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

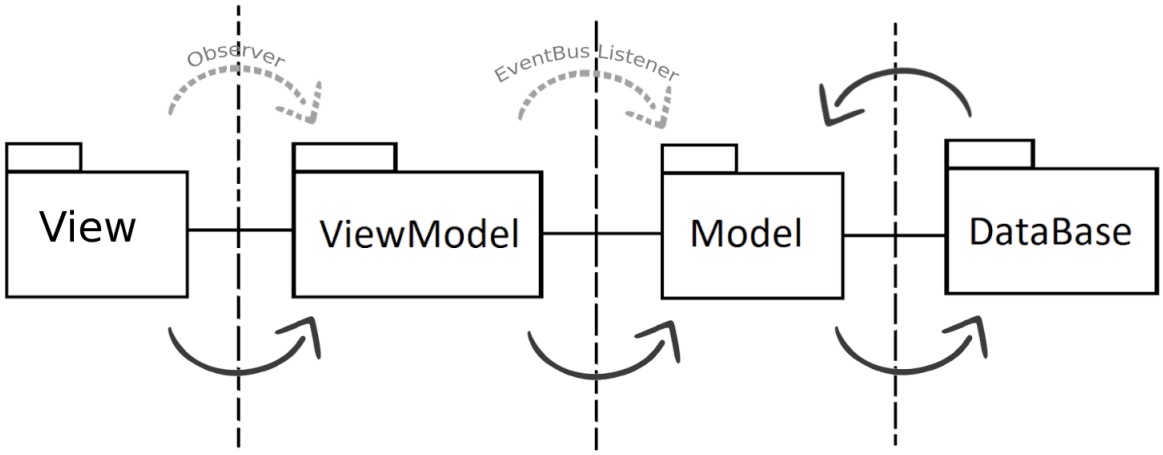

Figure 2.1: Components of the application.

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

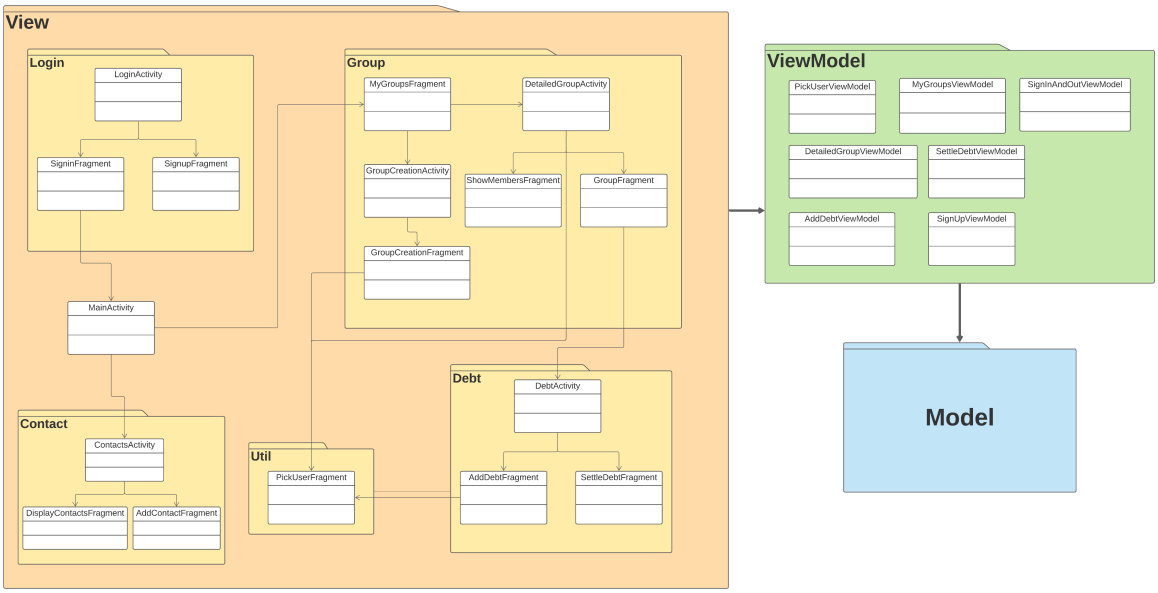

Figure 3.1: MVVM structure of Debtify

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

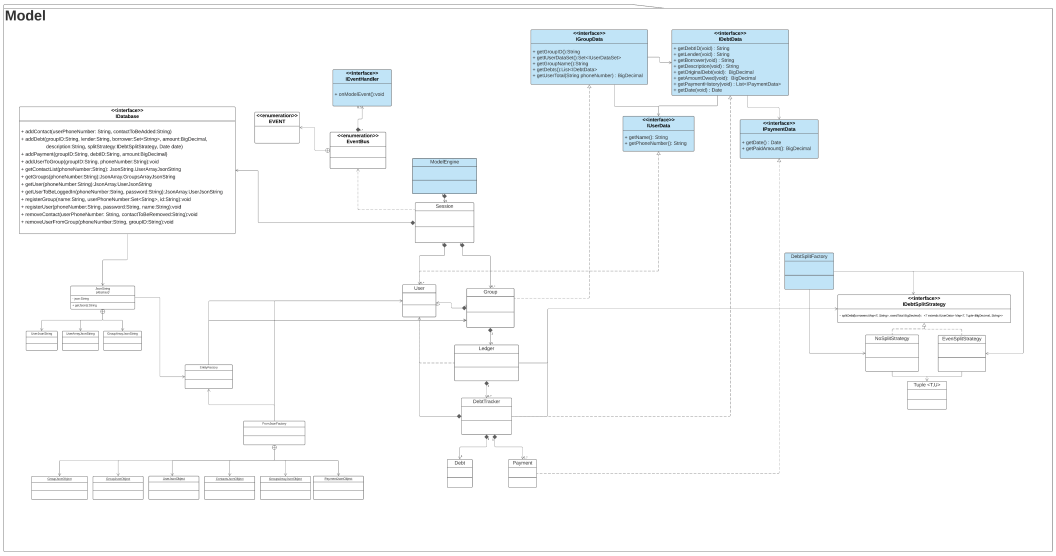

Figure 3.2: The UML diagram of the design model.

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.

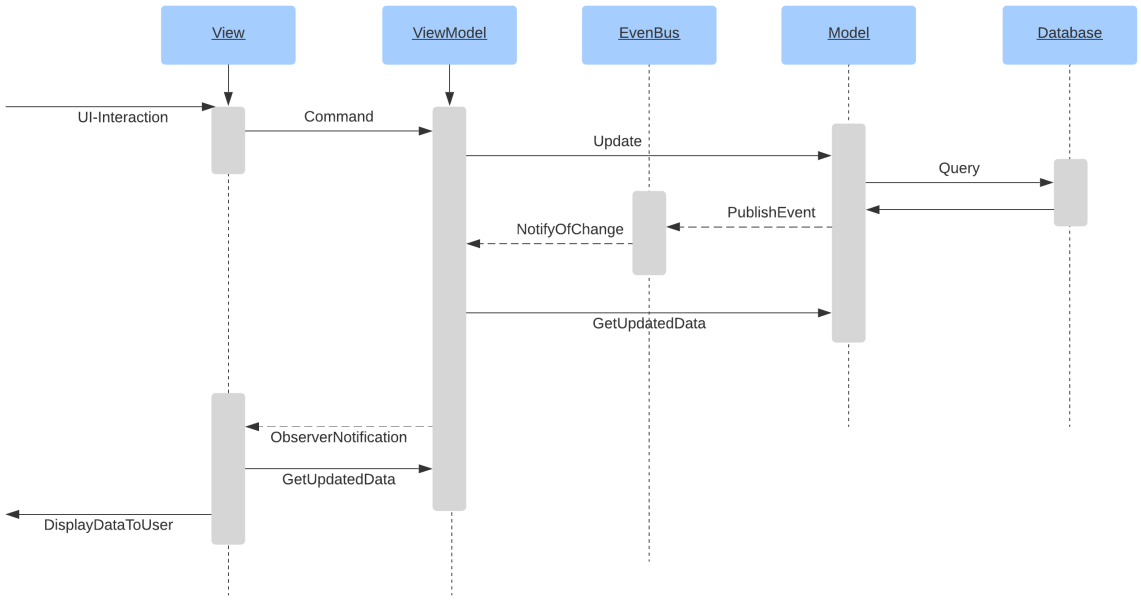

Figure 3.3: The run sequence for a generic example of a typical user interaction.

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.

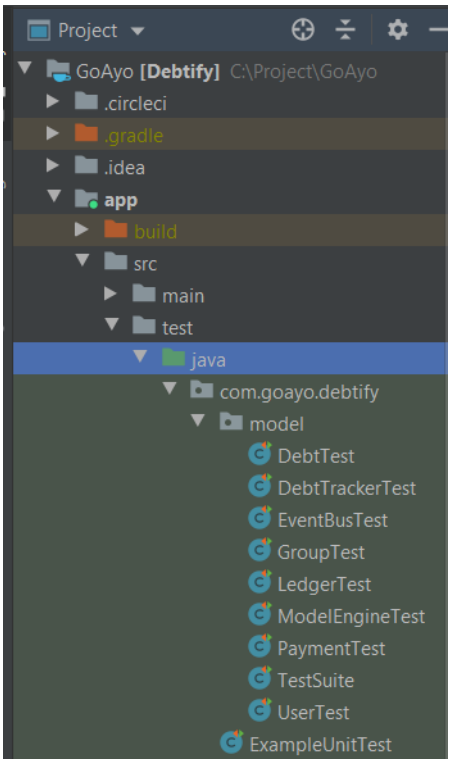

Figure 5.1: Path to the testing folder.

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.

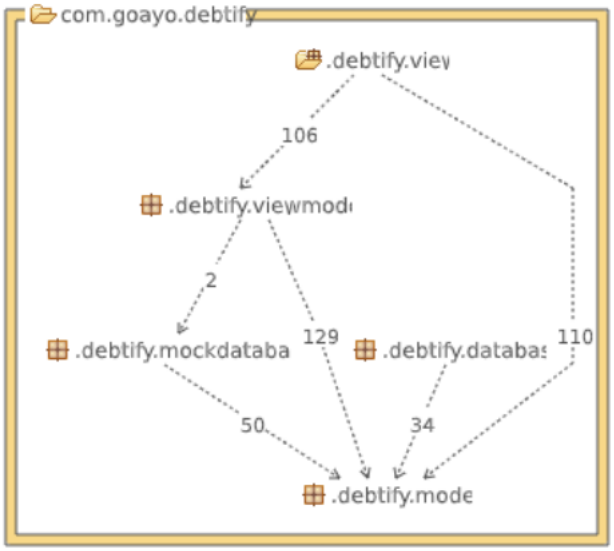

Figure B.1: Overall structure.

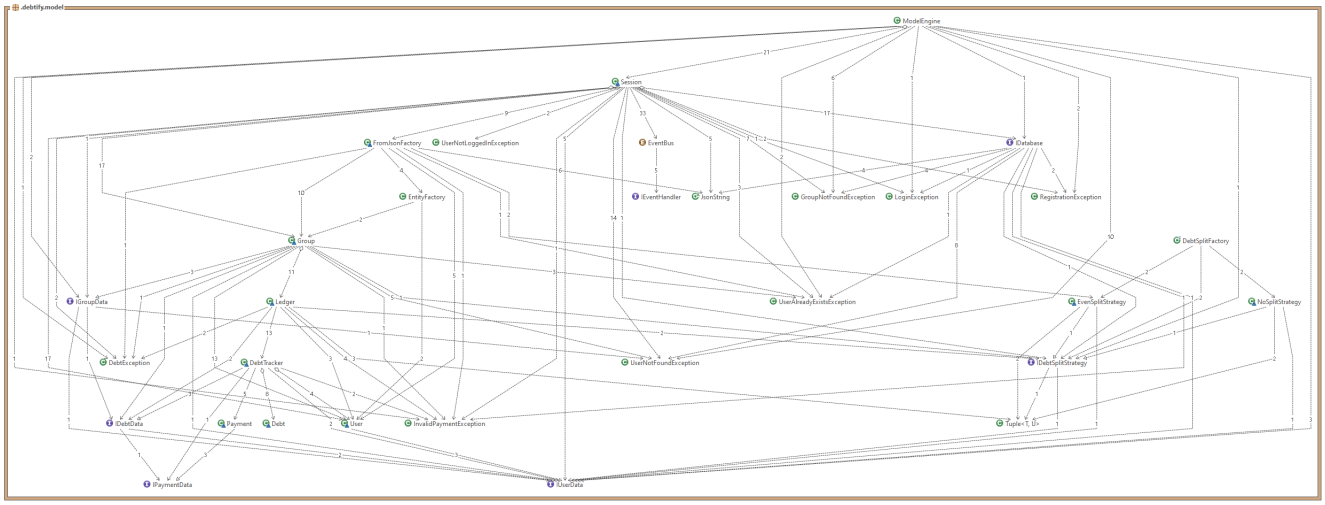

Figure B.2: Model structure.