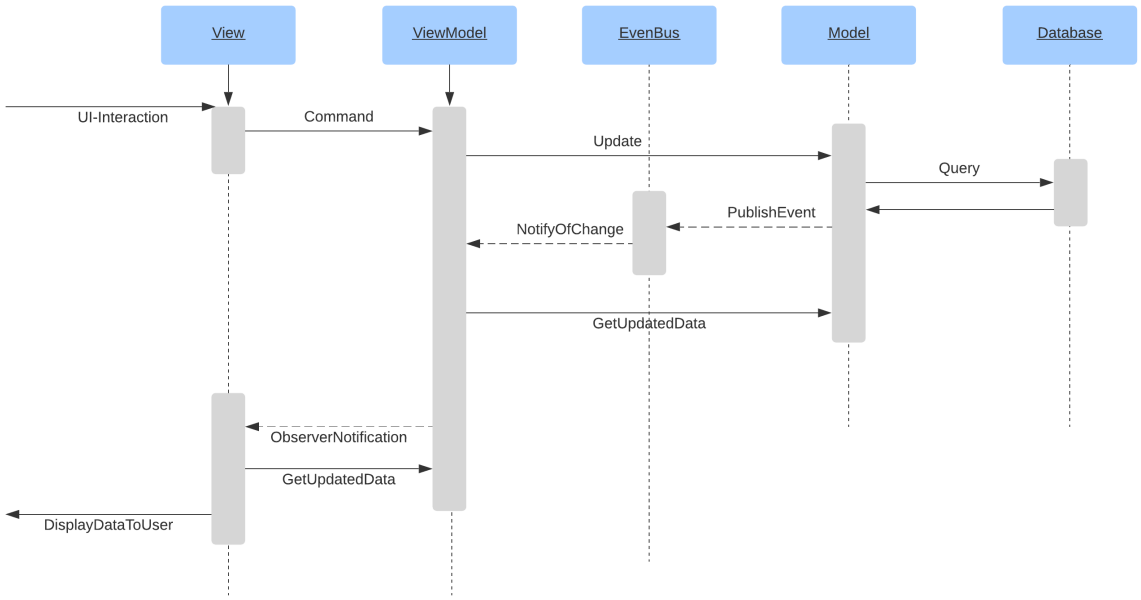

Figure 3.3: The run sequence for a generic example of a typical user interaction.

3.4.4 Life cycle and the Views

It was previously mentioned that by using ViewModels the life cycle management can be simplified as a result of how an Android application's life cycle works. The Views, or the

activities and fragments, as previously mentioned, can be terminated any moment due to various factors such as reaching the memory limit, switching to a new view or simply by turning off the screen.

However, the ViewModels are never terminated as they exist outside the Android's life cycle. This is precisely why, by using the built-in methods in the Android framework, the same instance of ViewModel can always be retrieved since the ViewModels are stored in a Factory class. Thereby, the Views can easily retrieve the same data they were left with the moment they are terminated, once the Views are recreated.

3.4.5 ModelEngineViewModel

Normally a concrete ViewModel class is created by extending the abstract class ViewModel from the Android framework. In that concrete ViewModel, it usually refers to some sort of Model interface. However in this application, since the ModelEngine is defined as a facade class that can be instantiated with a IDatabase of different sorts of implementations, it was considered ill-suited to make the ModelEngine a singleton. Consequently, it becomes a problem trying to retrieve the same instance of ModelEngine.

To solve this problem, the abstract class ModelEngineViewModel which extends from the ViewModel class has been created. The ModelEngineViewModel class aims to replace the ViewModel as a super class. The class instantiates a static instance of the ModelEngine which can be retrieved with a final method getModel(). Thus all classes extending ModelEngineViewModel will be able to retrieve an identical instance of ModelEngine.

3.5 Design patterns

The codebase of Debtify utilizes design patterns wherever it deems to be applicable. This section describes and explains the design patterns used.

3.5.1 Publish-Subscribe

Publish-Subscribe pattern is utilized in Debtify in order to communicate messages between publishers and subscribers. By using an EventBus, the publishers and subscribers are not aware of each other. Both parties are conscious of an EventBus who is responsible for filtering the published messages and distributes them to the subscribers that are listening to a matching Event.

3.5.2 Facade

As mentioned before, the ModelEngine class acts as the Model's exclusive facade that represents the entire Model package. See section 3.2.2 for more information.

3.5.3 Adapter

Every RecyclerView in the view package needs a corresponding adapter to make the existing Model classes compatible with Android's view components. The adapters can be found in the *view/adapter* package.

3.5.4 Strategy

The strategy pattern is apparent in the IDebtSplitStrategy and the concrete classes EvenSplitStrategy and NoSplitStrategy. By implementing the Strategy pattern, it allows the user to choose different methods/strategies to split a debt to multiple borrowers. It also allows for an easy extension for future debt split strategies.

3.5.5 Factory method

Both EntityFactory and DebtSplitFactorry follow the Factory method pattern, hence encapsulating the creation logic.

4 Persistent data management

Regardless of which type of database is injected to the model, no data is ever stored persistently in the Android OS. Persistence is achieved, either permanently by injecting the class known as RealDatabase or during run-time by injecting the MockDatabase class. When the RealDatabase is injected the application sends and retrieves data via HTTP-requests to a stationary server and thus requires an internet connection.

5 Quality

Testing and quality assurance is an essential part of software development. By consistently testing the application during the development, one can ensure that fewer bugs are delivered to the final product, which in turn assures the quality of the software. Since the application is written in Java, JUnit is chosen as the framework for unit testing the application.

5.1 Testing

The Model has been thoroughly tested with the unit testing framework JUnit. The testing environment can be found in the package *test/java/com/goayo/debtify/model*. The tests are not limited to single static values, but instead use randomized inputs which gives a wider range of testing. By using seeds, the failed tests can be easily traced down in contrast to completely random tests which may or may not fail. Furthermore, assertions are made in testing that exceptions are thrown when as expected when invalid inputs are provided.

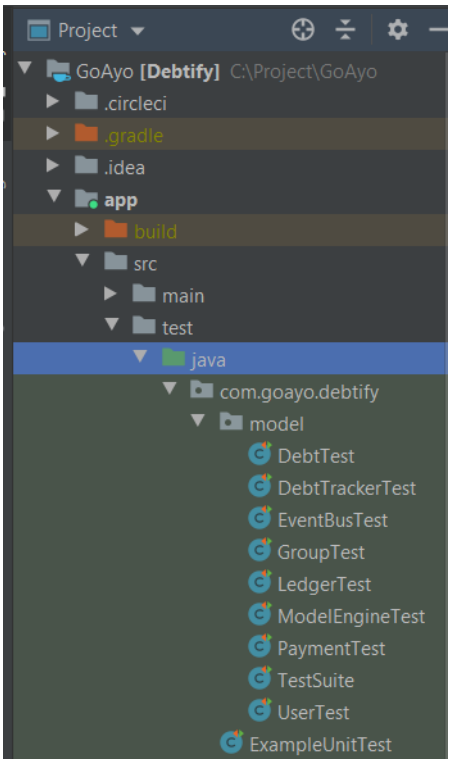

Figure 5.1: Path to the testing folder.

5.1.1 Continuous integration

The continuous integration system was incorporated with the project as a way to automate tests. By using the platform CircleCI, defects that have been pushed to the codebase could be identified and corrected sooner since CircleCI provides immediate feedback. The link can be found in the appendix.

5.2 Known issues

- The UI does not always respect the boundaries of the screen, on some devices, due to hard-coded dimension values.
- Logged in users are required to login again after reopening the app since the app never saves any cache.
- Due to the communication with the database being done on the main thread, the application may appear slow if the user has a poor internet connection.

5.3 Quality assurance reports

For quality assurance reports the project uses STAN, a tool for Java structure analysis. See appendix for the results.

5.4 Access control and security

Each time a user opens the application, the user is asked to provide a password to authenticate themselves. The user can never access groups that the user does not belong to, or any other sensitive data bound to other users.

The one piece of sensitive data handled by the application are the passwords sent in by users. These are stored in a persistent database hosted by the company server. The passwords are hashed using SHA256. This method is chosen due to the prototype scope of the application. In a real world application using a slower algorithm such as Argon2 is strongly recommended. However, the passwords are hashed outside of the model package and future deployments of the model will not require modification of the model to change the type of hashing. Furthermore the model will remain independent from any security libraries.

6 References

6.1 Bibliography

Bibliography

[1] *Android Developers*, sep. 2020. [Online]. Accessed: `https://developer.android.com/reference/android/app/Activity` [Retrieved 2020-10-22].

[2] "EventBusExplained," *GitHub*, oct. 2020. [Online]. Accessed: `https://github.com/google/guava/wiki/EventBusExplained` [Retrieved 2020-10-23].

[3] E. Gamma, J. Vlissides, R. Helm, R. Johnson, *Design Patterns: Elements of Reusable Object-Oriented Software*. USA: Addison-Wesley, 1994.

[4] "How to Write a Property Change Listener," *Oracle*, [Online]. Accessed: `https://docs.oracle.com/javase/tutorial/uiswing/events/propertychangelistener.html` [Retrieved 2020-10-22].

[5] "What is an XML file and How Do I Open One?", *Indeed*, oct. 2020. [Online]. Accessed: `https://www.indeed.com/career-advice/career-development/xml-file` [Retrieved 2020-10-23].

6.2 Tools

- Android Studio
  - The project's IDE.
  - `https://developer.android.com/studio`
- Lucidchart
  - An online software for creating UML diagrams.
  - `https://www.lucidchart.com/pages/`
- Slack
  - The project's communication platform to discuss confidential subjects, planning meetings and other work related topics.
  - `https://slack.com/intl/en-se/`
- Figma
  - A web-based platform for designing user interfaces.
  - `https://www.figma.com/`
- Zoom
  - For online meetings.
  - `https://zoom.us/`

6.3 Libraries

- JUnit
  - Java's unit testing framework.
  - `https://junit.org/junit4/`
- AndroidX
  - Android's API.
  - `https://developer.android.com/jetpack/androidx`
- GSON
  - Google's JSON parser library.
  - `https://github.com/google/gson`

A CircleCI

The link to the project on CircleCI. An account is required to access the site.

- `https://app.circleci.com/pipelines/github/OlofSjogren/GoAyo`

B Dependency analysis

Images from STAN regarding the core packages of Debtify.

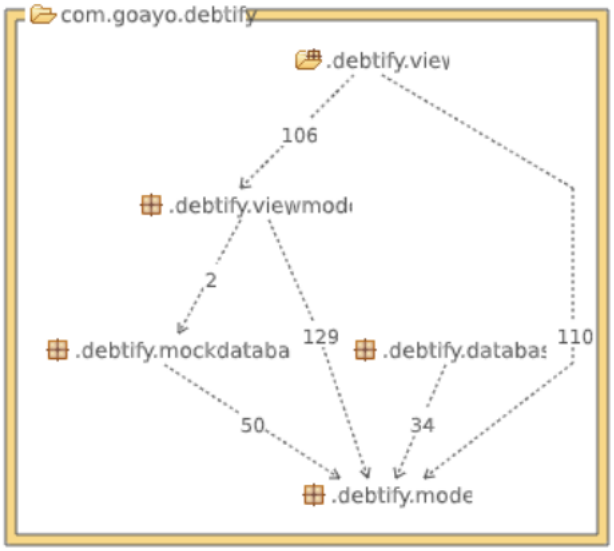

Figure B.1: Overall structure.

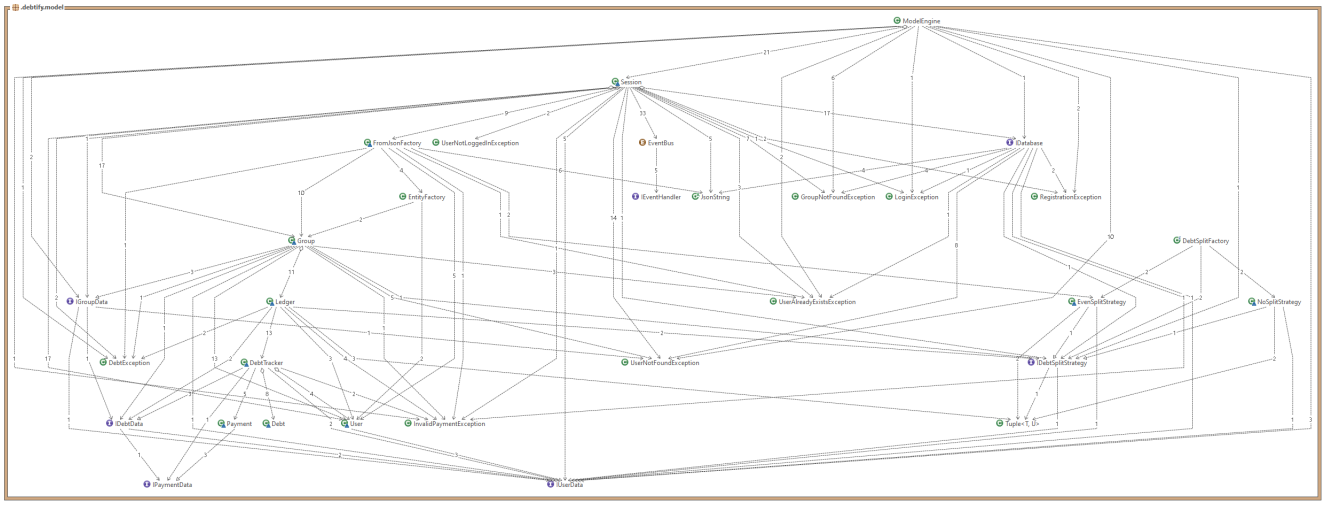

Figure B.2: Model structure.

Requirement & Analysis Document

Alex Phu, Oscar Sanner, Olof Sjögren, Yenan Wang

2020-10-20

Version 2.1

1. Introduction

The project aims to create a personal finance application that serves the purpose of handling debt between participants in a certain group. Keeping track of shared expenses between friends has never been easier.

The application is primarily aimed toward people who appreciate the convenience of smartphones and who want to utilize this to further organize their lives.

Debtify is not aimed primarily for business or professional use, but rather day-to-day use. It creates value for users by simplifying the menial and in some instances uncomfortable tasks of managing debts. By using the application, the experience of sharing expenses becomes more organized and enjoyable with an intuitive interface and pleasing design.

1.1 General characteristics of the application

- The application is an Android app written in Java.
- Users can register a new account with a name, phone number, and password.
- Users can add other users to their contacts.
- Users can create a group and add users from their contacts to the group.
- Users in a group can add new debts/payments to another user in the same group.
- Users can see a sorted log of all transactions in a group.

1.2 Scope of application

- No actual financial value will be transferred through the application. It will resemble a simplified ledger for personal debt between participants.
- The application will have access to a remote database where the application retrieves and stores data.

1.3 Definitions, acronyms, and abbreviations

- App
    - Abbreviation of application.
- View
    - A displaying page that the user can see.
- Endless Decimal:

    - A number which is not representable in decimals. As such the number will never be completely accurate. The most common example of this might be 10 divided by 3, resulting in 3,[an endless amount of threes].
- Hamburger-button
    - Is generally a button in the top-corner that unintentionally resembles a hamburger. That is, three horizontal lines stacked on each other.
- Notification
    - A message that pops up on a mobile device.
- Special Sign
    - A symbol which is not in the range of [A-Z], [a-z] or [0-9].
- JUnit
    - Java's unit testing framework.
- Master-branch
    - The repository's default and final branch. This is where the latest version of the application is found.

2. Requirements

2.1 User Stories

Initial user stories that the project was based on.

High Priority	Medium Priority	Low Priority
US01: Show debts for specific groups.	US07: Debt simplifier in Group.	US09: Activity feed.
US02: Create an account.	US12: My profile page.	US13: Graph functionality for debt in each group
US03: Log in.	US10: Add personal contacts.	US14: Graph functionality for debt for the logged-in account.
US04: Create groups.	US11: Access contacts from storage on the device	US15: Debt simplifier between individuals across groups
US05: Add user(s) to groups.		US16: Notification of actions involving debt
US06: Add debt to user(s) in a group.		
US08: Settle debt between users in a group.		
US19: Leave group.		

Figure 2.0: Completed User Stories are marked in green

High Priority

US01: Show debts for specific groups.

Implemented?

- **Yes**

Description

- As a user, I want to see all the added debts in a certain group so I can understand my debt situation.

Acceptance criteria

- Functional
  i. There exists a way to reach the View for the log of debts.
  ii. There exists a View for the log of debts.
  iii. The debts must be logged somewhere to be able to display them.
  iv. The debts are sorted by date and visible.
  v. There exists a way to exit the View.
- Non-functional
  i. The debt will be shown on a card.
  ii. The maximum amount of debts shown at once will be 4.

iii. If the list of debts exceeds 4, the user will be able to scroll down to see more debts.

US02: Create an account

Implemented?

- **Yes**

Description

- As a user, I want to be able to create an account so I can use the application and save my personal data.

Acceptance criteria

- Functional
  i. If the user is not logged in, the user is supplied with a button that leads the user to create an account.
  ii. The user is prompted to enter Phone number, Name, and Password on account creation.
  iii. After successfully registering, the data is automatically stored.
- Non-functional
  i. The user's credentials need to be stored in a database if the registration is successful and the user is connected to the internet.
  ii. If the user is not connected to the internet, the user shan't be able to register.

US03: Log in

Implemented?

- **Yes**

Description

- As a registered user, I want to log in so I can access my account and my saved data.

Acceptance criteria

- Functional
  i. The user will be introduced to the login page if the user is not logged in.
  ii. On the login page, a user is prompted to enter the phone number and password and can subsequently log in.
  iii. There exists a way to log out of the account.
- Non-functional
  i. If the user is not connected to the internet, the user shan't be able to log in.

US04: Create groups.

Implemented?

- **Yes**

Description

- As a user who owes/lends money to other user(s), I want to create a group of users so that I can keep track of my/their debt.

Acceptance criteria

- Functional
    i. There exists a group creation View.
    ii. There exists a way to navigate to the group creation View.
    iii. In the group creation View, the user can specify the name of the group.
    iv. In the group creation View, the user can invite other users to the group.
    v. Once the group is created, the user should be able to see it somewhere.
- Non-functional
    i. If the user is in several groups, the user should be able to scroll to see them all.

US05: Add user(s) to groups .

Implemented?
- **Yes**

Description
- As a user who just met my significant other, I want to add them to a group so I can put them in debt.

Acceptance criteria
- Functional
    i. There exists a way to add users to a group.
    ii. Ability to add other users to the group.
    iii. Multiple users should be able to be added at once.
- Non-functional
    i. The added user does not have to accept anything and joins the group immediately.

US06: Add debt to user(s) in a group.

Implemented?
- **Yes**

Description
- As a user, I want to put one or more users - in a group - in debt so that I can track it in the future.

Acceptance criteria
- Functional
    i. There exists a View where the user can create debt.
    ii. There is a way in the group view to navigate to the debt creation View.
    iii. The user can put multiple users in debt at once.
    iv. The user can specify the total amount of debt.
    v. The user can specify the reasoning behind the debt.
    vi. The user can choose to split the debt equally to the selected users.
    vii. The user can send the debt to the selected users.
- Non-functional
    i. The debt creation button should pop out with a distinct color.

ii. The debt should be added online and therefore backed up before it shows up as added in the group.

US08: Settle debt between users in a group.

Implemented?

- **Yes**

Description

- As a user, I want to settle a pending debt between one or more users in a group so that I am no longer indebted.

Acceptance criteria

- Functional
  i. There exists a View where the user can select and settle a pending debt.
  ii. The user can specify how much of the debt to settle.
  iii. The user cannot settle more than the existing debt amount.
  iv. The debt is updated after a settlement.
- Non-functional
  i. The pay button should pop out with a distinct color.
  ii. The payment should be added online and therefore backed up before it shows up as added in the group.

US19: Leave group.

Implemented?

- **Yes**

Description

- As a user, I want to be able to leave a group that I no longer want to be a part of.

Acceptance criteria

- Functional
  i. There exists a button where the user can use to leave the specified group.
- Non-functional
  i. All debts and lendings associated with the user will cease to exist when the user has left the group.

Medium Priority

US07: Debt simplifier in Group.

Implemented?

- No

Description

- As a user who is bad at math, I want the app to simplify the overall debt so that I can see debts in a comprehensible way.

Acceptance criteria

- Functional
  i. There exists a button that displays the simplified view of debts in a group when pressed.

- Non-functional
    i. The debts shall be simplified in the following ways.
        1. It only simplifies debts related to the logged-in user.
        2. If user A owes money to user B and user B owes money to user C, then user A owes money to user C, if and only if the amount of money user A owed to user B is less or equal to the amount user B is owed to user C. Else user A just owes to user B.
    ii. The simplifier is reliable to the degree that it's reversible with an error of 0.001 units when working with endless decimals.

US12: My profile page

Implemented?

- No

Description

- As a user, I want to be able to see and edit my profile to keep it updated.

Acceptance criteria

- Functional
    i. There is a name displayed on the profile page.
    ii. There is a visible profile picture on the page.
    iii. There is a visible biography paragraph on the page.
    iv. I should be able to edit the following information on my profile.
        1. Name
        2. Image
        3. Biography
- Non-functional
    i. The user can always see his/her name on the profile page.
    ii. The user can always see his/her image on the profile page.
    iii. The user can always see his/her biography on the profile page.
    iv. The user can fill out information with all special signs following UTF-8.

US10: Add personal contacts

Implemented?

- **Yes**

Description

- As a user with an account, I want to be able to add personal contacts so that I can save my acquaintances' information and quickly access them in groups.

Acceptance criteria

- Functional
    i. There exists a contact view that can be accessed.
    ii. On the contacts-page, all added contacts are listed.
    iii. On the contacts page, there exists a way to add and remove a contact.
- Non-functional
    i. The user can never add the same user twice.
    ii. The user can add an infinite amount of contacts.

US11: Access contacts from storage on the device

Implemented?

- No

Description

- As a user, I want to be able to add a new person through the inbuilt contacts (in the phone) so that I don't have to manually write in their information.

Acceptance criteria

- Functional
    i. Able to see contacts.
    ii. Able to select a contact.
    iii. The app should be able to see if it is an existing contact.
    iv. The app should verify that the added contact has an account on the app.
- Non-functional
    i. The view for showing contacts from the phone's contact book will show at most five contacts at a time.

Low Priority

US09: Activity feed.

Implemented?

- No

Description

- As a user, I want to be able to see a feed of the most recent events so that I can keep myself updated.

Acceptance criteria

- Functional
    i. When the user is logged in to the application, it is greeted by an Activity feed.
    ii. The Activity feed displays all events/actions done by the logged-in user.
- Non-functional
    i. The activity feed will show the recent events in reverse-chronological order.

US13: Graph functionality for debt in each group

Implemented?

- No

Description

- As a user, I want to be able to see all the debts in a group displayed in a graph so that I can easily understand it.

Acceptance criteria

- Functional
    i. There is a way to navigate to the group-specific debt-graph.
    ii. There is also a graph for displaying debt distribution by category.

iii. All the debts in the groups are displayed in a graph.
- Non-functional
  i. The graph is shown as a pie.

US14: Graph functionality for debt for the logged-in account.

Implemented?
- No

Description
- As a user, I want to be able to see all the debts to individuals displayed in a graph so that I can more easily understand it.

Acceptance criteria
- Functional
  i. There exists a way to navigate to the graph.
  ii. The graph displays a history of debts.
- Non-functional
  i. The graph displays the history for up to 12 months.

US15: Debt simplifier between individuals across groups

Implemented?
- No

Description
- As a user, I want my debts to a specific user across all mutual groups to be calculated to what I owe that user in total, so I can pay that user.

Acceptance criteria
- Functional
  i. If a debt is added to a group or a user in a group, from a user, the exact debt to each user is calculated.
  ii. There exists functionality to simplify the debt between users.
- Non-functional
  i. The simplifier is reliable to the degree that it's reversible with an error of 0.001 units when working with endless decimals.

US16: Notification of actions involving debt

Implemented?
- No.

Description
- As a user, I want to be notified of all actions done to the debts that affect me so I won't miss it.

Acceptance criteria
- Functional
  i. If a debt is added to a user or more, a notification will be sent to all users affected by it.
  ii. If a debt is settled between two or more users, a notification will be sent to all users affected by it.

- Non-functional
    i. The notification shall be sent as soon as possible whenever the user is logged in and is connected to the internet.

2.2 Definition of Done

- The code is peer-reviewed.
- Task has met the acceptance criteria.
- The code has passed JUnit tests.
- The application builds and compiles.
- The code is documented (JavaDoc).
- The code is merged into master-branch.
- The UML is updated.

2.3 User Interface

The focus of this section is to describe the navigation among the views and to visualize the GUI through screenshots.

2.3.1 Starting Page

The user is greeted by the login page where the user has the opportunity to register a new account or log in to an existing account.

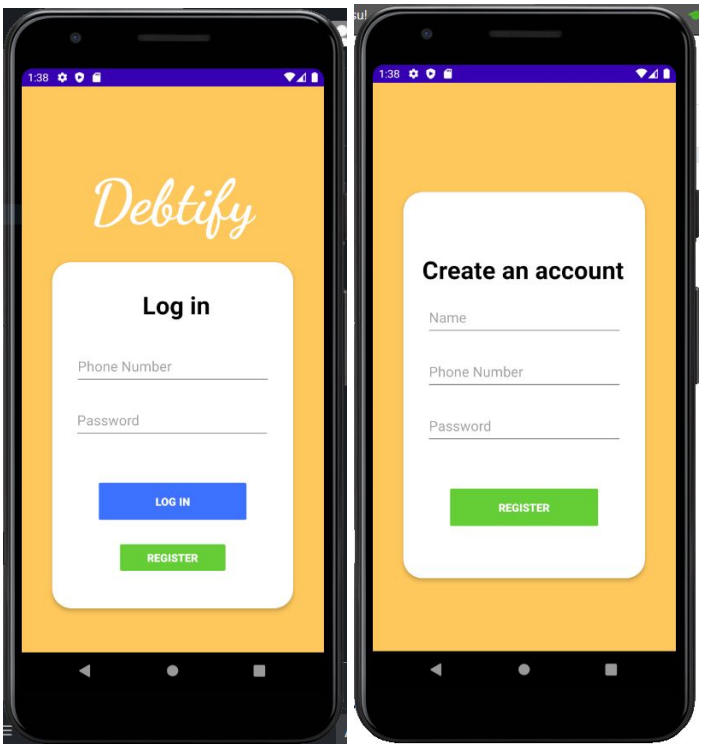

Figure 2.1: Login page to the left, registration page to the right.

2.3.2 Dashboard

After successfully logging in, the dashboard is displayed which contains minor user info at the top, a view that shows all of the groups - as card views - that are associated with the logged-in user, and the total debt balance for the user among the groups. There's also a floating action button in the bottom right which navigates to the group creation page.

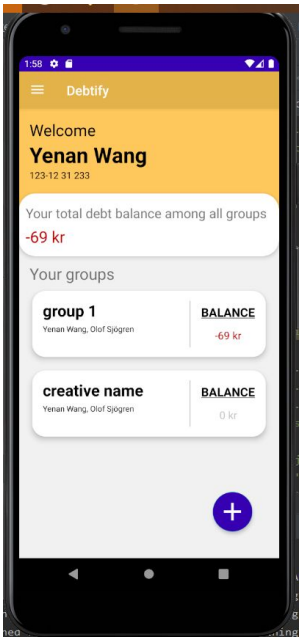

Figure 2.2: Dashboard.

2.3.3 Drawer Layout

A side panel appears by pressing the hamburger-button in the top-left corner of the dashboard, or alternately by dragging from the left edge to the right. The side panel acts as the application's global navigation.

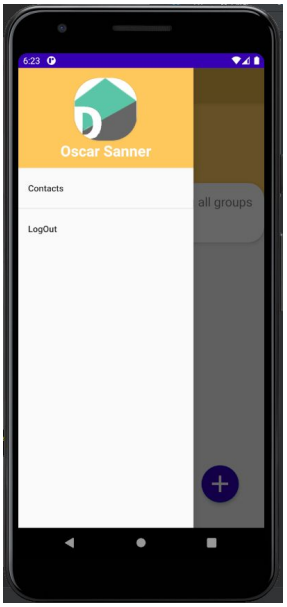

Figure 2.3: Drawerlayout which is accessed through the hamburger-button.

2.3.4 Contacts Page

The contacts page is accessed through the drawer layout which displays the currently logged-in user's contacts. The user can also add or remove contacts.

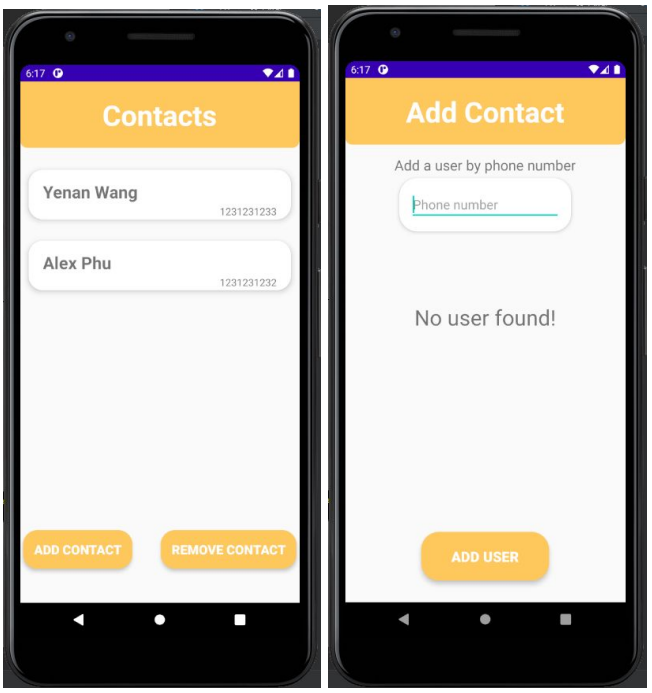

Figure 2.4: Contacts page to the left, page to add a contact to the right.

2.3.5 Create a Group Page

The group creation page is reached by pressing the floating action button. The user has to pick the participants of the new group before being able to name the group.

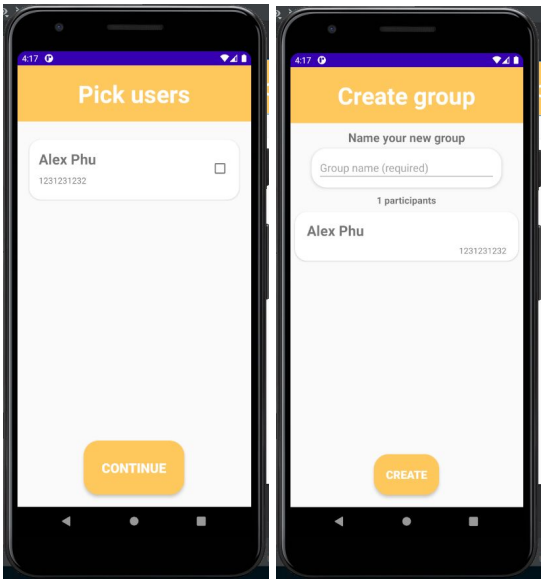

Figure 2.5: Pick user page on the left, page to name on the right.

2.3.6 Detailed Group View

The user enters the detailed view of a group by clicking on the group card views on the dashboard. This view presents a history of all payments and debts and the current debt balance for the logged-in user. Moreover, there's also an options menu in the top right corner which has three menu items, "Show members", "Add member", and "Leave group". At last, there's bottom navigation which consists of two buttons "Add payment" and "Add debt".

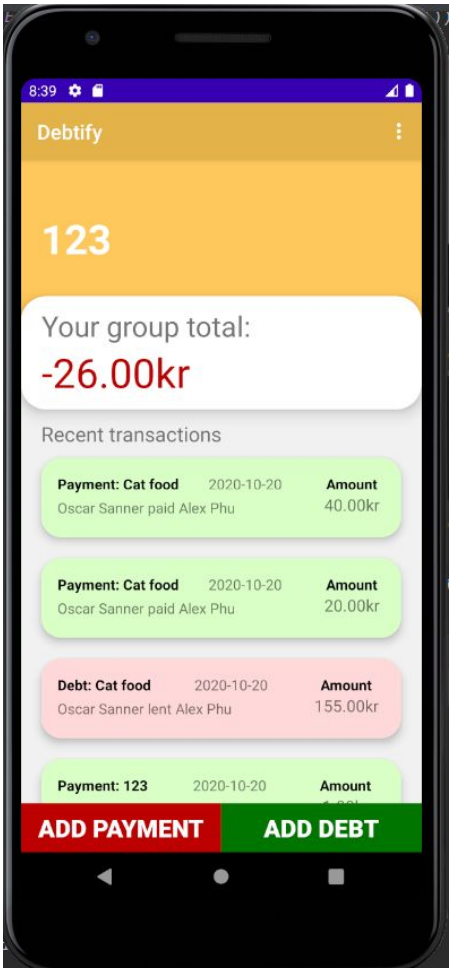

Figure 2.6: Detailed view of a group.

2.3.7 Add Debt Page

The user can navigate to the add debt page through a detailed group's bottom navigation. On this page, the user can specify who the lender and borrowers are, and whether or not the debt should be split evenly among the borrowers.

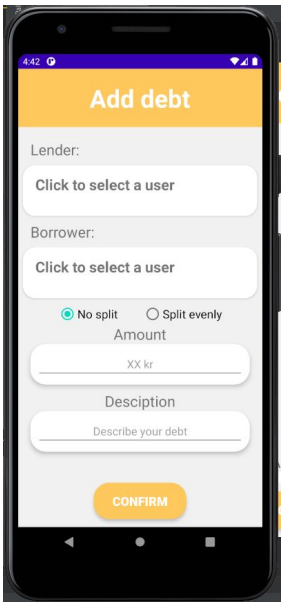

Figure 2.7: Add debt page.

2.3.8 Settle Debt Page

Similar to the “Add debt page”, the settle debt page can be accessed through a detailed group view’s bottom navigation. The user has the ability to settle any debts that are active in the group on this page.

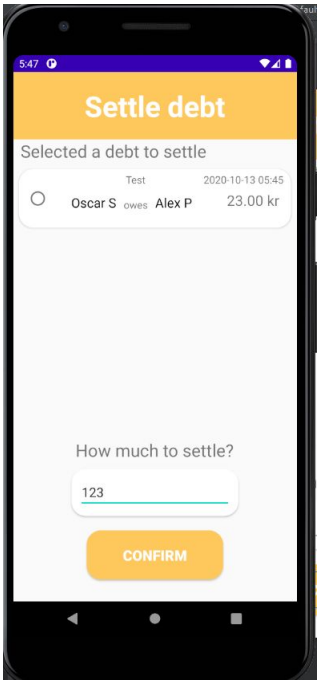

Figure 2.8: Settle debt page.

3. Domain model

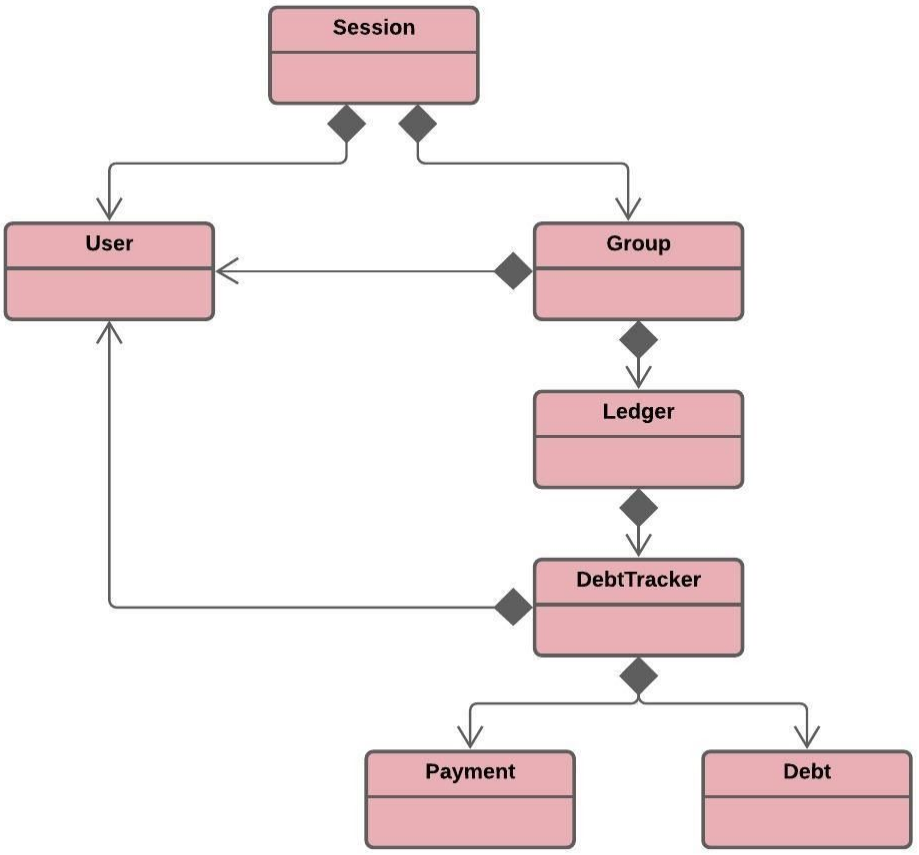

Figure 3.1: Domain model.

3.1 Class responsibility

Session
This class manages account-level interactions. This class could be considered as the main class. It holds the logged-in user as well as acts as an aggregate for the logged-in user's details, such as groups and contacts.

User
The *User* class is the representation of customers who are using the application.

Ledger
A ledger manages all the *DebtTrackers* in a specific group and serves as a traditional ledger. It's held by the group and organizes debts and payments.

DebtTracker
The *DebtTracker* represents the current debt balance between the lender and the borrower. Hence, this class keeps track of the details regarding the initial debt and all of the subsequent payments.

Debt
A *Debt* represents the amount of money that is owed.

Payment
A debt *Payment* represents the amount of money that is paid back.

4. References

4.1 Tools

- GitHub
  - Online version control using Git.
    - https://github.com/
- Android Studio
  - The project's IDE.
    - https://developer.android.com/studio
- Lucidchart
  - An online software for creating UML diagrams.
    - https://www.lucidchart.com/pages/
- Figma
  - A web-based platform for designing user interfaces.
    - https://www.figma.com/
- Slack
  - The project’s communication platform to discuss confidential subjects.
    - https://slack.com/intl/en-se/
- Zoom
  - For online meetings.
    - https://zoom.us/

4.2 Libraries

- JUnit
  - Java’s unit testing framework.
    - https://junit.org/junit5/
- AndroidX
  - Android framework
    - https://developer.android.com/jetpack/androidx
- Gson
  - Google’s library for parsing JSON files.
    - https://github.com/google/gson

Peer Review of Chans

Group GoAyo

Design principles and implementation

A notable design principle this codebase follows is the Single Responsibility Principle. The model package has a good separation for conceptual responsibilities. It is clear that they have thought thoroughly about their model classes' purposes and responsibilities. However, the controller classes hold multiple responsibilities by being both the view and controller which fails to achieve SRP, more on this later.

It is however a bit concerning that some classes have enormous if-statements in their methods. This breaks against OCP since it would not be possible to add more features without having to modify the existing condition checks. It is also worthy of mention that the model package does not use any polymorphism which might explain the usages of giant if-statements since it disables their ability to use a behavioural pattern like the State pattern. This topic will be touched on later.

The lack of abstraction and giant if-statements make the codebase difficult to maintain and require extra effort to remove/add functionality. For instance, if a new game phase were to be added then all the conditionals that check the phases would have to be modified which usually is not the ideal way to implement a feature.

Code documentation, readability, and naming

Most of the public classes in the model are well-documented, however, Java-documentation is not necessary for private methods as they aren't accessible for the end user. For instance, *resetSelectedSpace*() is missing Java documentation, even if the method name might be explicit. There should also be some comments inside the very large methods (see figure 1, appendix) that explain the more obscure codes.

Regarding the subject of coding style, there seem to be some inconsistencies. For instance, the indentation of curly brackets should be following the same style, even if both styles are valid. Another example would be the abundance of new lines that are superfluous. Lastly, most methods and classes have appropriate names that are inherently intuitive. However, there are a few variables that are incomprehensible. Particularly the *selectedSpace* and *selectedSpace2* in *Board* class as they aren't distinctly implying their context. Additional documentation would clear this up.

Code modularity, structure, dependencies, and abstractions

In terms of modularity, the model is completely independent of other packages, furthering the portability and re-deployability to other platforms. This is one of the most important rules with MVC. However, there are parts of the pattern implementation that could be looked over and improved. The controller concept is coupled in with the view concept in a way that seems unnatural. Each controller is also an anchor pane and acts as a view more or less, whilst also containing logic for handling different interactions with itself. The expected approach would be to have the logic in the controllers, perhaps as listeners to view elements placed in the view. More of this in the improvements section of this document.

The model has very few outgoing, non-Java-Class-Library dependencies. This, as previously mentioned, can make the model re-deployable to different platforms. The one problem, however, is that the *Player* and the *Board* class is dependent on the JavaFX library. While the model is still re-

deployable to other platforms supporting JavaFX, it's not at all reusable in environments where JavaFX is inapplicable. The developer would first have to handle the colour issue and modify the model to make use of it.

Little can be said about the use of abstractions, as there is no inheritance in the model. The model uses collection classes to handle sets of entities, and the highest abstraction applicable is always used, which is very good. I.e. the *List* type is used instead of the concrete *ArrayList*.

Testing, performance, and security

A few classes are thoroughly tested, albeit some other test classes are empty. A good way to ensure the appropriate classes and their functionality is being tested thoroughly is to make sure the test coverage includes the critical functionality in the model package.

There are some complications in running the game on Windows computers with a display scaling set above 100%. This could perhaps be fixed by using JavaFX's built-in methods *setMaximized()* or *setFullScreen()*. Other than that, the application runs well.

Most classes in the model are package-private but there are some inconsistencies (*Player* and *Space*), which perhaps should be set to package-private. Although these are minor concerns, they yield substantial security and assure that no other packages are dependent on the internal functionality of the model package. Consider adding an interface for accessing internal model components to allow change of the actual implementation without affecting the other components' dependencies.

Attributes and corresponding setters are marked private or package-private which is great. However, in some instances, the getters are returning direct references to attributes which may result in alias problems where the private-marked attribute is circumvented by the public getter. Consideration for defensive copying is highly relevant as well as a general notion for immutability.

Suggested improvements

A simple improvement to apply to the controller and view package would be to split every controller up in two different classes. One class would have the JavaFX components and the other class would have the responsibility of controlling these components. The classes would be put in the respective packages' view and controller. The relationship could then be such that a controller has initialised the components in the view class and adds listeners to the components. The controller would then call the model instance and execute various commands depending on which action occurred in the view. See figure 2 in the appendix for a concrete example.

Another potential improvement is to, instead of the enum *Phase*, utilize a State pattern where each state class corresponds to the current phases, e.g. *AttackState*, *DefenceState*. The *Round* class would have an attribute of a general state interface the aforementioned classes implements. The current *startPhase*(...)-method would be replaced with a general method that delegates through the attribute. This would reduce the currently massive conditionals and instead use polymorphism through a general interface. Furthermore, this allows for easy extension of the current phases, as well as the implementation of new phases of the round.

Appendix:

```
boolean startPhase(Space selectedSpace, Space selectedSpace2, Player currentPlayer, int units)
{
    if(selectedSpace != null)
    {
        if(currentPhase == Phase.DEPLOY)
        {
            return Deployment.startDeployment(selectedSpace, currentPlayer,units);
        }
        else if(selectedSpace2 != null){
            switch (currentPhase)
            {
                case ATTACK:
                    if(Attack.DeclareAttack(selectedSpace, selectedSpace2, selectedSpace.getUnits()) ) {
                        dices = Attack.calculateAttack(selectedSpace, selectedSpace2);
                        return true;
                    }
                    return false;
                case MOVE:
                    return Movement.MoveUnits(selectedSpace,selectedSpace2, units);
                default:
                    return false;
            }
        }
    }
    return false;
}
```

Figure 1: Giant if-statements makes it difficult to maintain.

```
public class ViewClass extends AnchorPane {
    Button button;

    public ViewClass() { button = new Button(); }

    //No logic what so ever. Just a dumb view.
}
```

```
public class ControllerClass {
    ViewClass viewClass;
    Model model;

    public ControllerClass(ViewClass viewClass){
        this.viewClass = viewClass;
        model = Model.getInstance();
        initViewButtons();
    }

    private void initViewButtons() {
        viewClass.button.setOnAction(new EventHandler<ActionEvent>() {
            @Override
            public void handle(ActionEvent actionEvent) {
                model.shutDown(); // Or some other model command.
            }
        });
    }
}
```

Figure 2: Example of how the relationship between a controller and a view could look like, to separate the responsibilities of the view and the controller.