System Design Document for Debtify

Oscar Scanner, Olof Sjögren, Alex Phu, Yenan Wang

2020-10-01

Version 1

Table of 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 should be extensible in order to establish few dependencies and low coupling.

1.2 Definitions, acronyms, and abbreviations

- US - User story, used to tag each user story. It does not mean the United States.
- MVVM - Model-View-ViewModel architecture in Android.
- Debitify - Placeholder name for our application. It is not decided. I repeat, **IT IS NOT DECIDED.**
- Entity - Identifiable object used in the application, such as users, groups, debts and payments.

2 System architecture

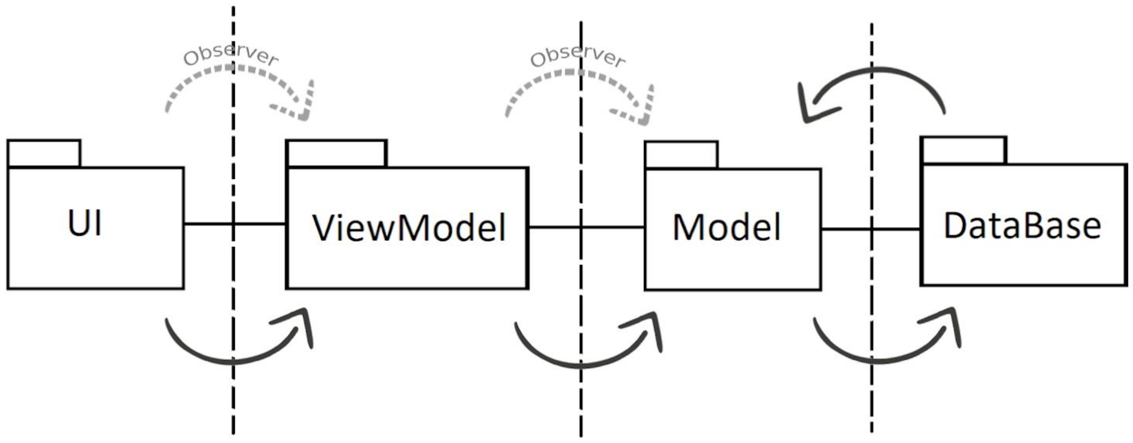


Figure 2.1: Components of the application.

The application is divided into four parts, the UI which acts as the presentation layer, the view-model which acts as the mitigator between the model and view package, the model which is the domain, and lastly the database. The UI package is the user interface of the app which is responsible for presenting relevant data as well as handling user input. The UI package communicates with the model through the view-model to get the desired data to display or to delegate user inputs to the model. Lastly, the model is dependent on the (for now) hardcoded database which is instantiated at the beginning of the app's lifecycle.

The most overall, top-level description of your application. If your application uses multiple components (such as servers, databases, etc.), describe their ***responsibilities*** *here and show how they are* ***dependent*** *on each other and* ***how they communicate*** *(which protocols, etc.)*

You will describe the ***'flow' of the application*** *at a high level. What happens if the application is started (and later stopped) and what the normal flow of operation is. Relate this to the different components (if any) in your application.*

3 System design

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

Make sure that these models stay in 'sync' during the development of your application.
Describe which (if any) ***design patterns*** *you have used.*
The above describes the static design of your application. It may sometimes be necessary to describe the dynamic design of your application as well. You can use a UML sequence diagram to show the different parts of your application to communicate in what order.

3.1 MVVM

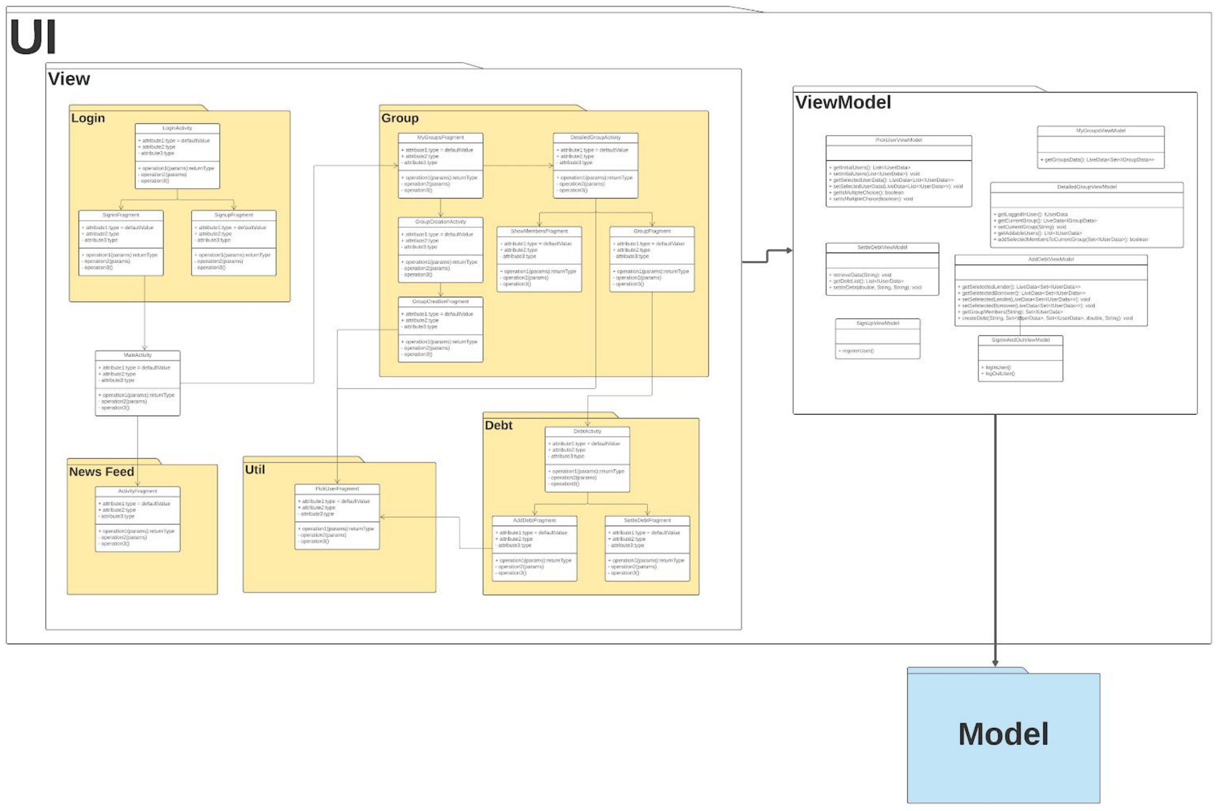


Figure 3.1: MVVM structure of Debtify

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 clean 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 single dedicated ViewModel, however, a ViewModel never becomes aware of a View with the purpose of negating effects of UI state change to ViewModels. Those ViewModels communicate with the Model for data and thus provide a simplistic and exclusively customized facade interface for Views.

3.1.1 Why MVC was disregarded

The core idea between MVVM and MVC is 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 refers to the pattern where the XML files are considered as the views and the Activity as a controller, as opposed to MVVM who 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 Android application's internal Lifecycle 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 Lifecycle works. By using ViewModels, an inbuilt class from the Android API, the obfuscation from managing Activities' life cycle becomes negligible.

3.1.2 Package relations

As aforementioned, each View class owns a dedicated ViewModel class and there will not be any extra layers between these two packages as that would be redundant and defies against the purpose of using ViewModel. Between the ViewModel and Model packages however, there exists a facade class with the intent of simplifying the Models' functionalities so that the ViewModels can have minimum business logic for data retrieval and naturally, not becoming directly dependent on the Model package. This also allows modification to the Model without directly affecting any dependencies.

There are a few Model interfaces which are globally accessible. Those interfaces create a typesafe and secure environment for data retrieval from the Model and neutralize the side effects while keeping the dependencies under control by eliminating dependencies to the core classes in the Model.

*Draw a UML package diagram for the top level for **all components that you have identified above** (which can be just one if you develop a standalone application). Describe the **interfaces and dependencies** between the packages. Describe **how you have implemented the MVC design pattern**.*

3.2 Model

The Debtify model package is a completely independent package. As such it can be driven by any other view or controller, for any system. In Debtify however, it's driven by the ViewModel package in an MVVM architecture. The model holds all instances of entities used in the application. It's 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. The highest level module of the package is the *Account* class. This class is supposed to always represent the account of the logged in user. The account holds several lists of 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.

The account also handles the database. The relation between the database and the model is such that an account instance holds a database of the static type *IDatabase*, which specifies the promises of a database class. The promise of a database class is to be able to make all the necessary calls to the database server, and to return Java objects in the case of queries. Therefore the database must know the specific implementation of the various entity type classes in the model and it should not be placed in a subpackage.

3.2.2 The facade class

The Model package is accessed through a single public facade class, known as the *ModelEngine*. This facade applies the singleton pattern and 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 package of the Model package to restrict coupling and to preserve the integrity of the package. Queries to the model thus returns 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.

If the client code using the model is required to keep itself updated to the model, the client code can subscribe via the implemented Observer pattern. Classes in the model are hence observable and an observer is passed through the *ModelEngine* and handed to the appropriate class.

3.2.3 Domain model vs design model

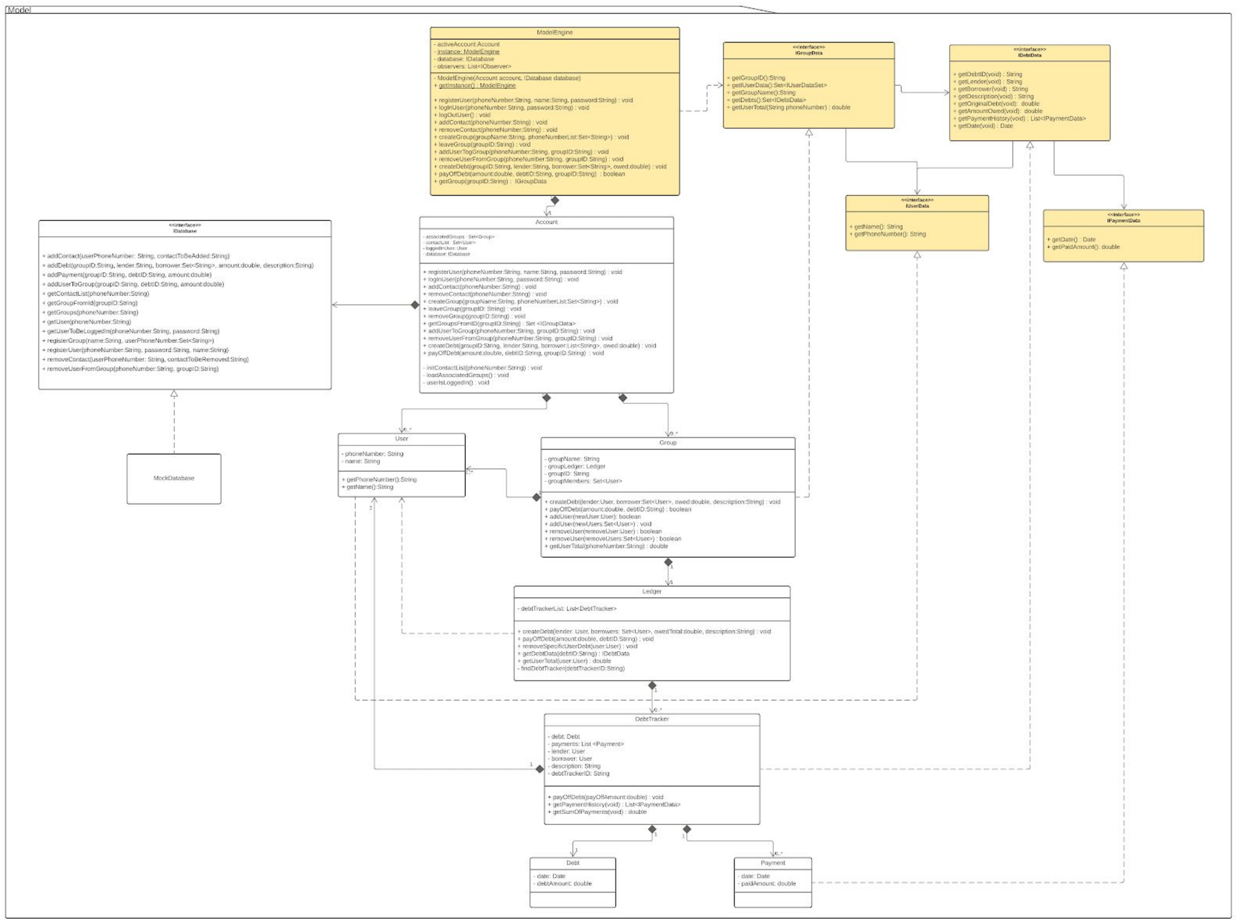

Model
ModelEngine
- activeAccount:Account
- instance: ModelEngine
- database: IDatabase
- observers: List<IObserver>
- ModelEngine(Account account, IDatabase database)
+ getInstance() : ModelEngine
+ registerUser(phoneNumber:String, name:String, password:String) : void
+ logInUser(phoneNumber:String, password:String) : void
+ logOutUser() : void
+ addContact(phoneNumber:String) : void
+ removeContact(phoneNumber:String) : void
+ createGroup(groupName:String, phoneNumberList:Set<String>) : void
+ leaveGroup(groupID:String) : void
+ addUserTogGroup(phoneNumber:String, groupID:String) : void
+ removeUserFromGroup(phoneNumber:String, groupID:String) : void
+ createDebt(groupID:String, lender:String, borrower:Set<String>, owed:double) : void
+ payOffDebt(amount:double, debtID:String, groupID:String) : boolean
+ getGroup(groupID:String) : IGroupData
<<interface>>
IGroupData
+ getGroupID():String
+ getIUserData():Set<IUserDataSet>
+ getGroupName():String
+ getDebts():Set<IDebtData>
+ getUserTotal(String phoneNumber) : double
<<interface>>
IDebtData
+ getDebtID(void) : String
+ getLender(void) : String
+ getBorrower(void) : String
+ getDescription(void) : String
+ getOriginalDebt(void): double
+ getAmountOwed(void): double
+ getPaymentHistory(void) : List<IPaymentData>
+ getDate(void) : Date
<<interface>>
IUserData
+ getName(): String
+ getPhoneNumber(): String
<<interface>>
IPaymentData
+ getDate() : Date
+ getPaidAmount(): double
<<interface>>
IDatabase
+ addContact(userPhoneNumber: String, contactToBeAdded:String)
+ addDebt(groupID:String, lender:String, borrower:Set<String>, amount:double, description:String)
+ addPayment(groupID:String, debtID:String, amount:double)
+ addUserToGroup(groupID:String, debtID:String, amount:double)
+ getContactList(phoneNumber:String)
+ getGroupFromId(groupID:String)
+ getGroups(phoneNumber:String)
+ getUser(phoneNumber:String)
+ getUserToBeLoggedIn(phoneNumber:String, password:String)
+ registerGroup(name:String, userPhoneNumber:Set<String>)
+ registerUser(phoneNumber:String, password:String, name:String)
+ removeContact(userPhoneNumber: String, contactToBeRemoved:String)
+ removeUserFromGroup(phoneNumber:String, groupID:String)
MockDatabase
Account
- associatedGroups : Set<Group>
- contactList : Set<User>
- loggedInUser: User
- database: IDatabase
+ registerUser(phoneNumber:String, name:String, password:String) : void
+ logInUser(phoneNumber:String, password:String) : void
+ addContact(phoneNumber:String) : void
+ removeContact(phoneNumber:String) : void
+ createGroup(groupName:String, phoneNumberList:Set<String>) : void
+ leaveGroup(groupID: String) : void
+ removeGroup(groupID:String) : void
+ getGroupsFromID(groupID:String) : Set <IGroupData>
+ addUserToGroup(phoneNumber:String, groupID:String) : void
+ removeUserFromGroup(phoneNumber:String, groupID:String) : void
+ createDebt(groupID:String, lender:String, borrower:List<String>, owed:double) : void
+ payOffDebt(amount:double, debtID:String, groupID:String) : void
- initContactList(phoneNumber:String) : void
- loadAssociatedGroups() : void
- userIsLoggedIn() : void
User
- phoneNumber: String
- name: String
+ getPhoneNumber():String
+ getName():String
Group
- groupName: String
- groupLedger: Ledger
- groupID: String
- groupMembers: Set<User>
+ createDebt(lender:User, borrower:Set<User>, owed:double, description:String) : void
+ payOffDebt(amount:double, debtID:String) : boolean
+ addUser(newUser:User): boolean
+ addUser(newUsers:Set<User>) : void
+ removeUser(removeUser:User) : boolean
+ removeUser(removeUsers:Set<User>) : boolean
+ getUserTotal(phoneNumber:String) : double
Ledger
- debtTrackerList: List<DebtTracker>
+ createDebt(lender: User, borrowers: Set<User>, owedTotal:double, description:String) : void
+ payOffDebt(amount:double, debtID:String) : void
+ removeSpecificUserDebt(user:User) : void
+ getDebtData(debtID:String) : IDebtData
+ getUserTotal(user:User) : double
- findDebtTracker(debtTrackerID:String)
DebtTracker
- debt: Debt
- payments: List <Payment>
- lender: User
- borrower: User
- description: String
- debtTrackerID: String
+ payOffDebt(payOffAmount:double) : void
+ getPaymentHistory(void) : List<IPaymentData>
+ getSumOfPayments(void) : double
Debt
- date: Date
- debtAmount: double
Payment
- date: Date
- paidAmount: double
1
0..*
2

Figure 3.2: The UML design model diagram.

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. These classes are as previously mentioned implementing interfaces to hide concrete implementation and to make objects immutable when accessed by client code. The addition of the *ModelEngine* class is also important to note as this serves as an access point to the package.

*Create a UML class diagram for every package. One of the packages will contain the model of your application. This will be the design model of your application, describe in detail **the relation between your domain model and your design model**. There should be a clear and logical relation between the two.*

3.3 UI

The View is the user interface of the application, which consists of Activities and Fragments. 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 the ViewModel instead. The ViewModel is an abstraction of the View who retrieves the required data from the Model and exposes such data for the View to display. The ViewModel sends the data through LiveData which is an observable. 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". Following this principle we have divided our activities into following classes.

- *DebtActivity*
- *DetailedGroupActivity*
- *GroupCreationActivity*
- *LoginActivity*
- *MainActivity*

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 or a portion of 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 can be seen as the presentation layer, whereas the fragment - a unique class for each XML file - controls its behaviour, thus separating the logic from the UI. Since fragments controls the XML files, the fragments are also called the UI controllers. For instance, a button is declared in the XML file but its behaviour will be decided in the fragment.

3.4 ViewModel

The ViewModel fills the role of acting as a connection between the UI and Model. Each View which 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 observers to invert dependencies and thus allow for a looser coupling.

3.4.1 Updating the ViewModels

A ViewModel will upon initialization add itself as an observer to a specific part of the Model. Which part depends on the ViewModel's area of responsibility, e.g. GroupViewModel will listen to Model updates regarding the logged in User's groups. This allows the ViewModel to always be updated when the Model is updated. This is especially important since the Model utilizes a database which in theory can be updated from outside sources and should in turn update the Model. The aforementioned Observer pattern makes sure the ViewModel is notified of such updates.

3.4.2 The Views and the ViewModels

The ViewModel itself contains MutableLiveData properties. The MutableLiveData class represents mutable data which can be observed for mutations by another object, which in this application is the corresponding View. The MutableLiveData reflects the data in the Model which is observed by ViewModel. The set-method for the MutableLiveData acts both as a setter for the property as well as a notifier for the listening View.

All ViewModel will thus have no dependency on any Views, the View will both execute methods it's ViewModel as well as listen to any updates the ViewModel announces. This also allows for many-to-one relationships between Views and ViewModels, where multiple Views utilize the same ViewModel thus reusing the ViewModel's functionality in different contexts.

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.

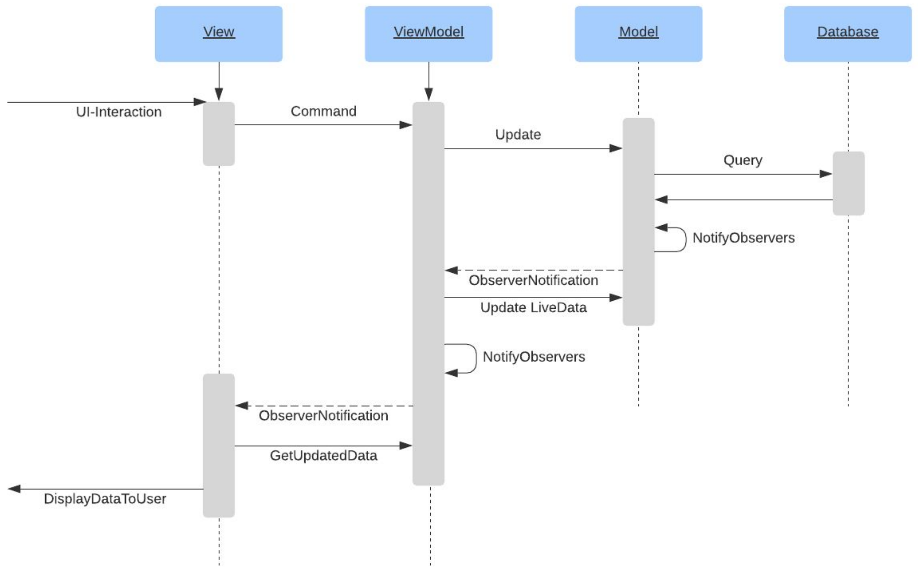


Figure 3.1: The run sequence for a generic example of a typical user interaction.

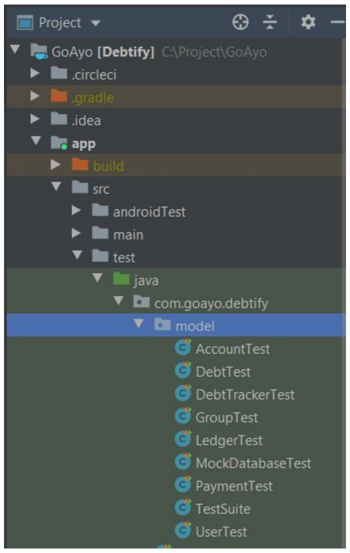
4 Persistent data management

The current implementation is a prototype and a proof of concept. As such it has no persistent data, accessible between sessions.

If your application makes use of persistent data (for example stores user profiles etc.), then explain how you store data (and other resources such as icons, images, audio, etc.).

5 Quality

Testing and quality assurance is an essential part of software development. An application cannot be kept up to date and survive the hardships of modern society without a consistently maintained testing environment. Since our application is written in Java, we have therefore chosen the JUnit testing framework to test our application.



```
Project
▼ GoAyo [Debtify] C:\Project\GoAyo
  ▶ .circleci
  ▶ .gradle
  ▶ .idea
  ▼ app
    ▶ build
    ▼ src
      ▶ androidTest
      ▶ main
      ▼ test
        ▼ java
          ▼ com.goayo.debtify
            ▼ model
                AccountTest
                DebtTest
                DebtTrackerTest
                GroupTest
                LedgerTest
                MockDatabaseTest
                PaymentTest
                TestSuite
                UserTest
```

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 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.

Describe how you test your application and where to find these tests. If applicable, give a link to your continuous integration.

5.2 Known issues

- The application is unfinished. Meaning it is still at the development stage.
- Functionality that has yet to be implemented:
  - Contacts
  - Group creation
- The UI may not update directly after value insertion into the Model.

5.3 Quality assurance reports

T.B.D.

Run analytical tools on your software and show the results. Use for example:

- *Dependencies: STAN or similar.*
- *Quality tool reports, like PMD.*

NOTE*: Each Java, XML, etc. file should have a header comment: Author, responsibility, used by ..., uses ..., etc.*

5.4 Access control and security

N/A

6 References

6.1 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/
- 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/

6.2 Libraries

- JUnit
  - Java’s unit testing framework.
    - https://junit.org/junit5/
- AndroidX
  - Android’s API
    - https://developer.android.com/jetpack/androidx

List all references to external tools, platforms, libraries, papers, etc. The purpose is that the reader can find additional information quickly and use this to understand how your application works.

7 Appendix

7.1 CircleCI

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

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