

Software Engineering 2

DESIGN REPORT

Team number:	0401
---------------------	------

Team member 1	
Name:	Benjamin Beslic
Student ID:	01568138
E-mail address:	a01568138@unet.univie.ac.at

Team member 2	
Name:	Adhurim Kuci
Student ID:	01576894
E-mail address:	a01576894@unet.univie.ac.at

Team member 3	
Name:	Felix Mühle
Student ID:	12138045
E-mail address:	a12138045@unet.univie.ac.at

Team member 4	
Name:	Arsen Keshishyan
Student ID:	01576524
E-mail address:	a01576524@unet.univie.ac.at

1 Design Draft

1.1 Design Approach and Overview

After we as a group looked at and analyzed the requirements for the project, we decided to follow a plan that leads us to a finished design and implementation of the project. The first step in this plan was for each of us to do a first iteration of the UML diagram so that we can compare these results for the next meeting and arrive at a correct or meaningful UML diagram.

Some of the results from the first iteration of the UML Diagram:

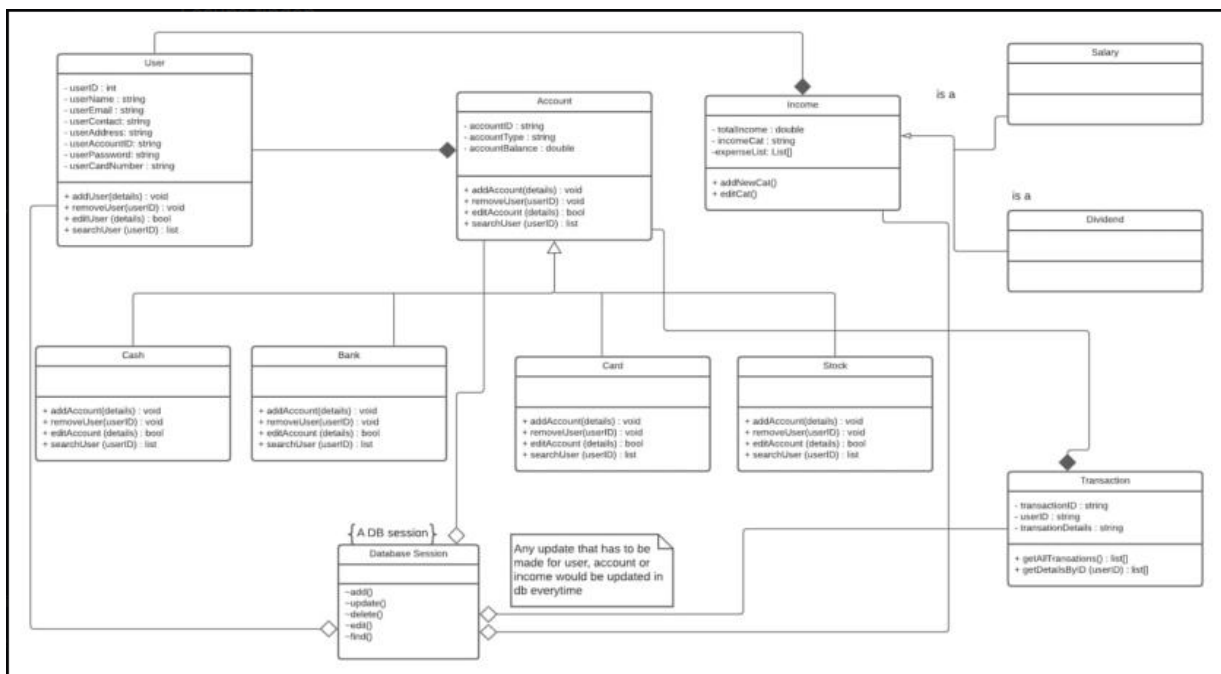


Figure 1: UML Prototype 1

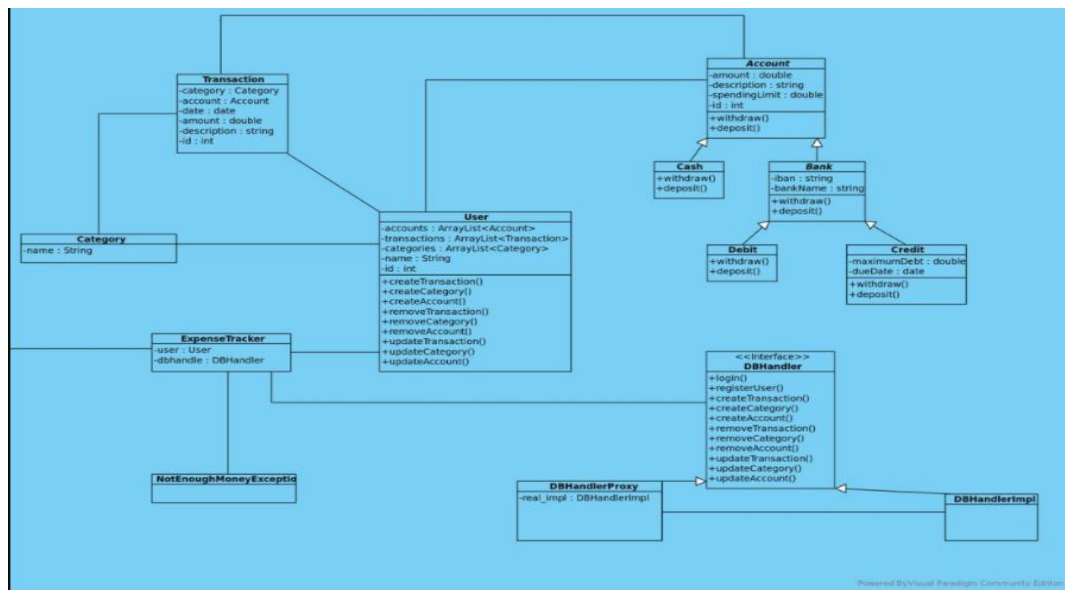


Figure 2: UML Prototype 2

We also considered where the design patterns should best be implemented and we have come to different results:

As an example, the use of a strategy pattern in Account klasse, which later proved to be more suitable for the factory pattern in other iterations.

Different approaches for design patterns:

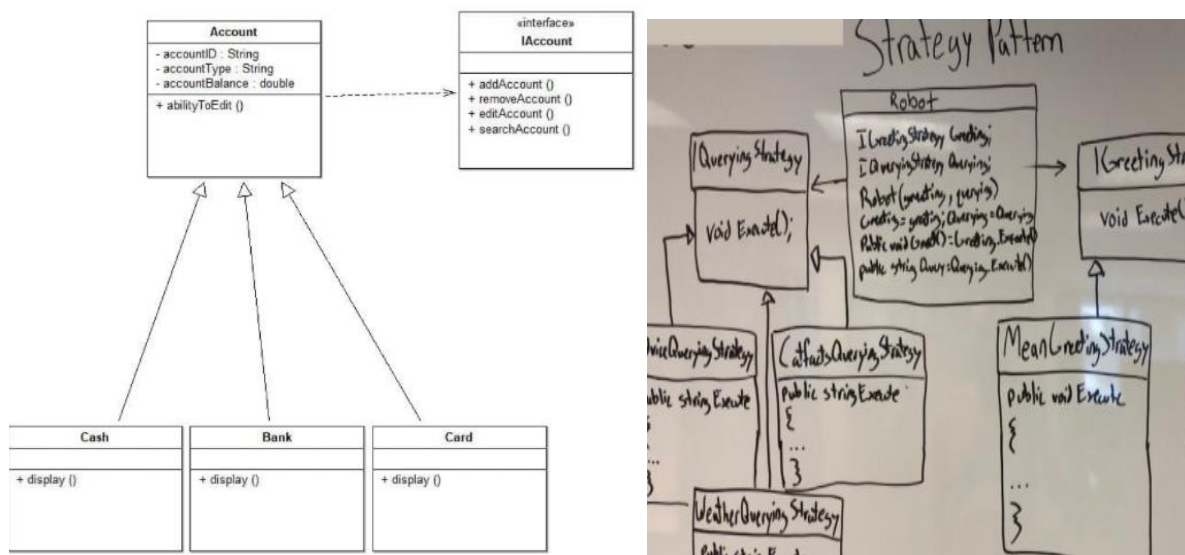


Figure 3 & 4: different approaches for design patters

Current solutions in our design phase can be seen in other point.

1.1.1 Class Diagrams

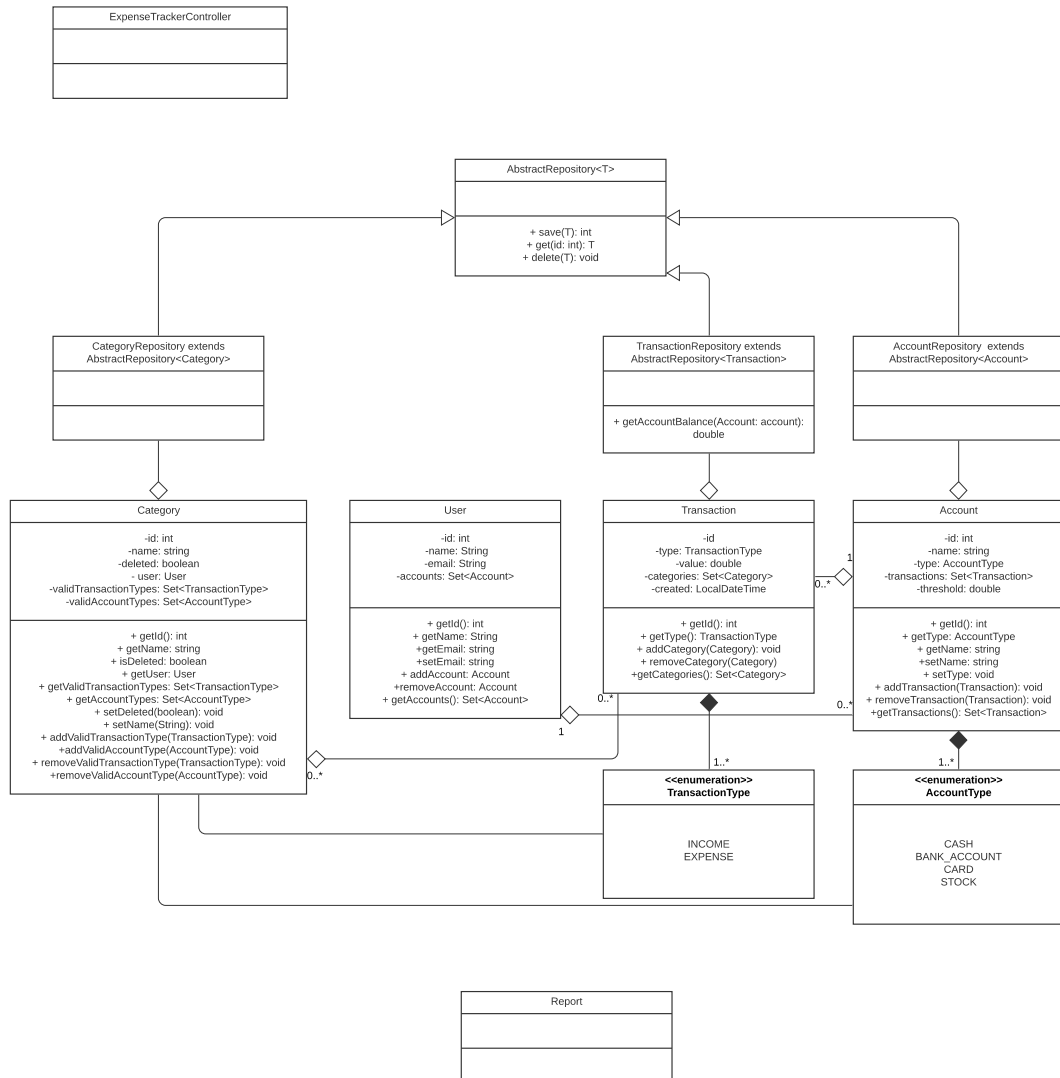


Figure 4: UML resulting from planning phase

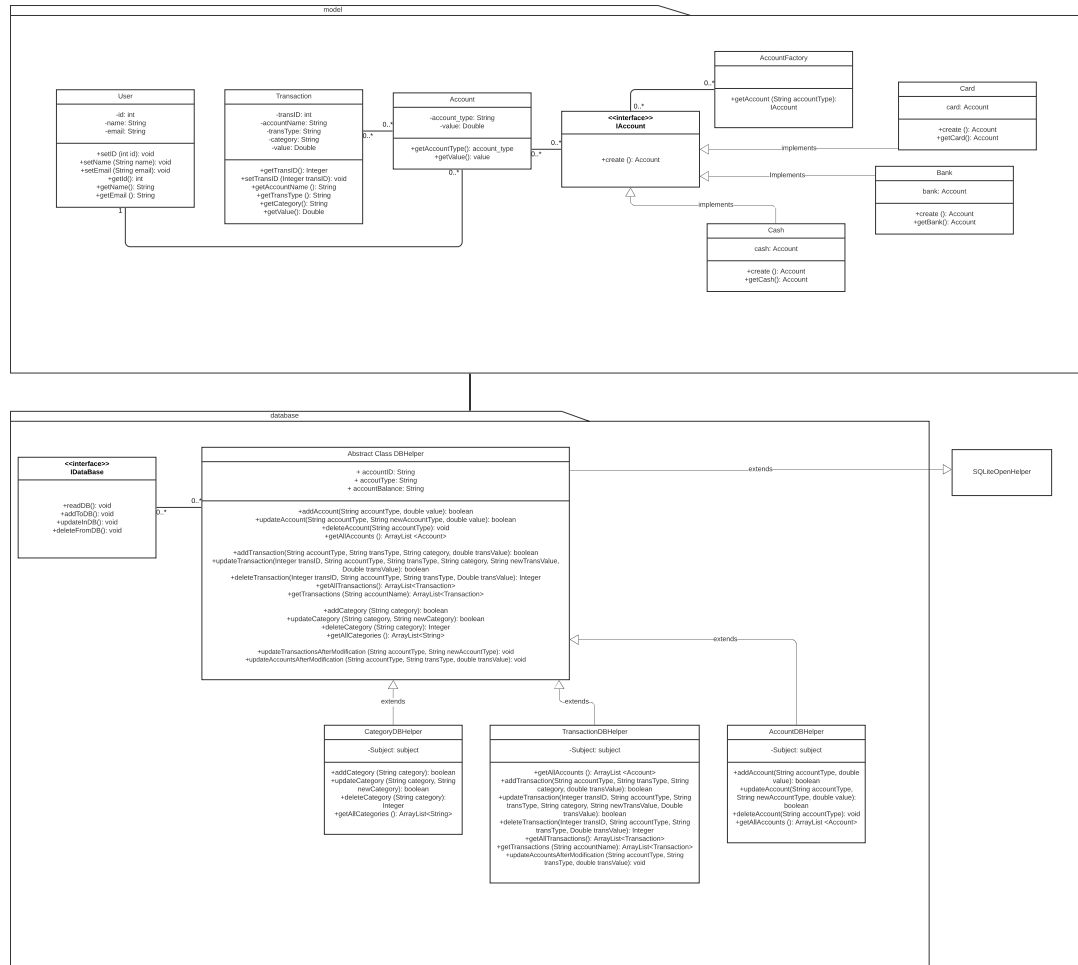


Figure 5: current UML based on actual implementation

1.1.2 Technology Stack

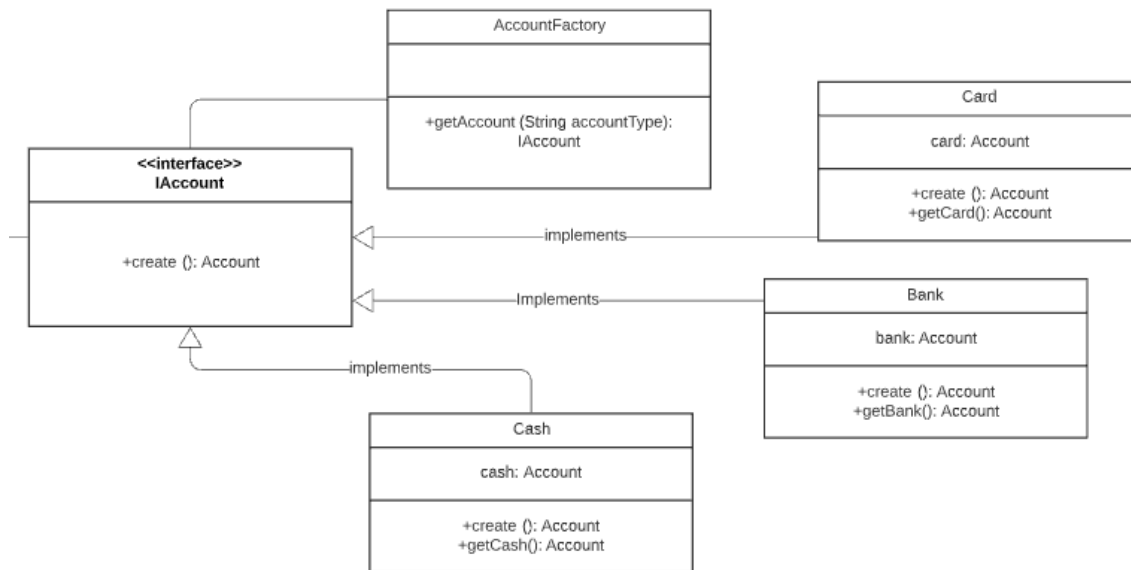
We have tried to make our technology decisions based on the project hints and FAQs. In general, our implementation includes predefined or provided features or libraries. For the Java classes we used the Eclipse IDE for Java Developers, because we thought that implementing and testing code would be faster and easier. The patterns that were used in code, we used the existing lecture materials. For the database, we decided to use the SQLite, as we have gained more experience with SQL database from previous projects. The Visual Paradigm was used to create UML diagrams. The app was tested on Pixel 3 XL API 28.

1.2 Design Patterns

1.2.1 Factory Pattern

This pattern enables the creation of objects inside a class using a factory method instead of creating an object directly, which is a better way and it has its own benefits. So we create object without exposing the creation logic to the client and refer to newly created object using a common interface. This kind of programming meet encapsulation and information hiding.

Based on our requirements, this pattern is useful because we have to create different account types for the account class. As mentioned above, this creation of objects is done in a better way by not creating these objects directly, but with a factory method. So in our program are many cases where at same place we need to create sometimes BANK instances, sometimes CARD instances and sometimes CASH instances. So based on what the user chooses we create an instance of class AccountFactory and we pass the account's data to the method `getAccount()` of the class AccountFactory, an on these methode the defined type well be created and its instance will be returned.



```

3  public interface IAccount {
4
5      public Account create();
6
7  }

```

The Interface IAccount will be implemented from the account types which override the method create() for creating instances of corresponding Object.

```

3  public class Bank implements IAccount{
4
5      Account bank;
6
7      |
8      @Override
9      public Account create() {
10         Account bank = new Account("BANK", 0.0);
11         return bank;
12     }
13     public Account getBank() { return bank; }
14
15
16
17 }
18

```

Class Bank implementing the IAccount interface.

```
public class AccountFactory {  
    //use getShape method to get object of type shape  
    public IAccount getAccount(String accountType){  
        if(accountType == null){  
            return null;  
        }  
        if(accountType.equalsIgnoreCase("BANK")){  
            return new Bank();  
        } else if(accountType.equalsIgnoreCase("CASH")){  
            return new Cash();  
        } else if(accountType.equalsIgnoreCase("CARD")){  
            return new Card();  
        }  
        return null;  
    }  
}
```

The AccountFactory class based on the input from the getAccount() method, creates an Object of this type. The parameter (accountType) is Spinner input converted to string.

```
final Spinner categories = new Spinner(this);  
categories.setAdapter(adp);  
layout.addView(categories);
```

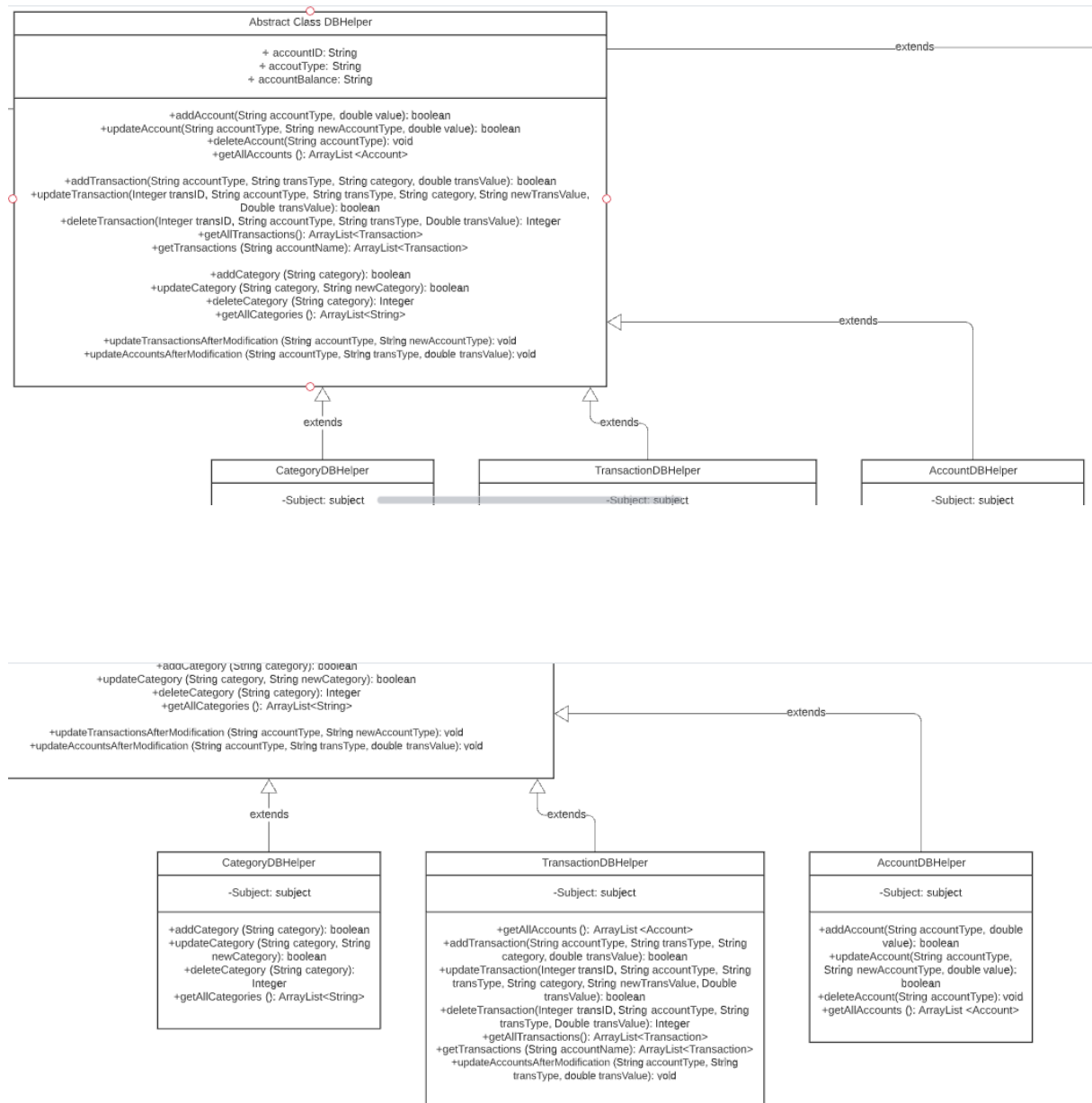
```
String transType = transTypes.getSelectedItem().toString();  
String category = categories.getSelectedItem().toString();  
String value = inputValue.getText().toString();  
saveData(accountName, transType, category, Double.parseDouble(value));  
readDB();
```

1.2.2 Observer Pattern

This kind of pattern is a behavioral pattern and its main purpose is notifying the system or the program for the changes that are made. This is a process between some classes (Observers) and Subjects, which are able to notify the observers, which then execute their update method.

In our project we thought we could implement observer pattern, as a notifier for changes that could/should be made in accounts, categories or transactions. We implemented for Observer -> the DBHelper class, which is extended from the three above mentioned classes also: AccountDBHelper, CategoryDBHelper and TransactionDBHelper. We also have implemented a subject class with a list of observers and we use AccountActivity class for initializing the changes. The method which is implemented in all three classes which extend DBHelper is **updateAccountsAfterModification()**;

With these method occurs the changes which are to be modified.



```
public abstract void updateAccountsAfterModification(String accountType, String transType, Double transValue);
```

Method for updates in DBHelper class. The same method is implemented in AccountDBHelper, CategoryDBHelper and TransactionDBHelper.

```
@Override
public void updateInDB(View view) {
    AlertDialog.Builder alertDialogBuilder = new AlertDialog.Builder(this);

    if(updateID != null) {
        alertDialogBuilder.setTitle("Update Transaction");
        alertDialogBuilder.setMessage("Enter values:");
        alertDialogBuilder.setCancelable(false);
    }
}
```

Method for update in AccountActivity for example updating a transaction by its value.

And the pattern that may find a usage in the next phase:

1.2.3 Strategy Pattern

We thought we could use strategy pattern by the Transaction class, which offers two ways of creating a transaction (income and outcome). The transaction would be the context and the strategy would be operationADD(value1,value2) for income and operationSubstract(value1,value2) for outcome.

```
viewHolder.getTransType().setText(localDataSet.get(position).getTransType());
if(localDataSet.get(position).getTransType().equals("Income")) {
    euro = "+ € ";
} else {
    euro = "- €";
}
```

1.2.4 Iterator Pattern

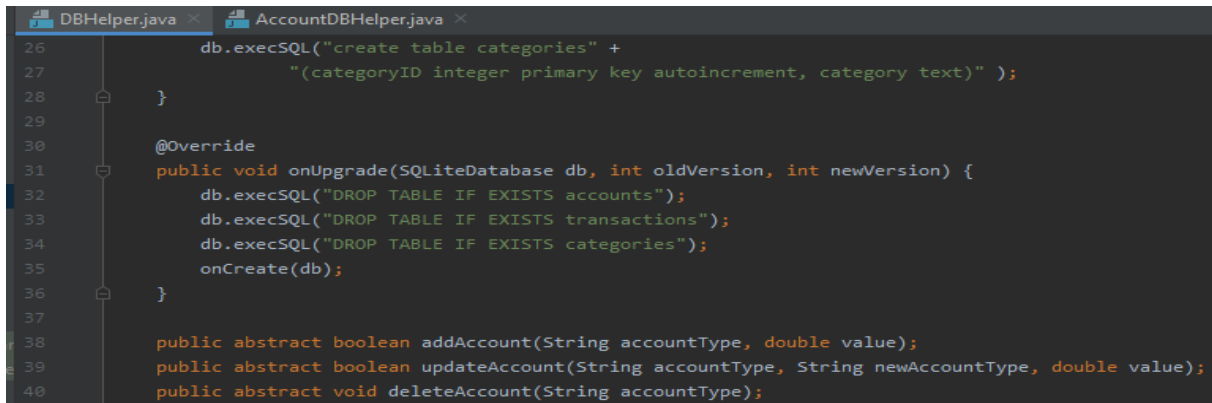
For iterating over the lists that we need in our program, for example: accountList, transactionList etc.

With this pattern we get a way to access the elements of a collection object in sequential manner without any need to know its underlying representation.

```
@Override
public ArrayList<Account> getAllAccounts() {
    ArrayList<Account> arrayList= new ArrayList<>();
    SQLiteDatabase db = this.getReadableDatabase();
}
```

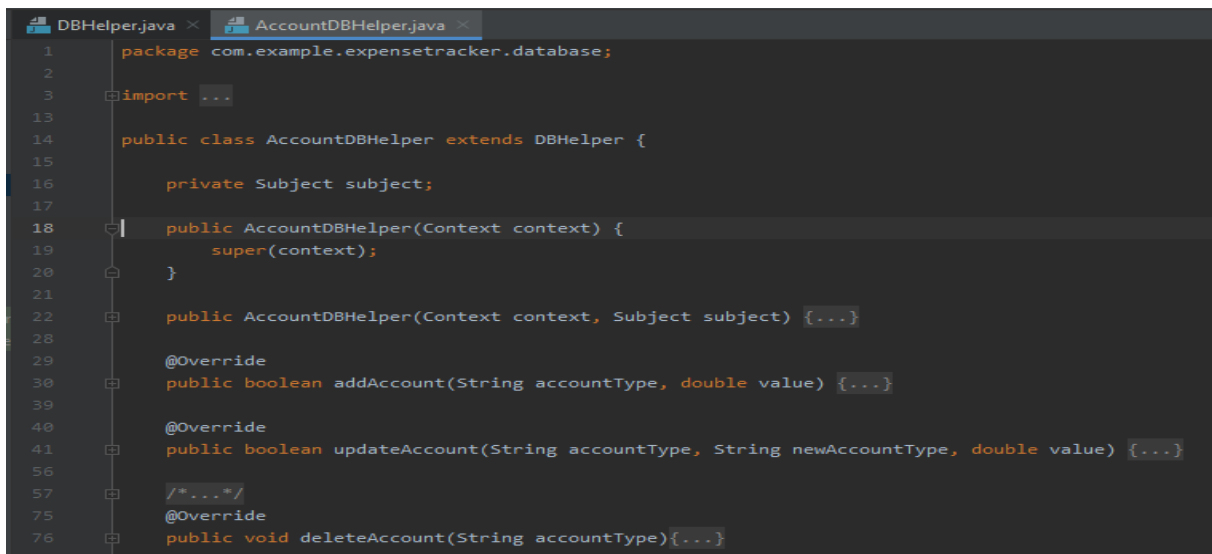
1.2.5 Template Pattern

Using an Abstract class and its defined template(s) to execute its methods. In our programm DBHelper.java and the classes that extend it : AccountDBHelper, CategoryDBHelper and TransactionDBHelper. These Subclasses use the same method as the Abstract Class (DBHelper) and follow a template (like rules how te methodes will be executed in order).



```

26         db.execSQL("create table categories" +
27             "(categoryID integer primary key autoincrement, category text)" );
28     }
29
30     @Override
31     public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
32         db.execSQL("DROP TABLE IF EXISTS accounts");
33         db.execSQL("DROP TABLE IF EXISTS transactions");
34         db.execSQL("DROP TABLE IF EXISTS categories");
35         onCreate(db);
36     }
37
38     public abstract boolean addAccount(String accountType, double value);
39     public abstract boolean updateAccount(String accountType, String newAccountType, double value);
40     public abstract void deleteAccount(String accountType);
  
```



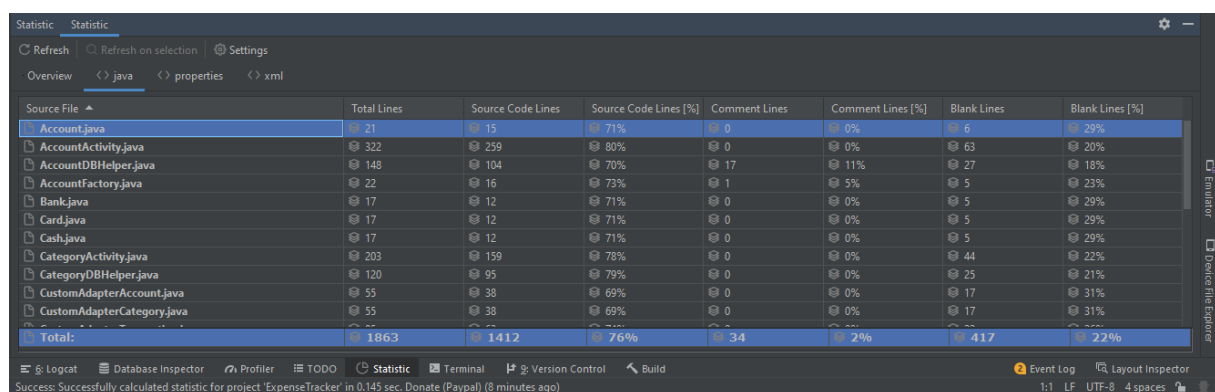
```

1  package com.example.expensetracker.database;
2
3  import ...
13
14  public class AccountDBHelper extends DBHelper {
15
16      private Subject subject;
17
18      public AccountDBHelper(Context context) {
19          super(context);
20      }
21
22      public AccountDBHelper(Context context, Subject subject) {...}
23
24      @Override
25      public boolean addAccount(String accountType, double value) {...}
26
27      @Override
28      public boolean updateAccount(String accountType, String newAccountType, double value) {...}
29
30      /*...*/
31
32      @Override
33      public void deleteAccount(String accountType){...}
34
35  }
  
```

2 Code Metrics

Our implementation includes three packages: model, database and activities. Model has 8 classes and is meant for storing data before it is stored in the database. Database has 5 classes and stores the data that can be called up. And the activities pack contains 11 classes, which more or less represent the functionalities that are called up by the user. So we come to a total of 24 classes.

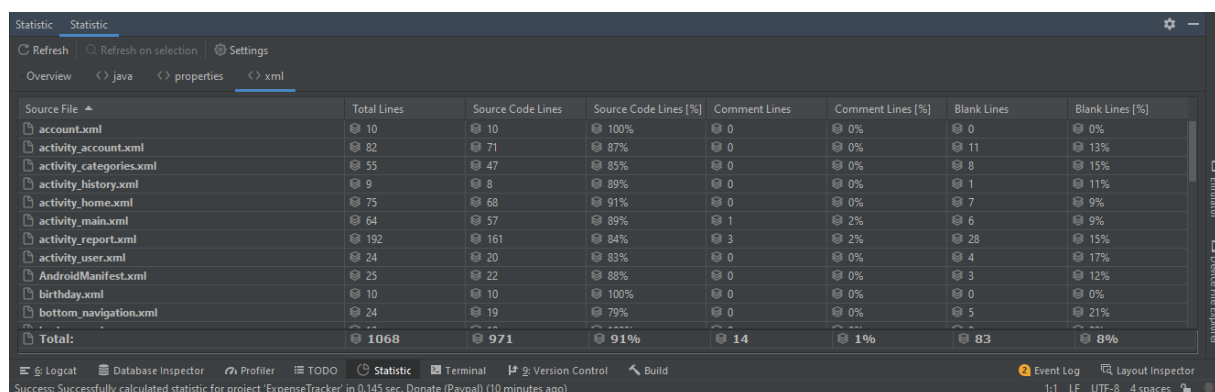
According to the statistic tool used in Android Studio we do have the results for the number of lines in our project. Java classes with total: 1863 lines (1446 without blank lines).



The screenshot shows the 'Statistic' window in Android Studio, displaying a table of code metrics for Java files. The table includes columns for Source File, Total Lines, Source Code Lines, Source Code Lines [%], Comment Lines, Comment Lines [%], Blank Lines, and Blank Lines [%]. The 'Total' row shows 1863 total lines, 1412 source code lines (76%), 34 comment lines (2%), and 417 blank lines (22%).

Source File	Total Lines	Source Code Lines	Source Code Lines [%]	Comment Lines	Comment Lines [%]	Blank Lines	Blank Lines [%]
Account.java	21	15	71%	0	0%	6	29%
AccountActivity.java	322	259	80%	0	0%	63	20%
AccountDBHelper.java	148	104	70%	17	11%	27	18%
AccountFactory.java	22	16	73%	1	5%	5	23%
Bank.java	17	12	71%	0	0%	5	29%
Card.java	17	12	71%	0	0%	5	29%
Cash.java	17	12	71%	0	0%	5	29%
CategoryActivity.java	203	159	78%	0	0%	44	22%
CategoryDBHelper.java	120	95	79%	0	0%	25	21%
CustomAdapterAccount.java	55	38	69%	0	0%	17	31%
CustomAdapterCategory.java	55	38	69%	0	0%	17	31%
Total:	1863	1412	76%	34	2%	417	22%

For the XML: Total lines 1068 (985 without blank lines)

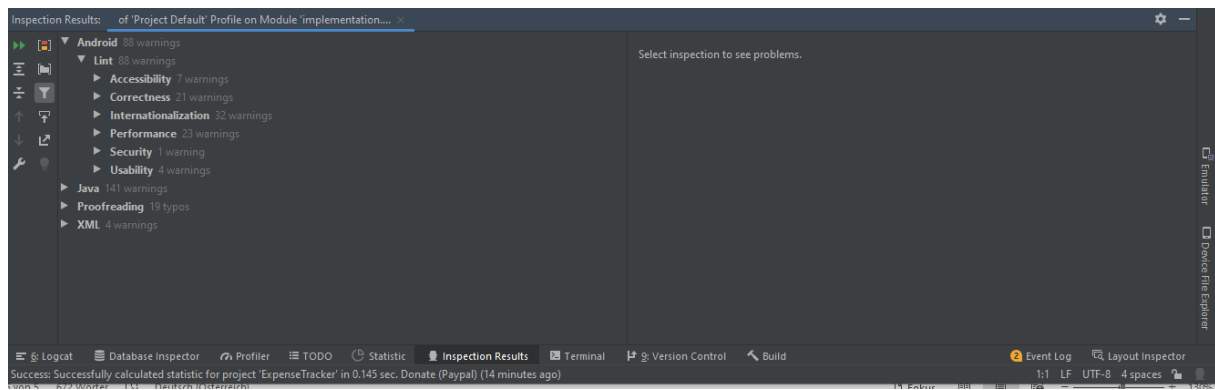


The screenshot shows the 'Statistic' window in Android Studio, displaying a table of code metrics for XML files. The table includes columns for Source File, Total Lines, Source Code Lines, Source Code Lines [%], Comment Lines, Comment Lines [%], Blank Lines, and Blank Lines [%]. The 'Total' row shows 1068 total lines, 971 source code lines (91%), 14 comment lines (1%), and 83 blank lines (8%).

Source File	Total Lines	Source Code Lines	Source Code Lines [%]	Comment Lines	Comment Lines [%]	Blank Lines	Blank Lines [%]
account.xml	10	10	100%	0	0%	0	0%
activity_account.xml	82	71	87%	0	0%	11	13%
activity_categories.xml	55	47	85%	0	0%	8	15%
activity_history.xml	9	8	89%	0	0%	1	11%
activity_home.xml	75	68	91%	0	0%	7	9%
activity_main.xml	64	57	89%	1	2%	6	9%
activity_report.xml	192	161	84%	3	2%	28	15%
activity_user.xml	24	20	83%	0	0%	4	17%
AndroidManifest.xml	25	22	88%	0	0%	3	12%
birthday.xml	10	10	100%	0	0%	0	0%
bottom_navigation.xml	24	19	79%	0	0%	5	21%
Total:	1068	971	91%	14	1%	83	8%

As we can see from the above statistics, our current implementation includes a large number of warnings that do not actually damage the execution of the project, but the quality features. Most of the warnings relate to aspects such as class structure, code style issues, correctness, usability etc.

Statistic from lint:



3 Team Contribution

3.1 Project Tasks and Schedule

Android Expense Tracker

Software Engineering 2
Adhurim Kuçi, Arsen Keshishyan, Benjamin Besic, Felix Mühle

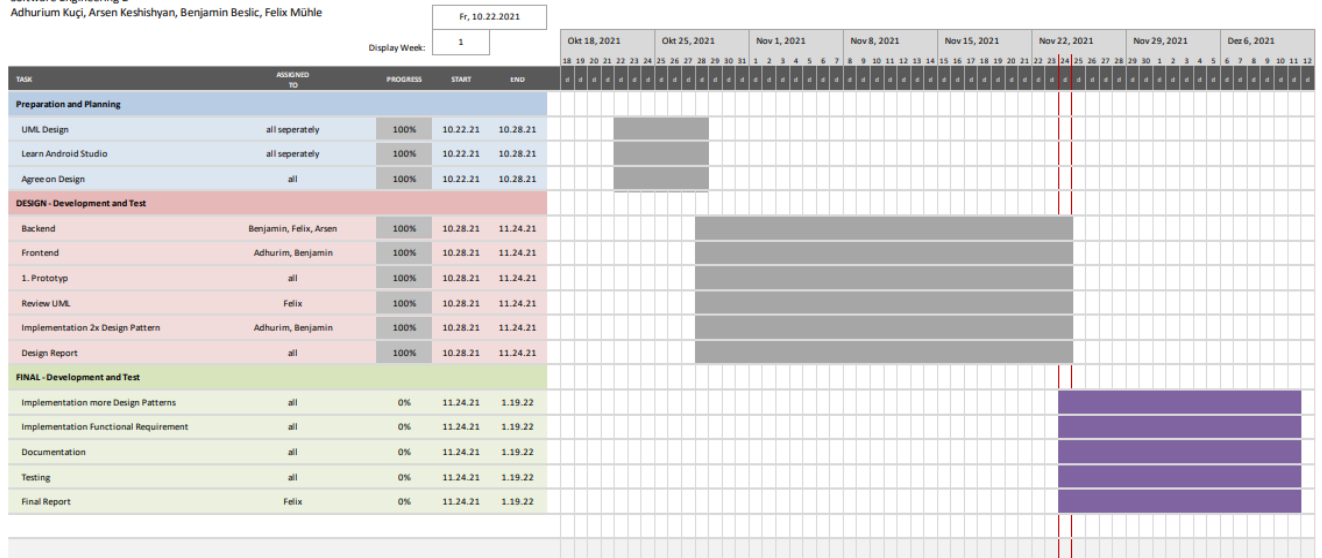


Figure 6: GANTT chart - project plan "Android Expense Tracker"

3.2 Distribution of Work and Efforts

Contribution of Adhurim Kuçi:

- first UML Design
- User Class & Login
- Frontend and Design of Implementation
- Implementation of Factory Pattern
- documentation & report: Factory Pattern, Code Metrics

Contribution of Arsen Keshishyan

- first UML Design
- Transaction Class
- unused version of Account Class
- alternative Implementation of frontend in local branch incl. exception class
- Implementation of Observer Pattern

- Documentation & Report: Design Draft, Observer Pattern

Contribution of Benjamin Beslic:

- first UML Design
- Database Backend with SQLite
- Backend review
- Connection of frontend with backend
- Implementation of Factory Pattern
- documentation & report: Technology Stack, Factory Pattern, other Pattern description

Contribution of Felix Mühle:

- first UML Design, final UML Design (planning phase), based on what we agreed on in discussion and rework of UML based on actual implementation
- unused version of category Class
- Implementation of Observer Pattern
- documentation & report: UML class diagrams, Observer Pattern, Team Contribution (GANTT, contribution documentation)