Software Engineering 2
FINAL 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 Final Design

1.1 Design Approach and Overview

We did not change our design approach from the design deadline very much, as we found it worked fairly well for us.

For every major design decision, all teammates were involved in consulting the solution approach. This was done to maximise the number of ideas and thus maximising the chance to find the most ideal solution. Furthermore we had an unwritten rule that each change to the code has to be looked at (reviewed) by at least one other team member. This hopefully significantly reduced the number of bugs in the code.

In this document I will focus on explaining the design steps taken after the DESIGN deadline. For steps taken before this deadline, please consult the DESIGN document.

We already knew when submitting our design deadline, even though it fulfilled the needed requirements, that we would need to rework large parts of our app. This was the first task we tried to tackle, as doing it later would only pile on code that would need to be refactored and reworked.

The first of this rework was the implementation of the MVP pattern. We briefly considered not implementing it at all, as it seemed to cause us more problems than good in the beginning. However ultimately we decided that the benefits outway the negatives, namely the ease of testing and the well defined interface for all activities, we have decided to proceed with the changes.

The second major rework that needed to happen was enabling the repository to communicate in an asynchronous manner. For this we have considered multiple possibilities, namely

- adding a field to each data passed to the repository, which will get modified once the operation is successful/not successful
- letting the repository offer an observable which would notify the observers of a given operation's outcomes. This would work for our case as we only have one View instantiated at a time.
- Letting the methods provided by the repository return observables. This would allow the classes calling the repository to observe the result and act accordingly

We have chosen the last option, as we thought it best fit our app and was the most modular and expandable approach.

After all these changes were done, nothing was in the way of implementing the missing functional requirements.

While working on the changes described above, we tried to improve our code as much as possible, to meet the quality requirements to the best level we could. We have added unit tests, javadocs and generally tried to improve the quality of our code as best we could.

Package structure

As the entire package diagram is too big we have decided to split it into its sub package components.

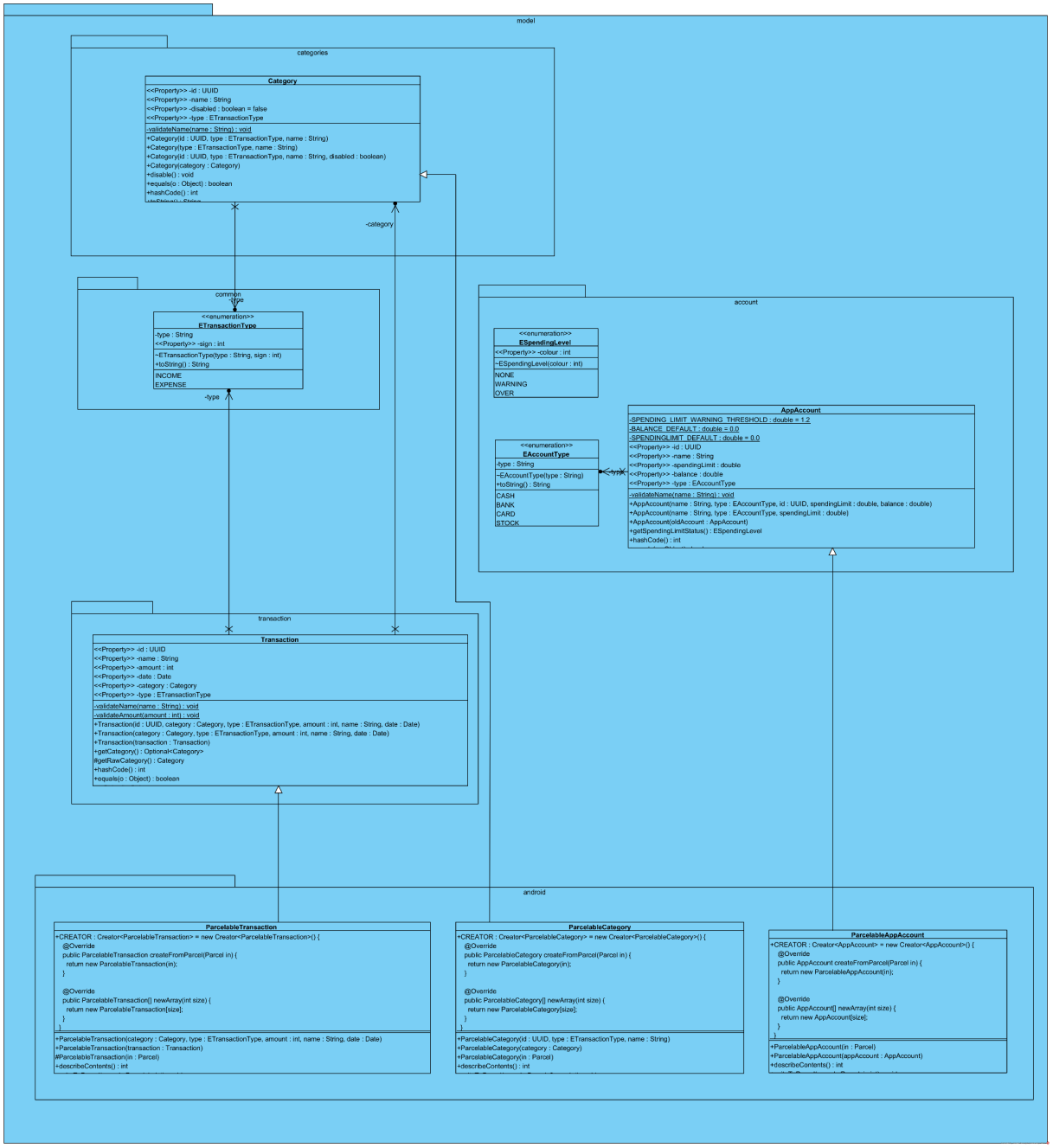
In general we have tried to reduce the number of dependencies on android (and other external) packages as much as we could inside of our app. Reducing dependencies is often a good idea, as we do not need to rely on the availability of the dependencies and potential major changes. As will be evident later 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.

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 the functional requirements. These are simple data store classes, which save all the relevant information needed to work with these classes in accordance with the requirements provided.

This class was extended from the DESIGN deadline by the fields needed for the functionality of FR4-FR6. The general structure of the classes has not changed much.



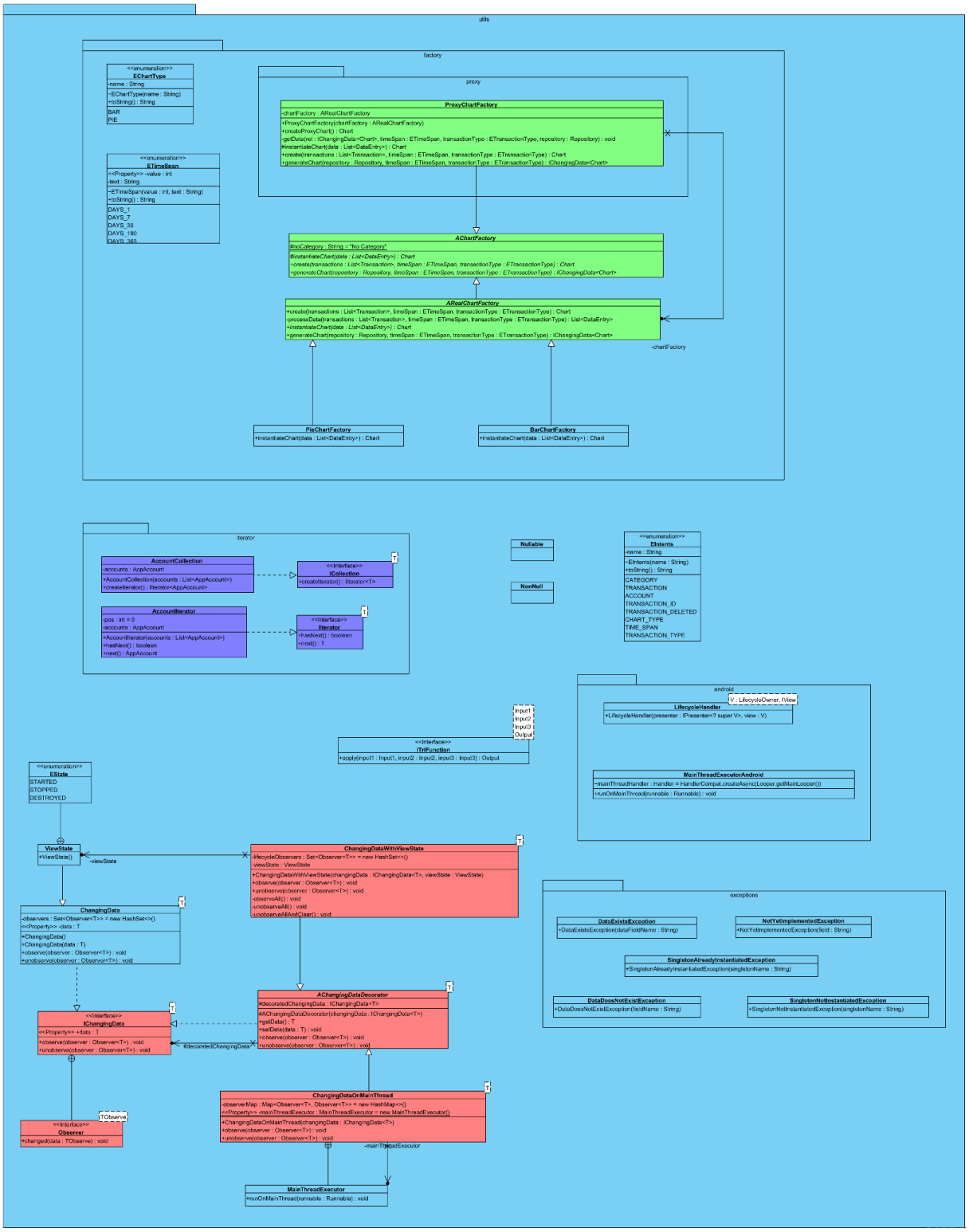
Util package

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.

Most notably, compared to the DESIGN deadline, this package was extended with the classes needed to generate and manage charts that are utilised in FR5 reports.



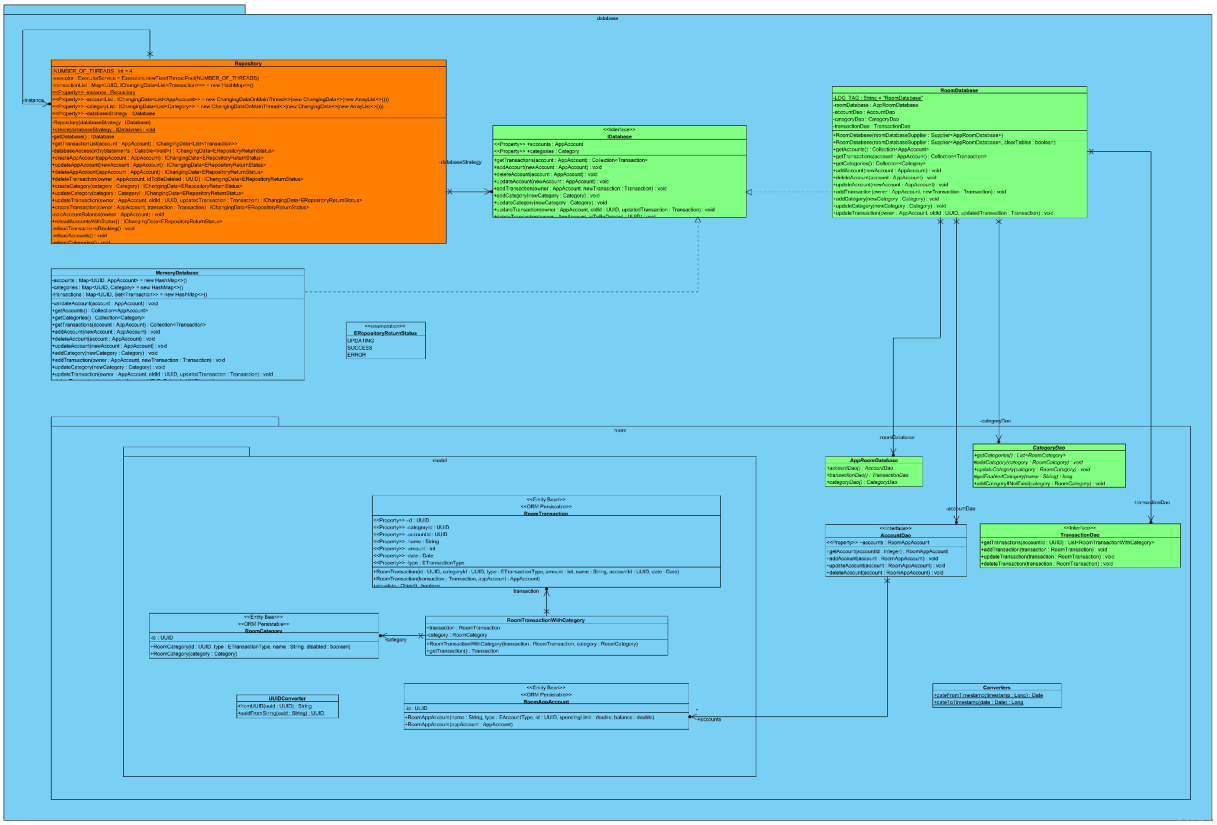
Database package

The database package holds all classes needed to use the database. It contains the interface IDatabase, the implementations of the interface MemoryDatabase and RoomDatabase and the Repository. Further, it contains all classes needed to manage and access the Room Database itself. Among those is the database.model

package, defining the structure of how the model classes (AppAccount, Category and Transaction) will be saved in the database.

The Repository is a singleton class. It uses a strategy pattern (This allows us to switch between the MemoryDatabase and the RoomDatabase for debugging and testing purposes). 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.

The repository is also the class which manages the execution of Database requests on a separate thread.



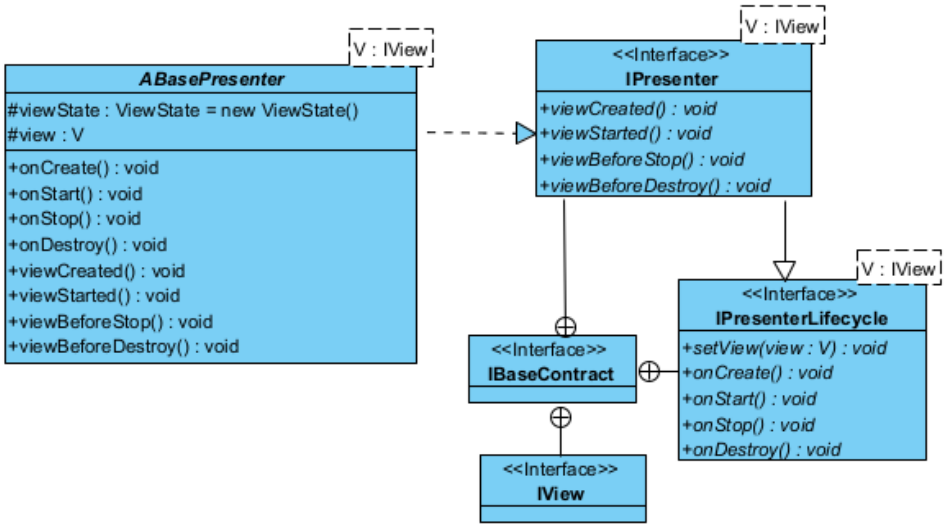
Ui package

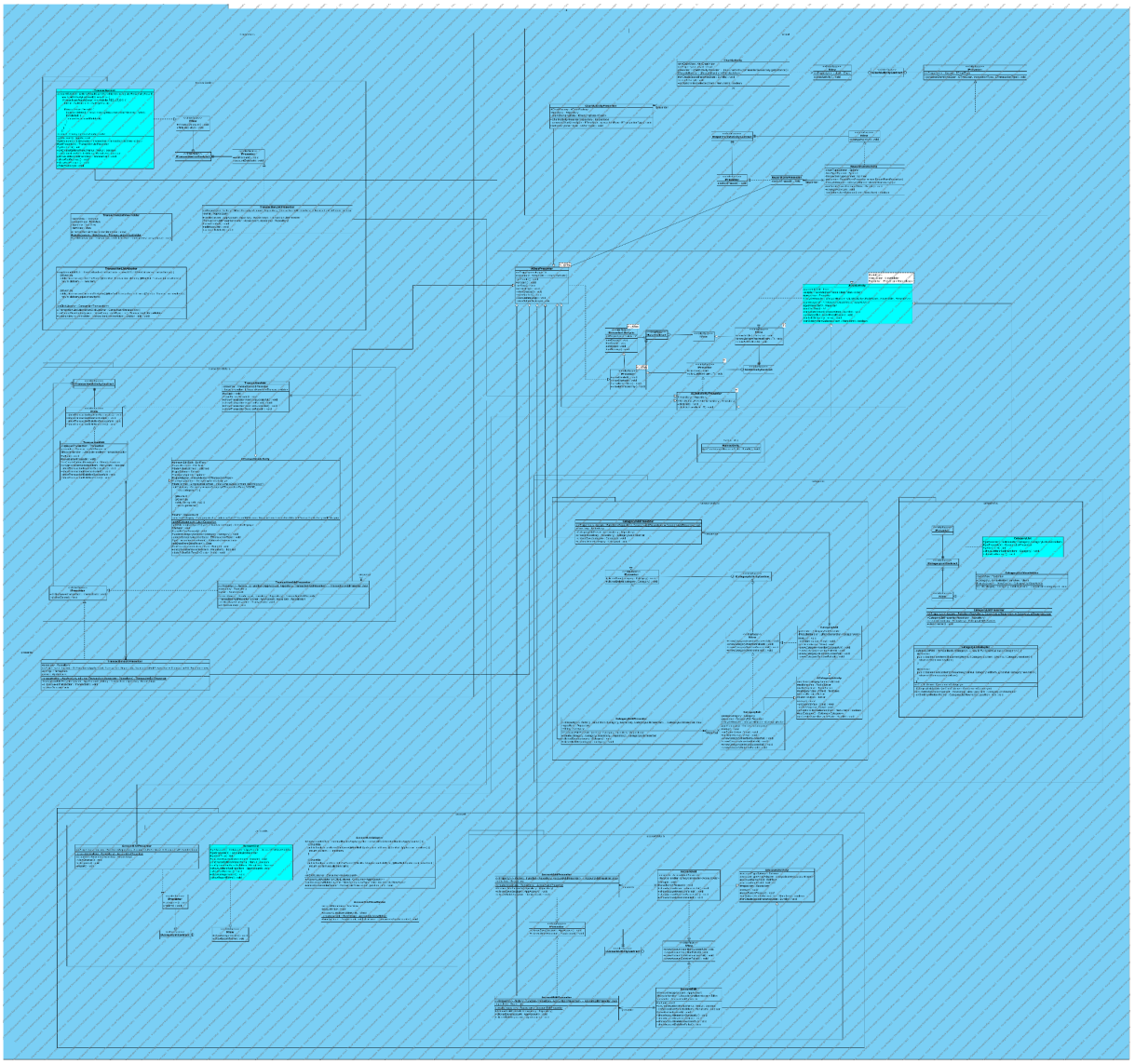
This class contains all classes related to displaying things on screen. It had a major redesign compared to the DESIGN deadline, as all Activity classes had to be split into the Activity(acting as the View in MVP), the Presenter and the appropriate contract between them.

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.

Further, we have noticed that the activities for adding, or editing accounts, categories and transactions ended up being the same thing respectively, with a few minor modifications on where a button reroutes or similar. We have thus implemented an abstract class for each of those activities, which allows us to more cleanly define the Add and Edit activities, without having repeated code.

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





1.1.1 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:

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

- 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 @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.

AnyChart-Android

https://github.com/AnyChart/AnyChart-Android

Version: 1.1.2

The AnyChart library was used to generate charts for FR5: Reports. We found that the library has a very intuitive API and streamlines the creation of charts.

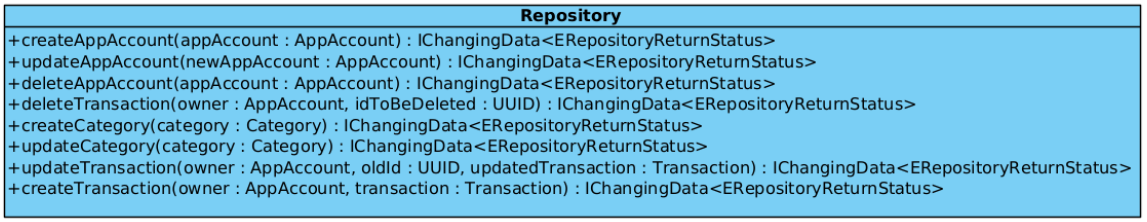
We have also considered using the library MPAndroidChart. Upon taking a closer look at it however, we found that the API for this library was a little unintuitive and provided too many features for our use case.

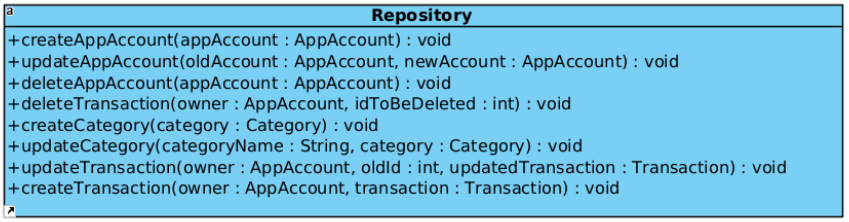
1.2 Major Changes Compared to DESIGN

Asynchronous Repository

Probably the biggest change we had to implement is make (almost) all methods in the repository asynchronous. We had to do this as our DESIGN submission only utilised a database, which saved the data into java collections. For the FINAL deadline we needed to implement the Room database. This database can (and should as it is a lot slower than java collections) be accessed from another thread. We have decided to tackle this issue by letting the Repository return a with ChangingData (an observable data holder) wrapped ERepositoryReturnStatus enum. This allows the class accessing data from the repository to observe this enum and

execute the appropriate code whenever its status changes to SUCCESS/ERROR.





Adding the MVP pattern to the code

Another big change that we made to our code is implementing the MVP pattern. We mentioned this as a goal in our DESIGN deadline, however we did not have it implemented yet at that time. Implementing the MVP pattern has allowed us to more easily create unit tests, even for the UI/Android part of the app.

Adding the MVP pattern has however significantly increased the complexity of our code. For every Activity we want to define we now have to create three classes instead of just one.

- A Contract which defines the methods the View and the Presenter will use to notify each other
- A Presenter, which contains all business logic and decides what gets shown on the UI when. This class needs to implement the Presenter interface from the Contract
- The Activity itself. This class implements the View interface from the contract and handles things like displaying data, or redirecting to other activities.

For an example of the View and Presenter classes, see for example the AccountAdd, AccountAddPresenter and IAccountActivityContract classes.

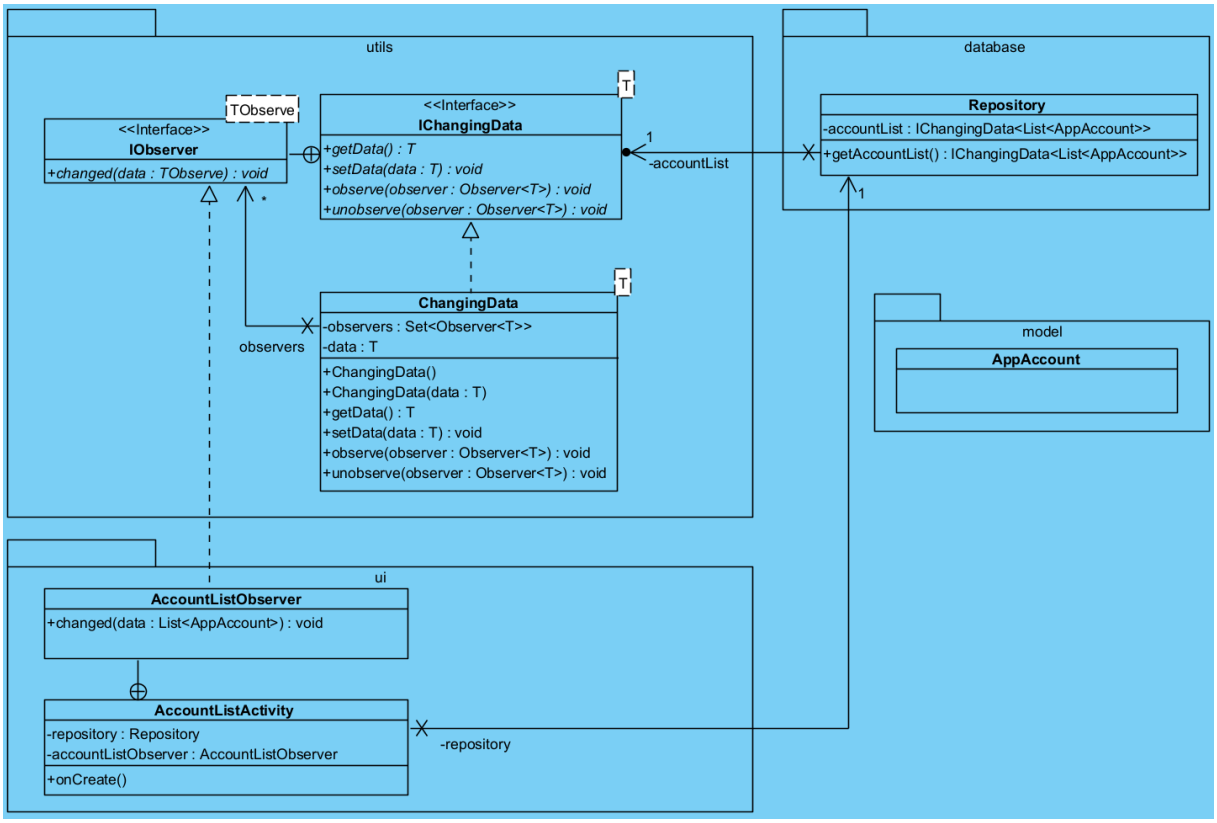
DESIGN Feedback: Information about transactions more accessible

We have added the transaction amount along with the information whether the transaction is an income or an expense to each list item in the transaction list. This should make it easier to see at a glance what each transaction represents.

1.3 Design Patterns

1.3.1 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);
  }

  ...
}
```

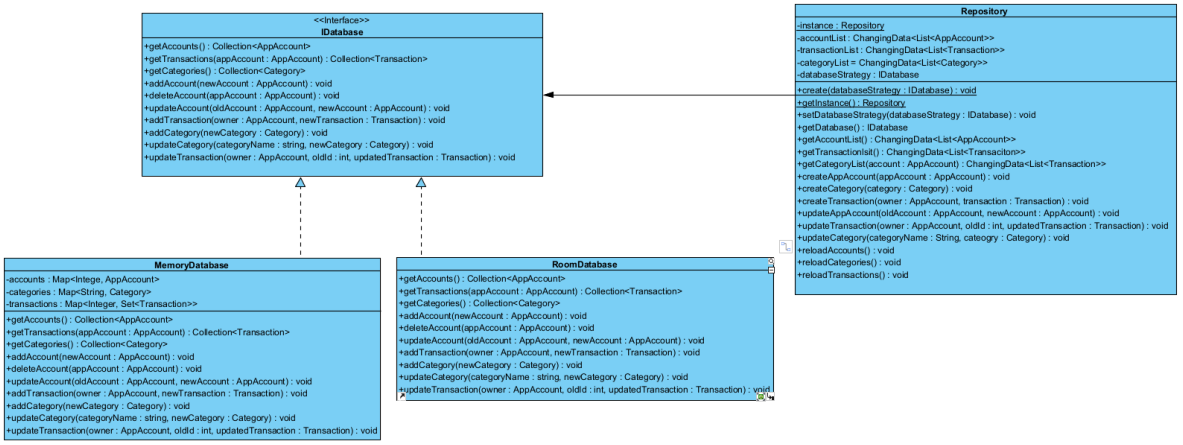
The following code fragment shows how the *onCreate()* method in *AccountListActivity* retrieves an *IChangingData* instance from the repository and subscribes an observer which updates the UI (in this case an *ArrayAdapter* of a *RecycleView*). The observer is implemented here as a lambda expression 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);
});
```

1.3.2 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.

Since milestone 1, we have upgraded the data storage to be able to be stored in a persistent Room database, not just application memory. The data access is handled by our Repository, which holds an interface field of the current database strategy. As we expected in milestone 1, moving to a persistent database didn't necessitate any changes to the existing infrastructure.



```
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;
  }
  ...
}
```

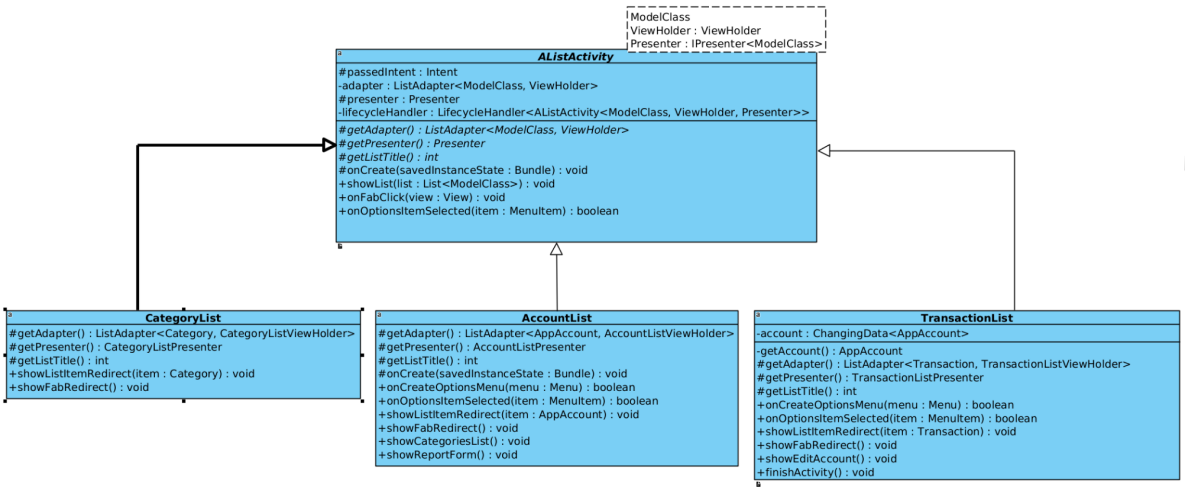
1.3.3 Template method pattern

The template method pattern allows to more easily define instances of a thing, by predefining some known common structure and defining which parts need to be implemented by each sub-instance. This allows us to extract common code, whilst defining the exact information a sub-instance needs to provide.

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.

In the diagram below AListActivity acts as the AbstractClass, whilst AccountList, TransactionList and CategoryList act as concrete classes each.




```
public abstract class AListActivity<ModelClass, ViewHolder extends
RecyclerView.ViewHolder, Presenter extends
IListActivityContract.IPresenter<ModelClass>> extends
   AppCompatActivity implements IView<ModelClass> {

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

```
 private LifecycleHandler<AListActivity<ModelClass, ViewHolder, Presenter>>
lifecycleHandler;
 protected Presenter presenter;

 protected abstract ListAdapter<ModelClass, ViewHolder> getAdapter();
 protected abstract Presenter getPresenter();
 protected abstract int getListTitle();

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

 @Override
 public void showList(@NonNull List<ModelClass> list) {
   adapter.submitList(list);
 }

 public void onFabClick(View view) {
   presenter.clickFab();
 }

 @Override
 public boolean onOptionsItemSelected(@NonNull MenuItem item) {
   // navigate back if back button is pressed

 }
}
```

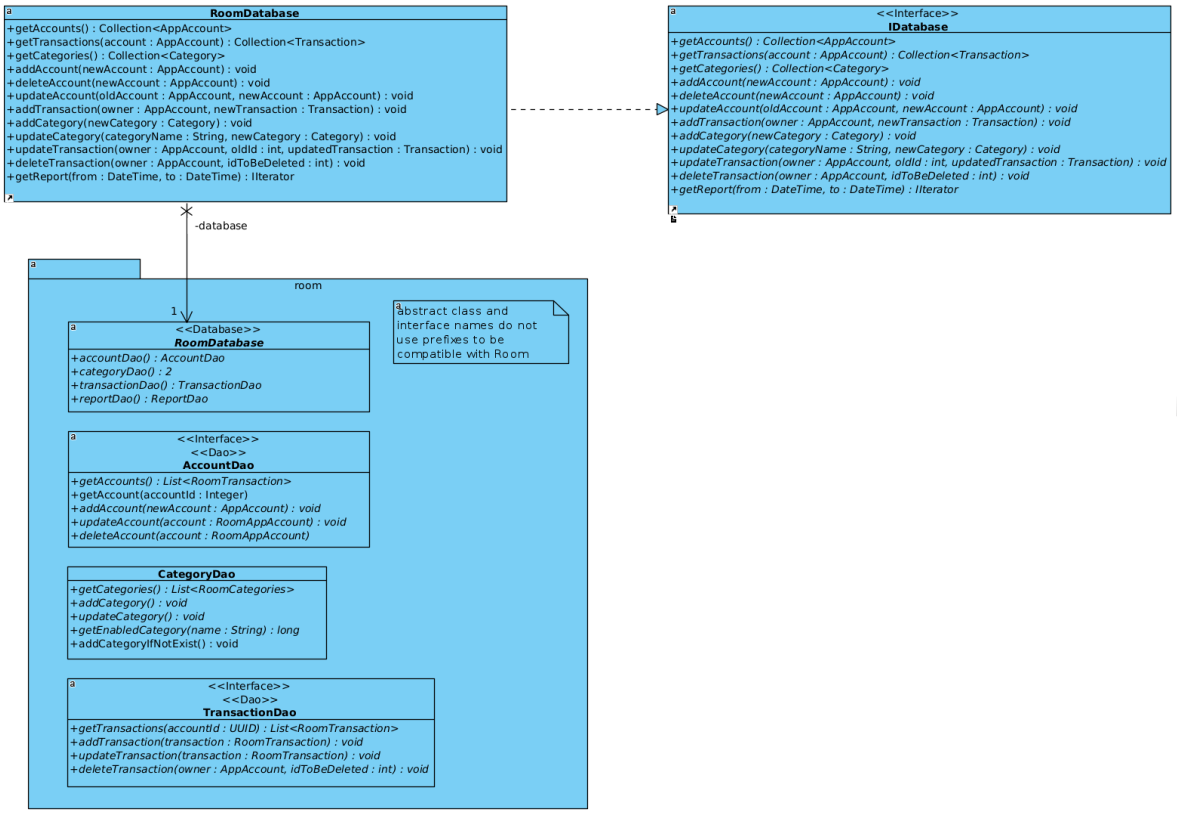
1.3.4 Adapter pattern

The adapter pattern serves to adapt one or more classes to a common interface. This allows to combine complex functionality together into a more simple whole and adapt functionality to an already used interface.

We have used this pattern to abstract the calls to the Room database. The AppRoomDatabase returns the different “Dao” objects needed to interact with the database. The job of the RoomDatabase class is to get the Dao objects from the room database and translate them to be compatible with the IDatabase implementation (and vice versa).

This helps us to abstract the entire Room implementation and use a much more straightforward approach to talk with our database. Further, it allows us to not have to reimplement the entire interface used in the app, when adding the Room Database.

In the diagram below, IDatabase is the Target, RoomDatabase is the Adapter and the different Dao classes are the Adpaptees.




```
public class RoomDatabase implements IDatabase {

 private final AppRoomDatabase roomDatabase;
 private final AccountDao accountDao;
 private final CategoryDao categoryDao;
 private final TransactionDao transactionDao;
```

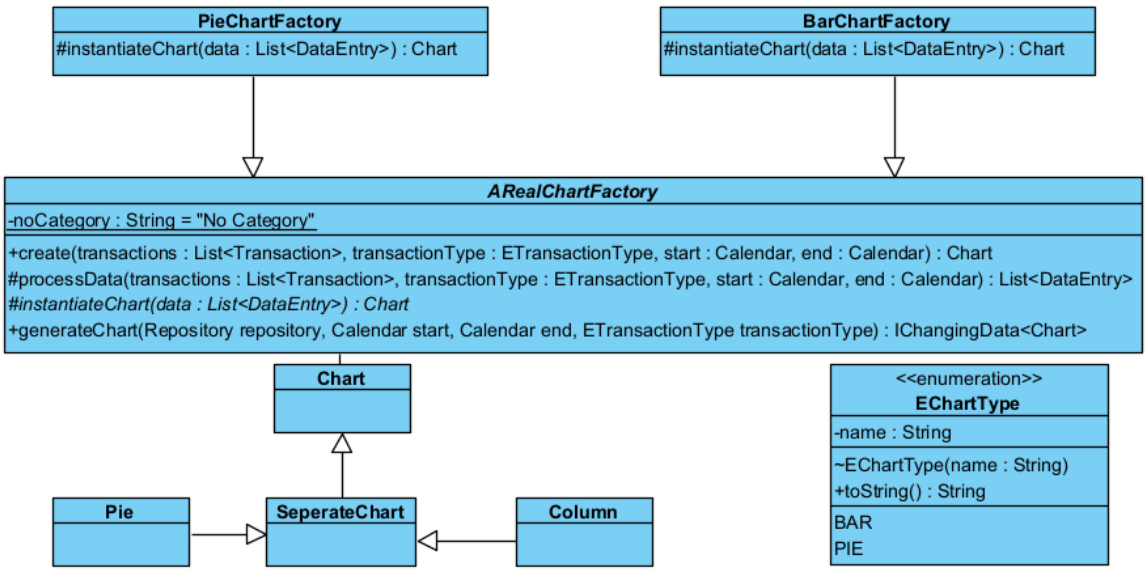
```
 public RoomDatabase(Supplier<AppRoomDatabase> roomDatabaseSupplier) {
   roomDatabase = roomDatabaseSupplier.get();
   accountDao = roomDatabase.accountDao();
   categoryDao = roomDatabase.categoryDao();
   transactionDao = roomDatabase.transactionDao();
 }

 @Override
 public Collection<AppAccount> getAccounts() {
   return accountDao.getAccounts().stream().map(roomAccount -> (AppAccount)
roomAccount).collect(
       Collectors.toList());
 }

 @Override
 public Collection<Transaction> getTransactions(@NonNull AppAccount account)
     throws DataDoesNotExistException {
   try {
     return transactionDao.getTransactions(account.getId()).stream()
.map(RoomTransactionWithCategory::getTransaction).collect(Collectors.toList());
   } catch (SQLiteException e) {
     throw new DataDoesNotExistException("Transaction");
   }
 }

 @Override
 public Collection<Category> getCategories() {
   return categoryDao.getCategories().stream().map(roomCategory -> (Category)
roomCategory).collect(Collectors.toList());
 }

 @Override
 public void addAccount(@NonNull AppAccount newAccount) throws DataExistsException {
   try {
     accountDao.addAccount(new RoomAppAccount(newAccount));
   } catch (SQLiteException e) {
     throw new DataExistsException("Account");
   }
 }

 // etc.
}
```

1.3.5 Factory Method pattern

When working on FR5 we noticed that the charts supported by the AnyChart-Android library have a similar structure when they are being created. Therefore, we decided to deploy the factory method pattern, which will allow us to easily aggregate the initialization process of charts in a single factory.

The abstract ARealChartFactory has a public method create method in which the creation process starts. The process method has an implementation in the abstract class which does a basic filtering of the transactions. The method was left protected to allow for subclasses to override it if they require different data entries to generate the chart. The abstract method instantiateChart has to be overridden by subclasses and defines the chart that the particular class will generate. The generateChart method is used by the ProxyChartFactory.

The Chart, SeperateChart, Pie and Column classes seen in the class diagram below belong to the AnyChart-Android library.




```
public abstract class ARealChartFactory extends AChartFactory {

    private final static String noCategory = "No Category";

    public Chart create(List<Transaction> transactions, Calendar
start, Calendar end, ETransactionType transactionType){

        List<DataEntry> data = processData(transactions, start,
end, transactionType);
        return instantiateChart(data);
    }

    protected List<DataEntry> processData(List<Transaction>
transactions, Calendar start, Calendar end, ETransactionType
transactionType){
        Map<String, Integer> filteredResult = new HashMap<>();
```

```
      //...
      // code that filters the passed data
      //...

        List<DataEntry> formattedData = new ArrayList<>();
        for (Map.Entry<String, Integer> entry :
filteredResult.entrySet()) {
            formattedData.add(new ValueDataEntry(entry.getKey(),
entry.getValue()));
        }
        return formattedData;
    }

    public abstract Chart instantiateChart(List<DataEntry> data);

    @Override
    public IChangingData<Chart> generateChart(Repository
repository, Calendar start, Calendar end, ETransactionType
transactionType) {
      //... method used by ProxyChartFactory
    }
}
```

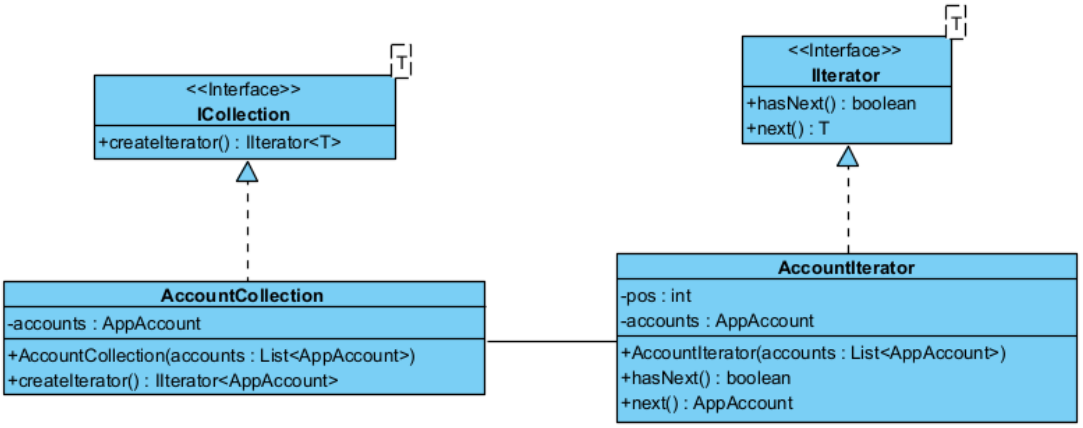
One of the subclasses that uses the AChartFactory class:

```
public class PieChartFactory extends ChartFactory{
    @Override
    protected Chart instantiateChart(List<DataEntry> data) {
        Pie pie = AnyChart.pie();
        pie.data(data);
        pie.title("Pie Chart");
        pie.legend().title().enabled(true);
        pie.legend().title()
                .text("Categories")
                .padding(0d, 0d, 5d, 0d);
        return pie;
    }
}
```

1.3.6 Iterator pattern

The iterator pattern allows us to easily traverse over the different AppAccounts in our application. The custom iterator is used in FR5 for the generation of the reports. We fetch all accounts in our database and then create an AccountCollection which allows us to iterate over it with the AccountIterator. When iterating over the collection we fetch all the transactions associated with the accounts and put them in a list, which is later used to generate the report.

An alternative would have been to just fetch all transactions from the database and skip adding an iterator altogether, but there are some drawbacks to it. Firstly, with our own custom iterator we can reduce the load that is sent to the database, by splitting up one big request into multiple smaller ones, which increases performance. And secondly fetching all transactions from our database would lead to transactions that don't belong to any account to be used in the generation of FR5. This is due to the fact that when an account is deleted the remaining transactions are not cleaned up, but instead remain in the database. With these arguments we chose to use the iterator pattern to solve this issue.




```
public interface IIterator<T> {
   boolean hasNext();
   T next();
}
```

```
public interface ICollection<T> {
   IIterator<T> createIterator();
}
```

```
public class AccountIterator implements IIterator<AppAccount>{

   private final List<AppAccount> accounts;
   private int pos = 0;

   public AccountIterator(List<AppAccount> accounts){
       this.accounts = accounts;
   }

   @Override
   public boolean hasNext() {
       return pos < accounts.size();
   }

   @Override
   public AppAccount next() {
       return accounts.get(pos++);
   }

}
```

```
public class AccountCollection implements ICollection<AppAccount> {

   private final List<AppAccount> accounts;

   public AccountCollection(List<AppAccount> accounts) {
       this.accounts = accounts;
   }

   @Override
   public IIterator<AppAccount> createIterator() {
       return new AccountIterator(accounts);
   }
}
```

1.3.7 Proxy pattern

This pattern is 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 ProxyChartFactory takes care of displaying a placeholder chart to the user, instead of a blank screen. The ProxyChartFactory class fulfils this purpose by holding an

instance of ARealChartFactory according to which chart type the user selected and displaying a proxy until the real factory produces a result.

```
public class ProxyChartFactory extends AChartFactory {
      private final ARealChartFactory chartFactory;

private void getData(IChangingData<Chart> ret, ETimeSpan timeSpan,
ETransactionType transactionType, Repository repository) {
// Re-fetch data from the db into the repository
        IChangingData<ERepositoryReturnStatus> status =
repository.reloadAccountsWithStatus();

        status.observe((newStatus) -> {
            switch (newStatus) {
                case SUCCESS:
            // once the data is updated, fetch it and manipulate it
to create necessary data for the real chart factory

                    try {
// Update the chart with the real data once ready
           ret.setData(chartFactory.create(transactions, timeSpan,
transactionType));
                    } catch (RuntimeException e) {
                       ...
                    }
                ...
            }
        });
    }

public IChangingData<Chart> generateChart(Repository repository,
ETimeSpan timeSpan, ETransactionType transactionType) {
      Chart proxy = this.createProxyChart();
      // instantiate ChangingData with the proxy
IChangingData<Chart> ret = new ChangingData<>(proxy);

      // asynchronously load the real chart
      getData(ret, timeSpan, transactionType, repository);

      // return the proxy in the mean while
      return ret;
      }
```

```
}
```

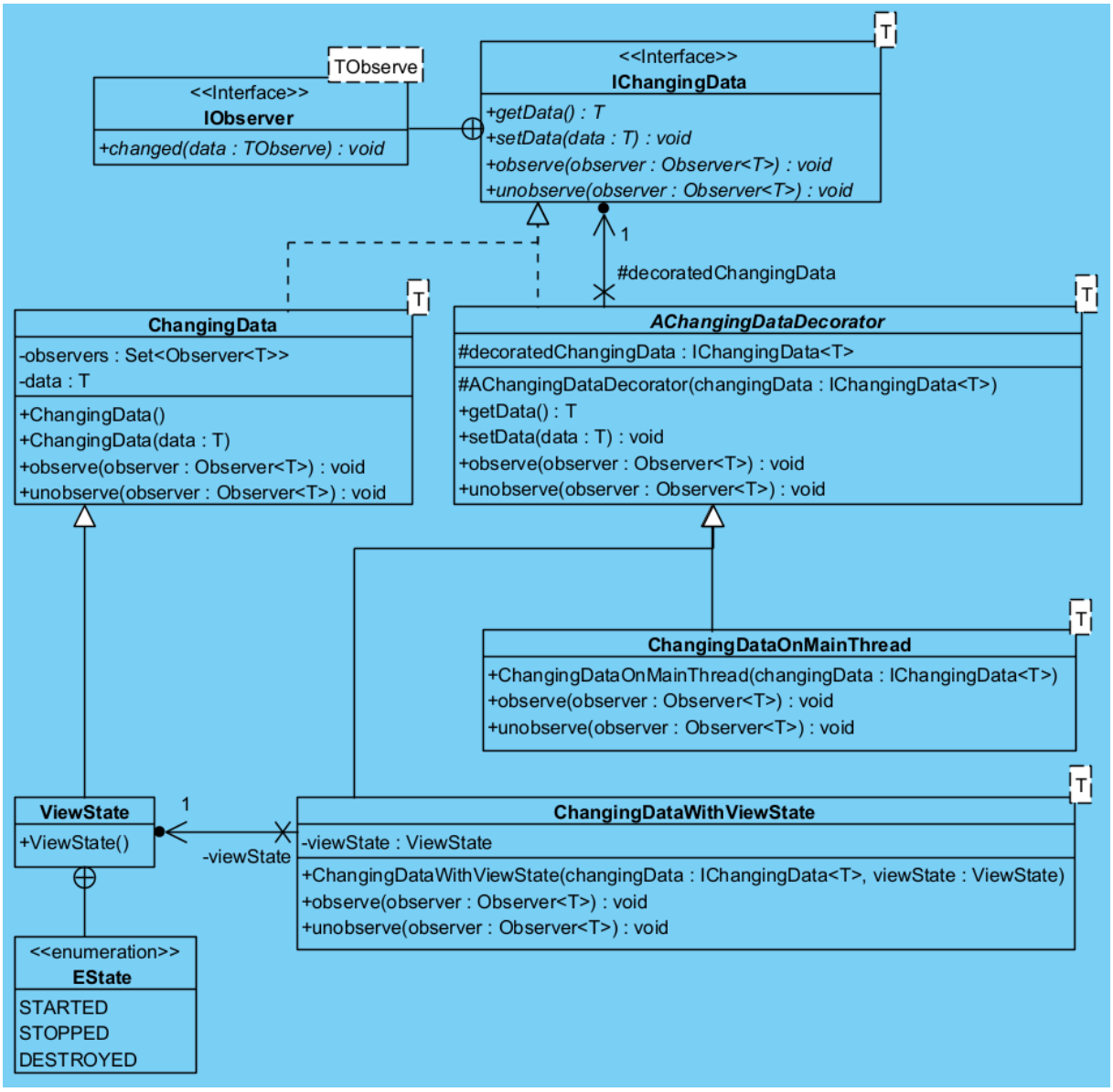
1.3.8 Decorator Pattern

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 1.3.1). 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.

```
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);
  }
}
```

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*.

```
abstract class AChangingDataDecorator<T> implements
IChangingData<T> {
  @NonNull
  protected final IChangingData<T> decoratedChangingData;

  protected AChangingDataDecorator(@NonNull IChangingData<T>
changingData) {
    this.decoratedChangingData = changingData;
  }

  @Override
  public T getData() {
    return decoratedChangingData.getData();
  }
  @Override
  public void setData(T data) {
    decoratedChangingData.setData(data);
  }

  @Override
  public void observe(@NonNull Observer<T> observer) {
    decoratedChangingData.observe(observer);
  }
  @Override
  public void unobserve(@NonNull Observer<T> observer) {
    decoratedChangingData.unobserve(observer);
  }
}
```

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 most interesting method is *observe(..)*:

```
@Override
public synchronized void observe(@NonNull Observer<T> observer) {
 Observer<T> newObserver = observerMap.get(observer);
 if (newObserver == null) {
   newObserver = data -> mainThreadExecutor.runOnMainThread(() ->
      observer.changed(data));
   observerMap.put(observer, newObserver);
 }
 super.observe(newObserver);
}
```

It creates a new observer as lambda, calling the original observer on the main thread. This new observer is registered at the decorated *ChangingData* instance instead of the passed one. This is done by calling the parent class, which is the abstract class *AChangingDataDecorator<T>*, which in turn calls observe on the decorated instance *decoratedChangingData*.

The second decorator we use: *ChangingDataWithViewState<T>*, 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).

Using those decorators is quite easy. The constructor of the decorator takes a reference to the *ChangingData* which should be decorated as a parameter. This can be directly done during construction of the ChangingData as done within the *Repository*:

```
new ChangingDataOnMainThread<>(new ChangingData<>(new
ArrayList<>()));
```

Or it can be done later by the user of the ChangingData, as we do it in our *Presenter* code:

```
new ChangingDataWithViewState<>(repository.getCategoryList(),
viewState);
```

2 Implementation

2.1 Overview of Main Modules and Components

The project is structured in the following 4 main modules that interact with each other:

Database:

The database allows us to store the data that is exchanged between the user and the app. The requests that come from the UI module are all sent to the repository, which handles and processes all requests. Depending on which database strategy is currently in use, it will save the data to either the MemoryDB or RoomDB. Apart from storing the data itself it also is responsible for keeping track of which accounts and transactions are associated with each other, preventing duplicate objects from being inserted, keeping the data up to date and much more.

UI:

The UI module allows the user to interact with the system by viewing the data that is stored in the database. It gives the user the ability to create new data or edit existing ones. The manipulatied data is then sent to the database module to store the changes that have occurred.

Model:

The model module is the backbone of the whole application since all other modules rely on the classes provided by this module to interact with each other and exchange data.

Util

Similarly, to the Model, this module is equally crucial for the functionality. It provides the rest of the system with helper functions which allow them to interact with each more efficiently. One of the major ones is ChangingData which allows our activities to be always up to date when the data changes somewhere in our database.

2.2 Coding Practices

Coding convention / style guide

We used the Google style guide to keep the code base consistent across all developers involved in the project.

Comments

For commenting in Java we used JavaDoc to document most of the classes in our project. This increases the readability and maintainability of our code.

Operations/Naming

We followed the java guidelines of using a camel-case for our project. We also tried to keep the naming style of functions and methods consistent across the team.

2.3 Defensive Programming

Assertions

In our project we leveraged the use of assertions mainly when changing between different activities. When transitioning from one activity to another we usually pass some sort of object or value in the Intent to be used in the other activity. For example, when going from AccountList to TransactionList we pass on the selected AppAccount object, so the TransactionList activity knows which transactions it must display in its list. There should never be a case where the object in the intent is null, so we verify it with an assert. The code snippet below showcases a method in the TransactionList class that fetches the serialised ParcelableAppAccount object from the intent.

```
@NonNull
private AppAccount getAccount() {
 if (account == null || account.getData() == null) {
   account = new
ChangingData<>(passedIntent.getParcelableExtra(EIntents.ACCOUNT.toString()));
 }

 assert (account.getData() != null);
 return account.getData();
}
```

This use of defensive programming can also be found in all activities which are responsible for editing existing objects in our database related to FR1, FR2 and FR3 as can be seen in the code example below. Here we make sure that the Transaction the user wants to edit was actually passed in the Intent.

```
@Override
protected void setup() {
 Intent intent = getIntent();
 displayedTransaction =
intent.getParcelableExtra(EIntents.TRANSACTION.toString());
 assert (displayedTransaction != null);
 // more unrelated logic
 //...
}
```

Error handling

In parts of our code where it's sometimes expected that bad values might enter our app, we make use of different error handling techniques to mitigate it. One of the main ways invalid values could enter is through the user that utilises our app to track their expenses. Every form field that the user can utilise is an attack vector for bad data to enter. For most of them we were able to use android studios built in features to ensure that doesn't happen by restricting the type of the data the user can enter (for example only a date object can be entered in a date field, etc.). In cases where that wasn't possible, we had to handle the data ourselves.

A good example to showcase this is when creating a new account. The user has the option of entering a name, spending limit and account type. The account type is selected through a spinner so there is no way a wrong value can be selected. For the spending limit field, the user can only input integers. In the case of nothing being entered in the field we use a neutral value of 0 when creating the account. Lastly, we have the account name field. Here we must verify that the entered string is not empty and that the account name isn't already used in our database. If either of that is the case, we throw an exception. This exception is then caught, and a toast notifies the user that the name was invalid.

3 Software Quality

3.1 Code Metrics

Lines of Code

The LOC code metrics analysis was performed on the 19th of January at 22:00. Some minor changes have happened since but they don't affect the analysis in a major way.

Our application codebase contains 97 classes in 28 java packages, along with 9 packages containing XML data for activities and other resources. Throughout these files we have a total of 6965 lines of code, 1270 of those are comments and 1038 lines of whitespace. We used cloc and IntelliJ's "Statistic" plugin to obtain these metrics.

Our testcases consist of 3044 lines of code, thereof 62 lines of comments and 562 lines of whitespace.

Compared to the DESIGN milestone we added 3685 lines of code in the application, approximately doubling it. Further, we added 2936 lines of code of testcases, which means nearly all test cases were written in the FINAL stage matching the planned milestone requirements.

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 *ATransactionActivity*, the most complex abstract base 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. The most complex class in the whole application is the class *Repository* from the *database* package, which is the facade of our database.

Java Lint Warnings

Using Android Studio's built in code analysis yielded many warnings regarding unused methods, which we could track down to the way we implemented our model view presenter pattern with abstract classes. We had to disable this specific warning, dropping the count to 32 warnings regarding Nullability problems, which we nearly all could track down to the way we have to initialize entities for the room database. Further warnings informed us that we could move fields to local, but we decided to keep them as this matches our design better. Many of the warnings we mentioned in

the DESIGN report are gone as we already expected back then. These were mainly caused by partially unimplemented functionality, which is now finished in the FINAL milestone.

Android Lint Warnings

As already mentioned in the DESIGN report, the Android Lint warnings are mainly about accessibility, usability and localization. As neither of these three were goals of the project, we mainly focused on other aspects of the application and only provided the UI functionality necessary for presenting the application.

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.

3.2 Testcases for Functional Requirements

We use a mix of unit tests running on the host and on the target platform together with UI tests to cover our functionality with tests. Due to our layered approach, using the MVP pattern and the design decision to use Android specific code only in places where it is absolutely necessary, our code reached a high degree of testability. Due to dependency injection and factory patterns we can mock most parts of the code and use unit testing for most classes. We have a total sum of 132 test cases.

FR1: Accounts

We have dedicated test cases to test the account functionality on various levels. The model.account package contains unit tests for the account model. The *ui.account.** packages contain test cases for account list, create account and edit account. These are again separated into tests for the presenter in *Account*PresenterTest* and *AccountListUnitTest* and into tests for the view in *Account*Test*. In sum we have 27 test cases for FR1. Apart from these tests the utility classes (e.g. *ChangingData*, *Parcelable**) have separate tests which can be counted to all functional requirements.

In our unit tests we use Mockito to mock dependencies and inject them either via constructor or via a factory pattern into the unit under test. An example is the injection of a mocked presenter into the views (Android Activities) via a factory pattern, which is necessary as the view is not constructed by our code, but by the Android framework and so we can not inject via the constructor:

```
public class AccountListUnitTest {
  @Mock
  AccountListPresenter mockPresenter;

  @Before
  public void setup() {
    AccountListPresenter.setFactory((Repository repository) ->
mockPresenter);
  }
}
```

FR2: Categories

Test cases for FR2 are designed similarly to FR1 and included in the packages *model.category* and *ui.categories.**. 23 test cases are covering FR2.

FR3: Transactions

The same is valid for FR3 with test cases in the packages *model.transaction* and *ui.transactions.**. In sum we have 43 test cases for FR3.

FR4: CRUD

The database system is based on the room database and therefore Android specific. It is tested with 19 test cases in the platform tests in the package *database.room*.

FR5: Reports

For the report view we use an external chart library. We run into limitations of testability of the library which we attribute to the fact that we are using the trial version. Our test cases cover the basic functionality of the system, but we can not test anything related to the chart views. So we are at the moment limited to manual testing in this area. As the reports are due to our modular system, a completely separate functionality, we can live with this limitation at the moment and manually test this part after changes in the area of reports. Additionally we test the chart proxy and chart factory code with test cases in *ProxyAChartFactoryTest*.

FR6: Spending Limit

Accounts over the spending limit are shown as colour annotations in the account list, therefore the tests for FR1 are also covering FR6.

3.3 Quality Requirements Coverage

In this chapter we show for each quality requirement how we achieved it and which parts cover it.

QR1: Comment & Documentation

We use JavaDoc comments throughout the code to document all classes and their interfaces. We generate out of the JavaDoc comments a complete code documentation which is available in the folder *implementation/JavaDoc*.

QR2: Style Guide

As coding style we agreed upon the Google Java Style Guide[2] 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*)

A common coding style helps to keep the code in a consistent format and improves the readability.

QR3: Common Coding Practices

To achieve good maintainability and readability of the code we applied common coding practices. As already mentioned in QR2, we defined a common coding style. This includes having a consistent naming convention throughout the code and our test cases. Apart from formal naming conventions we also tried to find good names for classes, methods and parameters to have code which stands for its own. Where good naming could not reach the goal, we used code comments.

We further used code comments in JavaDoc style to be able to generate a complete class and interface documentation as described in QR1.

Input from code reviews helped further to keep the code understandable. In cases where functions reached a high complexity level or did multiple things at once, code was split into separate methods and/or classes to reach higher cohesion.

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

QR4: Defensive Programming

We use techniques of defensive programming throughout the application. We use exceptions internally for communicating errors between methods, classes and modules, but we use exception handling before the exception reaches the UI.

We also introduced asynchronous error and state communication using our *ChangingData* observer class to be able to communicate results of asynchronous database operations up to the views. So we can inform the user much better of error situations and handle them gracefully. One example is the creation of a duplicate account or category entry. We prevent the user from creating duplicate entries, but as the operation is done in a background thread, we asynchronously inform the user about the outcome via a Toast message. With this technique the application continues to operate normally and the user is still informed if necessary.

Another defensive programming technique we use is to set values back to safe default values. One example of this is if a transaction gets changed from income to expense or vica versa. The category would not match any more in this case, as categories are either income or expense categories. To prevent an inconsistent situation, we automatically change the transaction to have no category and let the user choose a new one if she wants to.

QR5: Key Design Principles

We used object oriented design and design patterns to have a good base to be able to fulfil the key design principles:

- **Abstraction**
  We designed our system in modules/packages for the main functionality and defined the interfaces abstracting away the inner workings of those modules. We continued this approach on the class level, designing the system with the functionality and interfaces in mind, but keeping the internal details abstracted away. This allowed us to reach a better understanding of the complete application without knowing every detail upfront.
- **Hierarchy**
  We used hierachies during the design (packages, classes, interfaces) and in our software (business logic -> database adapter -> database strategy -> database implementation -> utility classes) to be able to have different views on the system with different levels of details.
- **Modularity**
  Our application is separated into many modules with low coupling between each other. We have modules containing the database logic completely separated from modules containing the business logic (the Presenter) again separated from modules for the views. Those modules interact only via interfaces with each other where necessary to reach low coupling between

modules. Within each module we tried to combine exactly those parts which need to work tightly together to increase the cohesion within the modules.

- **Information Hiding and Encapsulation**
  Every module and every class we use in our application encapsulates its own state and only provides a well-defined interface to its internal state to other modules and classes. This encapsulation allows changing the inner workings of a class without changing any other classes or modules. We did this to decrease the complexity of the system and increase the testability of every class in unit tests with mocked interfaces to other classes. We also used strict access modifiers (private, protected) where possible. In some cases (especially to allow Mockito to create mocks of classes) it was necessary to use more relaxed access modifiers.
- **Separation of Concerns**
  We tried to achieve separation of concerns on all levels. We separated the whole application into distinct parts responsible for different concerns (database, UI, ..). Down to the level of splitting methods which were handling 2 distinct tasks. We also separated code which was handling Android specific tasks from code which was handling generic tasks. With our Model-View-Presenter pattern we could also separate the view (UI) concern from the business logic (Presenter).

QR6: Testing & Review

We use many types of testing in our application to reach a good coverage of all our functional requirements as shown in chapter 3.2 of this document. Our test cases consist of unit tests and functional tests. Due to the use of the MVP pattern we can test model, view (UI tests) and presenter separately in unit tests while mocking the other parts via dependency injection. We also have test cases for the utility classes and the database implementation. On top of that we have integration UI tests testing all modules together.

We planned to achieve a high degree of automated testing to be able to detect new bugs early during the development cycle via regression testing, which can be easily integrated in continuous integration systems later on.

As testing can not achieve a high level of code quality alone, we also enforced a strict code review policy. Every code change had to be reviewed by at least one other team member and for central code changes all team members had to give their ok before the code got merged into the main development branch. Through testing and code review together with discussing design decisions in team meetings is the basis of our strategy to achieve a high quality of our code and a low number of defects.

QR7: Design Patterns

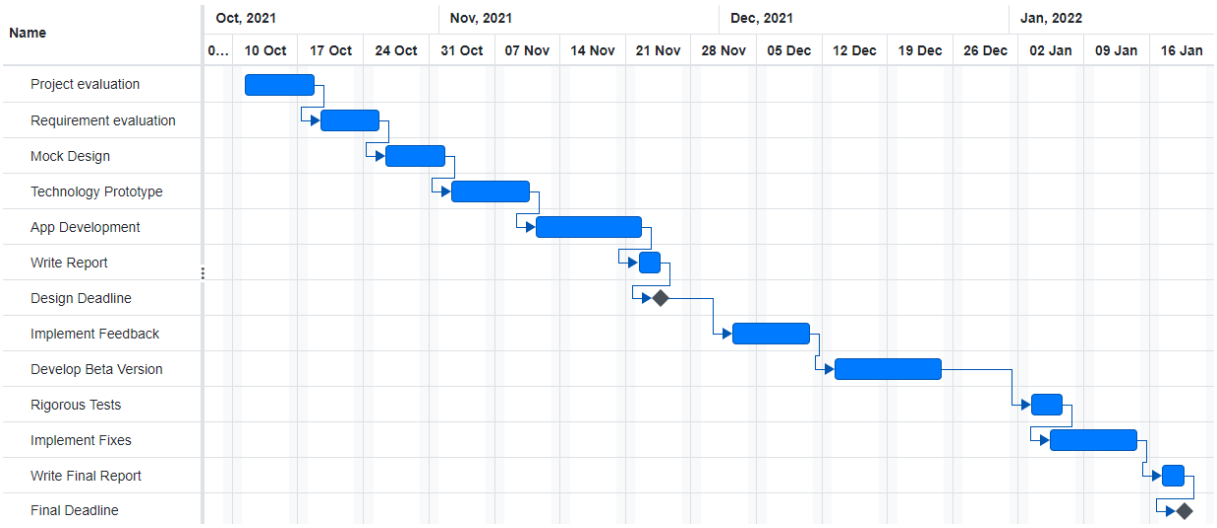
We implemented 8 design patterns and documented them in detail in chapter 1.3 of this document. The distribution of these patterns among the team members can be seen in chapter 4.2.

4 Team Contribution

4.1 Project Tasks and Schedule

Throughout the semester we tried to stick to a simple plan visualised by the Gantt chart below. We started our development cycle by evaluating the task at hand and creating small scale mocks of the app to familiarise ourselves with the Android development ecosystem. After each team member familiarised themselves with the development process, we shared our findings and refined our initial goals. After that we roughly divided the work and set out to develop a prototype. We tested the app synchronously during this time, with frequent code reviews to assure code quality, as well as to keep team members up to date with the parts of the codebase they didn't actively work on. After we finalised the design milestone requirements, we wrote the report and took a small break in development.

After receiving feedback for the design milestone, we restarted development with some refactors, as well as implementations of the remaining features. We also took this time to improve our codebase with documentation and tests. After all this was done, we wrote the final report.



4.2 Distribution of Work and Efforts

Bernhard Clemens Schrenk:

Completed work	• ChangingData* (Observer) • Model-View-Presenter: BaseContract & ABasePresenter • Android/Generic code abstractions • Package database.room.* • Technology Stack • Decorator pattern • Chapter 3. Software Quality • Unit Tests • Bug fixes • Code reviews
Design patterns	• Observer (DESIGN phase) • Decorator (FINAL phase)
Work time for FINAL milestone	40 hours

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) • Migration to the MVP pattern • Adapter pattern implementation • Repository on separate thread • JavaDocs • Unit tests • Code reviews
Design patterns	• Template method (already implemented) • Adapter (will be used to abstract the Room Database)
Work time for FINAL milestone	40 hours

Samuel Šulovský:

Completed work	• Proxy pattern • Reports structure • Spending limit (FR 6) • Categories (FR 2)

	● Repository (layer above DB implementing strategy pattern to abstract DB work from the app) ● Refactors, polish and bug fixes ● Document Review ● JavaDocs ● Unit tests ● Code reviews
Design patterns	● Strategy ● Proxy
Work time for FINAL milestone	40 hours?

Rumen Angelov:

Completed work	● Accounts (FR1) and it's associated database functionality ● Iterator Pattern ● Factory Method Pattern ● Reviews ● Documentation ● Unit Tests ● Bug testing and fixing ● JavaDocs ● Reports (FR5)
Design patterns	● Iterator ● Factory Method
Work time for FINAL milestone	37 hours

4.3 How-To Documentation

Installation and Start

1. Download app-debug.apk from GIT repository folder implementation
2. Start Android Virtual Device via Tools->AVD Manager in Android Studio
3. If not already done, use the button Create Virtual Device to create a new device:
4. Create a Pixel 2 API 29 device
5. Click the green play button on the Pixel 2 API 29 device in AVD manager
6. The virtual Android device starts
7. Wait until the home screen is shown

8. Drag the app-debug.apk file from your filesystem and drop it onto the home screen of the virtual Android phone
9. The application gets installed
10. Swipe up to reach the Android app launcher
11. Click on “App” to start the application

Testing

FR1: Accounts

1. Create a new Account with the “+” button on the bottom right side.
2. Fill out the new Account form and click save.
3. Tap the account in the list, then click the pencil icon in the top right to access the edit menu, where you can edit the account. You can also delete the account here by pressing the garbage bin icon in the top right.

FR2: Categories

4. Navigate back to the account list (for example by pressing back in the menu bar, pressing the back button, or relaunching the app).
5. Click on the 3 points menu in the header bar.
6. Click on Categories.
7. Tap the floating action button in the bottom right side to add a category.
8. Tap a category in the list to access the edit menu which also allows you to delete the category by pressing the appropriately named button.

FR3: Transactions

9. Navigate back to the account list (for example by pressing back in the menu bar, pressing the back button, or relaunching the app).
10. Click on your previously created account to reach the Transaction List of your new account.
11. Use the “+” button on the bottom right to create a new transaction.
12. Fill out the form and click save.
13. The transaction is now in the transaction list.
14. You can tap the transaction to access the edit dialog to edit or delete the transaction to your liking.

FR1-FR4: CRUD for FR1-FR3

15. Existing accounts, transactions and categories can be edited and deleted with the buttons in the header bar.

FR4:Persistent Storage

16. Kill the app: The task manager can be opened with the square button on the phone bottom right and then the application can be swiped upwards out of the screen.
17. Restart the app: Swipe up to reach the app launcher, click on “App”.
18. All data is still available.

FR5: Reports

19. Navigate back to the account list (for example by pressing back in the menu bar, pressing the back button, or relaunching the app).
20. Click on the 3 dots menu top right side in the header bar
21. Click on Report
22. Select a chart type and whether you'd like to see income or expense transactions.
23. Fill out the form with the date and the length of the report in days
24. Click on Generate Chart
25. The chart is shown

FR6: Spending Limit

26. Go back to the account list.
27. Select an account from the list by tapping on it.
28. Edit the account to add a spending limit by tapping the pencil button in the top right
29. Add a spending limit which is higher than the balance of the account.
30. Go back to the account list.
31. The account background is red, showing that the spending limit is reached.
32. Change the spending limit to 2 Euros below the account balance
33. Navigating back to the account list, you should see the account marked as yellow, as it is nearing the spending limit.