Software Engineering 2
DESIGN REPORT

Team number:	0404

Team member 1	
Name:	Bernhard Clemens Schrenk
Student ID:	11807255
E-mail address:	a11807255@unet.univie.ac.at

Team member 2	
Name:	Michal Robert Žák
Student ID:	11922222
E-mail address:	a11922222@unet.univie.ac.at

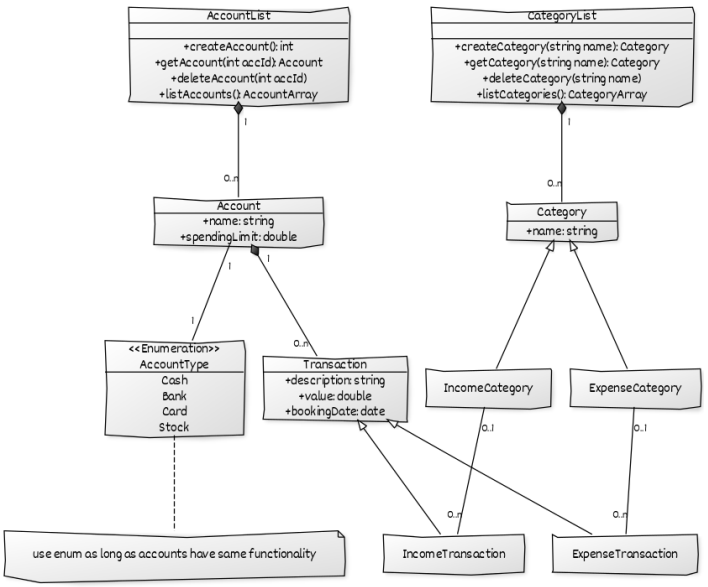
Team member 3	
Name:	Rumen Angelov
Student ID:	11911470
E-mail address:	a11911470@unet.univie.ac.at

Team member 4	
Name:	Samuel Šulovský
Student ID:	11915596
E-mail address:	a11915596@unet.univie.ac.at

1 Design Draft

1.1 Design Approach and Overview

While designing our app we tried to follow an interactive approach. At each step we tried to maximize the number of ideas, as we anticipated this would give us a higher chance of coming up with a good final design. As the first design step we decided to create sketches of UML class diagrams, visualizing how each team member thought the app architecture may end up looking, see image below for an example.



The entire team discussed the solutions in detail and we have selected some ideas we wanted to take with us to the next step. Among others:

- Enum for account type
- Using LiveData for the observer
- Proxy pattern to be used to load data into the reports
- Rough package division of the code

With the result from the first step, each team member set off to code a draft solution of the app. This solution was then discussed in detail in a similar style to the diagrams. The draft solutions highlighted some different coding approaches we may take. The draft solution which we finally agreed upon to be the best became the basis for our project. We have however also highlighted good sections from the other solutions, which we decided to take to the next step as well.

After that came the step to implement the app itself. We used a gitlab instance to manage our code versions and administer the workload. Here we created issues, which we divided amongst ourselves. To assure code quality and reduce the bugs

introduced, we had an unwritten rule that each code change needs to be inspected (reviewed) by at least one other team member. This has hopefully led to a fairly stable and robust code base, which won't be difficult to expand.

App Design

In general we have tried to reduce the number of dependencies on android (and other external) packages as much as we could inside of our app. Reducing dependencies is often a good idea, as we do not need to rely on the availability of the dependencies and potential major changes. As will be evident later in the document, this led us to reimplement some of the functionality provided by android. This allowed us to remove dependencies from parts of the app, where they were not necessary.

Underlying pattern: MVP

We evaluated some well known patterns for separating the UI from the data model and the business logic. We tried to find a good approach to use the MVC pattern, but came to the conclusion that in the Android environment the Activity and its related view description XML contain parts of View and of Controller. Apart from this mix up of two distinct functionalities we found that the business logic is not easily unit testable without having the whole Android environment runnable. And finally it did not fit in our approach to minimize platform dependent code.

We found a derivation of the MVC pattern called the Model-View-Presenter (MVP) pattern. The presenter takes over the functionality of the controller, but does not contain any view related functionality. It is more of a middleman between model and view and communicates with the view over a well-defined interface. It provides data to the view and receives user input from the view. All the business logic is contained in the presenter, but the presenter is not dependent on the Android environment. With this approach we could reach our goal to maximize testability, minimize platform dependency and separate the concerns.

We also evaluated the Model-View-ViewModel (MVVM) pattern, but found that the necessary data binding functionality introduces again Android dependency on various layers (including the model) and so we discarded this approach.

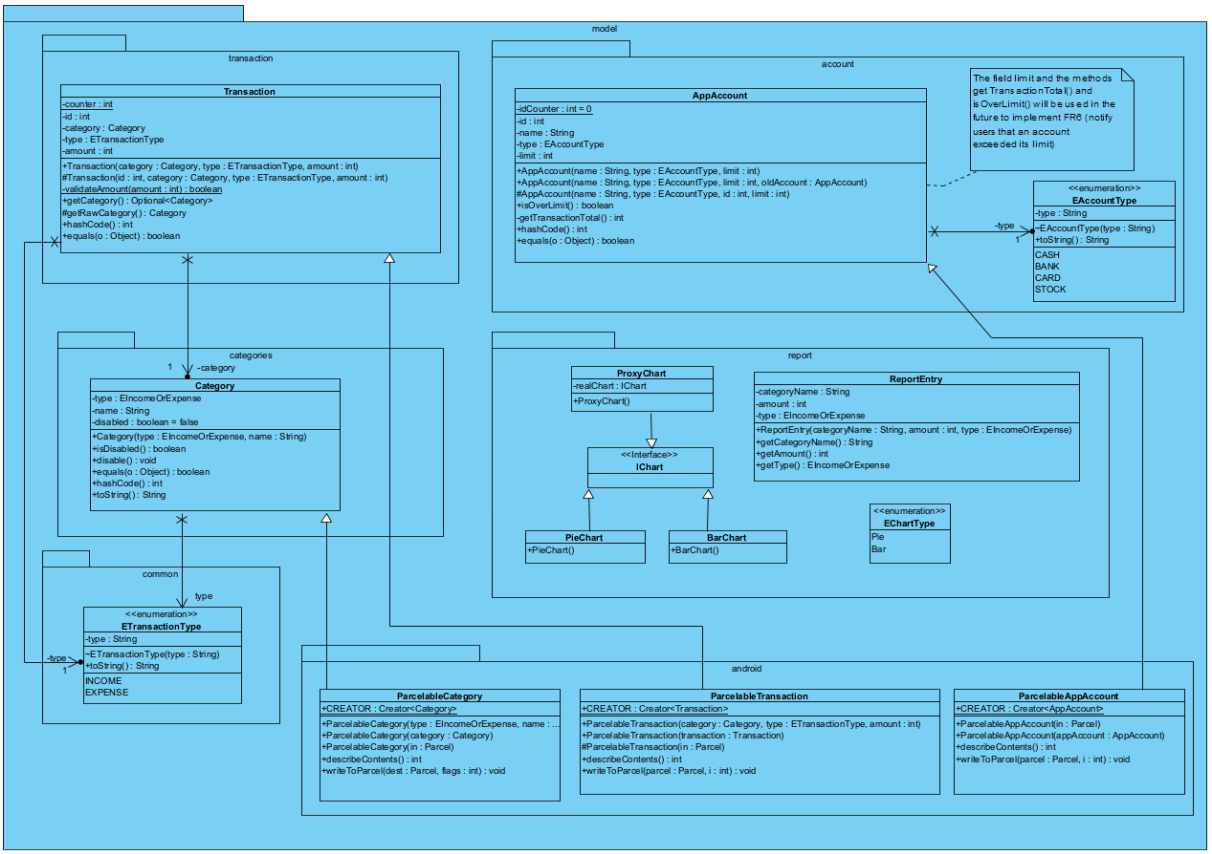
Note that, as we wanted to focus on adding some functionality first, the MVP pattern itself is not yet used in the code. The needed classes for this pattern are however already present and we just need to utilize them.

Package structure

The general structure of our app consists of four main packages. Namely database, model, ui and util. Each of these contains distinct classes and should interact with the other main packages only through a small number of classes (A big exception to this is the util package, which contains classes that are to be used by all other packages). All main packages will be discussed in detail in the next sections

Model package

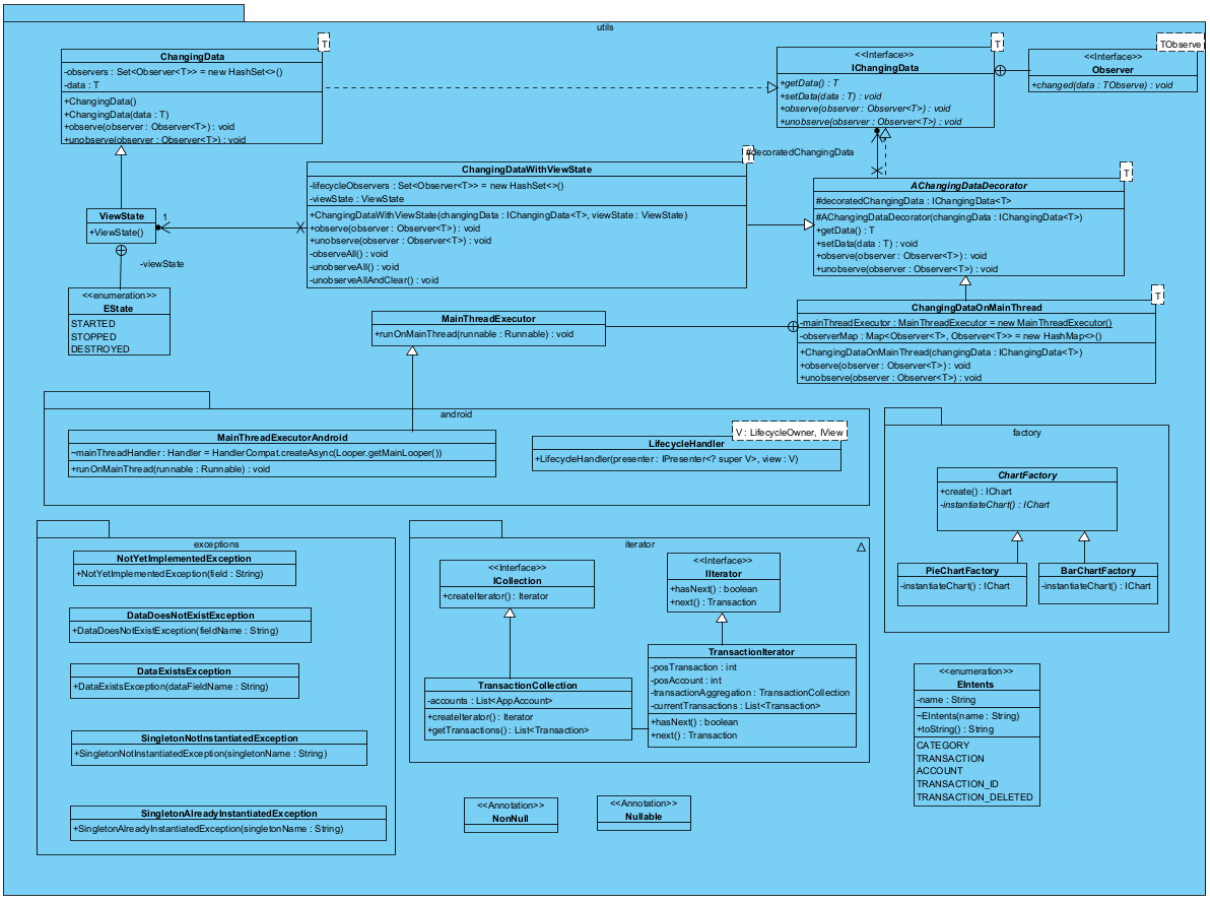
The model package provides implementation of the data models that are saved in the app. We tried to make the entire model package Android dependency free, to allow for a better modularity and reusability of this package. The main contents of this class are the classes AppAccount, Category and Transaction, which reflect the functionality described in FR1-FR3. These are simple data store classes, which save all the relevant information needed to work with these classes in accordance with the requirements provided.



Util package

The purpose of the util package is to provide low level functionality that is needed in most places of the app. Among other things it provides our own implementation of @Nullable and @NotNull. These are used in classes, where we do not want to introduce an Android dependency, such as the classes in the model package.

Furthermore this package provides the ChangingData implementation, which is a parallel of Android's LiveData. This is once again used in all classes, where we do not want to introduce an android dependency, but still want to take advantage of some of the powerful features Android provides.

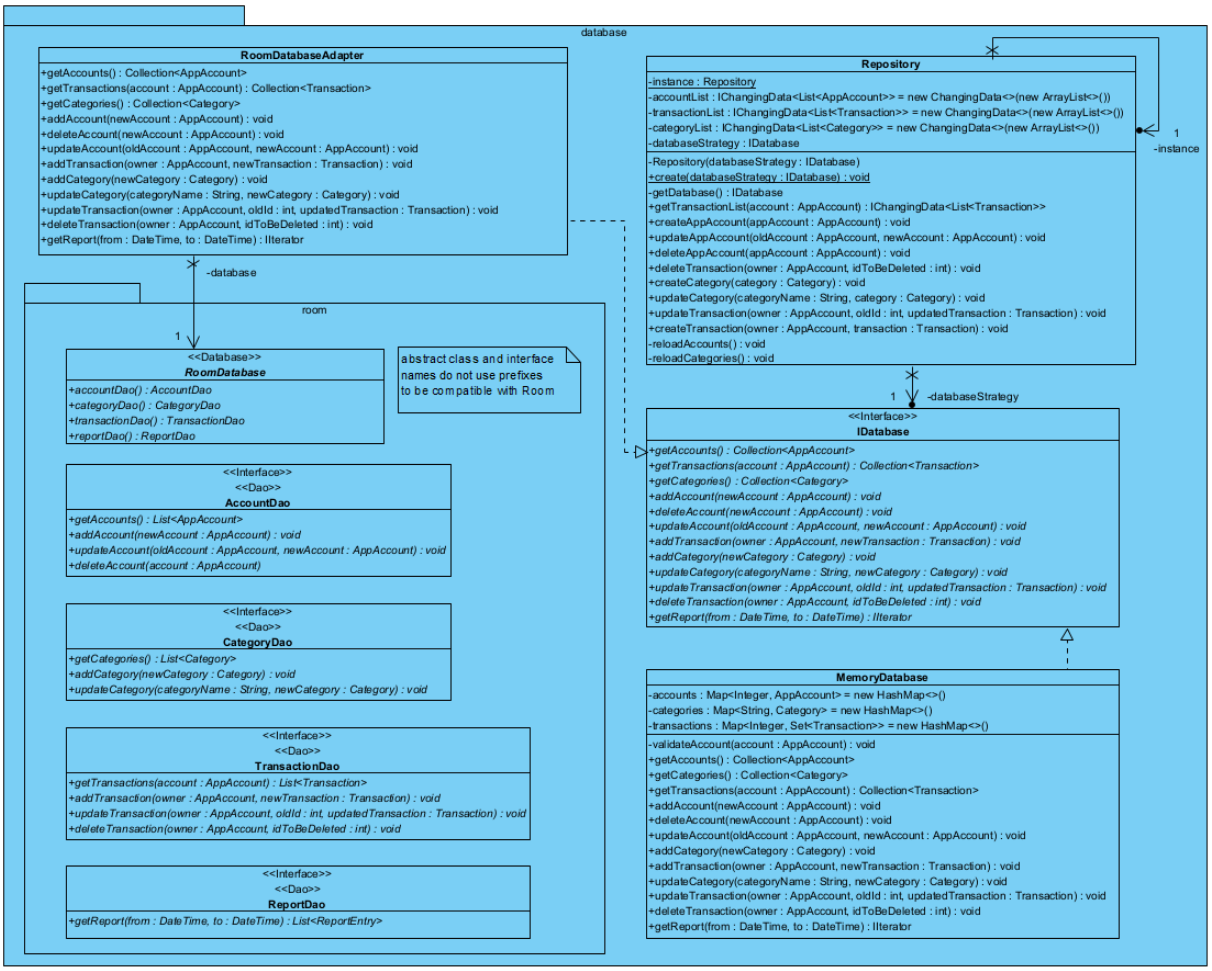


Database package

The database package holds all classes needed to use the database. For now only the interface IDatabase, the implementation of the interface MemoryDatabase and the Repository live inside of this package. In the future this package will be expanded to hold the RoomDatabase class, which will allow for persistent storage.

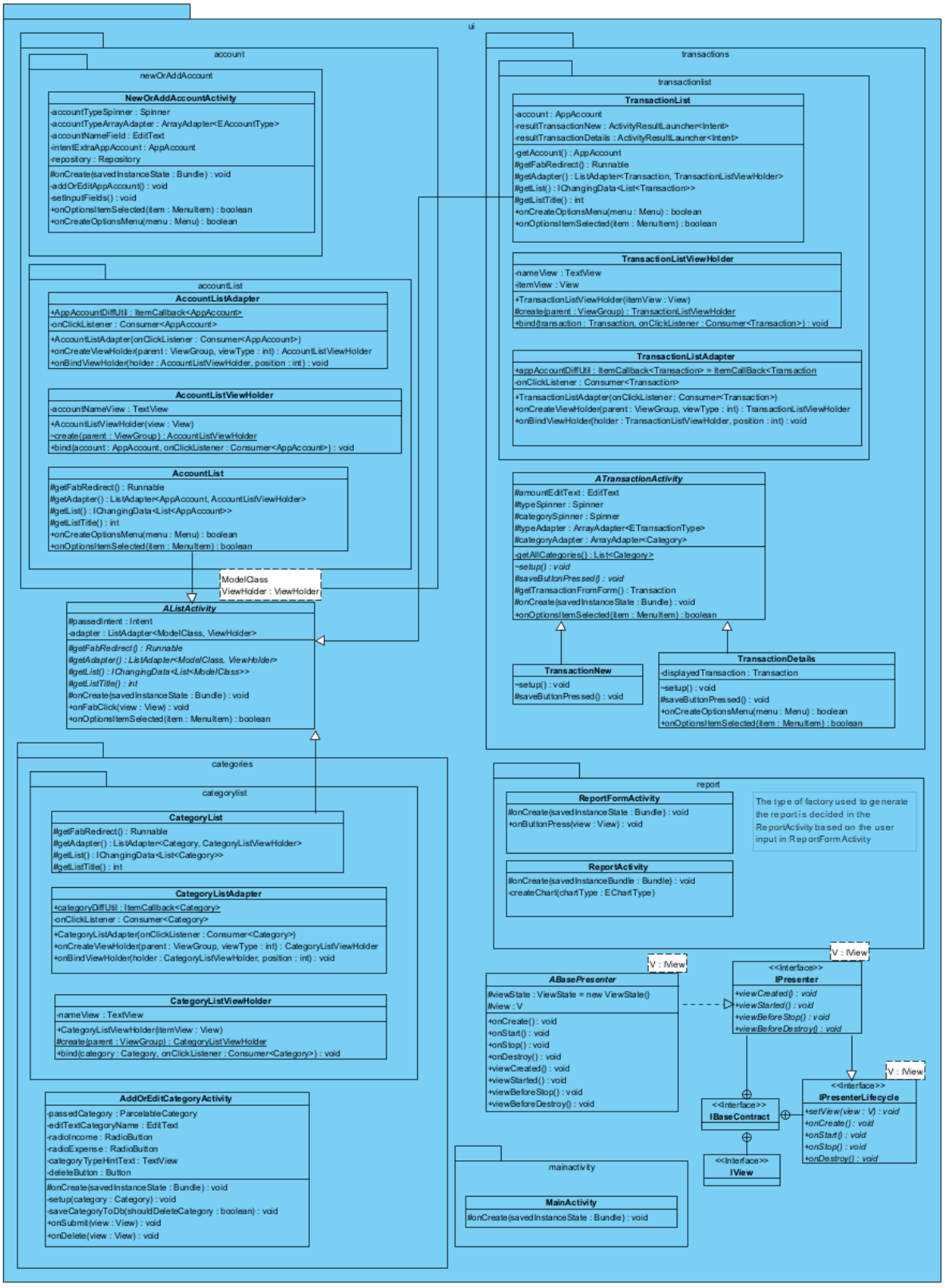
For now data is stored using the MemoryDatabase class. This is a simplified implementation of a Database, which simply holds all data inside of various java collections. This allowed us to quickly get a Database running, without having to focus on creating and managing the RoomDatabase.

The Repository is a singleton class. It uses a strategy pattern (This allows us to switch to the RoomDatabase once the time comes without problems). The Repository allows us to set an implementation of IDatabase as its database strategy. Further, it provides all fields as ChangingData, to allow the classes using it to observe changes to the data.

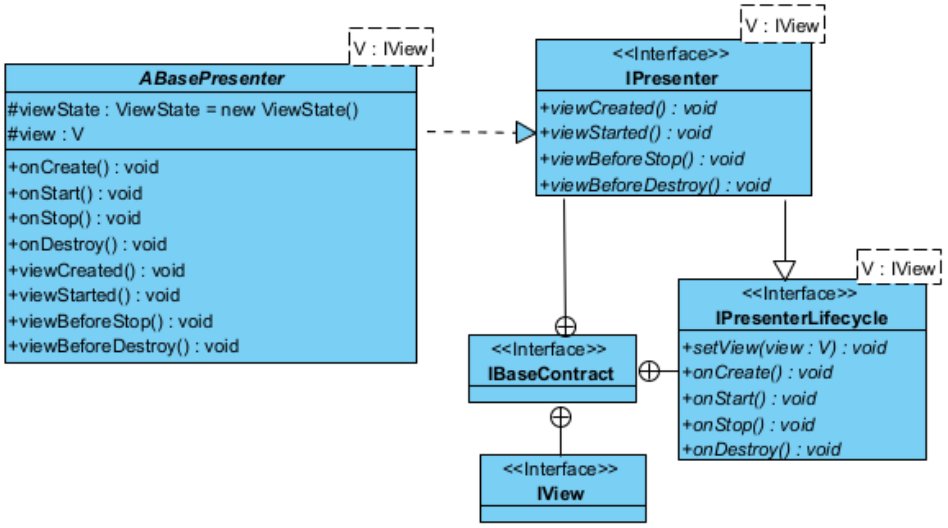


Ui package

The ui package holds all activities used by our app. As all three accounts, categories and transactions need a sort of list activity, we have decided to implement AListActivity, an abstract class which makes it very simple to create a list activity by just extending it and implementing all the abstract methods. AListActivity can be viewed as a TemplateMethod pattern. The abstract methods that are to be implemented are used in onCreate() to specify the details of the displayed data and the redirects of the elements on the page.



This package further holds all the activities that are going to be used to view/edit data of the application by applying the MVP pattern.



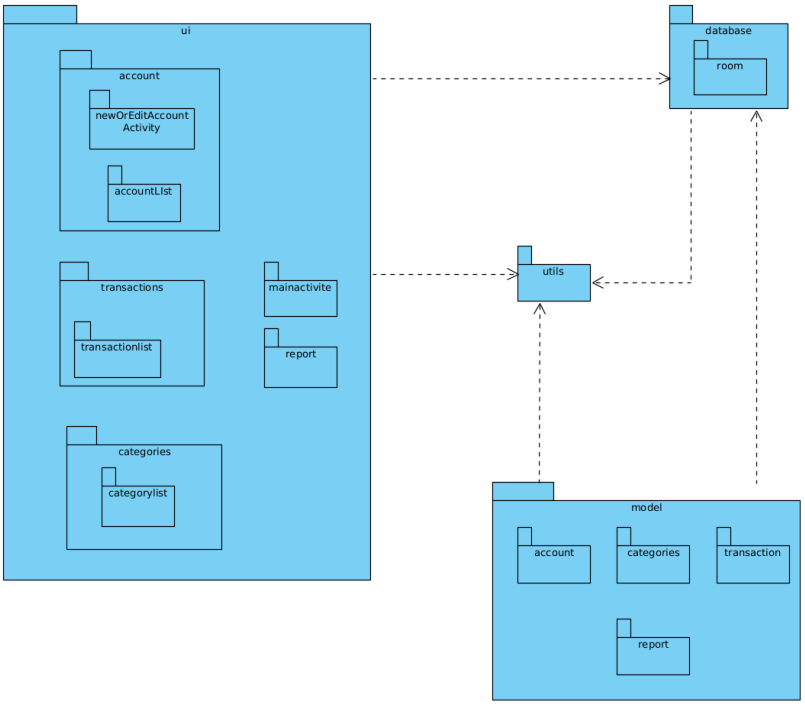
Another notable class is the ATransactionActivity, which is used to simplify the creation of transaction screens. When extending this subclass, the setup() method is to be overridden. In this method the subclass can provide further actions which need to happen when the activity is created. This method is called by the superclass (ATransactionActivity) at the end of the onCreate().

1.1.1 Package Diagram

The figure below shows a rough outline of how the packages described in the previous section interact with each other and how they are structured. As is visible, all packages depend on the util package.

The ui package should not directly communicate with the model package. All communication should be redirected through the database package. Later this communication will be extended to include the MVP pattern discussed above.

For details on the implementation and class structure of the packages, please refer to the diagrams in the previous sections, or to our detailed diagram export provided on our gitlab repository.



1.1.2 Technology Stack

Our solution is based on the Java programming language and uses libraries provided by the Android ecosystem for accessing system functionality. On top of it we used additional libraries as documented below.

Most of the code is designed to be platform independent and will not use Android specific libraries or APIs except those parts directly interacting with the platform (database, UI views).

During development we deployed our builds to a Pixel 2 (API level 29) AVD.

As coding style we agreed upon the Google Java Style Guide[1] with one exception to rule 5.1: We use the following prefixes:

- I: Interfaces (e.g. IDatabase)
- A: Abstract classes (e.g. ABasePresenter)
- E: Enums (e.g. EAccountType)

Minimum Android SDK
Version: 21

The application is designed to run with a minimum SDK version of 21. This is the minimum SDK version suggested in the Project Hints and FAQs document and allows it to run with Android version 5.0 (Lollipop) and higher. This will support, according to recent data, 98.0% of the currently active devices.

Using a higher version as an alternative is not necessary and would just reduce the number of supported devices as the application does not need any system functionality from higher Android/SDK versions.

Android Jetpack (AndroidX)
https://developer.android.com/jetpack/getting-started
Version: 1.3.1

The Jetpack libraries are refactored versions of the Android SDK not delivered with the operating system. They provide us with best practice APIs and backward compatibility. Most of the classes used are UI related like AppCompatActivity, RecyclerView, ConstraintLayout. There are no alternatives which are as good supported by the Android environment.

The @NonNull annotation of Jetpack will only be used in Android specific code. As an alternative for platform independent code we added our own @NonNull and

[1] https://google.github.io/styleguide/javaguide.html

@Nullable annotations to the code base and also in the IDE configuration. These are used for documenting possible null values of parameters, return values and fields.

Android Jetpack Room
https://developer.android.com/jetpack/androidx/releases/room
Version: 2.3.0

We will use Room as an object mapper on top of SQLite. Persistent storage is requested in FR4. Alternatively, own or third-party implementations on top of SQLite or plain files can be used, but Room is the upcoming standard for object persistence on Android and it is thoroughly tested.

Material Components
https://material.io/components?platform=android
Version: 1.4.0

We use the material components from Google for the styling of the UI components of the application. This is the standard on Android.

Java 8 API support
https://developer.android.com/studio/write/java8-support
Version: 1.1.5

We use the JDK desugar library to allow the code to use Java 8 API features without increasing the minimum SDK level. The features used are:

- Java Streams (java.util.stream)
- Optionals (java.util.Optional)

Alternatively we could refrain from using those features, but at least using Optionals is good practice in Java and so using it will provide better code quality.

JUnit 4
https://junit.org/junit4/
Version: 4.13.2

For tests we chose to use JUnit 4 due to the general team experience with this framework and the version 4. We use this both for unit tests and for instrumented tests. Later versions or alternative test frameworks have significantly different syntax and JUnit 4 is perfectly supported in the Android environment.

Mockito
https://site.mockito.org/
Version: 1.10.19

For mocking functionality (mainly in unit tests) we chose Mockito. Also here the team has experience with this framework and alternatives do not provide additional functionality which would be needed for this project.

Android Jetpack Test
https://developer.android.com/jetpack/androidx/releases/test
Version: 1.4.0 / 1.1.3 (ext:junit)

This is the de-facto standard library in the Android environment for running instrumented tests on the device or a device emulator. In this project this will be used to test Android specific code, mainly Room database access and UI code.

With alternative test frameworks, instrumented tests can also be run on the host platform. We evaluated Robolectric, but found that it does not provide 100% the same results as running the tests with Jetpack Test directly on the device (or emulator). We assume this comes from the fact that Robolectric is a reimplementation of the Android framework and so not 100% equivalent to the real operating system and libraries.

Android Jetpack Espresso
https://developer.android.com/training/testing/espresso
Version: 3.4.0

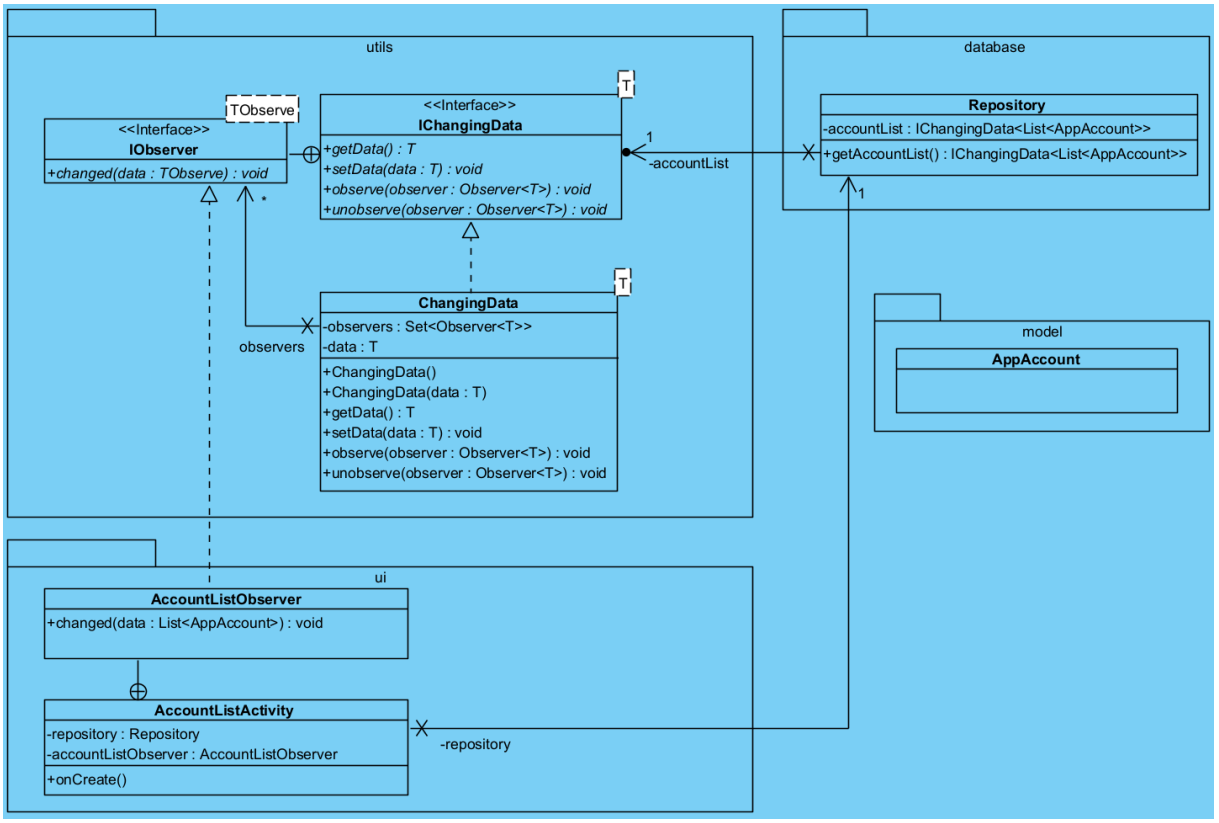
The Espresso library provides all necessary functionality to interact with the UI during instrumented tests. The library is perfectly integrated with the Android and Jetpack components.

Alternative libraries like Appium were evaluated as they would support a wider range of platforms, but the evaluation showed that the loose Android integration caused problems accessing the UI elements directly. This was causing harder to write test code and due to timing issues also unstable test results. Furthermore, the complex framework was slow, especially on startup of every test.

1.2 Design Patterns

1.2.1 Observer Pattern

For solving FR1 - FR3 we must ensure that the UI gets updated if the data in the model (or the database) changes. As the UI classes (like the *Presenter*, *Activity* and *RecyclerView*) are not known by the database, they can not be directly called with new data. Here the observer pattern gives us the possibility to register the UI as observers at the data source. In case of data changes, the registered observers will be called.



The class diagram shows in the *utils* package the *IChangingData* interface and its default implementation *ChangingData*. It contains an interface called *IObserver* which needs to be implemented by the concrete observer, as shown by *AccountListObserver* in the *ui* package.

As an example, the *AccountListActivity* is shown which retrieves a *ChangingData* instance for the list of *AppAccount*s. It gets this from the *Repository* by calling *getAccountList()* in its *onCreate()* method. It immediately subscribes to the *ChangingData* instance as an observer to get notified if the *Repository* changes the account list. As soon as notified, it will update its UI.

The following source code shows the interface of *ChangingData*. It is a generic class and can be used for any form of data. It is important that new data gets set with the

setData(T data) method so that the implementation can then notify the subscribed observers.

```
public interface IChangingData<T> {
  T getData();
  void setData(T data);

  void observe(@NonNull Observer<T> observer);
  void unobserve(@NonNull Observer<T> observer);

  interface Observer<TObserve> {
    void changed(TObserve data);
  }
}
```

The most important parts of the implementation of *IChangingData* are shown below. It contains a *Set* of *Observer*s and a reference to the data. If the data gets changed with *setData(T data)* the value is stored and all subscribed observers get notified by calling their *changed(T data)* method.

Observers can get subscribed by calling the *observe(Observer<T> observer)* method. This method will add the observer and immediately notify it with the current data. This reduces the effort of the programmer so that a separate handling of the data value during setting up the observer is not necessary. Furthermore, it removes a possible race condition in multithreaded environments. For multithreaded environments the methods are synchronized.

```
public class ChangingData<T> implements IChangingData<T> {
 private final Set<Observer<T>> observers = new HashSet<>();
 private T data;

 @Override
 public synchronized void setData(T data) {
   this.data = data;
   for (Observer<T> observer : observers) {
     observer.changed(this.data);
   }
 }

 @Override
 public synchronized void observe(@NonNull Observer<T> observer) {
   observers.add(observer);
```

```
    observer.changed(this.data);
  }

  ...
}
```

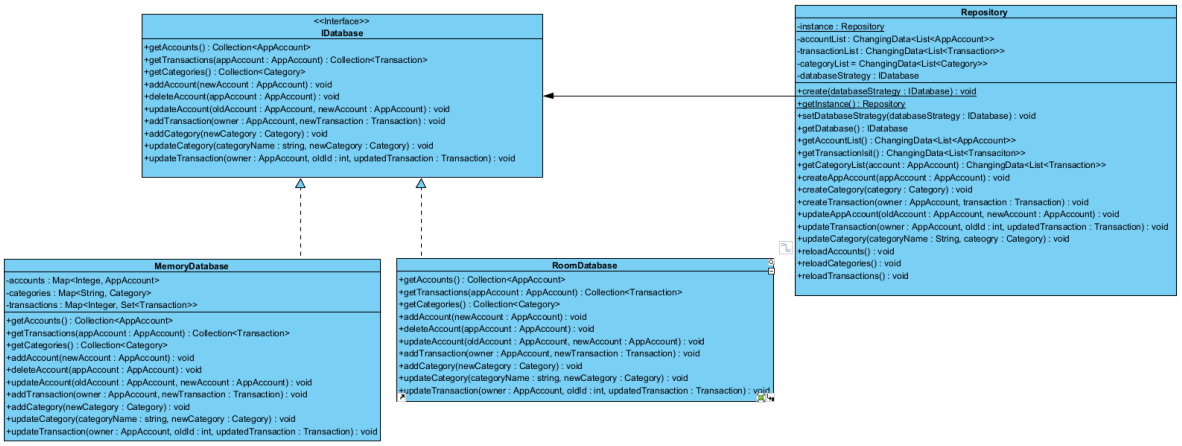
The following code fragment shows how the *onCreate()* method in *AccountListActivity* retrieves an *IChangingData* instance from the repository and subscribes an observer which updates the UI (in this case an *ArrayAdapter* of a *RecycleView*). The observer is implemented here as a lambda expression which will be provided to the *observe(..)* method by Java as an anonymous subclass of *Observer<List<AppAccount>>*. The *changed(..)* method will be overloaded with the lambda expression.

```
IChangingData<List<AppAccount>> listData =
  repository.getAccountList();
listData.observe(data -> {
  adapter.submitList(data);
});
```

1.2.2 Strategy Pattern

The strategy pattern allows us to use a more primitive database system during the initial development stages without needing to make any changes when we implement our persistent storage solution to meet FR4.

Our current solution is to store the data created from FR1, FR2 and FR3 in the memory of the app. The data access is handled by our Repository, which holds an interface field of the current database strategy. When persistent storage is implemented, we won't need to make any changes to the existing code base. Instead, the user will be able to simply toggle between the 2 modes of storage through the UI.



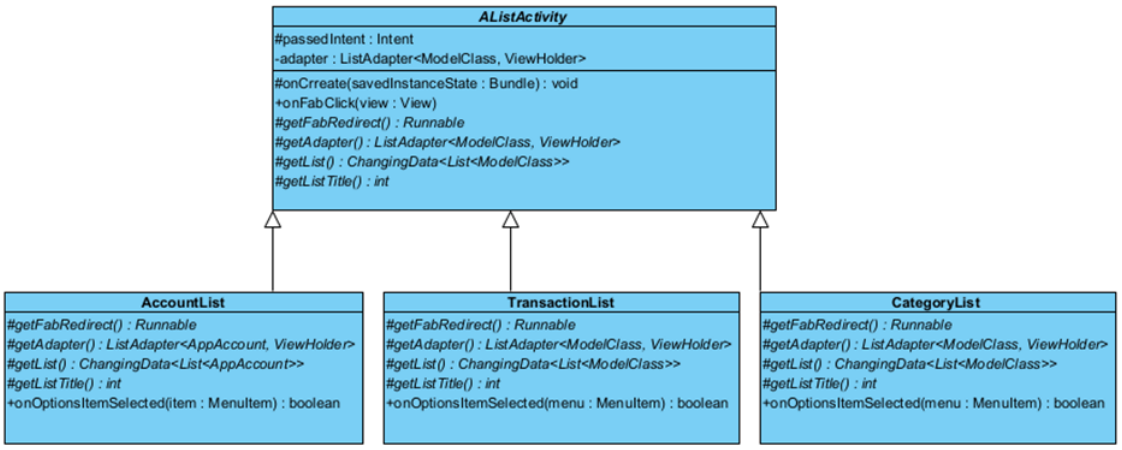

```
public class Repository {

  ...
  private static Repository instance;
  private IDatabase databaseStrategy;

  private Repository(IDatabase databaseStrategy) {
    this.databaseStrategy = databaseStrategy;
  }

  public static void create(IDatabase databaseStrategy) {
    if (instance != null) {
```

```
      throw new
SingletonAlreadyInstantiatedException("Repository");
    }
    instance = new Repository(databaseStrategy);
  }

  public static Repository getInstance() {
    if (instance == null) {
      throw new SingletonNotInstantiatedException("Repository");
    }
    return instance;
  }

  public void setDatabaseStrategy(IDatabase databaseStrategy) {
    this.databaseStrategy = databaseStrategy;
  }
  ...
}
```

1.2.3 Template Method Pattern

This pattern has no direct relation to any of the functional requirements, however it allows us to abstract the creation of simple lists in our app. These lists are all needed by FR1, FR2, FR3

During the development of the app we noticed that the activities that display the list of accounts, transactions, and categories have a similar structure and function. Thus, we decided that it would be beneficial to extract the common code parts of all 3 activities into one abstract template activity. This reduces code redundancy and adds consistency across the different list activities. The base logic that generates the list is handled in the abstract class. Any activity-specific data that is needed is provided by a getter that is overridden by the subclass.



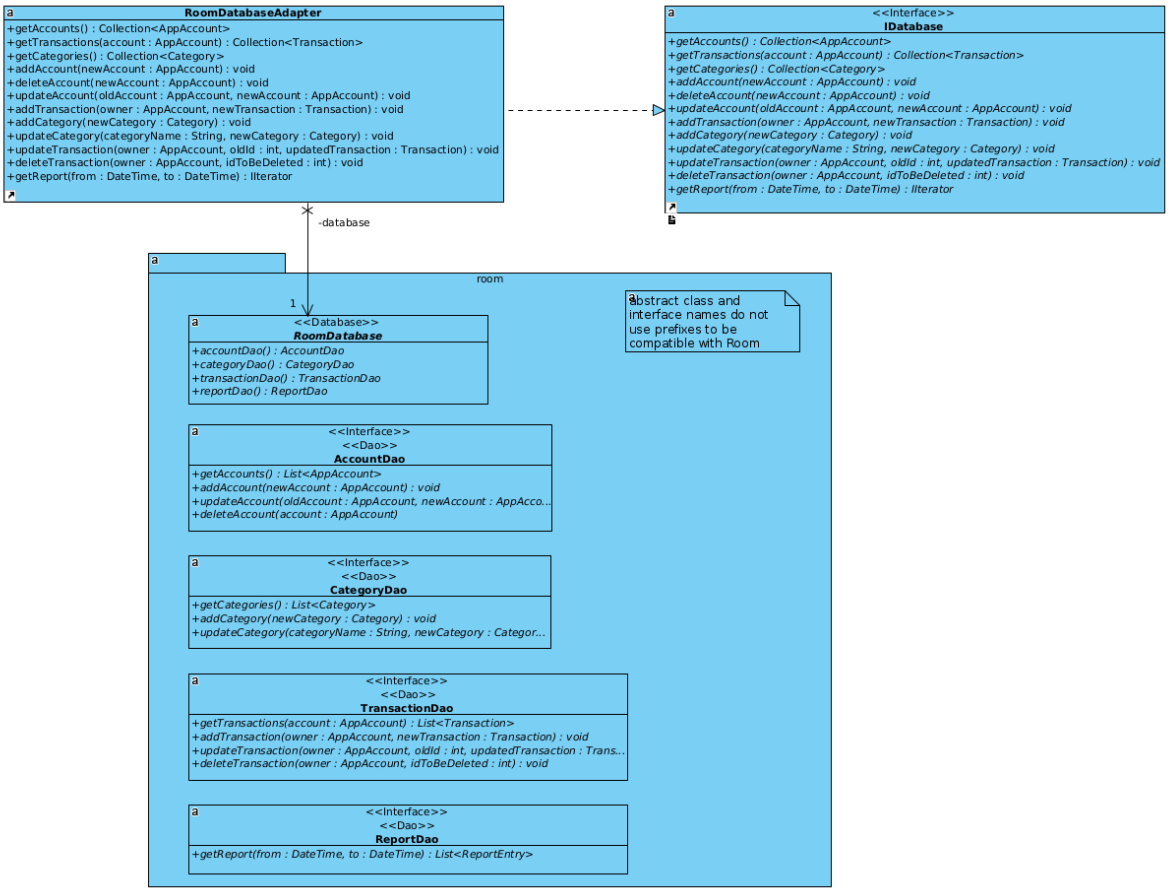

```
public abstract class AListActivity<ModelClass, ViewHolder extends
RecyclerView.ViewHolder> extends
    AppCompatActivity {

  protected Intent passedIntent;
  private ListAdapter<ModelClass, ViewHolder> adapter;

  protected abstract Runnable getFabRedirect();

  protected abstract ListAdapter<ModelClass, ViewHolder>
getAdapter();

  protected abstract ChangingData<List<ModelClass>> getList();

  protected abstract int getListTitle();
```

```
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    // General logic that creates the list activity
  }

  public void onFabClick(View view) {
    getFabRedirect().run();
  }

  @Override
  public boolean onOptionsItemSelected(@NonNull MenuItem item) {
    // menuitem onclick code
  }
}
```

1.2.4 Adapter pattern (not yet implemented)

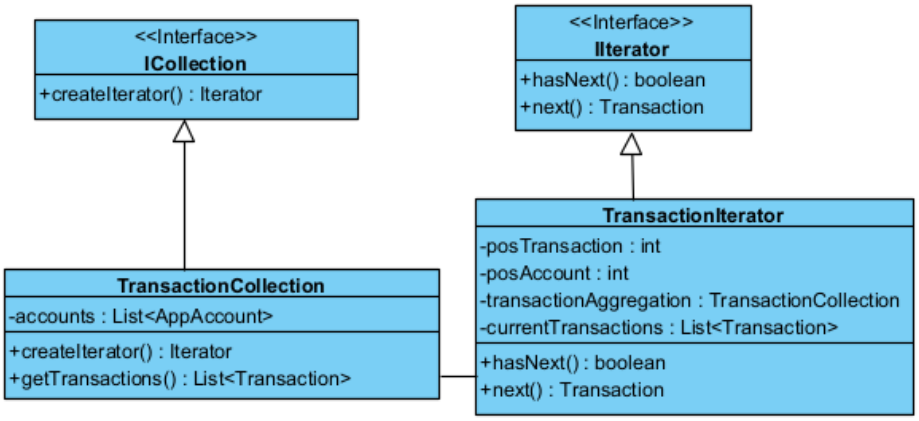
We will use this pattern to abstract the calls to the Room database. The RoomDatabase will return the different “Dao” objects needed to interact with the database. The job of the RoomDatabaseAdapter will be to get the Dao objects from the room database and translate them to be compatible with the IDatabase implementation (and vice versa).

This will help us to abstract the entire Room implementation and use a much more straightforward approach to talk with our database.



1.2.5 Iterator Pattern (not yet implemented)

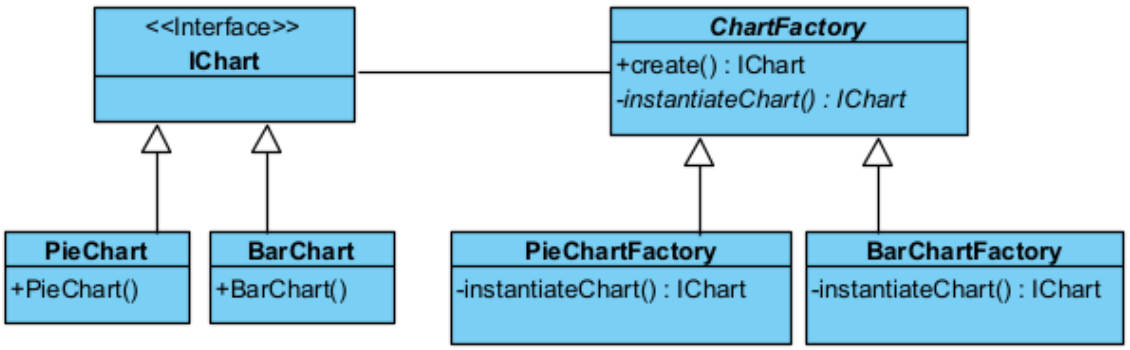
We will develop an iterator that will allow us to easily traverse accounts and their associated transactions. This will be used in the calculations needed to generate the reports for FR5. The iterator will be able to find the next transaction that is connected to the account, or go to the next account, without the developer having to worry how the underlying data structure works.



1.2.6 Factory Method Pattern (not yet implemented)

The factory method pattern will be used to create the needed charts for the report for FR5. Depending on what type of chart the user has selected, a different factory method will be called to generate the desired chart.

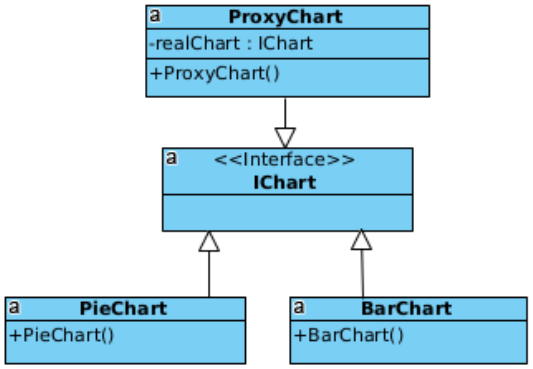
The abstract ChartFactory class has a create() method in which the abstract method instantiateChart() will be called. The abstract method will be realized in the subclass of the ChartFactory. Each subclass will create a different type of chart.



1.2.7 Proxy Pattern (not yet implemented)

The pattern is currently not implemented.

This pattern will be used in the report generation feature. While the factory pattern mentioned above will provide the user with a desired chart, the process of creating this chart might take some time, especially for larger data sets. In this case, the user should be shown a placeholder. The ProxyChart class fulfills this purpose by displaying a placeholder until it can display the chart generated by the factory.



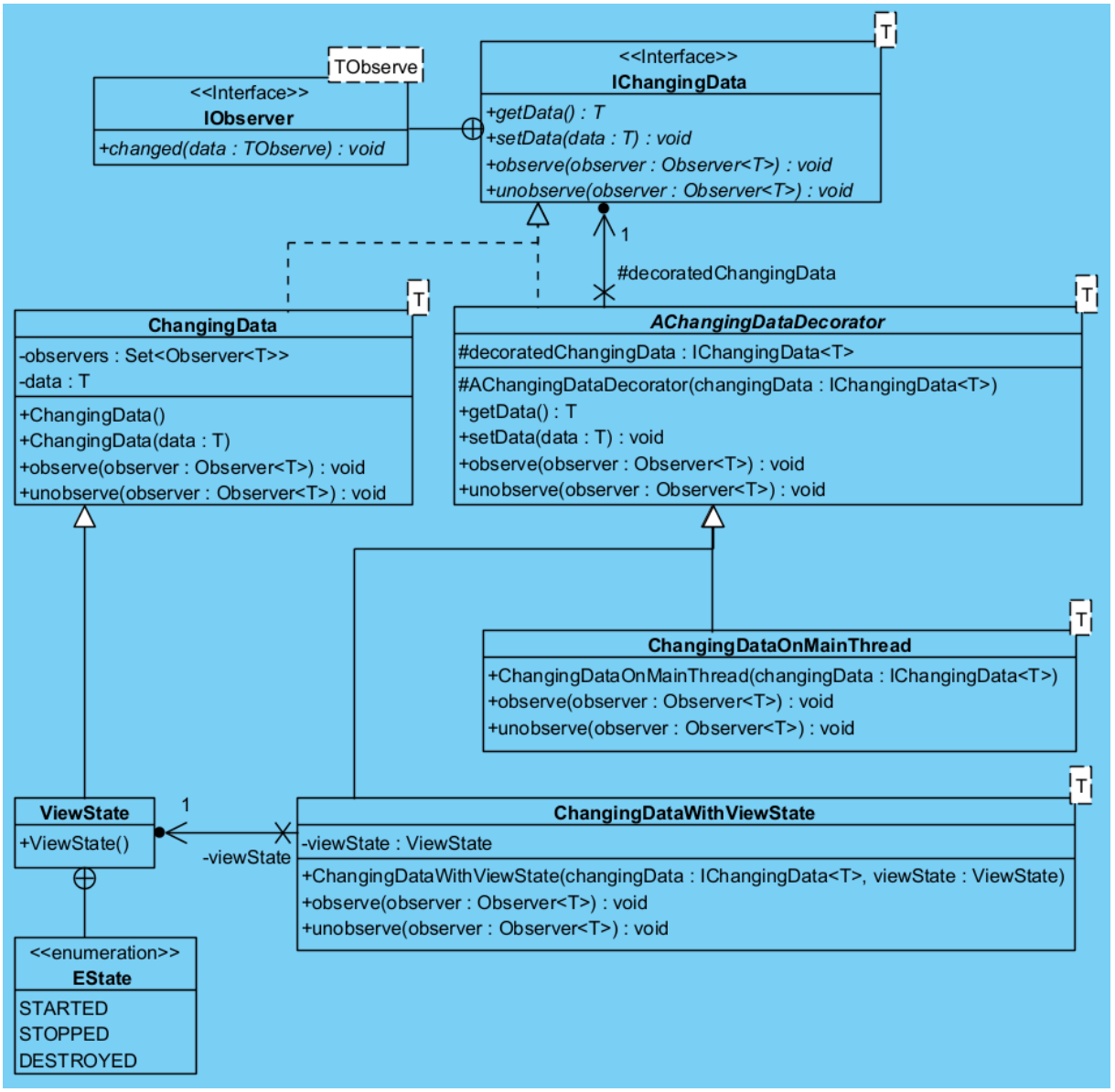
1.2.8 Decorator Pattern (not yet used)

For solving FR1 - FR3 we need to receive updated data from the database to our UI. For this we introduced an instance of the observer pattern in our class *ChangingData<T>* (see chapter 1.2.1). Depending on the position where this class is used we need to add additional features on top of it. So we decided to apply the decorator pattern to the *ChangingData<T>* class.

We need 2 additional functionalities:

- ***ChangingDataOnMainThread***: if *ChangingData<T>* gets used in a multithreaded environment (as we will have it within our *RoomDatabase* implementation), we need to ensure that the observer gets called on the main (UI) thread. For this we introduce a decorator called *ChangingDataOnMainThread*.
- ***ChangingDataWithViewState***: if *ChangingData<T>* is used with observers which are related to a view, they need to consider the view state. If the view is not visible, notifications can be stopped and if the view gets visible again, missed notifications must be delivered. If the view gets destroyed, the observer must be unregistered to prevent a reference loop. For this we introduce a decorator called *ChangingDataWithViewState*.

The following class diagram shows the involved classes for *ChangingData<T>* and its 2 decorators:



ChangingData<T> is the class which gets decorated. To accomplish this an interface *IChangingData<T>* is necessary. This interface is also implemented by the abstract decorator class *AChangingDataDecorator<T>*. This abstract class contains a protected reference to the instance of *IChangingData<T>* which it decorates, called *decoratedChangingData*. This reference gets initialized by its protected constructor. The implementations of the interface methods just call the equivalent methods on the decorated instance *decoratedChangingData*.

There are 2 derived classes of this abstract class, one for each decorator. The *ChangingDataOnMainThread<T>* decorator is quite an easy decorator. It just ensures that all observers which subscribe will be notified on the main (UI) thread. For this it overloads the *observe(..)* and *unobserve(..)* methods. It adds some intermediate code before calling its base class *observe(..)* and *unobserve(..)* methods. The constructor just passes its parameter (the instance which needs decoration) to the base class constructor.

The *ChangingDataWithViewState<T>* decorator works in a similar way, but it also needs a *ViewState* instance to know if the observers need to be notified. This view state is provided as a second parameter to the constructor.

The reason why *ViewState* is derived from ChangingData<T> is not directly related to the decorator pattern: *ChangingDataWithViewState<T>* needs to react to changes of the view state. So *ViewState* is itself a *ChangingData<T>* subclass to allow *ChangingDataWithViewState<T>* to subscribe as an observer to it (see Observer Pattern).

The implementation of this pattern is mostly done, but as it is not yet used, the details about the code will be provided in the Final report.

2 Code Metrics

Lines of Code

The LOC code metrics analysis was performed on the 24th of November at 14:00. Some minor changes have happened since but they don't affect the analysis in a major way.

Our codebase contains 50 classes in 24 java packages, along with 9 packages containing XML data for activities and other resources. Throughout these files we have a total of 3280 lines of code, 427 of which are comments and 541 lines of whitespace. We used cloc and IntelliJ's "Statistic" plugin to obtain these metrics.

Our code follows a good complexity to LOC coherence, with the most complex parts of our codebase gravitating to larger LOC figures. Our most complex data class - Transaction - is the largest in the model package, while TransactionList, the most complex activity, is the largest in the ui package. LOC metrics also highlight the good abstraction achieved with abstract classes like ATransactionActivity or AListActivity which, while having high LOC counts by themselves, allow for their specific implementations to have LOC counts as low as 44 LOC for CategoryActivity which extends AListActivity.

Java Lint Warnings

Using Android Studio's built in code analysis yielded some expected results regarding the linting warnings of our application in its current state. As could be expected, most of the errors (42 out of 94) were Java lint warnings for unused declarations, stemming from the parts of the codebase which haven't been integrated into the app yet, but will be in the next milestone. A good example is the setDatabaseStrategy method in the Repository class, which will, in the future, be able to be used to switch between local storage and Room based persistent storage. For now, the RoomDatabase is not implemented and its methods throw a NotYetImplementedException.

There are some notable linting warnings that we will address in the future, as they provide insight into our code flow and potential issues. For example, there is a data flow lint warning in the Transaction class which warns us that the method validateAmount is always inverted in its calls, which signifies to us that the method should probably be renamed and inverted, so that it's calls don't always need to be inverted, leading to more clarity and readability.

Android Lint Warnings

Other issues of interest that we will tackle in the next milestone are Android lint warnings, specifically in the realms of accessibility and correctness. For example, some parts of our app have images missing contentDescription parameters for hard of sight users, or don't have accessibility labels. Additionally, not all of our xml layouts are standardised, we are still using some hardcoded strings and dimensions, which we will move into constant declarations in the string or dimens xml files respectively.

The linter also checks for typos, but its findings were not useful, as the typos it found were not actually typos, just unrecognised words, or misplaced checks, such as in the word unobserve which was used as a method name, even though it is commonly used in the software development sphere.

Known Issues

There are also a few bugs we are aware of but will fix in the next milestone. The first of which is an integer overflow error which causes the app to crash if an integer overflow occurs when creating or editing a transaction.

Finally, we are aware of our current Repository layer between the database and the app having some potential conceptual issues, which we are investigating, writing down and iterating on while brainstorming solutions. The main problems we see are potential code duplication, the lack of clarity for the app of what data is held in memory at which point and the necessity for the Repository to know the internal structure of the database to provide data efficiently.

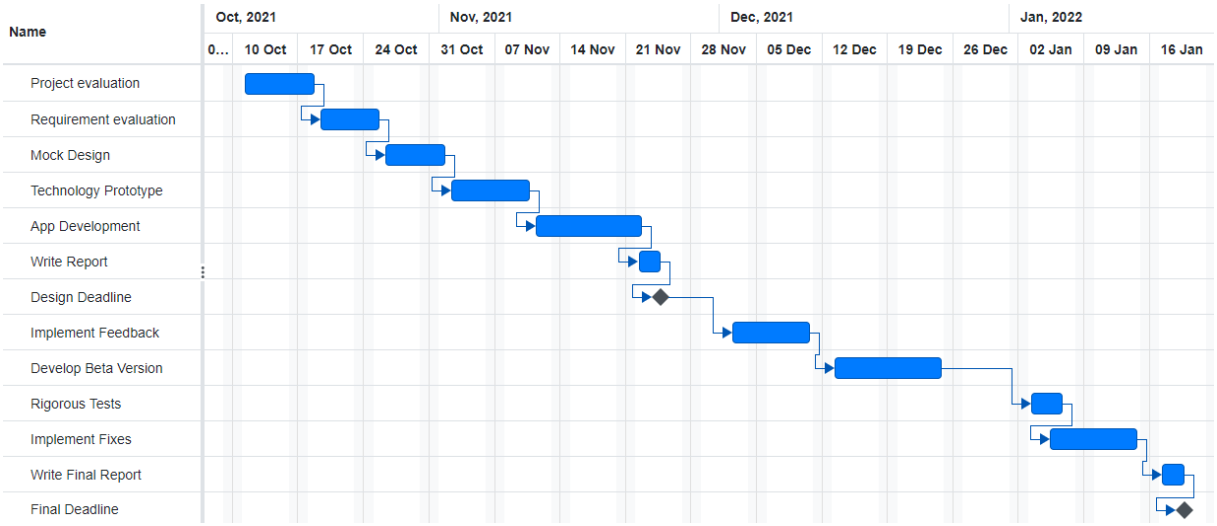
We have created Gitlab issues for each of these known bugs and are going to implement them in the next milestone.

3 Team Contribution

3.1 Project Tasks and Schedule

All tasks shown in the Gantt chart below will be/were performed by all team members. Once this document is published, we will have completed the “Design Deadline" milestone. After this milestone we will implement the feedback we get on our current project status.

We plan to finish the functionality of the app before christmas. This allows us to dedicate enough time to test the application and fix any bugs found. The testing stage will be basically running until the final deadline.



3.2 Distribution of Work and Efforts

Bernhard Clemens Schrenk:

Completed work	● ChangingData* (Observer) ● Unit Test: ChangingDataImplTest ● Model-View-Presenter: BaseContract & BasePresenter ● @NonNull & @Nullable annotations ● Bug fixes
Design Patterns	● Observer (already implemented) ● Decorator (will be used to add optional features to ChangingData)

Michal Robert Žák:

Completed work	● FR3 (Transactions),the associated database functionality and views related to FR3 ● General bug fixes ● Abstract List Activity (used by all list activities, providing a layer of abstraction to reduce complexity)
Design patterns	● Template method (already implemented) ● Adapter (will be used to abstract the Room Database)

Rumen Angelov:

Completed work	● FR1 (Accounts) and it’s associated database functionality and views ● Bug fixes
Design Patterns	● Iterator (will be used to easily traverse accounts and transactions) ● Factory Method (will be used to generate different types of reports for FR5)

Samuel Šulovský:

Completed work	● Categories (FR 2) ● Repository (layer above DB implementing strategy pattern to abstract DB work from the app) ● Larger code reviews ● Refactors, polish and bug fixes ● Document Review
Design Patterns	● Strategy (already implemented) ● Proxy (will be used in chart generation and related placeholders)