Software Engineering 2
DEAD REPORT

Deadline Reports

Team number:	4

Team member 1	
Name:	Nemanja Srdanovic
Student ID:	01576891
E-mail address:	a01576891@unet.univie.ac.at

Team member 2	
Name:	Jelena Mikic
Student ID:	01329687
E-mail address:	a01329687@unet.univie.ac.at

Team member 3	
Name:	Veljko Radunovic
Student ID:	01329687
E-mail address:	a01329687@unet.univie.ac.at

Team member 4	
Name:	Marija Zivanovic
Student ID:	01549452
E-mail address:	a01549452@unet.univie.ac.at

1 Final Design

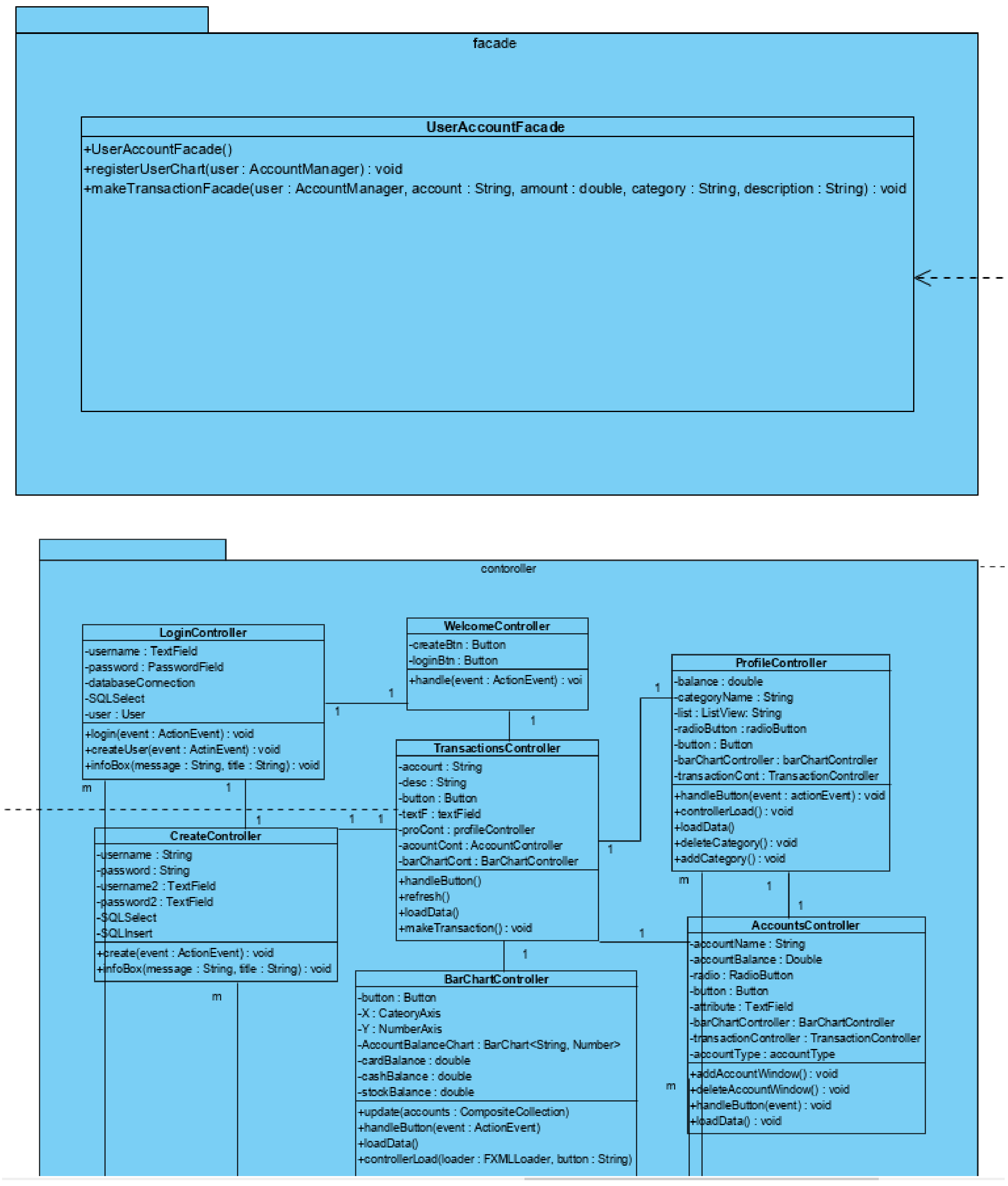
1.1 Design Approach and Overview

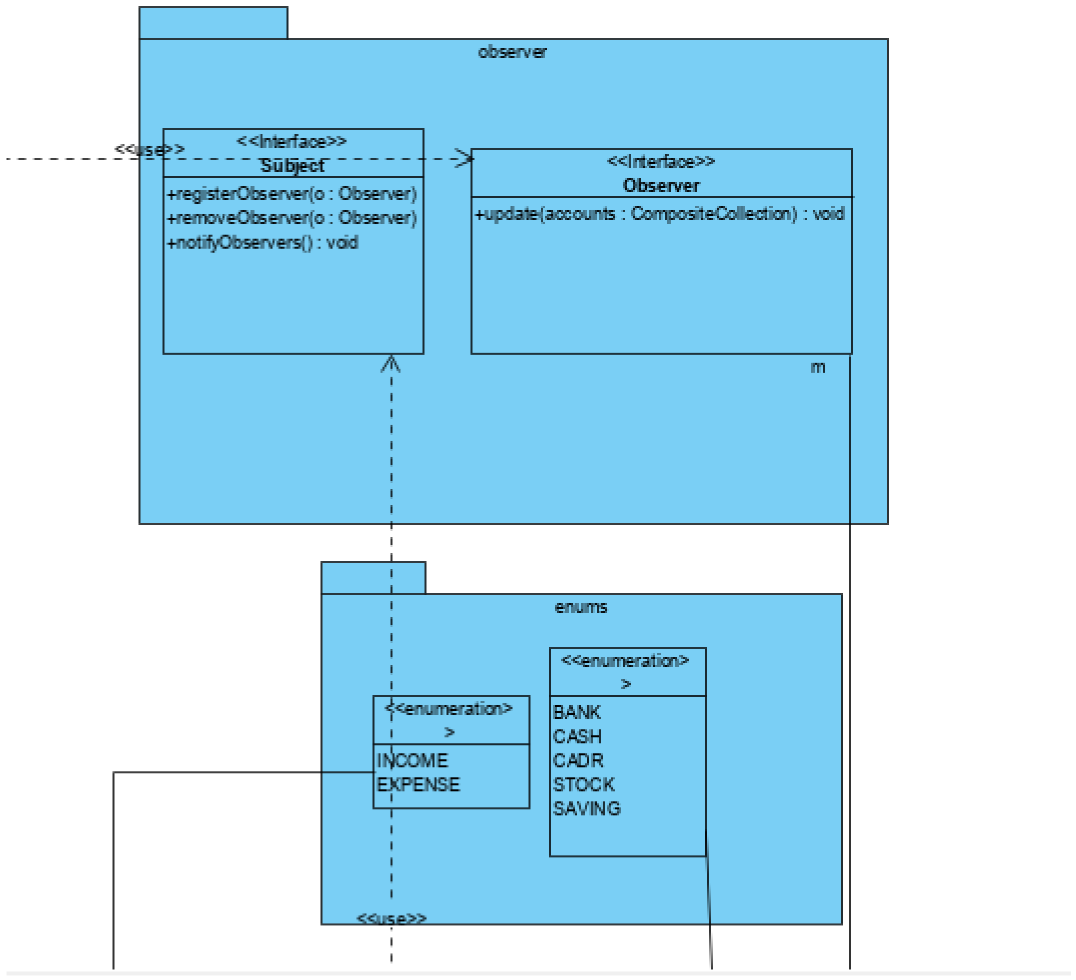
Our design approach is shown on the following Class diagrams.

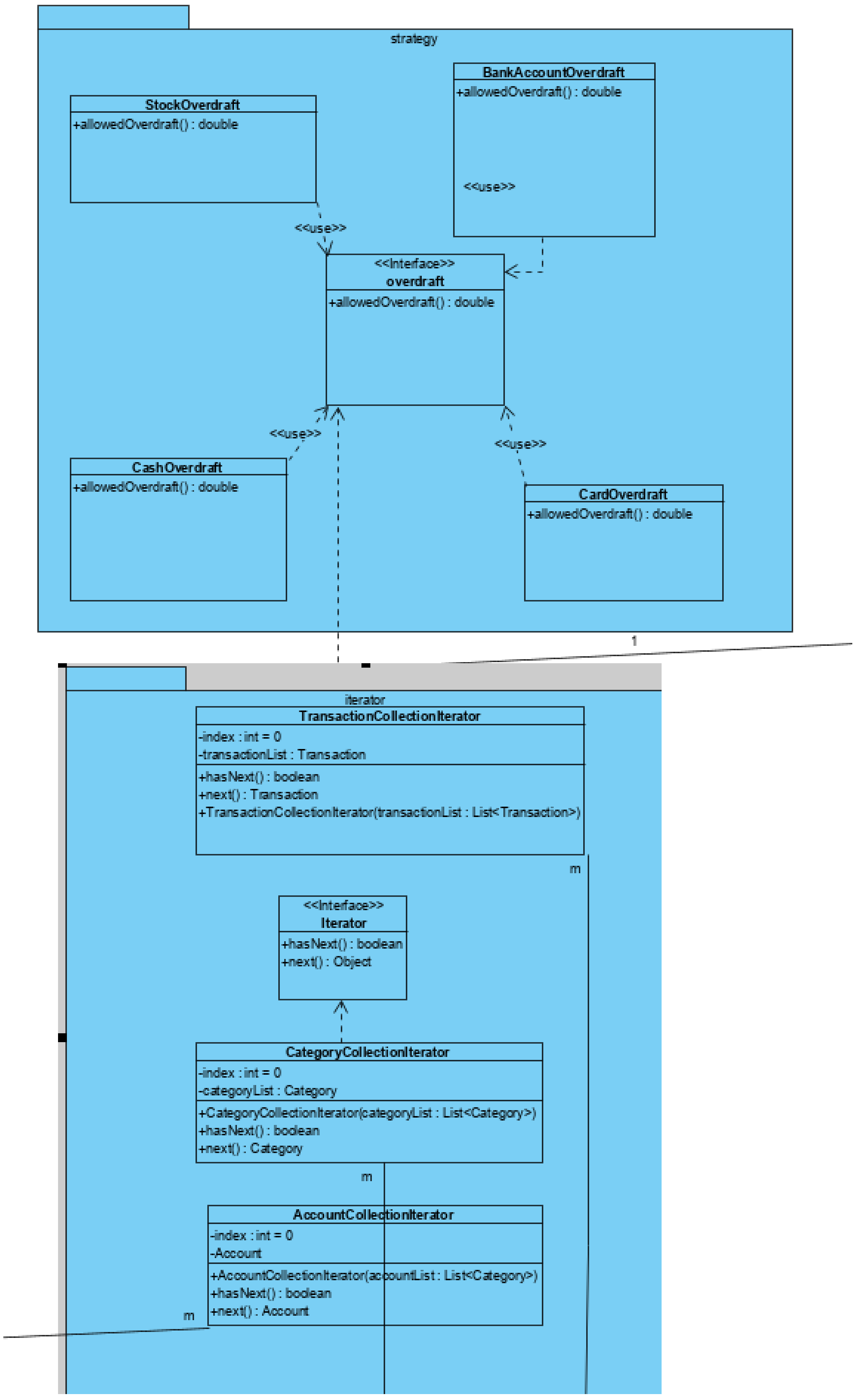
We started discussing our project when we met at University. We have studied similar applications in order to get an idea of where to start with our project. There were a lot of suggestions on how the application should look, we discussed the design and all the features that the project will contain.

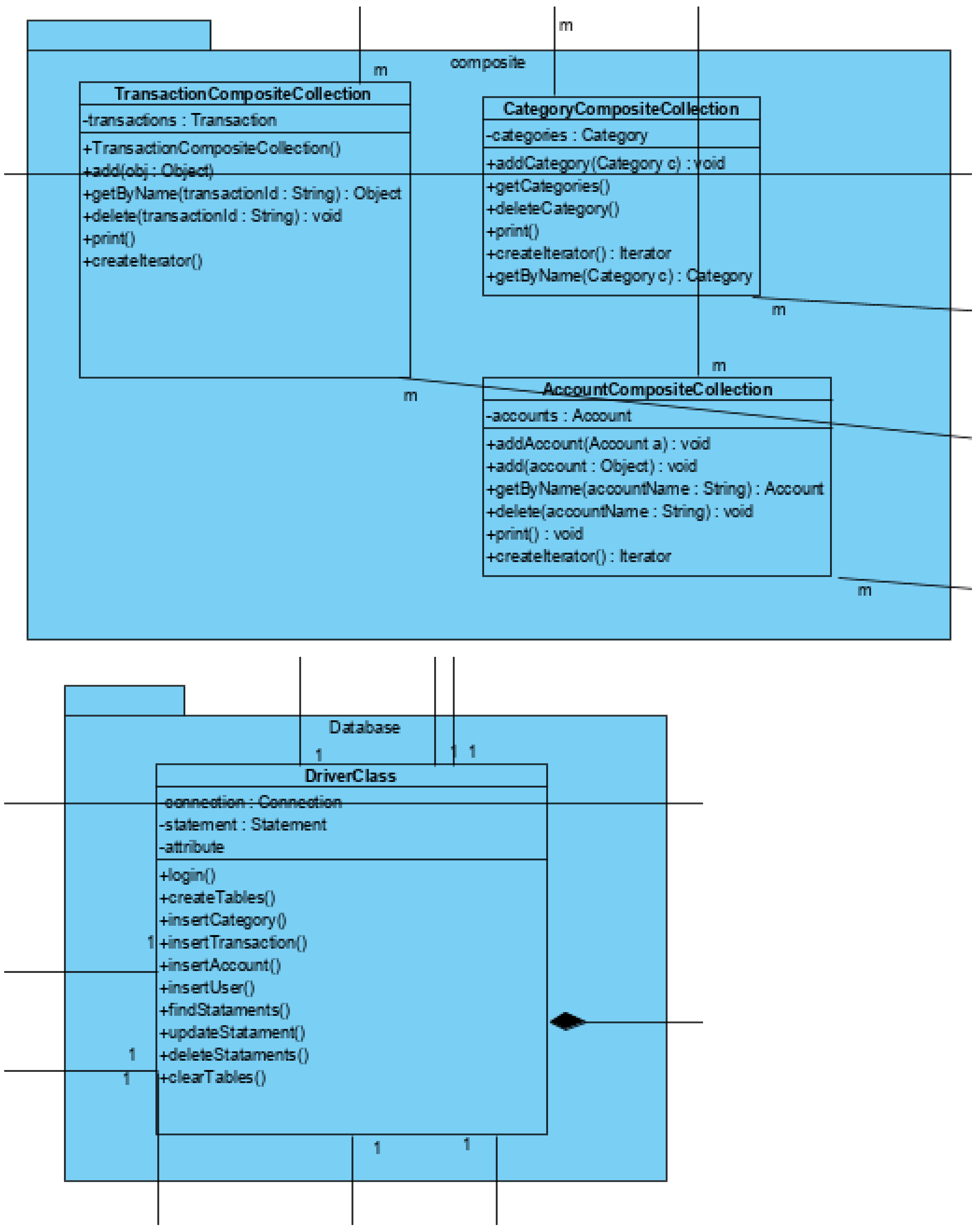
We started drawing on our whiteboard our project. We have sketched our program in order to make it easier for us to plan our program and we agreed to meet the basic needs that an application should contain, we did not want to complicate the development of our application, so we have agreed to create a good simple basis, which we will elaborate in the further stages of the project. From there, we started building the base of our application, the skeleton we uploaded to git so that each of the group members could do their job in the development of the project.

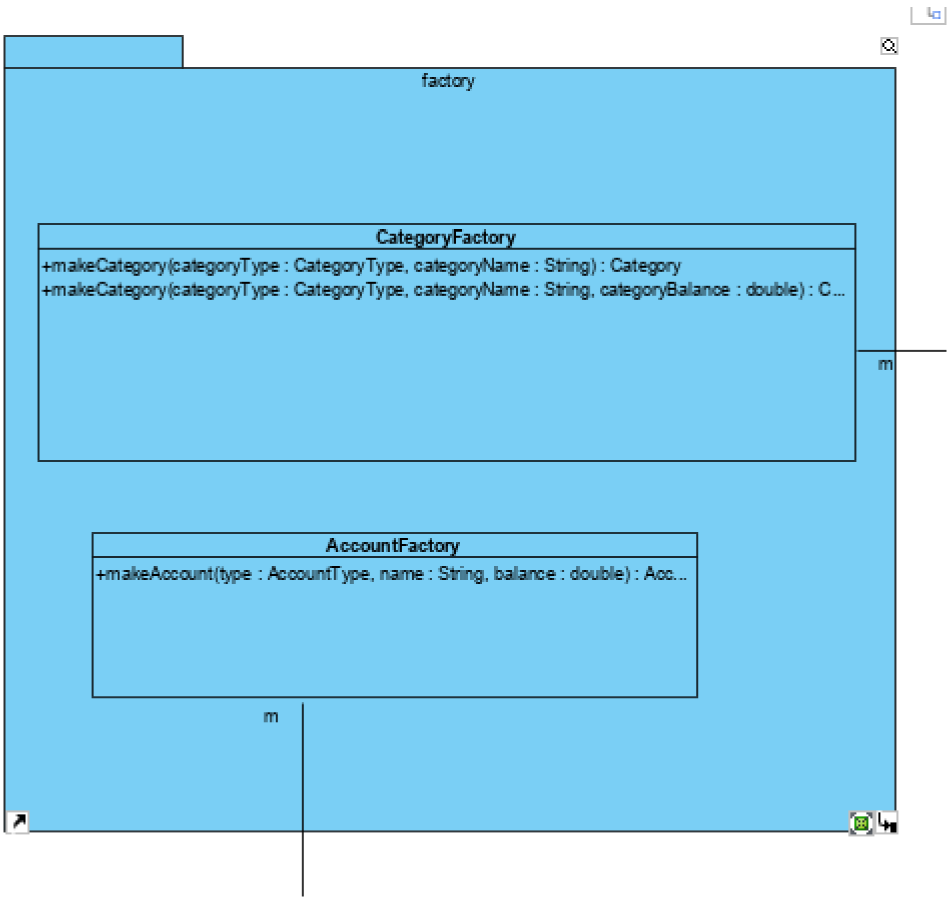
1.1.1 Class Diagrams

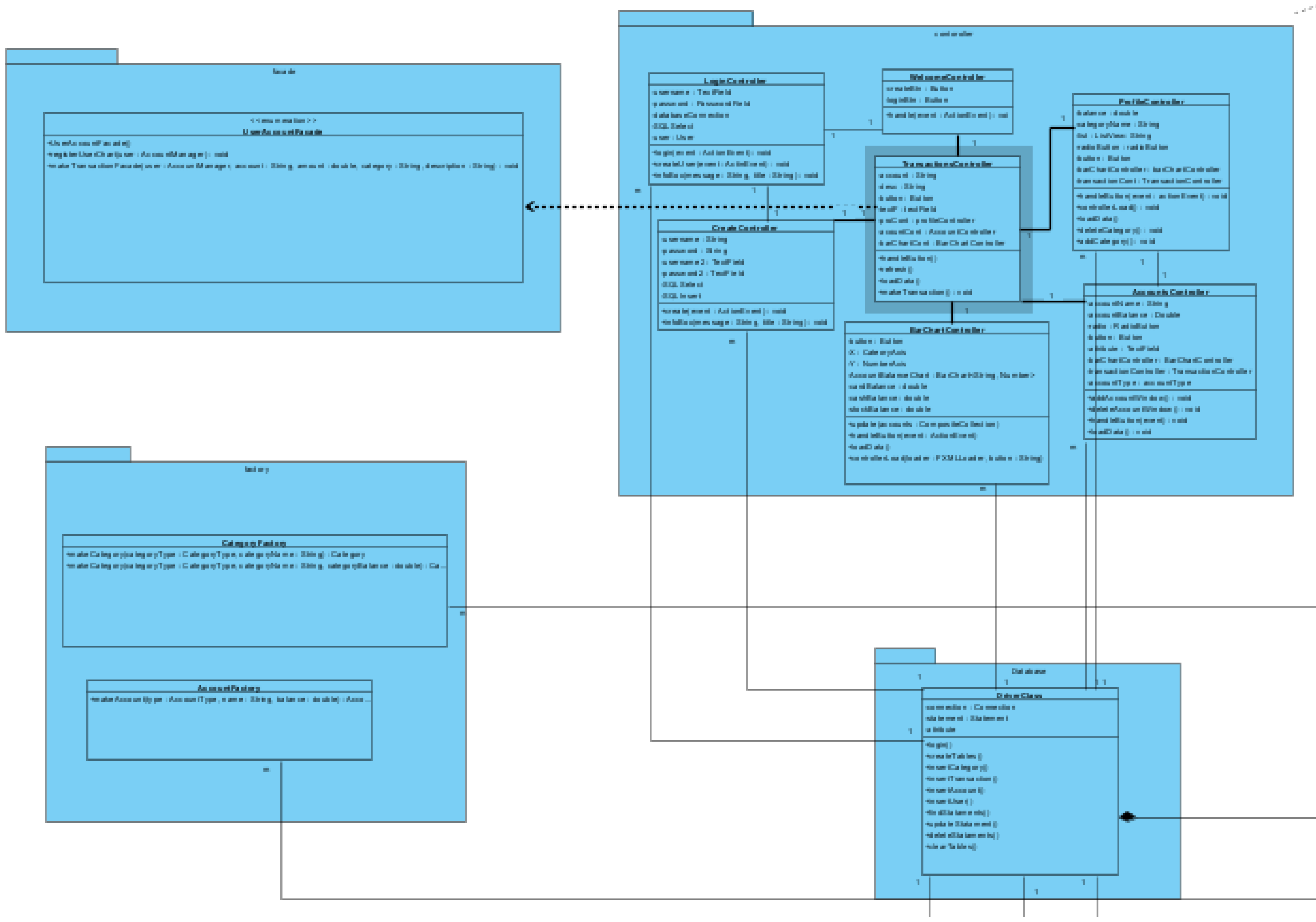

facade
UserAccountFacade
+UserAccountFacade()
+registerUserChart(user : AccountManager) : void
+makeTransactionFacade(user : AccountManager, account : String, amount : double, category : String, description : String) : void
contoroller
LoginController
-username : TextField
-password : PasswordField
-databaseConnection
-SQLSelect
-user : User
+login(event : ActionEvent) : void
+createUser(event : ActinEvent) : void
+infoBox(message : String, title : String) : void
WelcomeController
-createBtn : Button
-loginBtn : Button
+handle(event : ActionEvent) : voi
ProfileController
-balance : double
-categoryName : String
-list : ListView: String
-radioButton : radioButton
-button : Button
-barChartController : barChartController
-transactionCont : TransactionController
+handleButton(event : actionEvent) : void
+controllerLoad() : void
+loadData()
+deleteCategory() : void
+addCategory() : void
TransactionsController
-account : String
-desc : String
-button : Button
-textF : textField
-proCont : profileController
-acountCont : AccountController
-barChartCont : BarChartController
+handleButton()
+refresh()
+loadData()
+makeTransaction() : void
CreateController
-username : String
-password : String
-username2 : TextField
-password2 : TextField
-SQLSelect
-SQLInsert
+create(event : ActionEvent) : void
+infoBox(message : String, title : String) : void
AccountsController
-accountName : String
-accountBalance : Double
-radio : RadioButton
-button : Button
-attribute : TextField
-barChartController : BarChartController
-transactionController : TransactionController
-accountType : accountType
+addAccountWindow() : void
+deleteAccountWindow() : void
+handleButton(event) : void
+loadData() : void
BarChartController
-button : Button
-X : CateoryAxis
-Y : NumberAxis
-AccountBalanceChart : BarChart<String, Number>
-cardBalance : double
-cashBalance : double
-stockBalance : double
+update(accounts : CompositeCollection)
+handleButton(event : ActionEvent)
+loadData()
+controllerLoad(loader : FXMLLoader, button : String)
1
1
1
1
1
m
1
1
1
1
1
1
m
1
1
1
m
m


observer
<<use>>
<<Interface>>
Subject
+registerObserver(o : Observer)
+removeObserver(o : Observer)
+notifyObservers() : void
<<Interface>>
Observer
+update(accounts : CompositeCollection) : void
m
enums
<<enumeration>>
INCOME
EXPENSE
<<enumeration>>
BANK
CASH
CADR
STOCK
SAVING
<<use>>

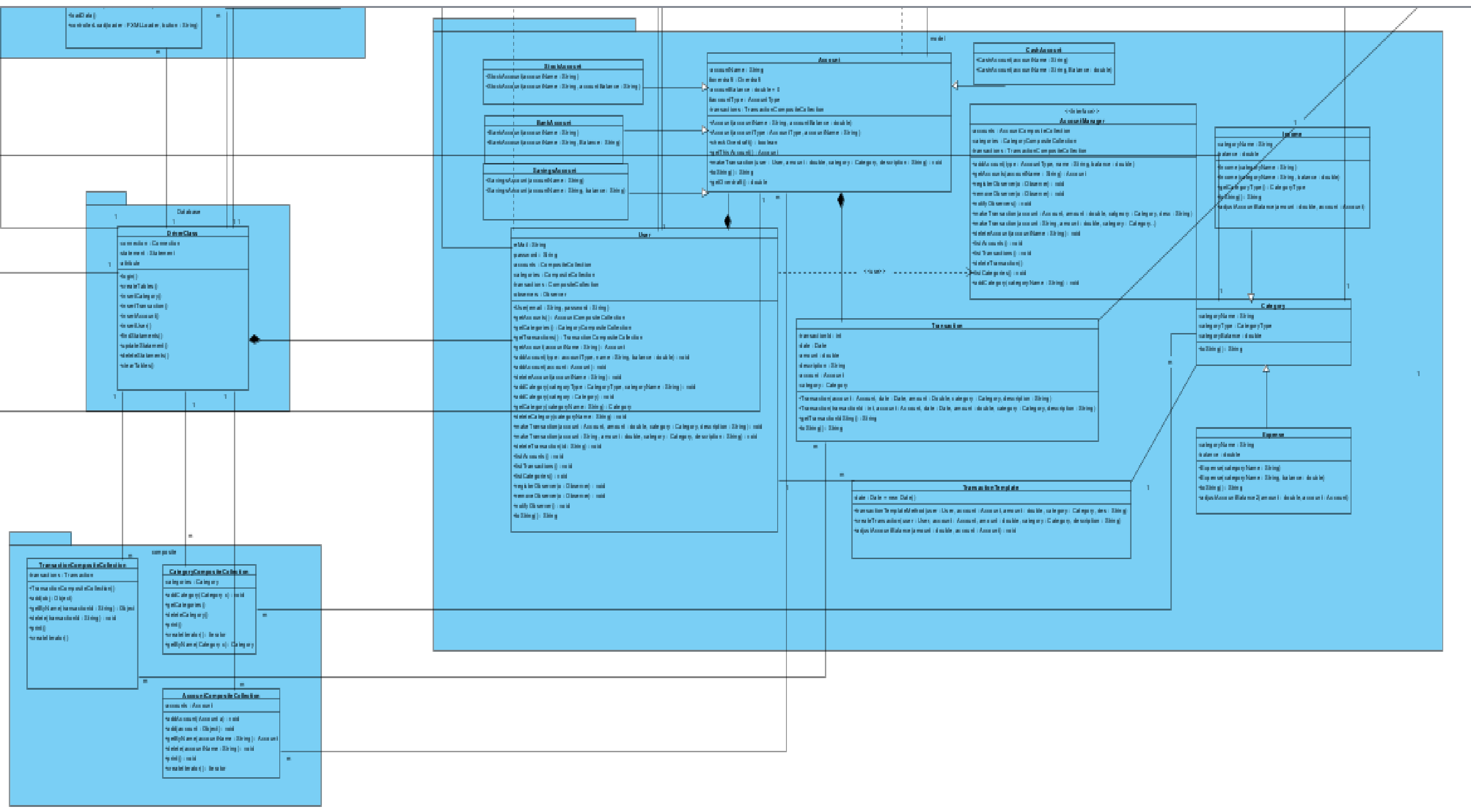

strategy
BankAccountOverdraft
+allowedOverdraft() : double
<<use>>
StockOverdraft
+allowedOverdraft() : double
<<use>>
<<Interface>>
overdraft
+allowedOverdraft() : double
<<use>>
<<use>>
CashOverdraft
+allowedOverdraft() : double
CardOverdraft
+allowedOverdraft() : double
1
iterator
TransactionCollectionIterator
-index : int = 0
-transactionList : Transaction
+hasNext() : boolean
+next() : Transaction
+TransactionCollectionIterator(transactionList : List<Transaction>)
m
<<Interface>>
Iterator
+hasNext() : boolean
+next() : Object
CategoryCollectionIterator
-index : int = 0
-categoryList : Category
+CategoryCollectionIterator(categoryList : List<Category>)
+hasNext() : boolean
+next() : Category
m
AccountCollectionIterator
-index : int = 0
-Account
+AccountCollectionIterator(accountList : List<Category>)
+hasNext() : boolean
+next() : Account
m


composite
TransactionCompositeCollection
-transactions : Transaction
+TransactionCompositeCollection()
+add(obj : Object)
+getByName(transactionId : String) : Object
+delete(transactionId : String) : void
+print()
+createIterator()
CategoryCompositeCollection
-categories : Category
+addCategory(Category c) : void
+getCategories()
+deleteCategory()
+print()
+createIterator() : Iterator
+getByName(Category c) : Category
AccountCompositeCollection
-accounts : Account
+addAccount(Account a) : void
+add(account : Object) : void
+getByName(accountName : String) : Account
+delete(accountName : String) : void
+print() : void
+createIterator() : Iterator
m
m
m
m
m
m
Database
DriverClass
-connection : Connection
-statement : Statement
-attribute
+login()
+createTables()
+insertCategory()
+insertTransaction()
+insertAccount()
+insertUser()
+findStataments()
+updateStatament()
+deleteStataments()
+clearTables()
1
1 1
1
1
1
1
1

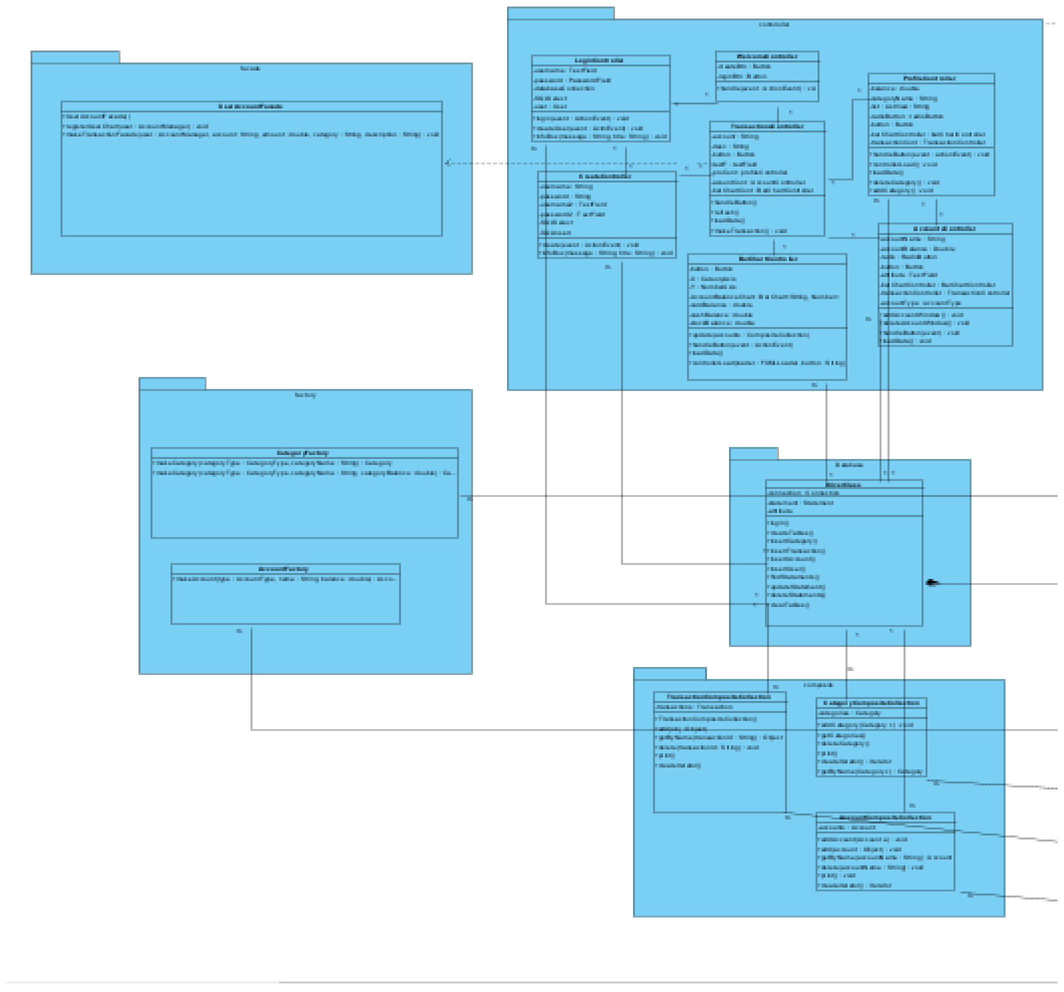

factory
CategoryFactory
+makeCategory(categoryType : CategoryType, categoryName : String) : Category
+makeCategory(categoryType : CategoryType, categoryName : String, categoryBalance : double) : C...
m
AccountFactory
+makeAccount(type : AccountType, name : String, balance : double) : Acc...
m



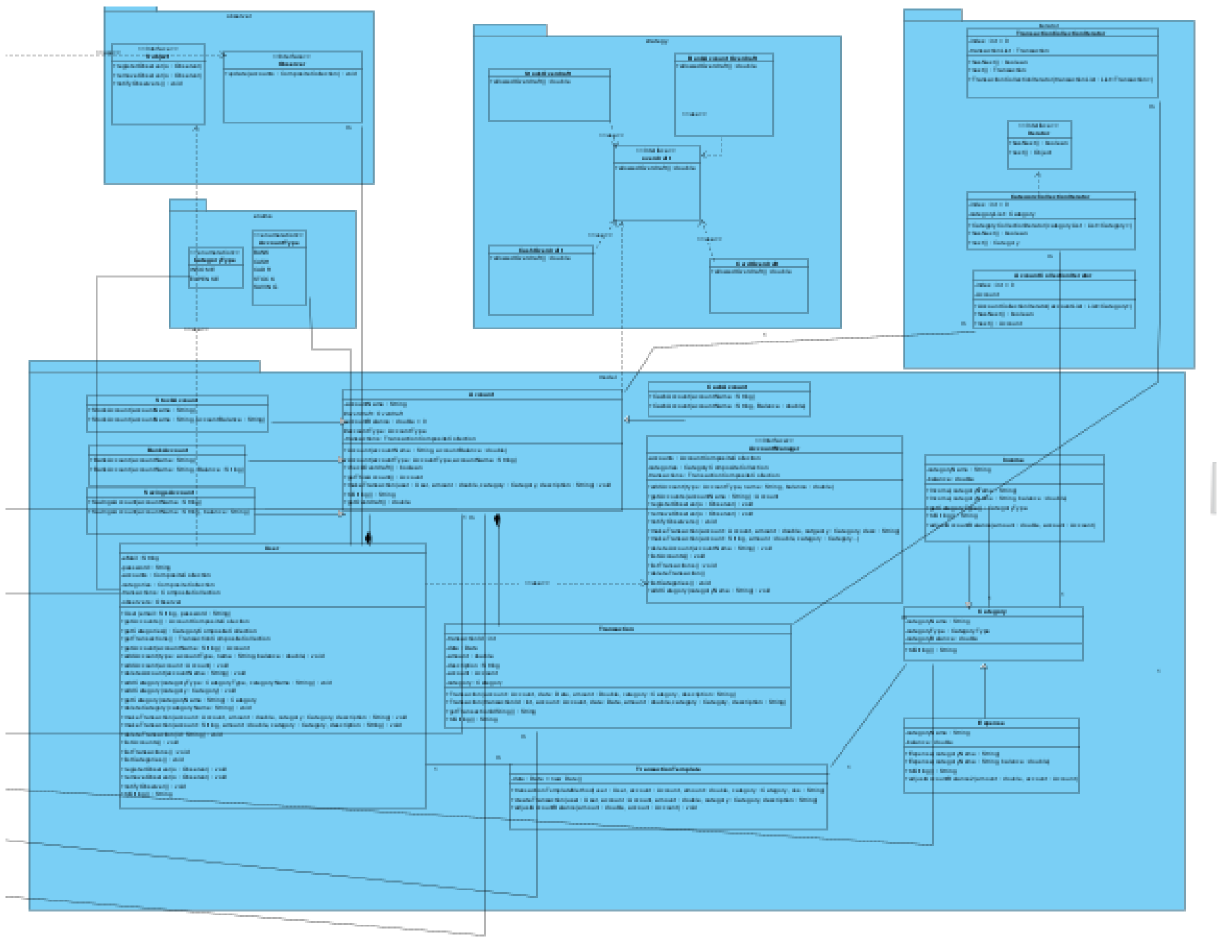
facade
<<enumeration>>
UserAccountFacade
controller
LoginController
WelcomeController
ProfileController
TransactionsController
CreateController
AccountsController
BarChartController
factory
CategoryFactory
AccountFactory
Database
DriverClass


model
StockAccount
Account
CashAccount
BankAccount
SavingsAccount
<<interface>>
AccountManager
Income
Database
DriverClass
User
Transaction
Category
Expense
TransactionTemplate
composite
TransactionCompositeCollection
CategoryCompositeCollection
AccountCompositeCollection

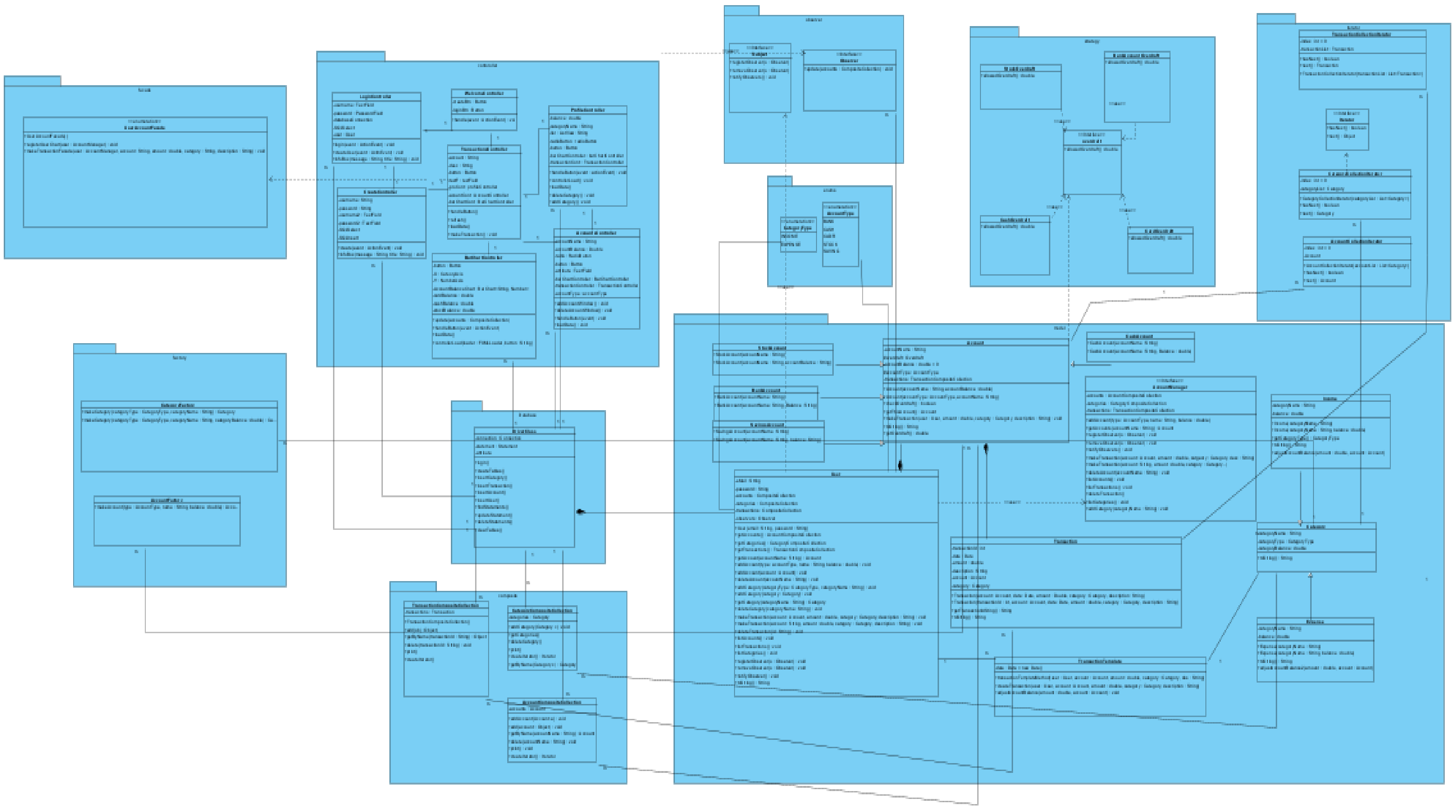
This is a left side of the complete diagram:



This is a right side of the complete diagram:



This is a complete UML-Class Diagram



1.1.2 Technology Stack

From the technology stack, for our development environment we are going to use the latest InteliJ IDE for Java 1.8.[1]

User Interface:

As we have planned in the previous phase, we have used JavaFX for our User Interface. We did not have any major problems using this technology, as it was perfectly supported from Eclipse and Intellij IDE we have used. Screen Builder was a great advantage and we discovered it's good sides specially by creating bar charts. Generally, we did not have problems with User GUI itself, but we had some problems connecting all application, with database and backend. All in all, we will use this technology again.

Database:

As one of the major changes compared to SUPD is our database, as we switched at the end from the H2 Database to the SQLite Database. According to H2 Database, we had problems installing and running it on different devices, because of some internal H2 database errors that exists in the latest version. Namely, the version we tried to use, had some security setup that disallowed us creating new databases in a web browser. That is why we could test it only at one machine, what was inefficient. We switched then to SQLite, what worked very fine, because we have also included our testDB.db file in our implementation folder. In that way, out database file was always directly up to date. We used DB Browser for SQLite to check our database tables. This was very practical and solved all problems we had with H2 Webserver.

1.2 Major Changes Compared to SUPD

From the major changes compared to SUPD, as we have already discussed, it was a database. We have switched from H2 to SQLite Database, because we had problems with H2 Webserver that did not allowed us to create new databases. We solved our problem by using SQLite database file directly in out implementation folder.

According to our SUPD Feedback, our documentation was completed, with one with one disadvantage related to our UML Diagram. We created new UML Diagram as well and corrected all cardinalities that failed last time.

[1] https://www.eclipse.org/downloads/packages/release/luna/sr2/eclipse-ide-java-developers

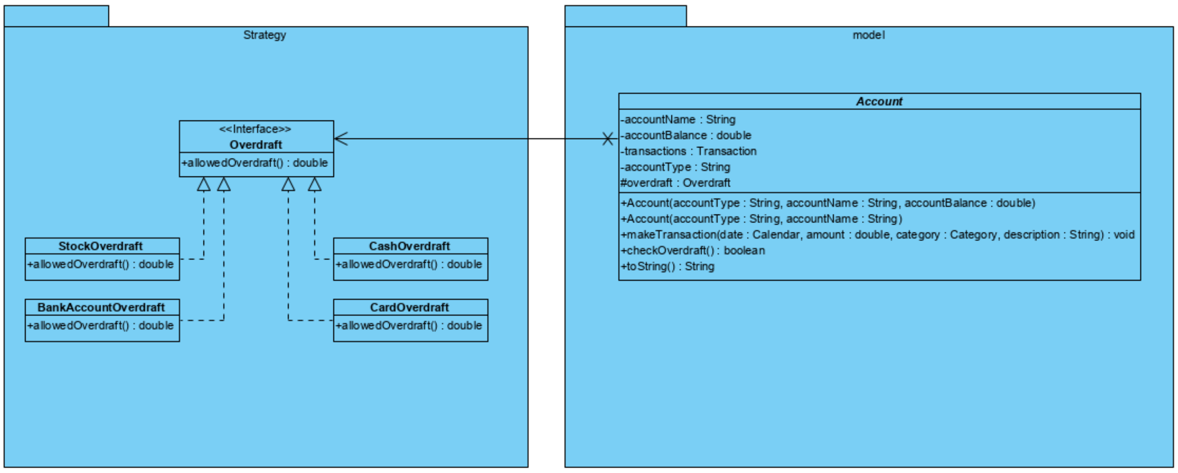
1.3 Design Patterns

1.3.1 Strategy Design Pattern

The Strategy pattern is used to abstract out behaviours that are different and vary among subclasses and make it interchangeable. In that way, clients that are using those behaviours needn't know about specifics of implementation, they are just using it. And because those behaviours aren't tied up to the superclass or subclasses, they can change without affecting them.

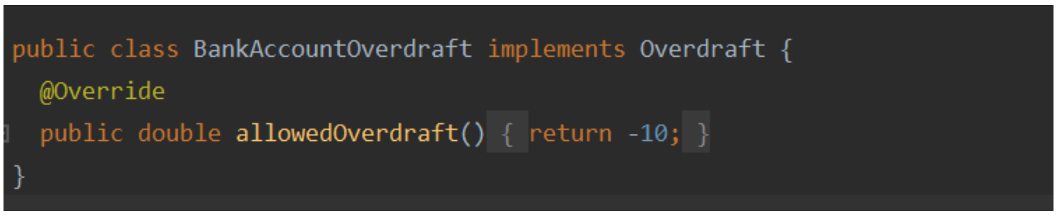
- Our strategy pattern is used as an "overdraft" for our accounts. All four types of accounts have different allowed overdrafts.
- This pattern we implemented through the interface "Overdraft" and four different classes that are implementing the Overdraft interface for each of our account types. Those classes return through double interface method different allowed overdraft.
- In our abstract Account class, we implemented a protected overdraft variable and two methods, setter for overdraft and boolean method to check if the current account balance is smaller than an allowed overdraft for that account. The overdraft variable becomes a value in constructors of classes that extend the account, different values for each account type.

Fig.3. Class Diagram: Strategy Pattern


Strategy
<<Interface>>
Overdraft
+allowedOverdraft() : double
StockOverdraft
+allowedOverdraft() : double
BankAccountOverdraft
+allowedOverdraft() : double
CashOverdraft
+allowedOverdraft() : double
CardOverdraft
+allowedOverdraft() : double
model
Account
-accountName : String
-accountBalance : double
-transactions : Transaction
-accountType : String
#overdraft : Overdraft
+Account(accountType : String, accountName : String, accountBalance : double)
+Account(accountType : String, accountName : String)
+makeTransaction(date : Calendar, amount : double, category : Category, description : String) : void
+checkOverdraft() : boolean
+toString() : String

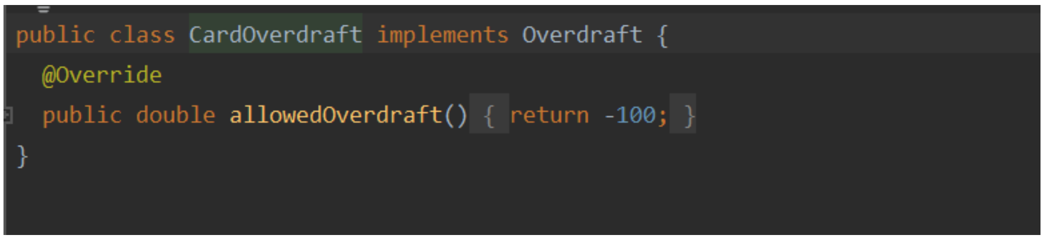
```
    double allowedOverdraft();
}
```

Definition of interface Overdraft



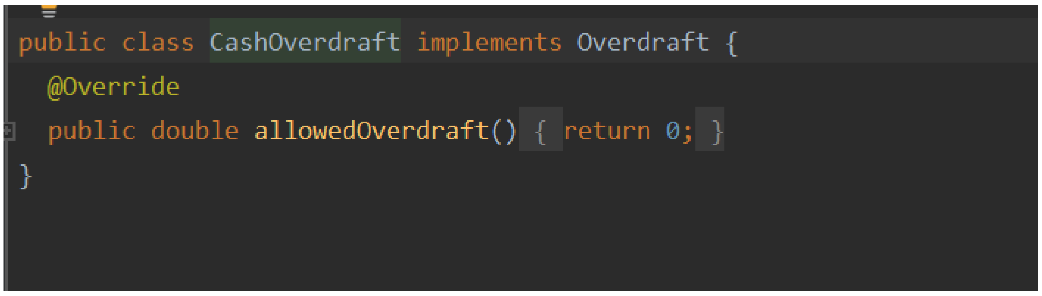
```java
public class BankAccountOverdraft implements Overdraft {
  @Override
  public double allowedOverdraft() { return -10; }
}
```

Definition of class BankAccountOverdraft



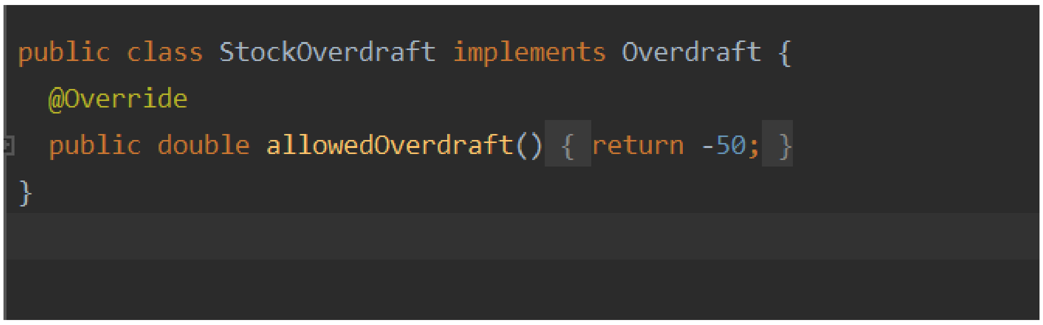
```java
public class CardOverdraft implements Overdraft {
  @Override
  public double allowedOverdraft() { return -100; }
}
```

Definition of class CardOverdraft



```java
public class CashOverdraft implements Overdraft {
  @Override
  public double allowedOverdraft() { return 0; }
}
```

Definition of class CashOverdraft



```java
public class StockOverdraft implements Overdraft {
  @Override
  public double allowedOverdraft() { return -50; }
}
```

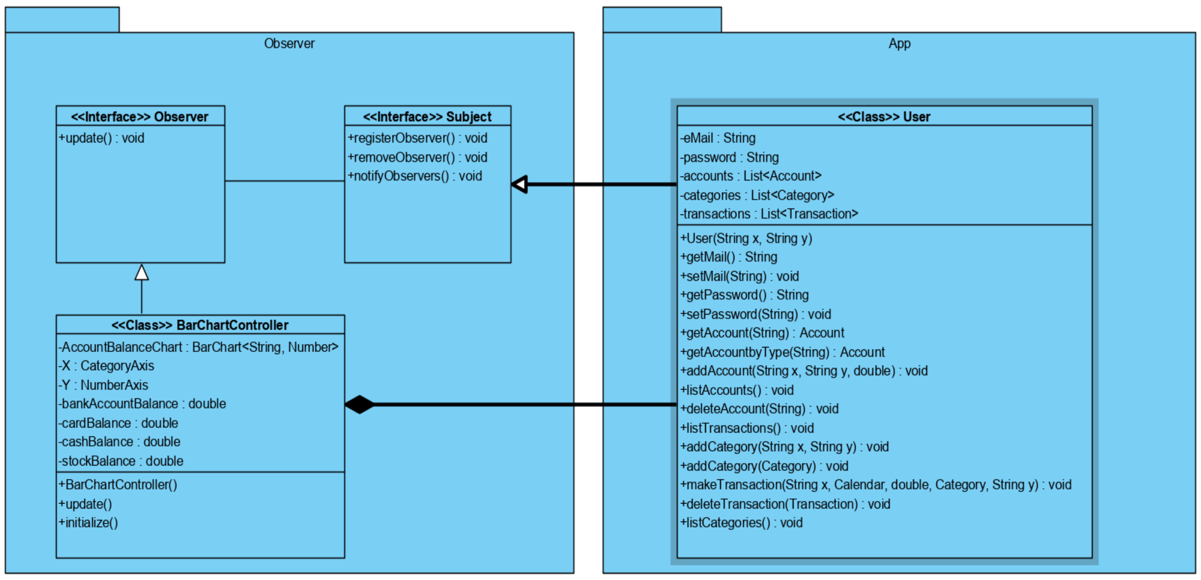
Definition of class StockOverdraft

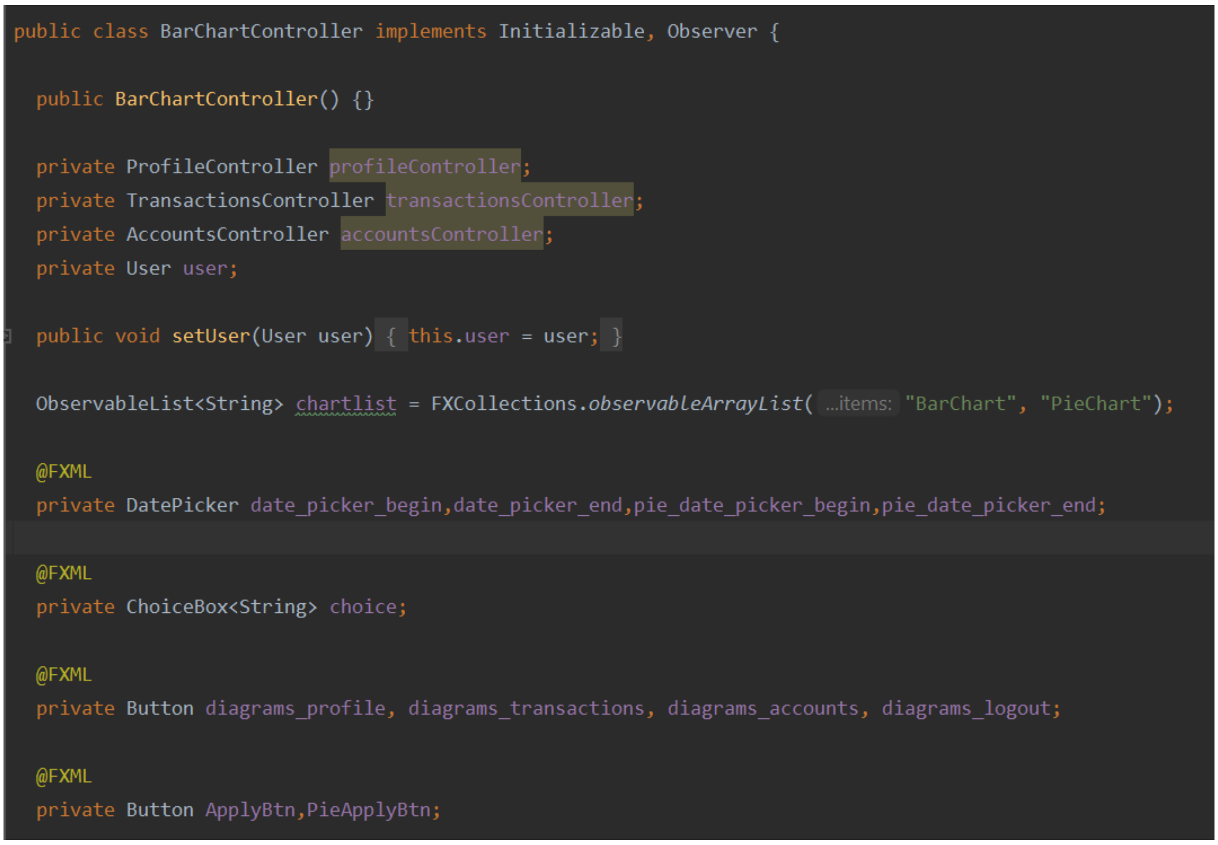
1.3.2 Observer Design Pattern

The observer pattern represents a one-to-many dependency relationship between the observable object, i.e. subject that contains some data and observers who are somewhat dependent on that subject’s data and are automatically notified when that data changes. In some cases, dependent objects can then also be updated accordingly. Benefit of this pattern is that subject and observers can be very different objects without any special knowledge about each other, but they can still interact.

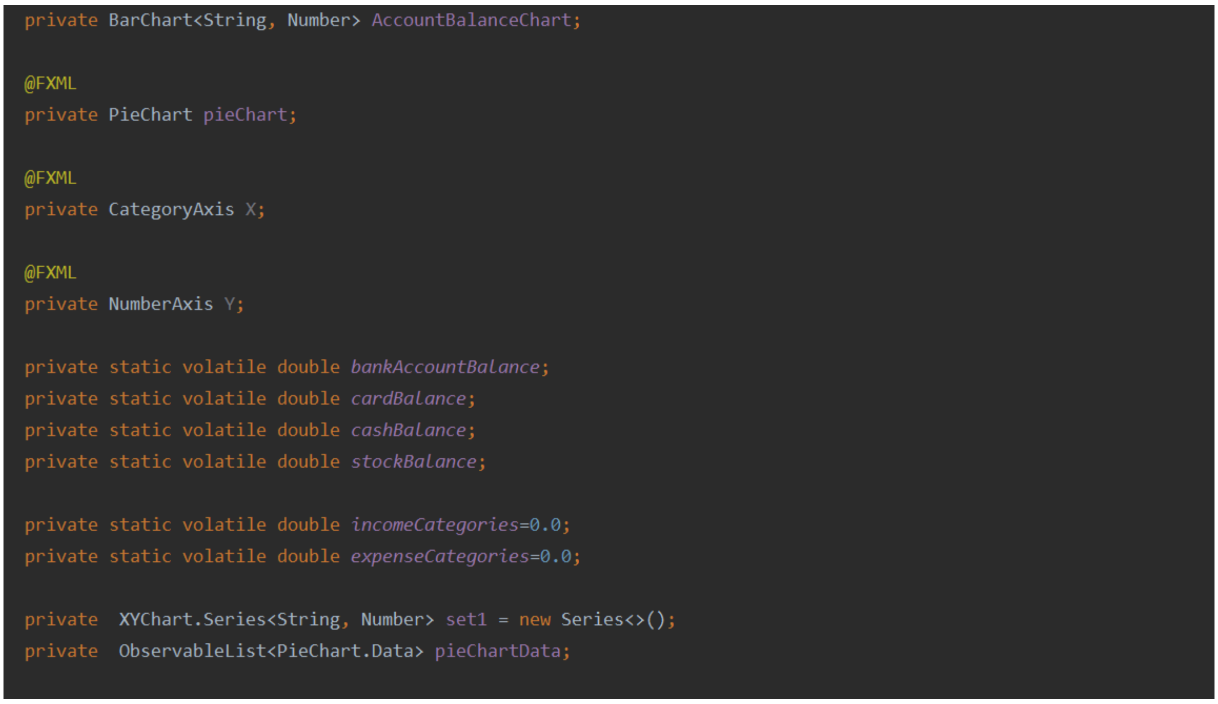
- We decided to use observer pattern instance in order to graphically display current balance for different types of accounts, as these and other similar reports are part of functional requirements.
- Class User is our concrete implementation of Subject interface and Class BarChartController is registered as Observer. They should interact in that way that whenever user makes transaction on some account or deletes it, or adds a new account, the BarChartController object will get notified and updated so the user can get graphical representation of current balance.

Fig.4. Class Diagram: Observer Pattern


Observer
<<Interface>> Observer
+update() : void
<<Interface>> Subject
+registerObserver() : void
+removeObserver() : void
+notifyObservers() : void
<<Class>> BarChartController
-AccountBalanceChart : BarChart<String, Number>
-X : CategoryAxis
-Y : NumberAxis
-bankAccountBalance : double
-cardBalance : double
-cashBalance : double
-stockBalance : double
+BarChartController()
+update()
+initialize()
App
<<Class>> User
-eMail : String
-password : String
-accounts : List<Account>
-categories : List<Category>
-transactions : List<Transaction>
+User(String x, String y)
+getMail() : String
+setMail(String) : void
+getPassword() : String
+setPassword(String) : void
+getAccount(String) : Account
+getAccountbyType(String) : Account
+addAccount(String x, String y, double) : void
+listAccounts() : void
+deleteAccount(String) : void
+listTransactions() : void
+addCategory(String x, String y) : void
+addCategory(Category) : void
+makeTransaction(String x, Calendar, double, Category, String y) : void
+deleteTransaction(Transaction) : void
+listCategories() : void

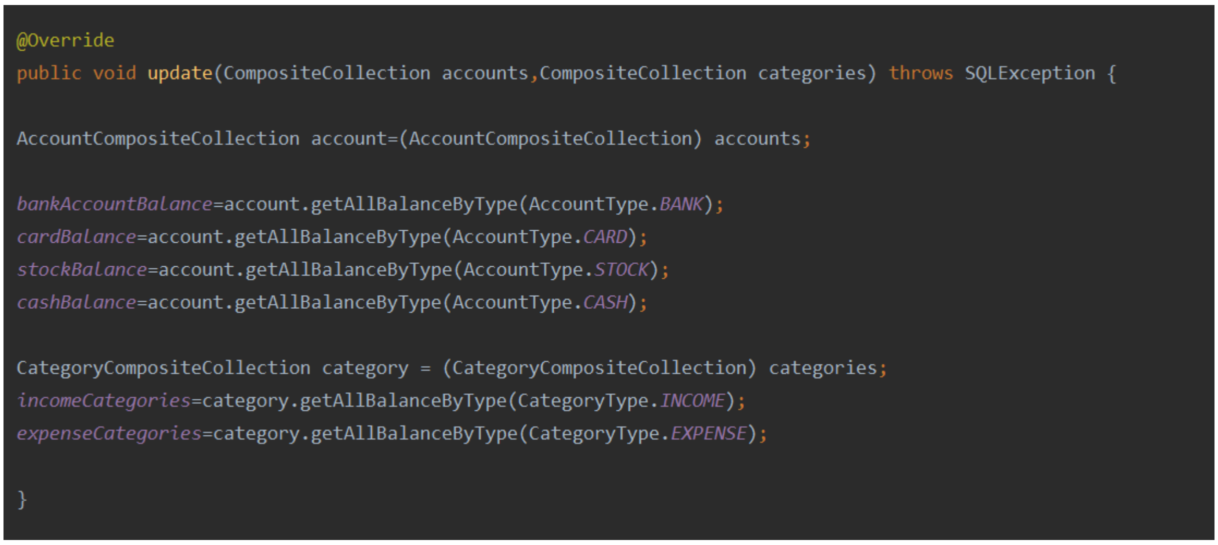


```
public class BarChartController implements Initializable, Observer {

  public BarChartController() {}

  private ProfileController profileController;
  private TransactionsController transactionsController;
  private AccountsController accountsController;
  private User user;

  public void setUser(User user) { this.user = user; }

  ObservableList<String> chartlist = FXCollections.observableArrayList( ...items: "BarChart", "PieChart");

  @FXML
  private DatePicker date_picker_begin,date_picker_end,pie_date_picker_begin,pie_date_picker_end;

  @FXML
  private ChoiceBox<String> choice;

  @FXML
  private Button diagrams_profile, diagrams_transactions, diagrams_accounts, diagrams_logout;

  @FXML
  private Button ApplyBtn,PieApplyBtn;
```



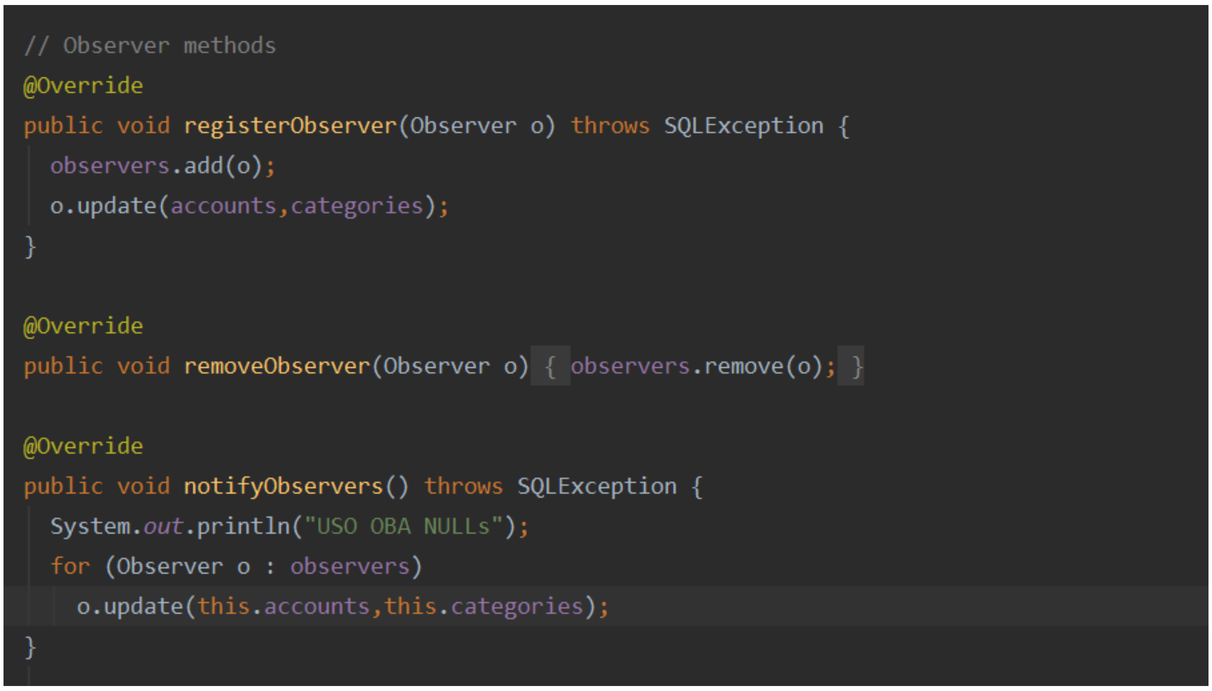
```
private BarChart<String, Number> AccountBalanceChart;

@FXML
private PieChart pieChart;

@FXML
private CategoryAxis X;

@FXML
private NumberAxis Y;

private static volatile double bankAccountBalance;
private static volatile double cardBalance;
private static volatile double cashBalance;
private static volatile double stockBalance;

private static volatile double incomeCategories=0.0;
private static volatile double expenseCategories=0.0;

private  XYChart.Series<String, Number> set1 = new Series<>();
private  ObservableList<PieChart.Data> pieChartData;
```

Definition of class BarChartController



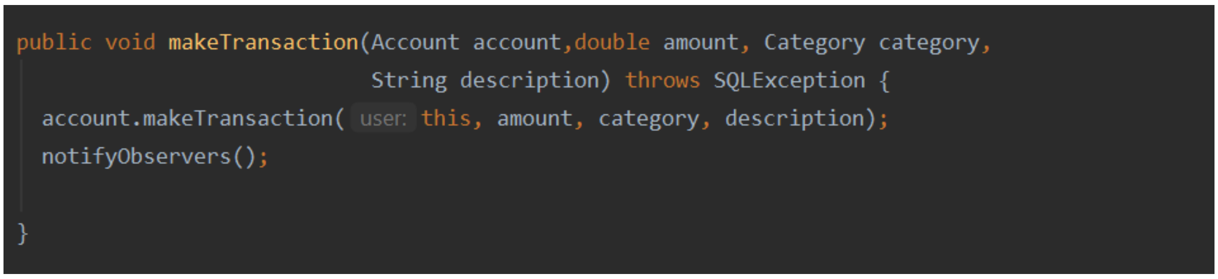
```
@Override
public void update(CompositeCollection accounts,CompositeCollection categories) throws SQLException {

AccountCompositeCollection account=(AccountCompositeCollection) accounts;

bankAccountBalance=account.getAllBalanceByType(AccountType.BANK);
cardBalance=account.getAllBalanceByType(AccountType.CARD);
stockBalance=account.getAllBalanceByType(AccountType.STOCK);
cashBalance=account.getAllBalanceByType(AccountType.CASH);

CategoryCompositeCollection category = (CategoryCompositeCollection) categories;
incomeCategories=category.getAllBalanceByType(CategoryType.INCOME);
expenseCategories=category.getAllBalanceByType(CategoryType.EXPENSE);

}
```

Update method used in BarchartController



```
// Observer methods
@Override
public void registerObserver(Observer o) throws SQLException {
    observers.add(o);
    o.update(accounts,categories);
}

@Override
public void removeObserver(Observer o) { observers.remove(o); }

@Override
public void notifyObservers() throws SQLException {
    System.out.println("USO OBA NULLs");
    for (Observer o : observers)
        o.update(this.accounts,this.categories);
}
```

Methods in User for registering, removing and notifying observers

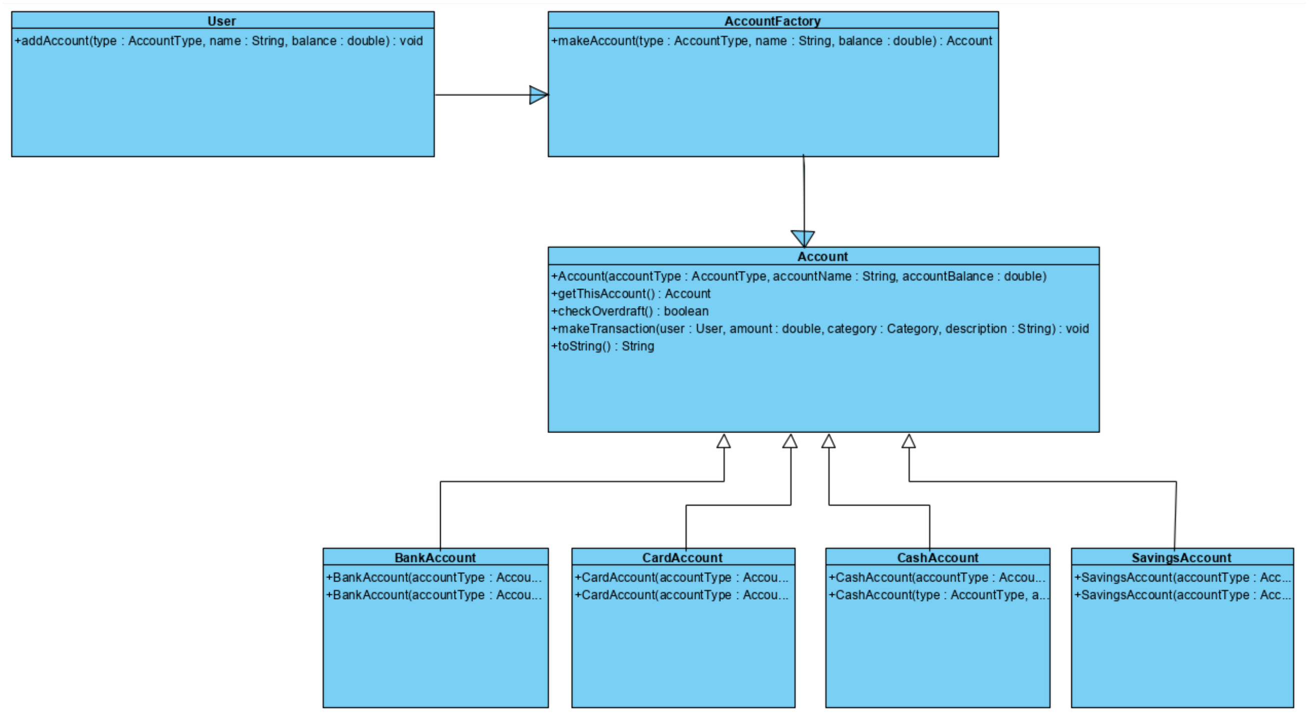


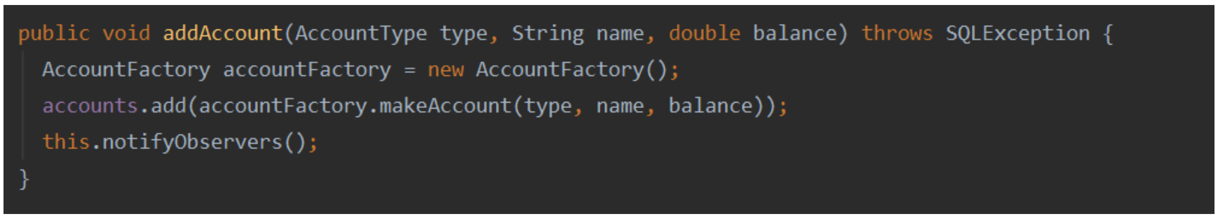
```
public void makeTransaction(Account account,double amount, Category category,
                            String description) throws SQLException {
    account.makeTransaction( user: this, amount, category, description);
    notifyObservers();

}
```

Method makeTransaction where observers get notified

1.3.3 Factory Design Pattern

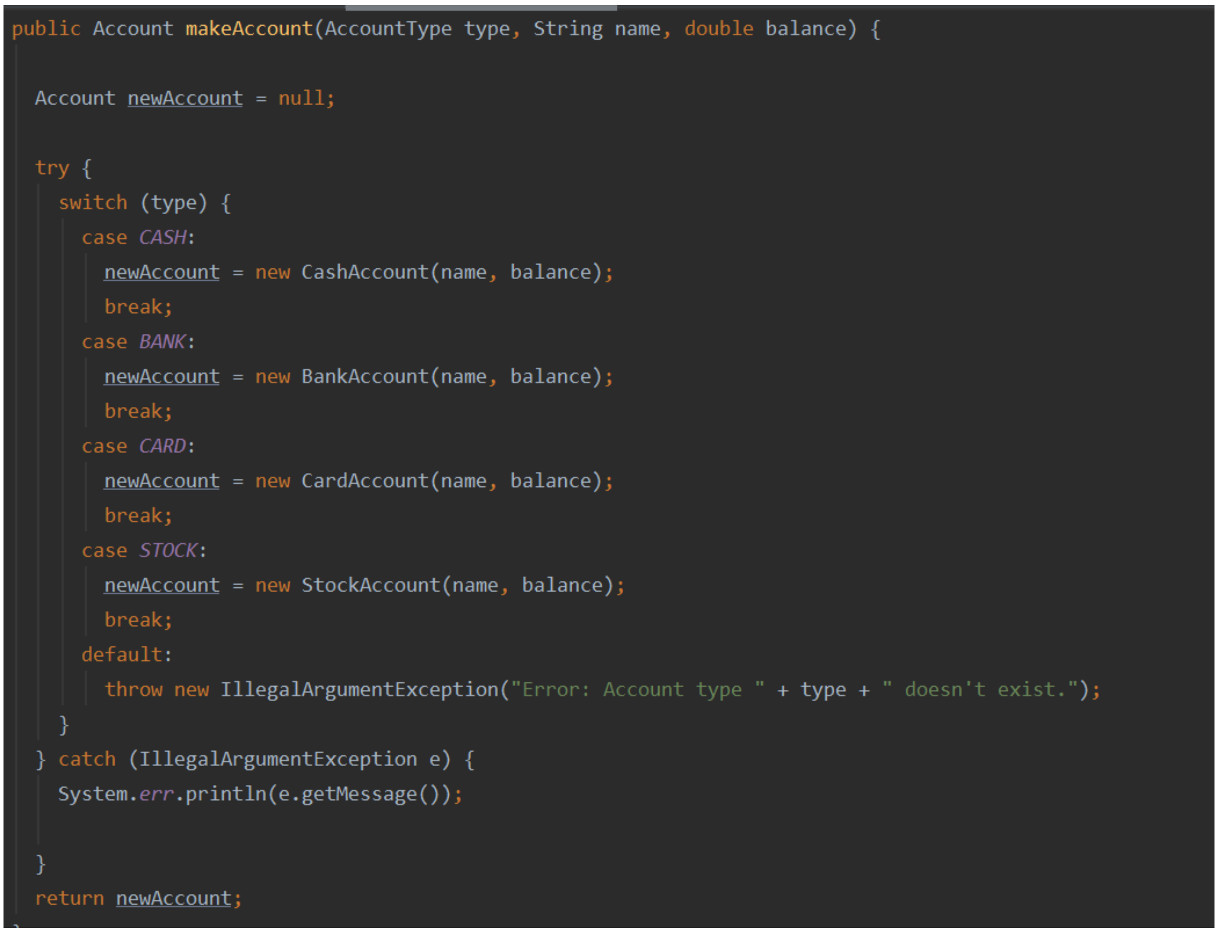
The Factory Design Pattern is used whenever we want a method to return one of several possible classes that share a common superclass. We used this pattern twice in our project. The factory's only job is creating accounts or categories, by encapsulating the creations. One of the benefits of this design pattern is that we only have one place to make modifications.


User
+addAccount(type : AccountType, name : String, balance : double) : void
AccountFactory
+makeAccount(type : AccountType, name : String, balance : double) : Account
Account
+Account(accountType : AccountType, accountName : String, accountBalance : double)
+getThisAccount() : Account
+checkOverdraft() : boolean
+makeTransaction(user : User, amount : double, category : Category, description : String) : void
+toString() : String
BankAccount
+BankAccount(accountType : Accou...
+BankAccount(accountType : Accou...
CardAccount
+CardAccount(accountType : Accou...
+CardAccount(accountType : Accou...
CashAccount
+CashAccount(accountType : Accou...
+CashAccount(type : AccountType, a...
SavingsAccount
+SavingsAccount(accountType : Acc...
+SavingsAccount(accountType : Acc...



```
public void addAccount(AccountType type, String name, double balance) throws SQLException {
    AccountFactory accountFactory = new AccountFactory();
    accounts.add(accountFactory.makeAccount(type, name, balance));
    this.notifyObservers();
}
```

addAccount method used in User

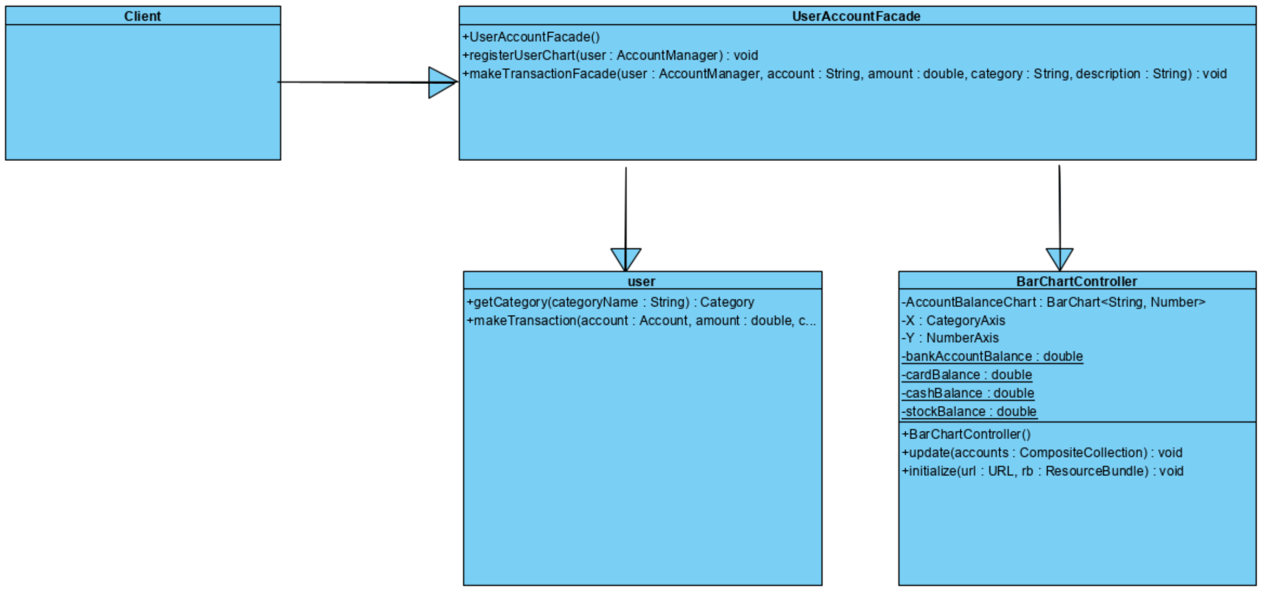


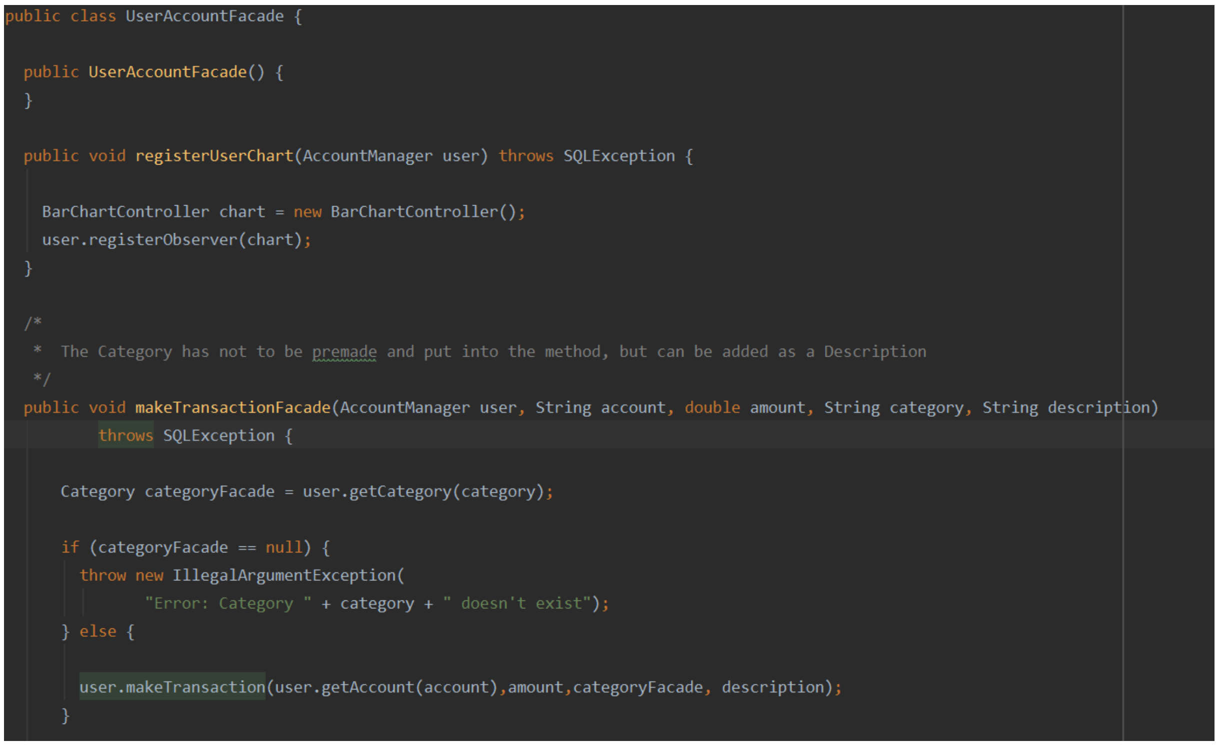
```java
public Account makeAccount(AccountType type, String name, double balance) {

  Account newAccount = null;

  try {
    switch (type) {
      case CASH:
        newAccount = new CashAccount(name, balance);
        break;
      case BANK:
        newAccount = new BankAccount(name, balance);
        break;
      case CARD:
        newAccount = new CardAccount(name, balance);
        break;
      case STOCK:
        newAccount = new StockAccount(name, balance);
        break;
      default:
        throw new IllegalArgumentException("Error: Account type " + type + " doesn't exist.");
    }
  } catch (IllegalArgumentException e) {
    System.err.println(e.getMessage());

  }
  return newAccount;
```

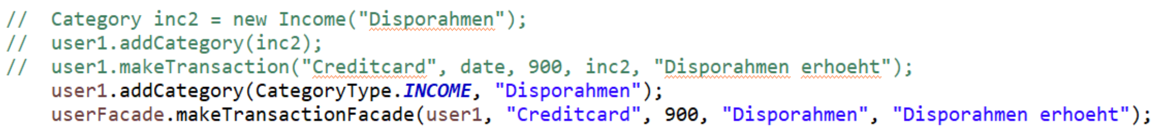
makeAccount method used in AccountFactory

1.3.4 Facade Design Pattern

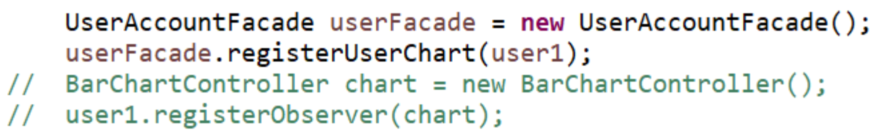
We used the facade design pattern to create a simplified interface that performs many other actions behind the scenes. The Facade Design Pattern decouples or separates the client from all of the sub components, the aim is to simplify interfaces so the user doesn't have to worry about what is going on under the hood.


Client
UserAccountFacade
+UserAccountFacade()
+registerUserChart(user : AccountManager) : void
+makeTransactionFacade(user : AccountManager, account : String, amount : double, category : String, description : String) : void
user
+getCategory(categoryName : String) : Category
+makeTransaction(account : Account, amount : double, c...
BarChartController
-AccountBalanceChart : BarChart<String, Number>
-X : CategoryAxis
-Y : NumberAxis
-bankAccountBalance : double
-cardBalance : double
-cashBalance : double
-stockBalance : double
+BarChartController()
+update(accounts : CompositeCollection) : void
+initialize(url : URL, rb : ResourceBundle) : void


```
public class UserAccountFacade {

  public UserAccountFacade() {
  }

  public void registerUserChart(AccountManager user) throws SQLException {

    BarChartController chart = new BarChartController();
    user.registerObserver(chart);
  }

  /*
   *  The Category has not to be premade and put into the method, but can be added as a Description
   */
  public void makeTransactionFacade(AccountManager user, String account, double amount, String category, String description)
          throws SQLException {

      Category categoryFacade = user.getCategory(category);

      if (categoryFacade == null) {
        throw new IllegalArgumentException(
               "Error: Category " + category + " doesn't exist");
      } else {

        user.makeTransaction(user.getAccount(account),amount,categoryFacade, description);
      }
```
Methods used in UserAccountFacade



```
//  Category inc2 = new Income("Disporahmen");
//  user1.addCategory(inc2);
//  user1.makeTransaction("Creditcard", date, 900, inc2, "Disporahmen erhoeht");
    user1.addCategory(CategoryType.INCOME, "Disporahmen");
    userFacade.makeTransactionFacade(user1, "Creditcard", 900, "Disporahmen", "Disporahmen erhoeht");
```

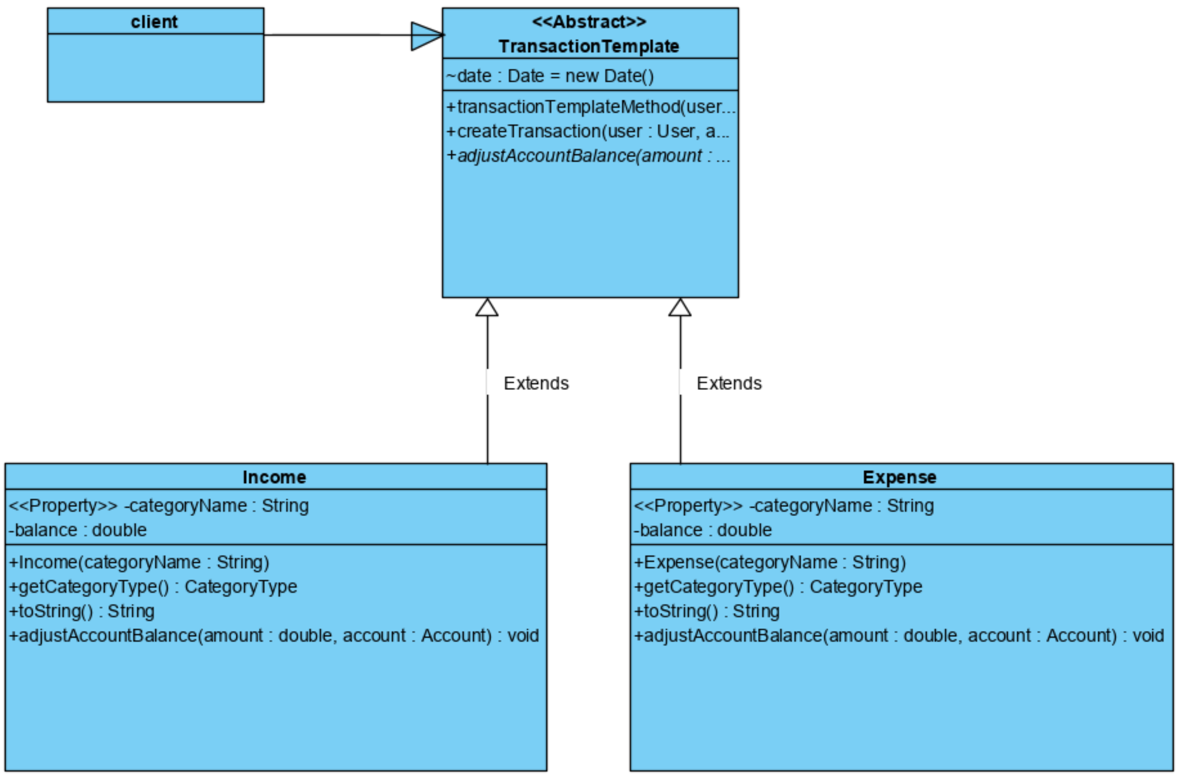


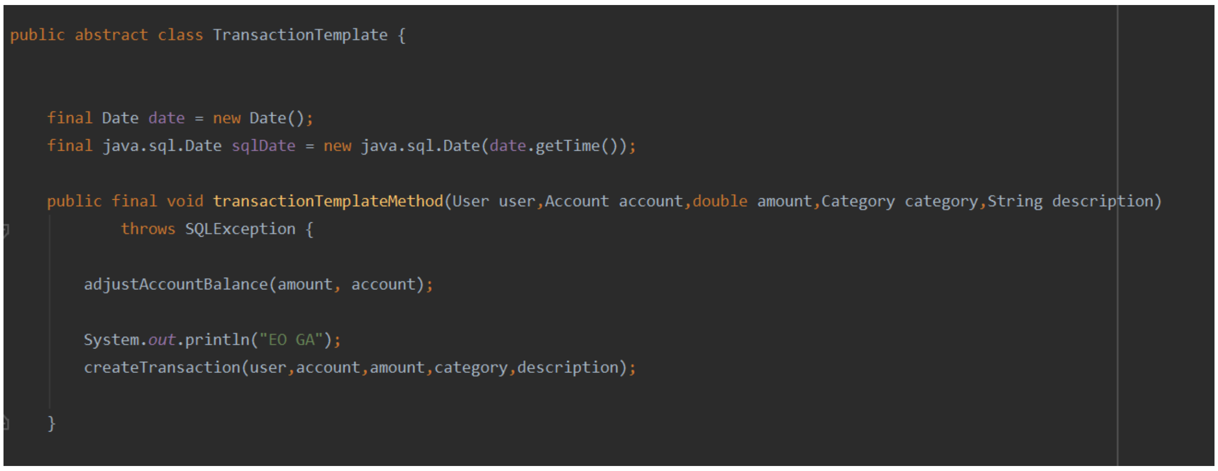
```
    UserAccountFacade userFacade = new UserAccountFacade();
    userFacade.registerUserChart(user1);
//  BarChartController chart = new BarChartController();
//  user1.registerObserver(chart);
```

Difference in the code before and after the facade pattern

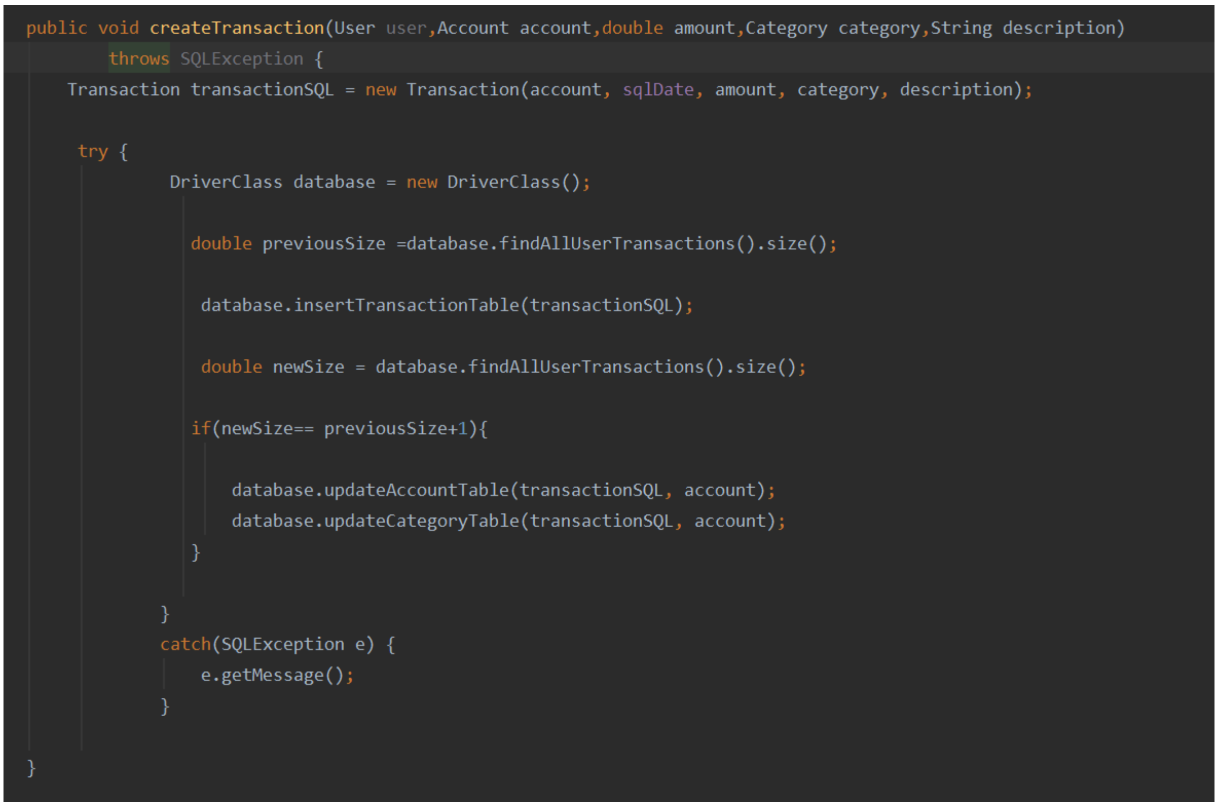
1.3.5 Template Design Pattern

The template method design pattern is used in the project to create an group of subclasses that execute a similar group of methods or better said an algorithm that contains a series of method calls that every subclass object will call. Considering that the subclass object can override some of the method calls.

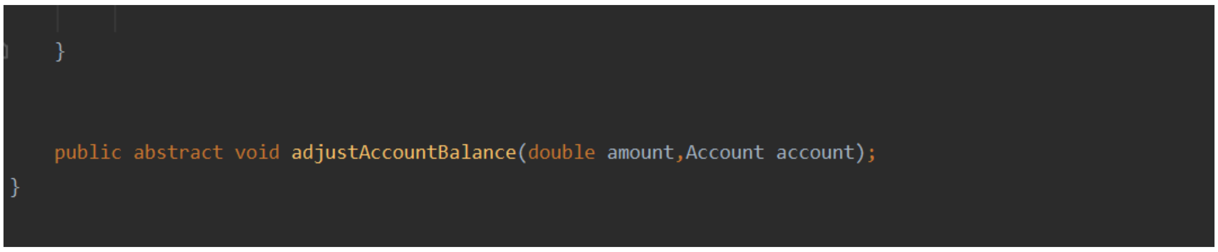

client
<<Abstract>>
TransactionTemplate
~date : Date = new Date()
+transactionTemplateMethod(user...
+createTransaction(user : User, a...
+adjustAccountBalance(amount : ...
Extends
Extends
Income
<<Property>> -categoryName : String
-balance : double
+Income(categoryName : String)
+getCategoryType() : CategoryType
+toString() : String
+adjustAccountBalance(amount : double, account : Account) : void
Expense
<<Property>> -categoryName : String
-balance : double
+Expense(categoryName : String)
+getCategoryType() : CategoryType
+toString() : String
+adjustAccountBalance(amount : double, account : Account) : void



```java
public abstract class TransactionTemplate {


    final Date date = new Date();
    final java.sql.Date sqlDate = new java.sql.Date(date.getTime());

    public final void transactionTemplateMethod(User user,Account account,double amount,Category category,String description)
            throws SQLException {

        adjustAccountBalance(amount, account);

        System.out.println("EO GA");
        createTransaction(user,account,amount,category,description);

    }
```

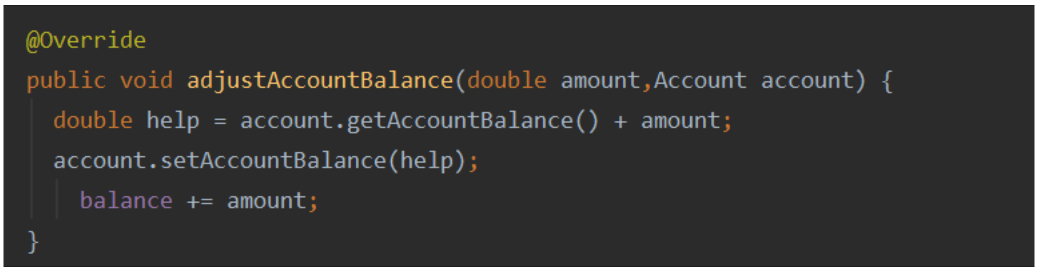


```java
  public void createTransaction(User user,Account account,double amount,Category category,String description)
          throws SQLException {
     Transaction transactionSQL = new Transaction(account, sqlDate, amount, category, description);

      try {
               DriverClass database = new DriverClass();

                 double previousSize =database.findAllUserTransactions().size();

                  database.insertTransactionTable(transactionSQL);

                  double newSize = database.findAllUserTransactions().size();

                 if(newSize== previousSize+1){

                     database.updateAccountTable(transactionSQL, account);
                     database.updateCategoryTable(transactionSQL, account);
                 }

              }
              catch(SQLException e) {
                  e.getMessage();
              }

  }
```



```java
    }


    public abstract void adjustAccountBalance(double amount,Account account);
}
```

transactionTemplateMethod in TransactionTemplate

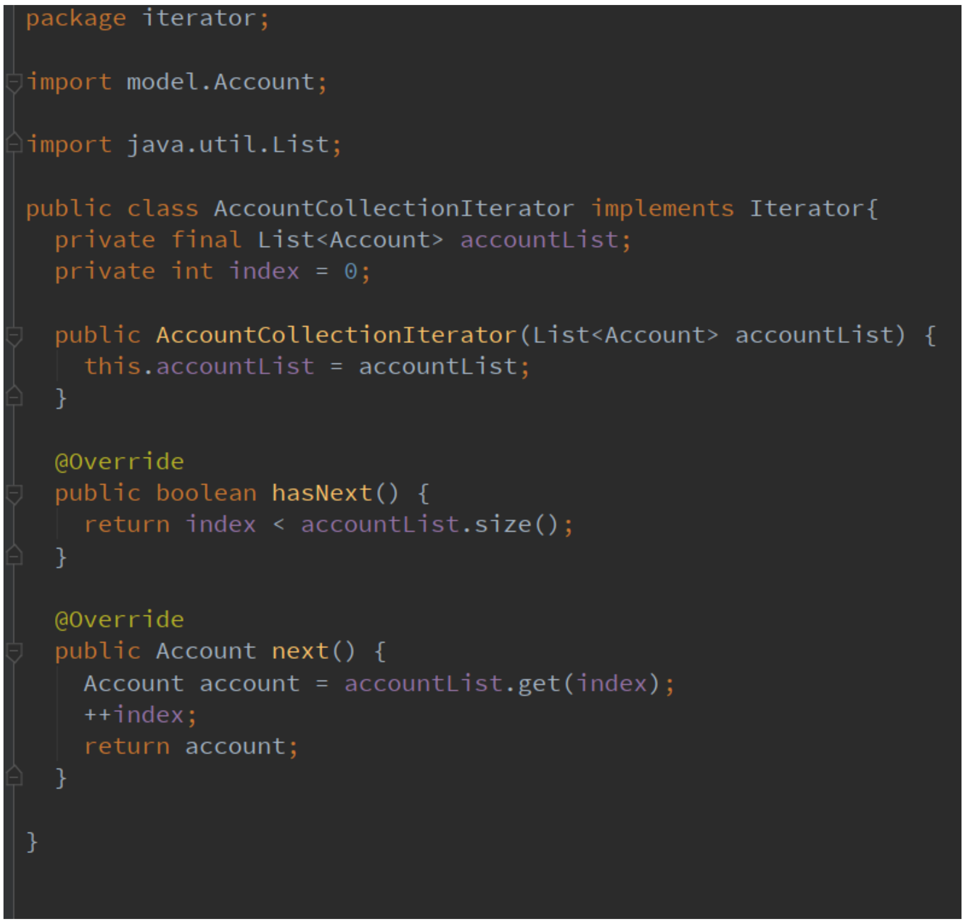


```
@Override
public void adjustAccountBalance(double amount,Account account) {
  double help = account.getAccountBalance() + amount;
  account.setAccountBalance(help);
    balance += amount;
}
```

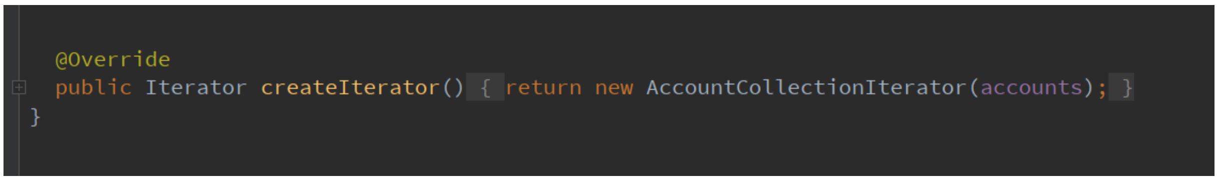
adjustAccountBalance method in Income

1.3.6 Iterator Design Pattern

Iterator pattern is used to provide us with a way of traversing user collection objects in order and displaying them, but without showing their underlying implementation.

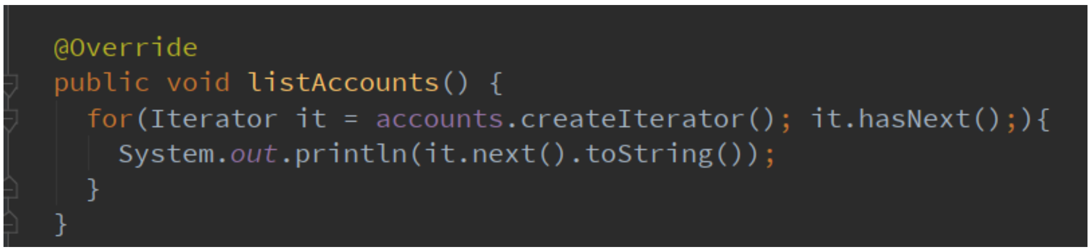


```
package iterator;

import model.Account;

import java.util.List;

public class AccountCollectionIterator implements Iterator{
  private final List<Account> accountList;
  private int index = 0;

  public AccountCollectionIterator(List<Account> accountList) {
    this.accountList = accountList;
  }

  @Override
  public boolean hasNext() {
    return index < accountList.size();
  }

  @Override
  public Account next() {
    Account account = accountList.get(index);
    ++index;
    return account;
  }

}
```



```
   @Override
   public Iterator createIterator() { return new AccountCollectionIterator(accounts); }
}
```

User class using AccountCollectionIterator to display accounts

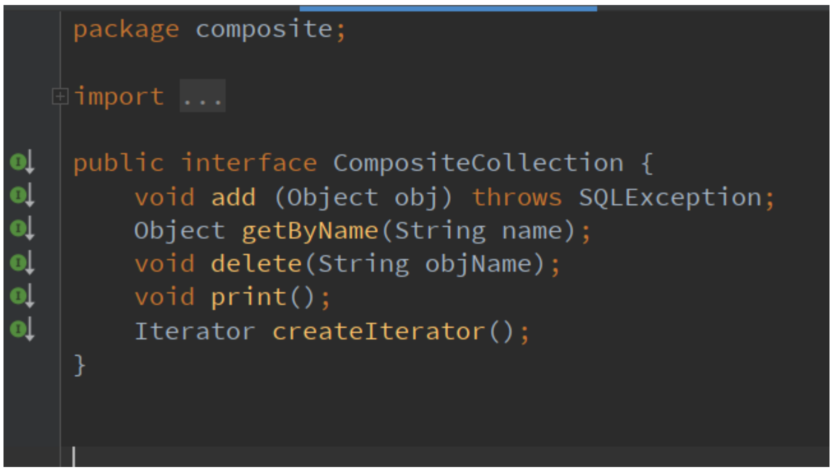


```
@Override
public void listAccounts() {
  for(Iterator it = accounts.createIterator(); it.hasNext();){
    System.out.println(it.next().toString());
  }
}
```

1.3.7 Composite Design Pattern

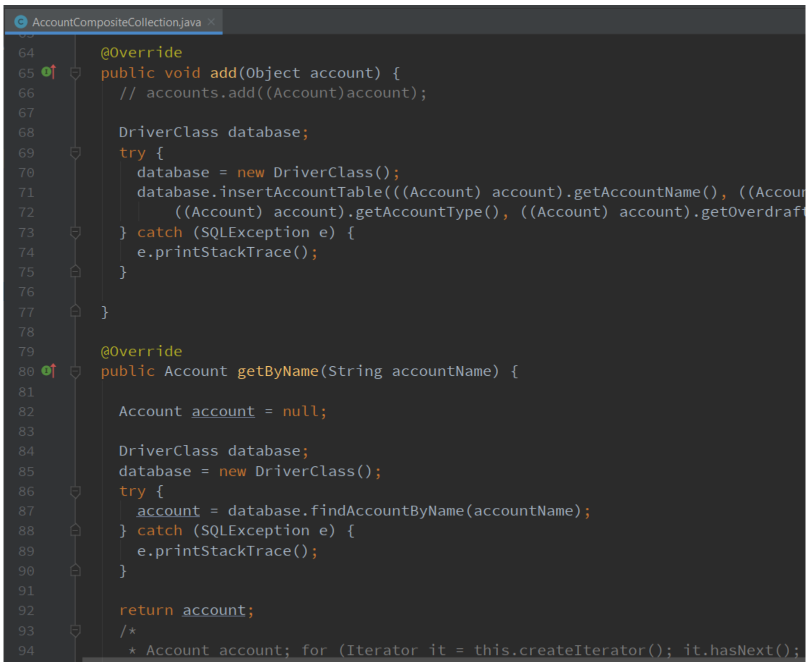
Composite Pattern is a way to let clients treat single objects and nested objects uniformly, that is to present part-whole hierarchies. Regarding our implementation, we used the Composite Design Pattern to easily store and manage user collections, and to treat different collections, as for example TransactionCollection and AccountCollection the same way.

CompositeCollection interface



```
package composite;

import ...

public interface CompositeCollection {
    void add (Object obj) throws SQLException;
    Object getByName(String name);
    void delete(String objName);
    void print();
    Iterator createIterator();
}
```

AccountCompositeCollection implementing interface



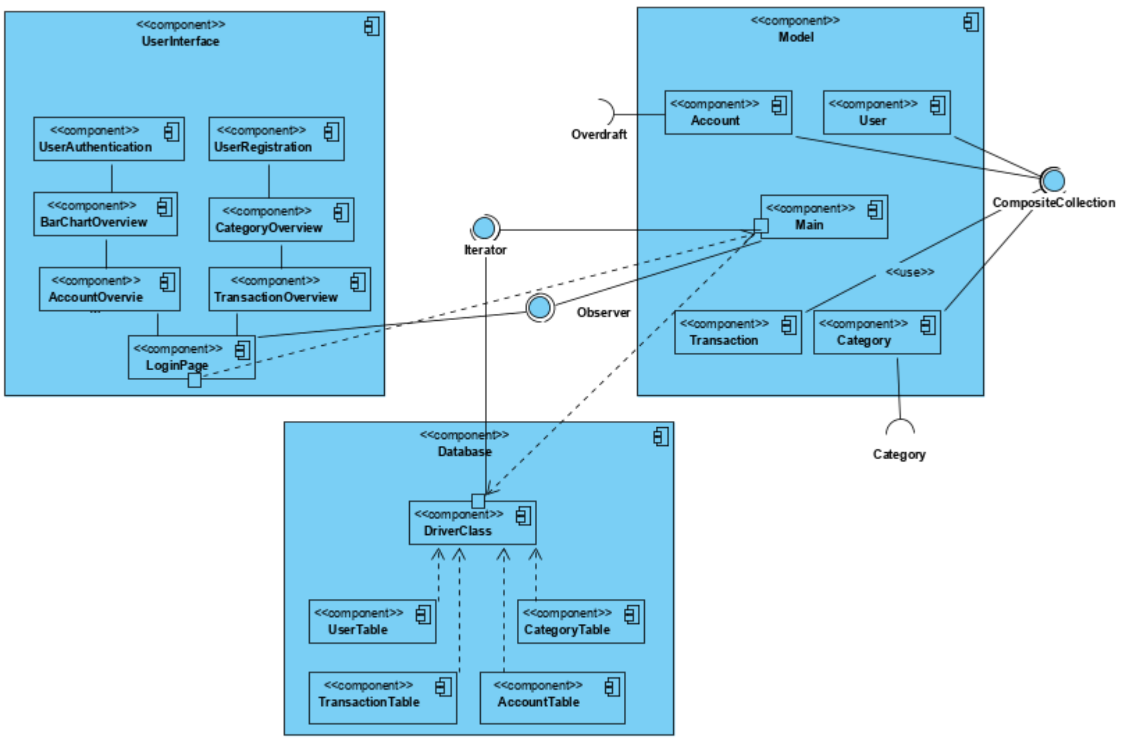
AccountCompositeCollection.java

```
@Override
public void add(Object account) {
  // accounts.add((Account)account);

  DriverClass database;
  try {
    database = new DriverClass();
    database.insertAccountTable(((Account) account).getAccountName(), ((Accour
        ((Account) account).getAccountType(), ((Account) account).getOverdraft
  } catch (SQLException e) {
    e.printStackTrace();
  }

}

@Override
public Account getByName(String accountName) {

  Account account = null;

  DriverClass database;
  database = new DriverClass();
  try {
    account = database.findAccountByName(accountName);
  } catch (SQLException e) {
    e.printStackTrace();
  }

  return account;
  /*
   * Account account; for (Iterator it = this.createIterator(); it.hasNext();
```

2 Implementation

2.1 Overview of Main Modules and Components

Our implementation bases on three main components: model, where backend with patterns is implemented, database, that is directly connected with a backend and where our data is saved and called from. User Interface is third part, that is the only connection with user. After user interaction and input in our Application, User Interface will connect our model classes, that will forward data to database, for saving. On the other side, when user wants to list his account transactions or categories, or to see his transactions in some time in the past, frontend will connect model classis, that will after all, call SQL statements for calling data from database. In that way our system depends on a good connection between those systems. With interfaces like Iterator or Observer, that connection will be well maintained. Observer Interface, for example, from Observer Pattern take cares that all date is actual, to follow all changes user made and to notify backend. In the following component diagram, we showed main app components with its dependence.

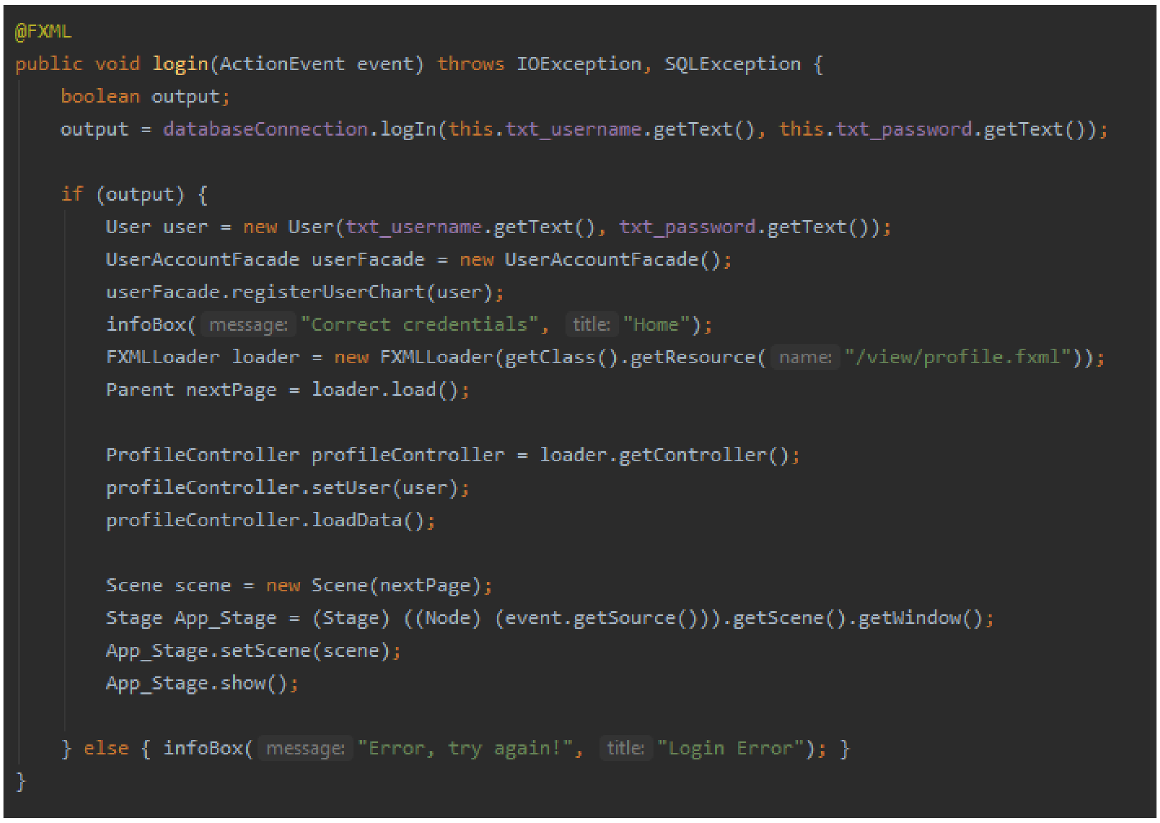


<<component>> UserInterface
<<component>> UserAuthentication
<<component>> UserRegistration
<<component>> BarChartOverview
<<component>> CategoryOverview
<<component>> AccountOvervie...
<<component>> TransactionOverview
<<component>> LoginPage
<<component>> Model
<<component>> Account
<<component>> User
Overdraft
CompositeCollection
<<component>> Main
Iterator
Observer
<<use>>
<<component>> Transaction
<<component>> Category
Category
<<component>> Database
<<component>> DriverClass
<<component>> UserTable
<<component>> CategoryTable
<<component>> TransactionTable
<<component>> AccountTable

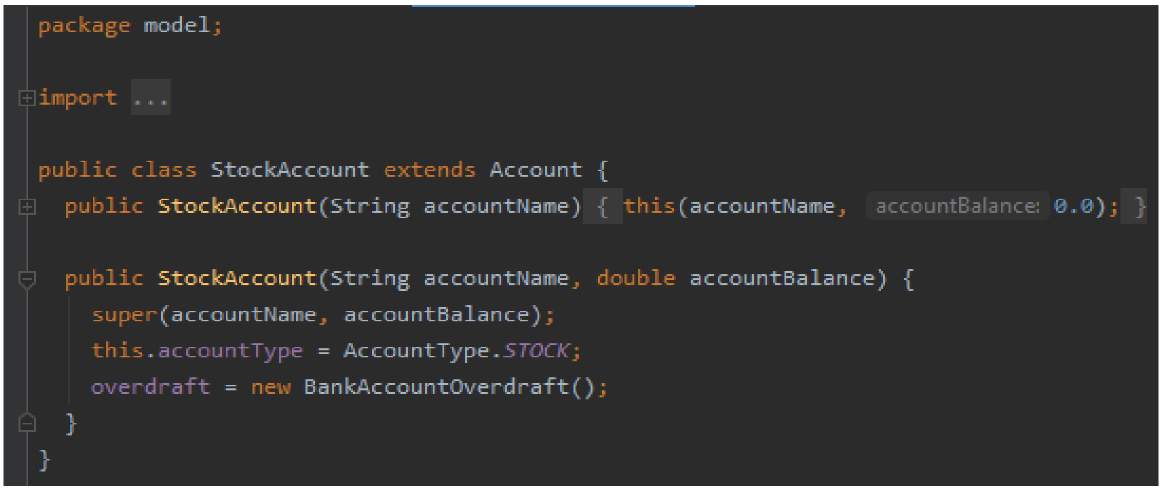
2.2 Coding Practices

Consistent style

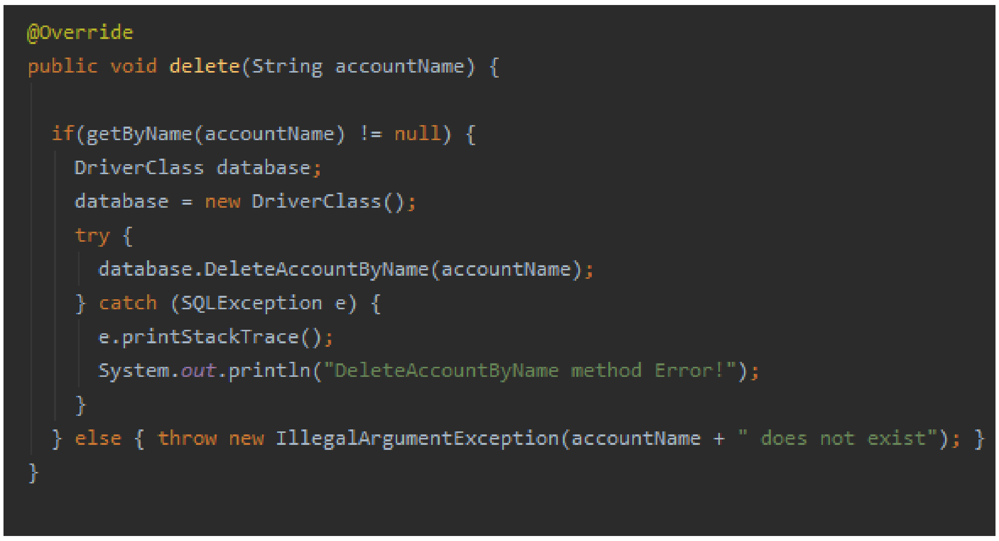
In the development of our program, we were using the Google Java Style Guide, to make our code smooth, logical and understandable. It is a simple way to get a precise shape for your code and with that make a *consistent style* in all sections of the program. That made the formatting of the code very manageable. Rules of the Google Style Guide assisted us to put all different types of code writing in one common, natural and understandable format to all members of our group.



```java
@FXML
public void login(ActionEvent event) throws IOException, SQLException {
    boolean output;
    output = databaseConnection.logIn(this.txt_username.getText(), this.txt_password.getText());

    if (output) {
        User user = new User(txt_username.getText(), txt_password.getText());
        UserAccountFacade userFacade = new UserAccountFacade();
        userFacade.registerUserChart(user);
        infoBox( message: "Correct credentials",  title: "Home");
        FXMLLoader loader = new FXMLLoader(getClass().getResource( name: "/view/profile.fxml"));
        Parent nextPage = loader.load();

        ProfileController profileController = loader.getController();
        profileController.setUser(user);
        profileController.loadData();

        Scene scene = new Scene(nextPage);
        Stage App_Stage = (Stage) ((Node) (event.getSource())).getScene().getWindow();
        App_Stage.setScene(scene);
        App_Stage.show();

    } else { infoBox( message: "Error, try again!",  title: "Login Error"); }
}
```



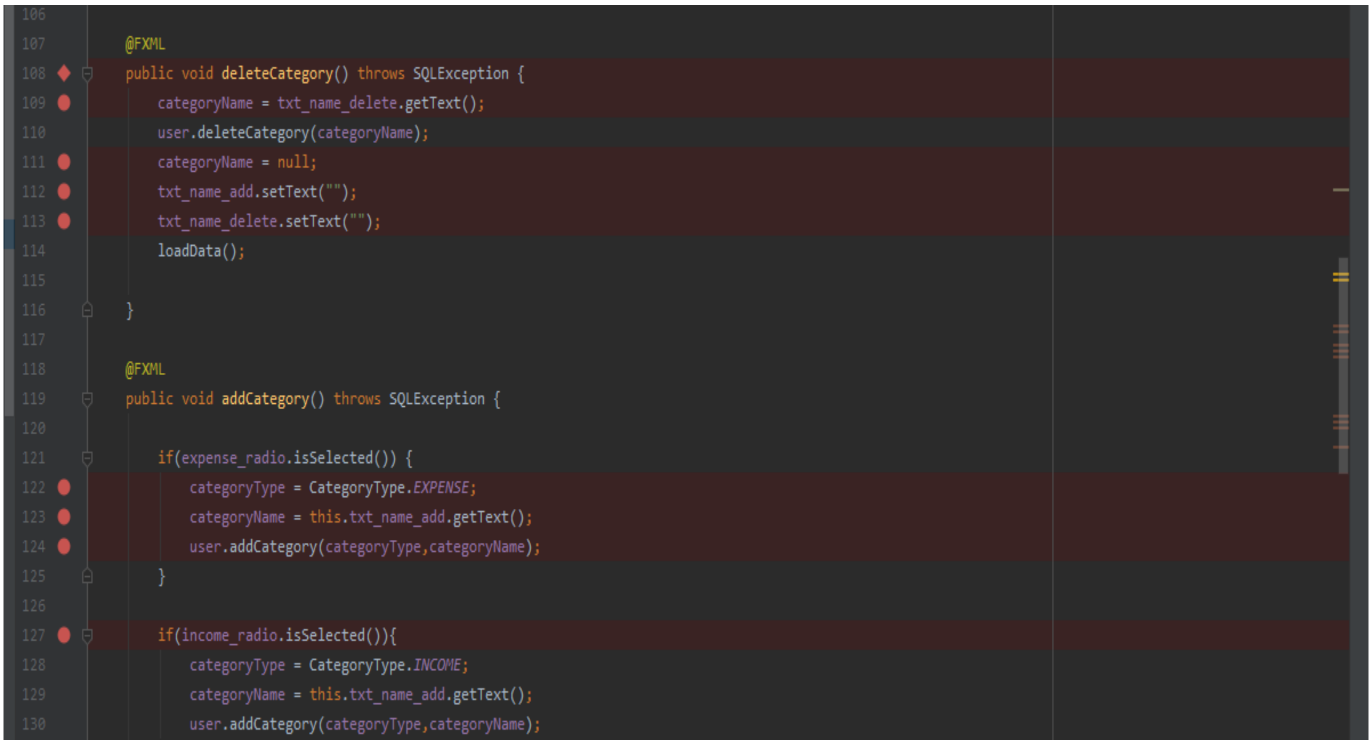
```java
package model;

import ...

public class StockAccount extends Account {
  public StockAccount(String accountName) { this(accountName,  accountBalance: 0.0); }

  public StockAccount(String accountName, double accountBalance) {
    super(accountName, accountBalance);
    this.accountType = AccountType.STOCK;
    overdraft = new BankAccountOverdraft();
  }
}
```



```java
@Override
public void delete(String accountName) {

  if(getByName(accountName) != null) {
    DriverClass database;
    database = new DriverClass();
    try {
      database.DeleteAccountByName(accountName);
    } catch (SQLException e) {
      e.printStackTrace();
      System.out.println("DeleteAccountByName method Error!");
    }
  } else { throw new IllegalArgumentException(accountName + " does not exist"); }
}
```

Naming

We kept our classes as small as possible to be able to find the methods in them and update those methods if that is demanded. All classes are separated into the packages, which secured an easy job for us to keep the entire code clean and organized. We were keeping names of the variables short and reasonable, to provide the reader of the code opportunity to tell through the name of the variable what that variable represents. The same idea is implemented in the naming of the methods. The methods are short and named accurately, they are practically verbs that are mixed with internal words because readers of the code viewing the name of the method can understand what that method should be doing.

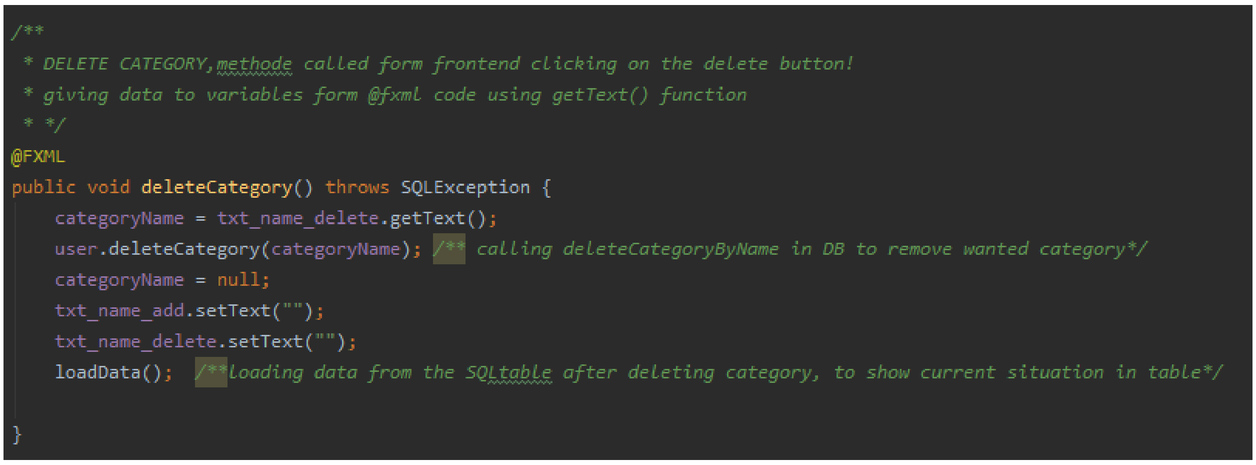


```java

@FXML
public void deleteCategory() throws SQLException {
    categoryName = txt_name_delete.getText();
    user.deleteCategory(categoryName);
    categoryName = null;
    txt_name_add.setText("");
    txt_name_delete.setText("");
    loadData();

}

@FXML
public void addCategory() throws SQLException {

    if(expense_radio.isSelected()) {
        categoryType = CategoryType.EXPENSE;
        categoryName = this.txt_name_add.getText();
        user.addCategory(categoryType,categoryName);
    }

    if(income_radio.isSelected()){
        categoryType = CategoryType.INCOME;
        categoryName = this.txt_name_add.getText();
        user.addCategory(categoryType,categoryName);
```

Comments and Exceptions

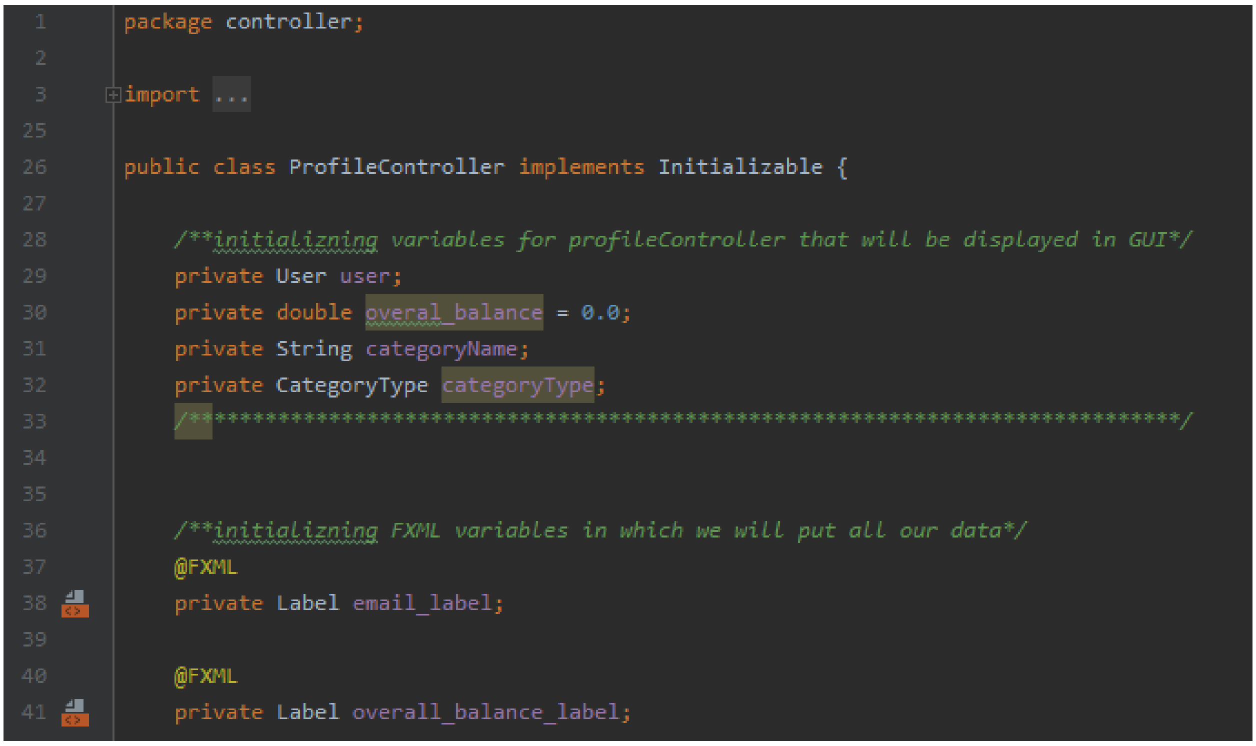
Every part of the code is explained in comments, to provide an explanation to the reader and a reminder to the developer, and we were avoiding putting obvious comments in our code because it can be overdone. Commenting in our code is useful and relevant, it commonly makes orientation in code much more manageable. We debugged our methods, to be able to catch the flow of the program in the console while using a graphical user interface. We were throwing exceptions with messages that show where the program breaking. We separated our often-used methods and reconstructed them to avoid duplicating the code. Putting information in our console allowed us to easier test our interface, and to see if the object in GUI is functional.

This is how we implemented our comments:



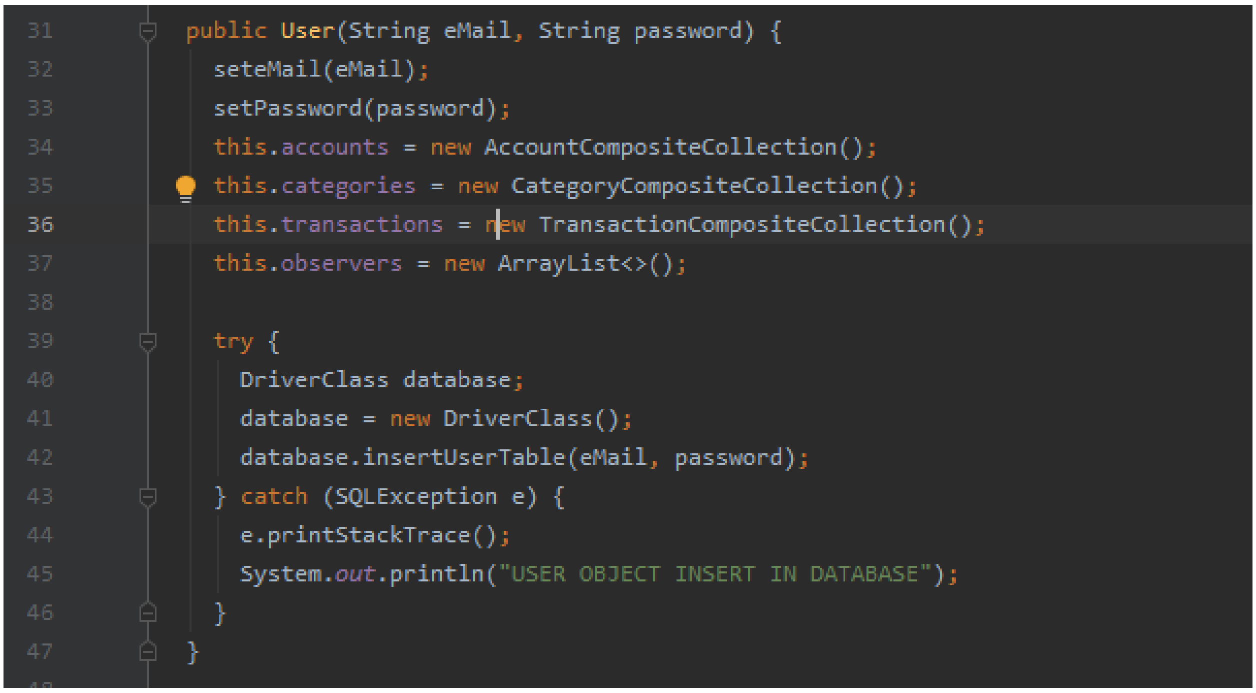
```
/**
 * DELETE CATEGORY,methode called form frontend clicking on the delete button!
 * giving data to variables form @fxml code using getText() function
 * */
@FXML
public void deleteCategory() throws SQLException {
    categoryName = txt_name_delete.getText();
    user.deleteCategory(categoryName); /** calling deleteCategoryByName in DB to remove wanted category*/
    categoryName = null;
    txt_name_add.setText("");
    txt_name_delete.setText("");
    loadData();  /**loading data from the SQLtable after deleting category, to show current situation in table*/

}
```

This is what we try to avoid commenting code:



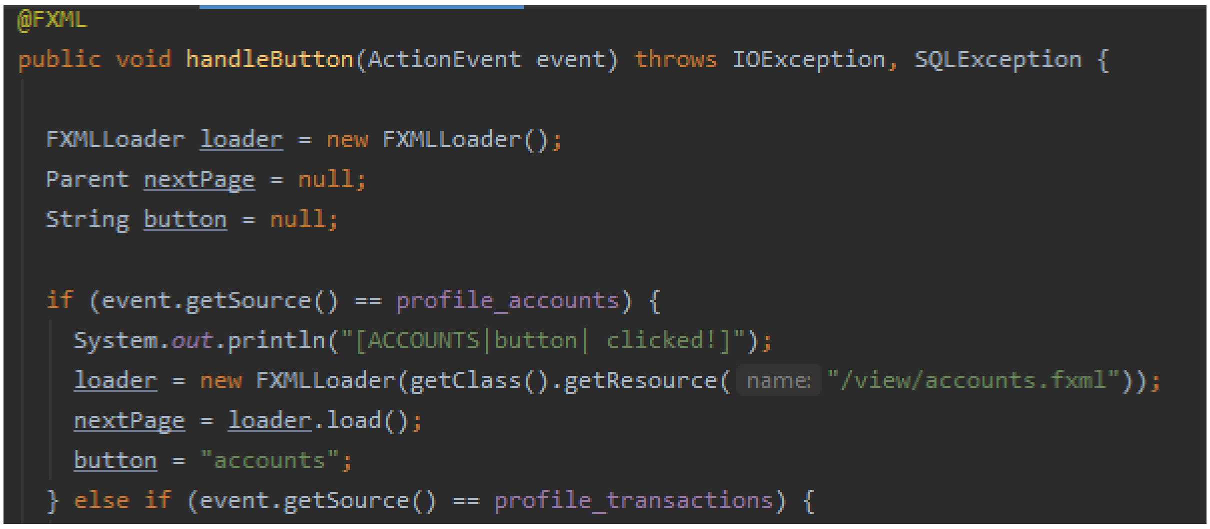
```
package controller;

import ...

public class ProfileController implements Initializable {

    /**initializning variables for profileController that will be displayed in GUI*/
    private User user;
    private double overal_balance = 0.0;
    private String categoryName;
    private CategoryType categoryType;
    /*************************************************************************************/


    /**initializning FXML variables in which we will put all our data*/
    @FXML
    private Label email_label;

    @FXML
    private Label overall_balance_label;
```

Example of Exceptions in our code:



```
public User(String eMail, String password) {
    seteMail(eMail);
    setPassword(password);
    this.accounts = new AccountCompositeCollection();
    this.categories = new CategoryCompositeCollection();
    this.transactions = new TransactionCompositeCollection();
    this.observers = new ArrayList<>();

    try {
        DriverClass database;
        database = new DriverClass();
        database.insertUserTable(eMail, password);
    } catch (SQLException e) {
        e.printStackTrace();
        System.out.println("USER OBJECT INSERT IN DATABASE");
    }
}
```

An Example of information in backend, to make functional frontend:

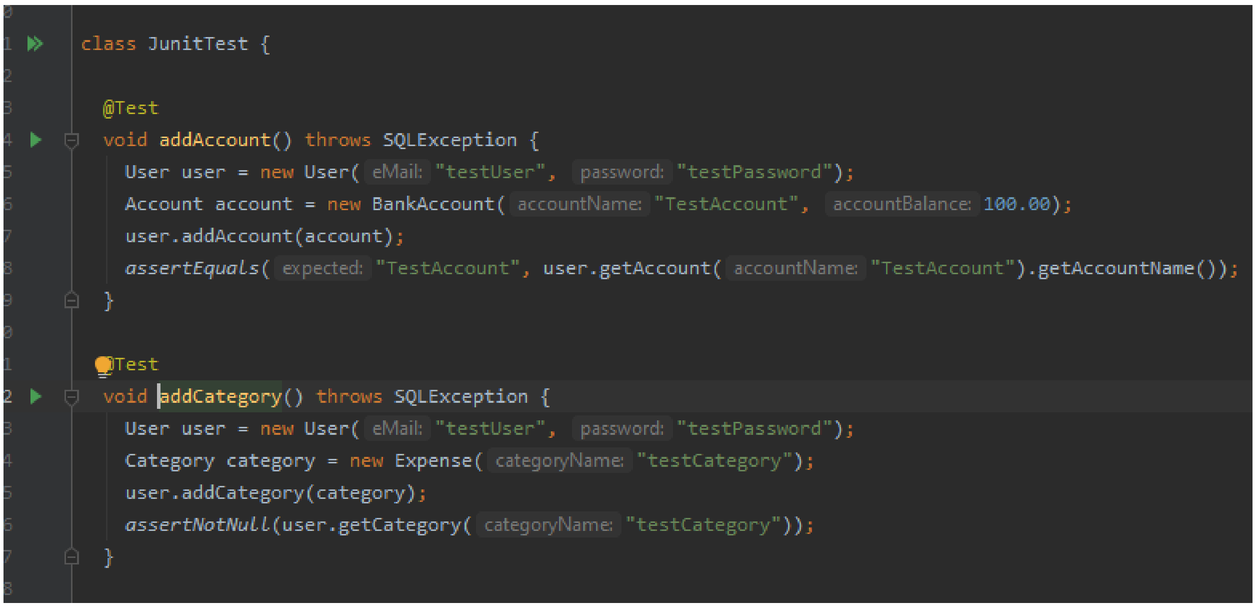


```
@FXML
public void handleButton(ActionEvent event) throws IOException, SQLException {

    FXMLLoader loader = new FXMLLoader();
    Parent nextPage = null;
    String button = null;

    if (event.getSource() == profile_accounts) {
        System.out.println("[ACCOUNTS|button| clicked!]");
        loader = new FXMLLoader(getClass().getResource( name: "/view/accounts.fxml"));
        nextPage = loader.load();
        button = "accounts";
    } else if (event.getSource() == profile_transactions) {
```

Overriding and Enumeration

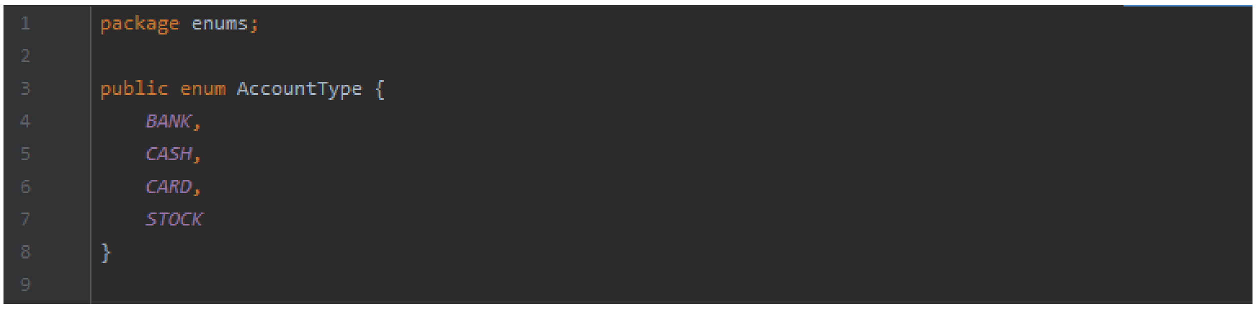
We were overriding methods to avoid mistakes of wrong typing a method name and to make our code much more understandable because it is an easier method of connecting things when the methods are overwritten. We also attempt to avoid an empty catch block, to provide us an easier way to find bugs in our development. Testing our program with the JUnit advanced testing of our code in a much more professional way. Using this type of coding practice helped us to verify the behaviour of code and identifying the software bugs. Implementation of the enumeration optimized code and allowed us easily to deal with a different type of categories by directly using pre-defined enumerations for account types and category types.

An example of testing program with Junit Test:

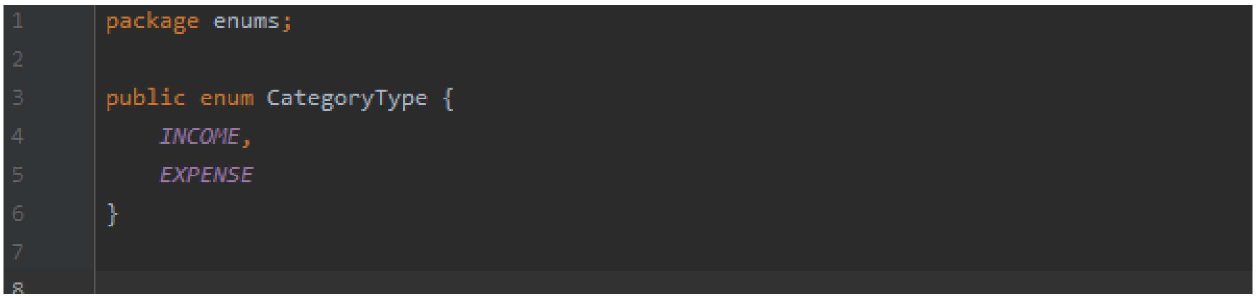


```
class JunitTest {

  @Test
  void addAccount() throws SQLException {
    User user = new User( eMail: "testUser", password: "testPassword");
    Account account = new BankAccount( accountName: "TestAccount", accountBalance: 100.00);
    user.addAccount(account);
    assertEquals( expected: "TestAccount", user.getAccount( accountName: "TestAccount").getAccountName());
  }

  @Test
  void addCategory() throws SQLException {
    User user = new User( eMail: "testUser", password: "testPassword");
    Category category = new Expense( categoryName: "testCategory");
    user.addCategory(category);
    assertNotNull(user.getCategory( categoryName: "testCategory"));
  }
```

Enumeration in our code:

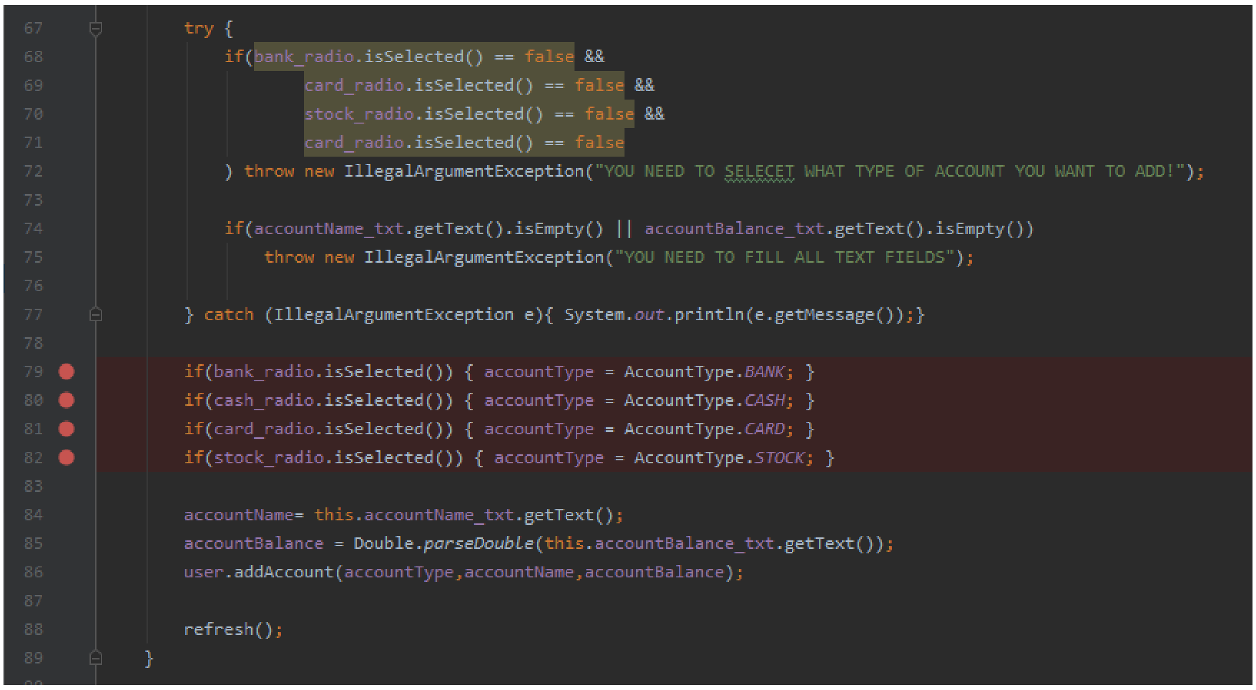


```
package enums;

public enum AccountType {
    BANK,
    CASH,
    CARD,
    STOCK
}
```



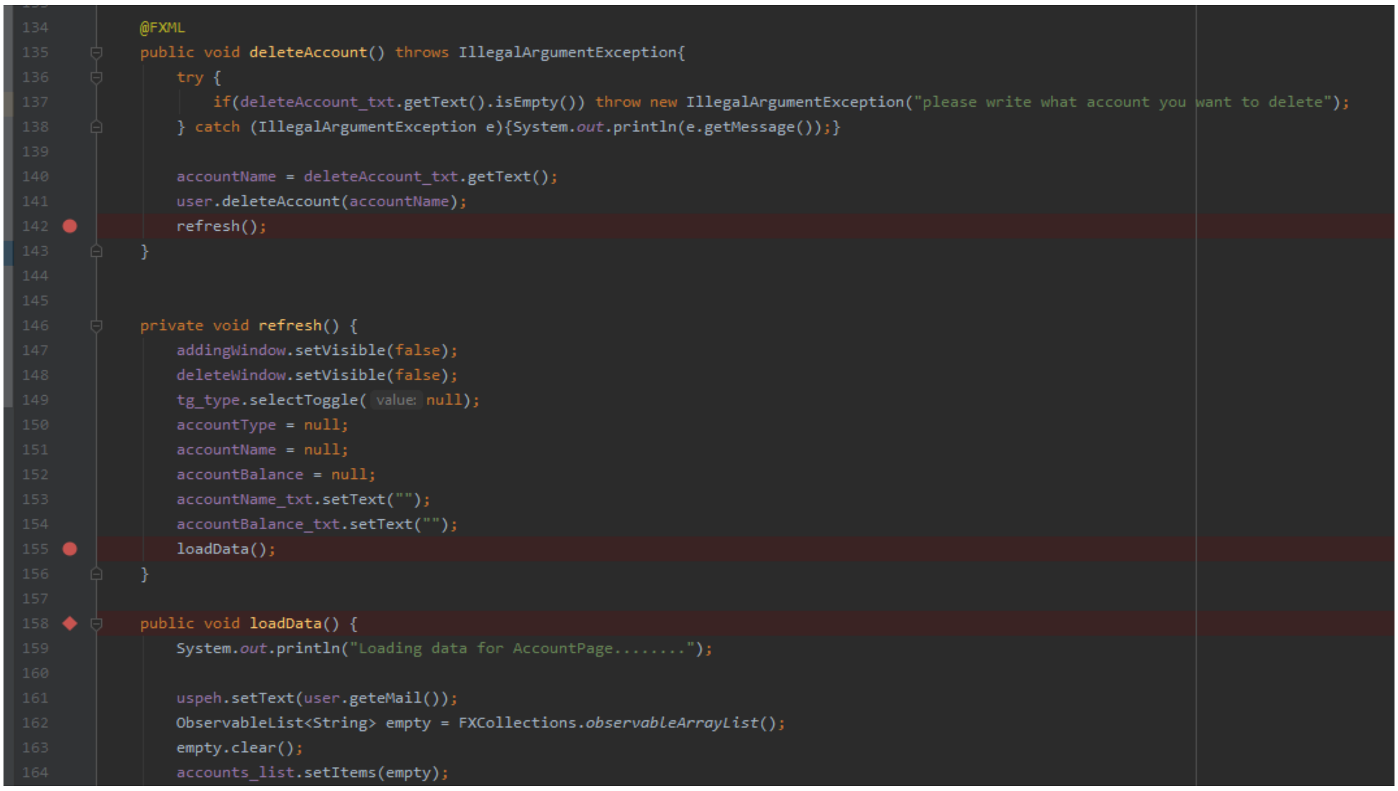
```
package enums;

public enum CategoryType {
    INCOME,
    EXPENSE
}
```

An Example of using Enumerations in AccountsController:



```java
        try {
            if(bank_radio.isSelected() == false &&
                    card_radio.isSelected() == false &&
                    stock_radio.isSelected() == false &&
                    card_radio.isSelected() == false
            ) throw new IllegalArgumentException("YOU NEED TO SELECET WHAT TYPE OF ACCOUNT YOU WANT TO ADD!");

            if(accountName_txt.getText().isEmpty() || accountBalance_txt.getText().isEmpty())
                throw new IllegalArgumentException("YOU NEED TO FILL ALL TEXT FIELDS");

        } catch (IllegalArgumentException e){ System.out.println(e.getMessage());}

        if(bank_radio.isSelected()) { accountType = AccountType.BANK; }
        if(cash_radio.isSelected()) { accountType = AccountType.CASH; }
        if(card_radio.isSelected()) { accountType = AccountType.CARD; }
        if(stock_radio.isSelected()) { accountType = AccountType.STOCK; }

        accountName= this.accountName_txt.getText();
        accountBalance = Double.parseDouble(this.accountBalance_txt.getText());
        user.addAccount(accountType,accountName,accountBalance);

        refresh();
    }
```

An Example of the method that avoid duplicating code:

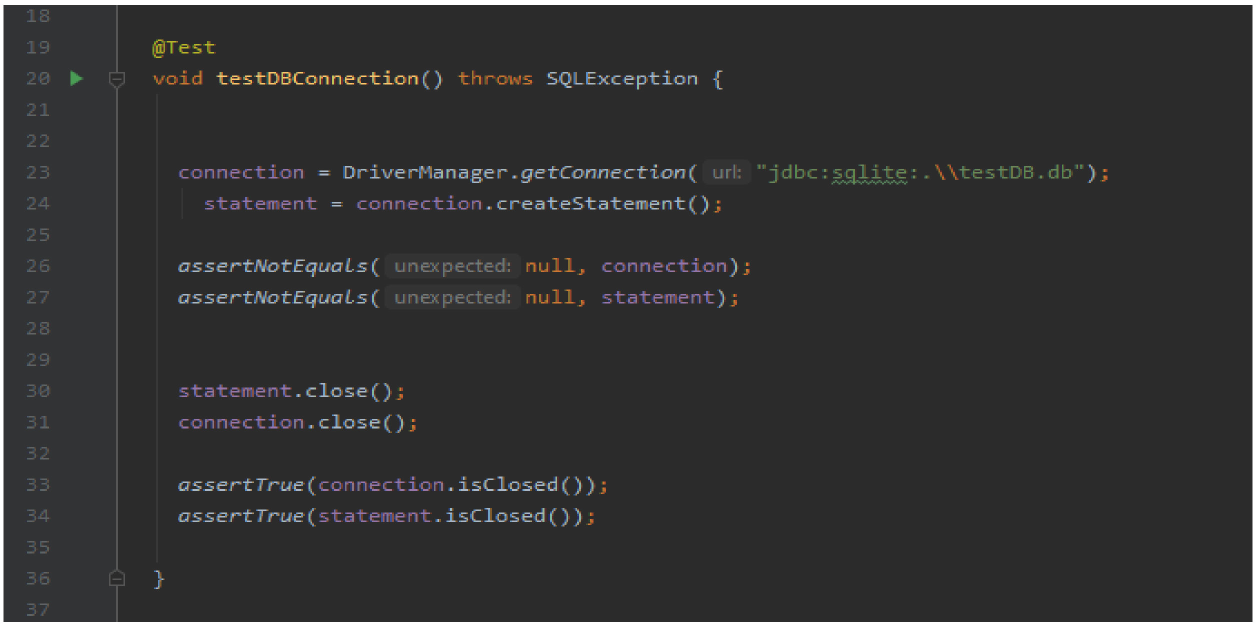


```java
    @FXML
    public void deleteAccount() throws IllegalArgumentException{
        try {
            if(deleteAccount_txt.getText().isEmpty()) throw new IllegalArgumentException("please write what account you want to delete");
        } catch (IllegalArgumentException e){System.out.println(e.getMessage());}

        accountName = deleteAccount_txt.getText();
        user.deleteAccount(accountName);
        refresh();
    }


    private void refresh() {
        addingWindow.setVisible(false);
        deleteWindow.setVisible(false);
        tg_type.selectToggle( value: null);
        accountType = null;
        accountName = null;
        accountBalance = null;
        accountName_txt.setText("");
        accountBalance_txt.setText("");
        loadData();
    }

    public void loadData() {
        System.out.println("Loading data for AccountPage........");

        uspeh.setText(user.geteMail());
        ObservableList<String> empty = FXCollections.observableArrayList();
        empty.clear();
        accounts_list.setItems(empty);
```

2.3 Defensive Programming

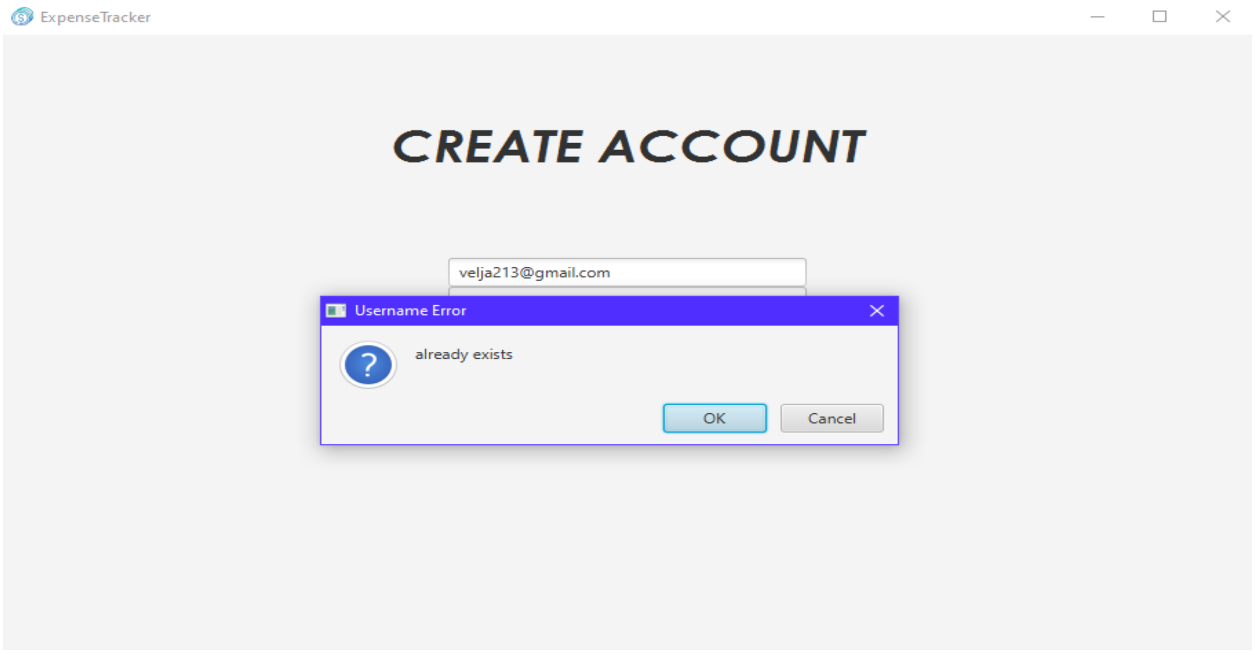
Our defensive programming is mostly presented through the JUnit test for our classes and our database code also thought JUnit test for database. The code is written step by step, all-time caring about every part of the code. We tried to test as a lot as possible to provide secured initializing of variables, parameters, and objects. Knowing that we provided correctness of the code.

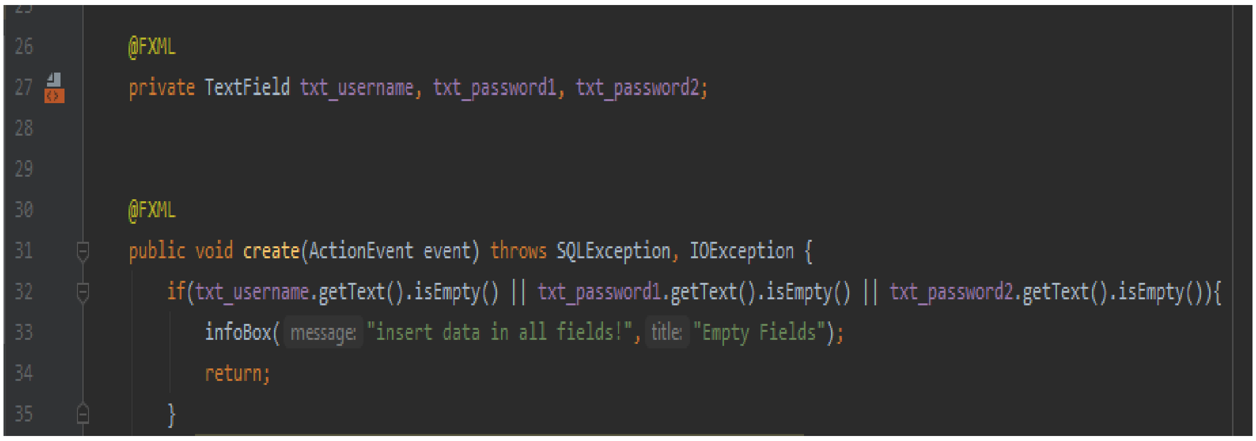
An example of testing program with Junit Test, assert:



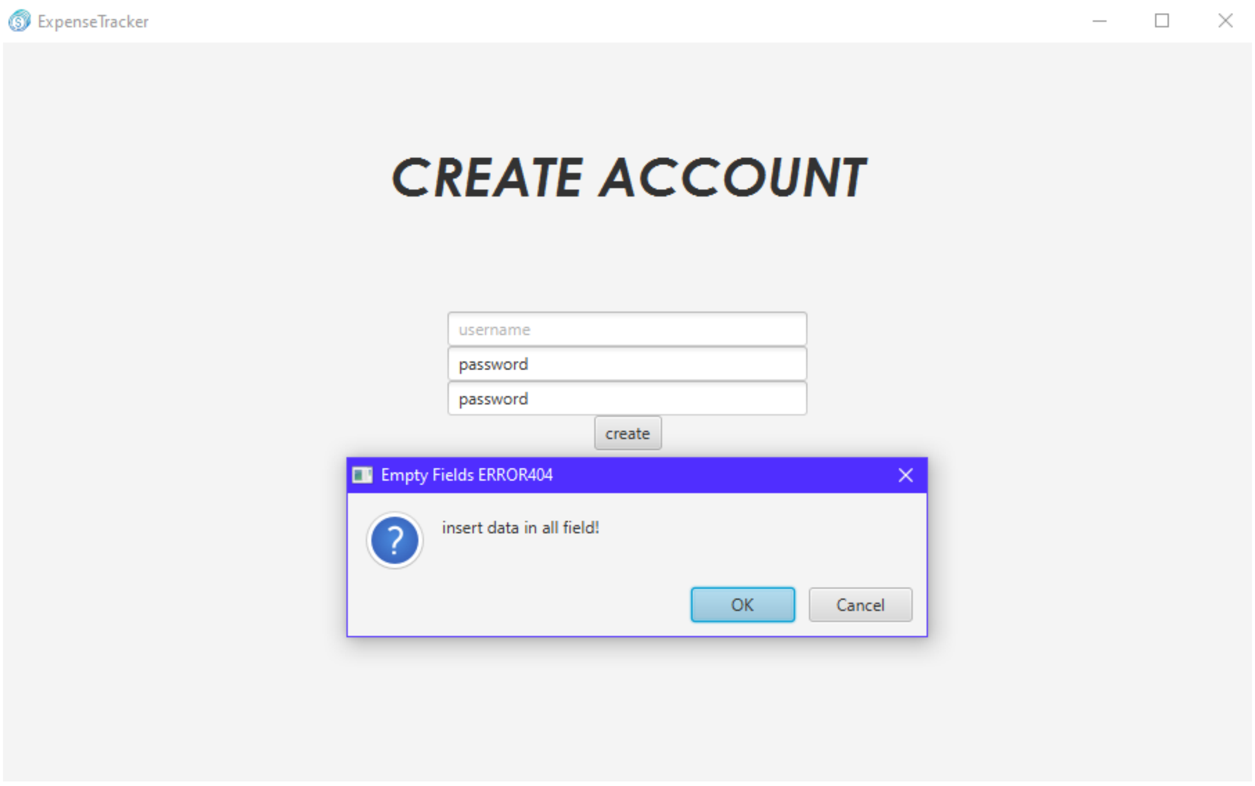
```
@Test
void testDBConnection() throws SQLException {


    connection = DriverManager.getConnection( url: "jdbc:sqlite:.\\testDB.db");
      statement = connection.createStatement();

    assertNotEquals( unexpected: null, connection);
    assertNotEquals( unexpected: null, statement);


    statement.close();
    connection.close();

    assertTrue(connection.isClosed());
    assertTrue(statement.isClosed());

}
```

Examples of showing information on screen when the inputs are wrong inserted:





```
    @FXML
    private TextField txt_username, txt_password1, txt_password2;


    @FXML
    public void create(ActionEvent event) throws SQLException, IOException {
        if(txt_username.getText().isEmpty() || txt_password1.getText().isEmpty() || txt_password2.getText().isEmpty()){
            infoBox( message: "insert data in all fields!", title: "Empty Fields");
            return;
        }
```



ExpenseTracker

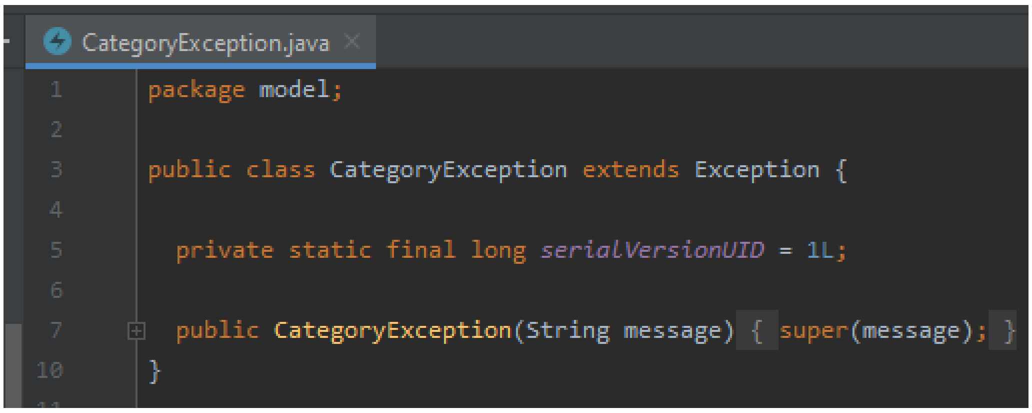
CREATE ACCOUNT

Empty Fields ERROR404

insert data in all field!

We wanted to avoid returning inaccurate results and continue to have a clean program with fewer chances to break, on the other side we also needed to have a lot of back information from the code, we did it printing back information for calling every method. That acknowledged us to follow the flow of the program and correct all parts of the program that need to be fixed, and with that minimize the robustness of the code. We tried to keep the program safe from Invalid Inputs, and if things go wrong, we manage to keep the program perform correctly. We wanted to always have some type of return info or message to the user because that is always a better alternative then crashing of the program in the middle of using the application.

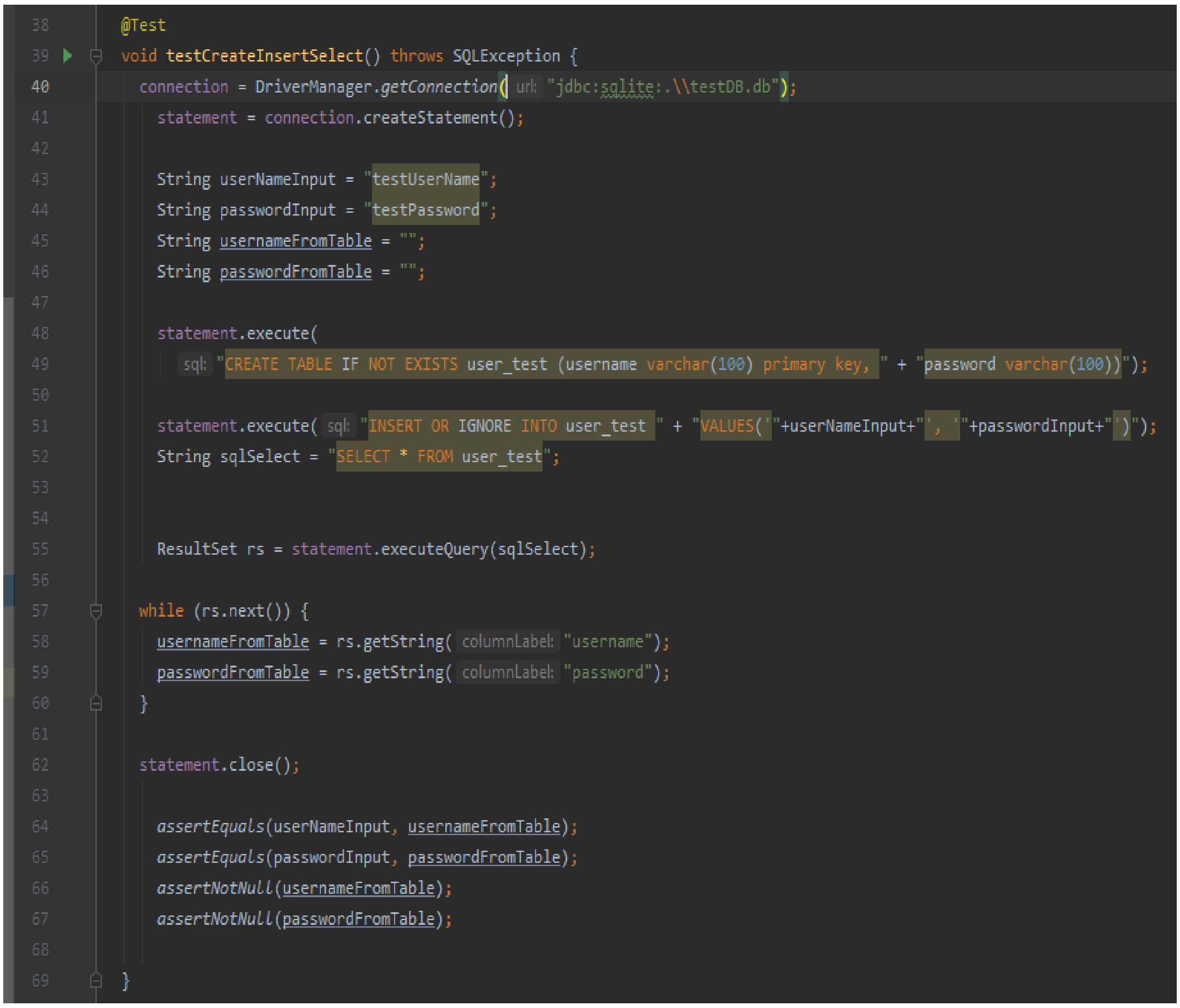
Exception class for Category:



CategoryException.java

```java
package model;

public class CategoryException extends Exception {

  private static final long serialVersionUID = 1L;

  public CategoryException(String message) { super(message); }
}
```

Junit test for the SQL insert:



```java
@Test
void testCreateInsertSelect() throws SQLException {
  connection = DriverManager.getConnection("jdbc:sqlite:.\\testDB.db");
    statement = connection.createStatement();

    String userNameInput = "testUserName";
    String passwordInput = "testPassword";
    String usernameFromTable = "";
    String passwordFromTable = "";

    statement.execute(
      "CREATE TABLE IF NOT EXISTS user_test (username varchar(100) primary key, " + "password varchar(100))");

    statement.execute("INSERT OR IGNORE INTO user_test " + "VALUES('"+userNameInput+"', '"+passwordInput+"')");
    String sqlSelect = "SELECT * FROM user_test";


    ResultSet rs = statement.executeQuery(sqlSelect);

  while (rs.next()) {
    usernameFromTable = rs.getString("username");
    passwordFromTable = rs.getString("password");
  }

  statement.close();

    assertEquals(userNameInput, usernameFromTable);
    assertEquals(passwordInput, passwordFromTable);
    assertNotNull(usernameFromTable);
    assertNotNull(passwordFromTable);

}
```

3 Software Quality

3.1 Code Metrics

3.2 Testcases for Functional Requirements

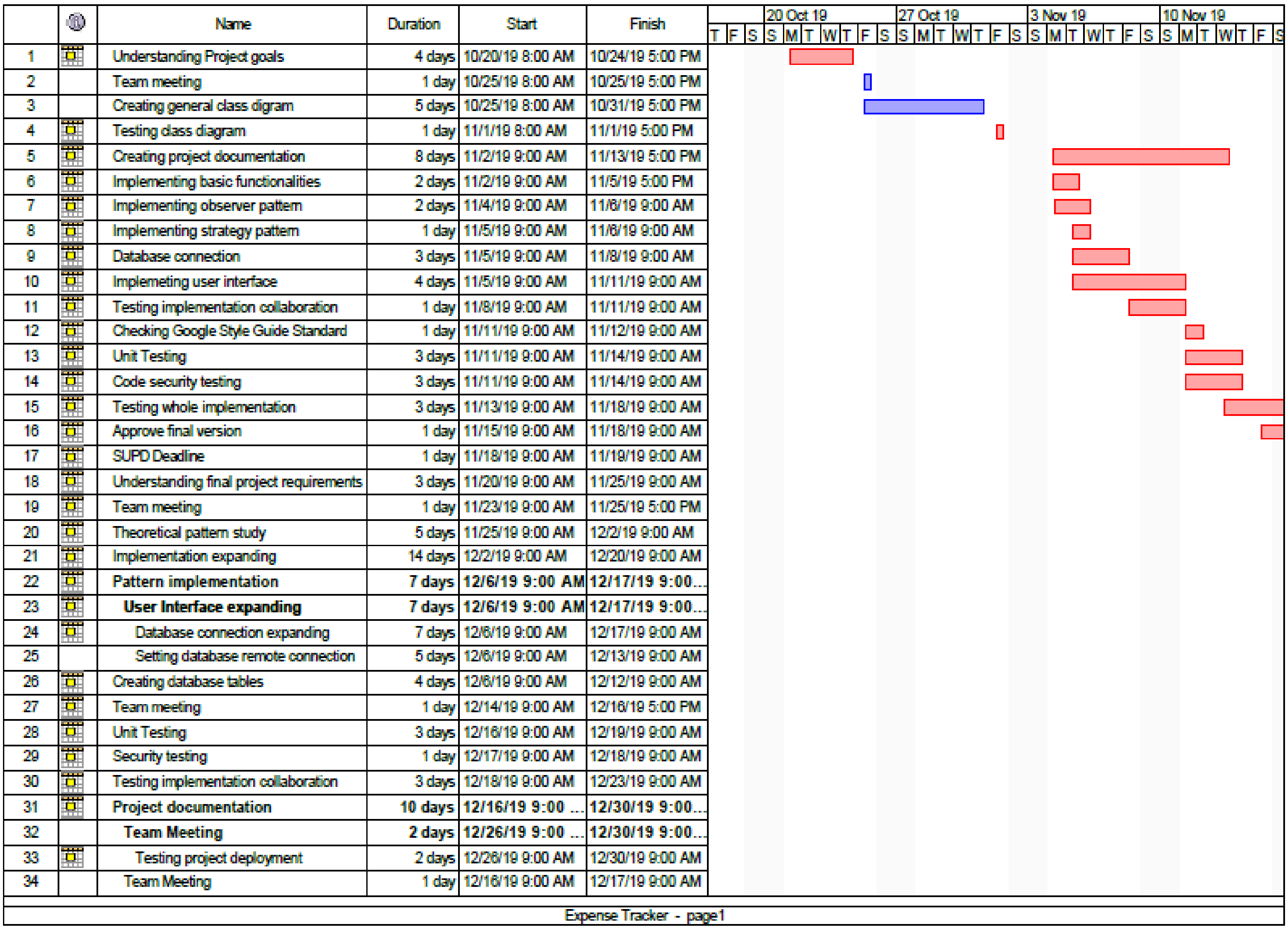
We created test package, with our Test Case Classes for Patterns Functionalities and Database Connection and Implementation. We used JUnit Tests and checked our code coverage. We tested multiple times our user interface and all functionalities manually and with our Main Class, that is a connection to the project main components. All tests are successfully evaluated. We have also tested security aspects, and provided input checks in backend and also in frontend.

3.3 Quality Requirements Coverage

We covered many internal and external software quality requirements.
As we worked in a team, it was very important to keep our code clean and readable, what was very useful for later changes and maintainability. For example, we implemented our application in the first line without database, to be sure that our patterns work correctly. We have tested it, and after it, because of our clean code, we connected the application with database easily.
Our code testability is also good, what we have also proved with many test cases.
We have also respected code usability, as we implemented very intuitive user interface, that is also very clean and practical for users. We have tested our Application usability with one external group of people who were not familiar with our Application. They have concluded, that they did not have problems understanding Application functionalities and how all buttons work.
Correctness was also important for our team, as we wanted to create correct system, that will add, update and delete data from our database and show it correctly on the screen.
Application robustness was very important, as we didn’t want our application to crush when user accidentally write wrong input. That’s why we have secured our code from many different types of wrong inputs and therefore caught many Exceptions.

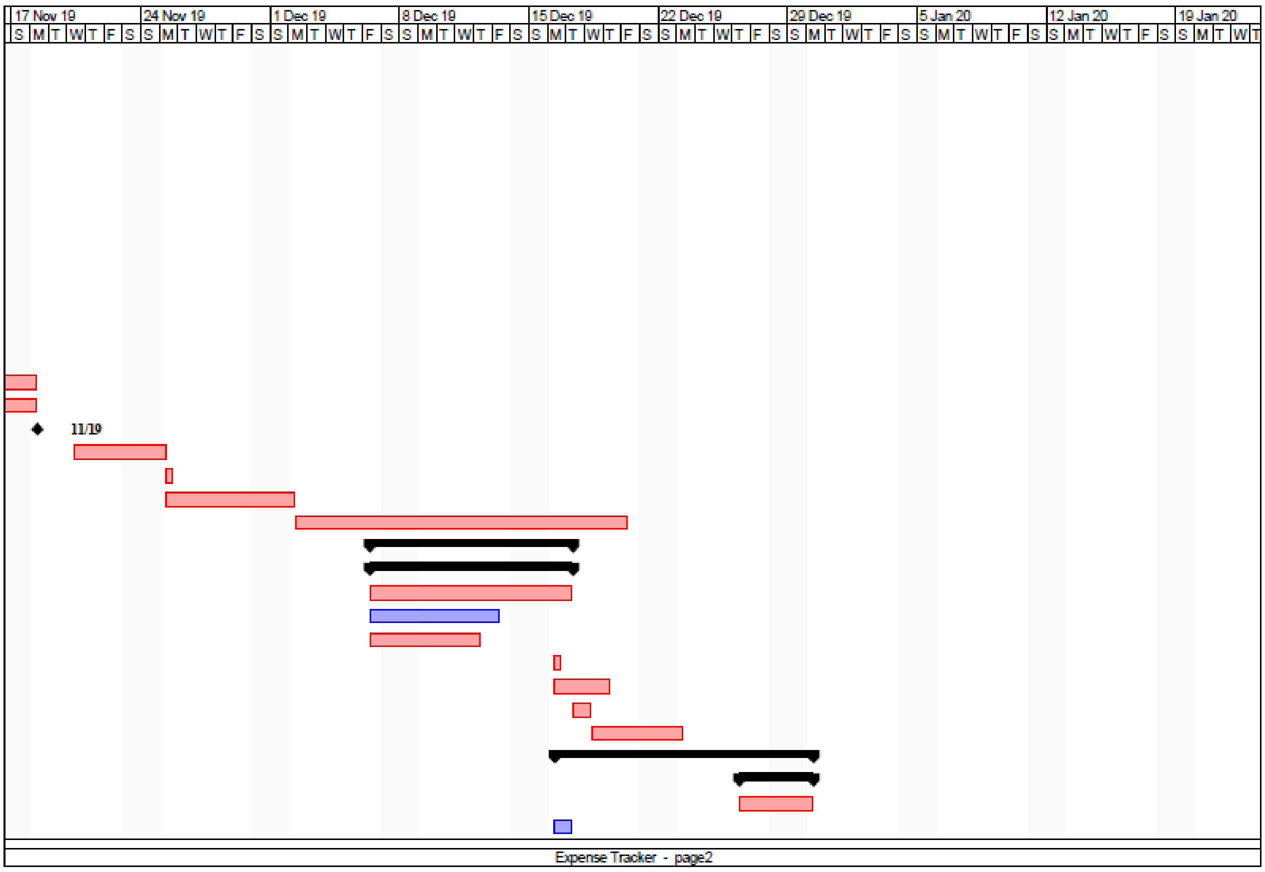
1 Team Contribution

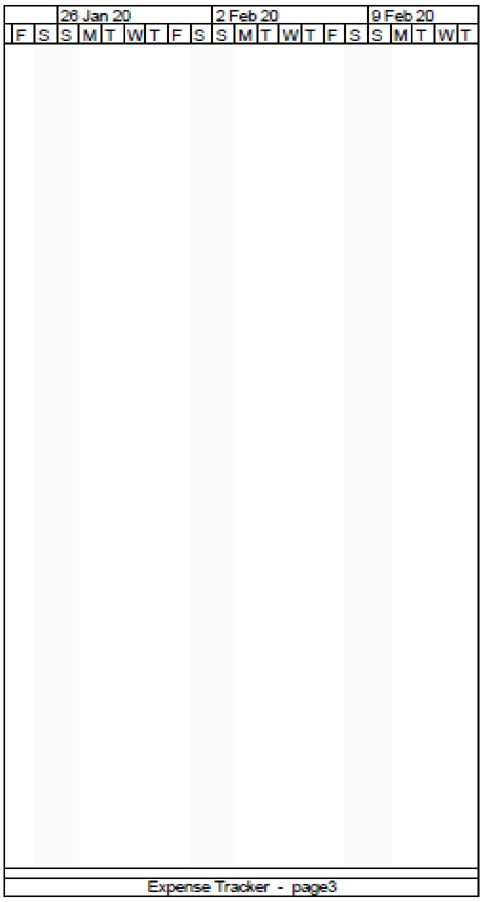
1.1 Project Tasks and Schedule



		Name	Duration	Start	Finish
1	▦	Understanding Project goals	4 days	10/20/19 8:00 AM	10/24/19 5:00 PM
2		Team meeting	1 day	10/25/19 8:00 AM	10/25/19 5:00 PM
3		Creating general class digram	5 days	10/25/19 8:00 AM	10/31/19 5:00 PM
4	▦	Testing class diagram	1 day	11/1/19 8:00 AM	11/1/19 5:00 PM
5	▦	Creating project documentation	8 days	11/2/19 9:00 AM	11/13/19 5:00 PM
6	▦	Implementing basic functionalities	2 days	11/2/19 9:00 AM	11/5/19 5:00 PM
7	▦	Implementing observer pattern	2 days	11/4/19 9:00 AM	11/6/19 9:00 AM
8	▦	Implementing strategy pattern	1 day	11/5/19 9:00 AM	11/6/19 9:00 AM
9	▦	Database connection	3 days	11/5/19 9:00 AM	11/8/19 9:00 AM
10	▦	Implemeting user interface	4 days	11/5/19 9:00 AM	11/11/19 9:00 AM
11	▦	Testing implementation collaboration	1 day	11/8/19 9:00 AM	11/11/19 9:00 AM
12	▦	Checking Google Style Guide Standard	1 day	11/11/19 9:00 AM	11/12/19 9:00 AM
13	▦	Unit Testing	3 days	11/11/19 9:00 AM	11/14/19 9:00 AM
14	▦	Code security testing	3 days	11/11/19 9:00 AM	11/14/19 9:00 AM
15	▦	Testing whole implementation	3 days	11/13/19 9:00 AM	11/18/19 9:00 AM
16	▦	Approve final version	1 day	11/15/19 9:00 AM	11/18/19 9:00 AM
17	▦	SUPD Deadline	1 day	11/18/19 9:00 AM	11/19/19 9:00 AM
18	▦	Understanding final project requirements	3 days	11/20/19 9:00 AM	11/25/19 9:00 AM
19	▦	Team meeting	1 day	11/23/19 9:00 AM	11/25/19 5:00 PM
20	▦	Theoretical pattern study	5 days	11/25/19 9:00 AM	12/2/19 9:00 AM
21	▦	Implementation expanding	14 days	12/2/19 9:00 AM	12/20/19 9:00 AM
22	▦	**Pattern implementation**	**7 days**	**12/6/19 9:00 AM**	**12/17/19 9:00...**
23	▦	**User Interface expanding**	**7 days**	**12/6/19 9:00 AM**	**12/17/19 9:00...**
24	▦	Database connection expanding	7 days	12/6/19 9:00 AM	12/17/19 9:00 AM
25		Setting database remote connection	5 days	12/6/19 9:00 AM	12/13/19 9:00 AM
26	▦	Creating database tables	4 days	12/6/19 9:00 AM	12/12/19 9:00 AM
27	▦	Team meeting	1 day	12/14/19 9:00 AM	12/16/19 5:00 PM
28	▦	Unit Testing	3 days	12/16/19 9:00 AM	12/19/19 9:00 AM
29	▦	Security testing	1 day	12/17/19 9:00 AM	12/18/19 9:00 AM
30	▦	Testing implementation collaboration	3 days	12/18/19 9:00 AM	12/23/19 9:00 AM
31	▦	**Project documentation**	**10 days**	**12/16/19 9:00 ...**	**12/30/19 9:00...**
32		**Team Meeting**	**2 days**	**12/26/19 9:00 ...**	**12/30/19 9:00...**
33	▦	Testing project deployment	2 days	12/26/19 9:00 AM	12/30/19 9:00 AM
34		Team Meeting	1 day	12/16/19 9:00 AM	12/17/19 9:00 AM

Expense Tracker - page1


17 Nov 19
24 Nov 19
1 Dec 19
8 Dec 19
15 Dec 19
22 Dec 19
29 Dec 19
5 Jan 20
12 Jan 20
19 Jan 20
11/19
Expense Tracker - page2


26 Jan 20
2 Feb 20
9 Feb 20
Expense Tracker - page3

Compared to our last Gantt Diagram, where we have planned our Final Submission, our work did not deviate much from the practice. What have changed, are our Team Meetings that were much often than planned, because we had to meet every time, checking out connection and the application live. We have also experienced throw out team work, that Team Meetings did bring as much effort that the individual work, because of the constant Brain Storming and Live Supporting.

1.2 Distribution of Work and Efforts

2 In the following tables we presented our distribution of work, responsibilities as like as time effort by each of our team member until this phase of our project.

3

Team member	*Task*	*Time*
Veljko Radunovic	Class Diagram, Frontend	34 h
Veljko Radunovic	Strategy pattern, Documentation	5 h
Veljko Radunovic	Strategy pattern, Class Diagram	5 h
Jelena Mikic	Rough architecture coding	10 h
Jelena Mikic	Pattern documentation	6 h
Jelena Mikic	Iterator, Composite Pattern	15 h
Nemanja Srdanovic	Observer, Factory, Template, Facade Pattern	12 h
Nemanja Srdanovic	User Interface and Implementation for Observer Pattern (Diagram Tab)	10 h
Nemanja Srdanovic	Architecture coding	5 h
Nemanja Srdanovic	Observer, Factory, Template, Facade Pattern, Class Diagram	3 h
Marija Zivanovic	Documentation,	4 h
Marija Zivanovic	Database connection, Database	2 h
Marija Zivanovic	User Interface, Basic, UML	2 h
All members	Meeting, Communication	4 h

Team member	Total time invested in the project
Veljko Radulovic	44 h
Jelena Mikic	44 h
Nemanja Srdanovic	44 h
Marija Zivanovic	44 h

A1 HowTo

This deployment process is based on Eclipse 1.8 development environment. To successfully deploy our current code, you will have to follow these following steps:

Running out Application by double clicking on .jar file

1. In the folder implementation\out\artifacts\implementation_jar you'll find the .jar file
2. By clicking on the .jar application will be launched. *
3. To use our default user, go to login page.
4. To successfully log in, type in username: "admin" and password "admin"
5. Application contain 5 different buttons on the left side of the screen(profile, account, transactions, diagrams and logout)
6. Each one of them will lead you to the new window of the application
7. On the "Profile" page you can add new categories or delete them from the user. List of all categories will be displayed in the list.
8. "Accounts" page is the similar to category page, adding new account with some balance or delete them from the user profile. All account will be displayed in the list on the page.
9. On "Transactions" page you can make new transactions or delete them from the history. To correctly make transaction, you need to put the name of the an existing account, an existing category, an amount and the description for that transaction. To be able to delete one of the transaction from the list, you'll need the put the id of the transaction and then press the delete button. The id of the transaction you can see in the displayed list above.
10. "Diagrams" page will display the stats of all account balances through the "bar chart" and all stats of categories through the "pie chart". Both Charts can display different period of those stats choosing the dates(from-to). In the right top corner you can select what chart you will to be displayed. Clicking on the load button it will display chart.

* in case the double click on the jar icon does not open the application, navigate in a command line to our folder: "implementation\out\artifacts folder where we saved our .jar file

Execute "java -jar ExpenseTracker.jar" in your command line to start the app.