Software Engineering 2
DEAD REPORT

Deadline Report

Team number:	33

Team member 1	
Name:	Vincent Zettl
Student ID:	01404131
E-mail address:	a01404131@unet.univie.ac.at

Team member 2	
Name:	Lukas Höwarth
Student ID:	01447910
E-mail address:	a01447910@unet.univie.ac.at

Team member 3	
Name:	Alexander Garber
Student ID:	01568645
E-mail address:	a01568645@unet.univie.ac.at

~~**Team member 4**~~	
~~**Name:**~~	~~TODO~~
~~**Student ID:**~~	~~TODO~~
~~**E-mail address:**~~	~~TODO~~

1 Final Design

1.1 Design Approach and Overview

We continued working on DEAD after we were done with SUPD. Most of the basic functionalities were incorporated during the SUPD development process so this time we mostly focused on patterns, coding practices, defensive programming, FR5,FR6 and the usability of our project. After reviewing the feedback we got for SUPD we had another team meeting, created a timetable and discussed upcoming tasks and challenges. We made multiple to-do lists regarding diffents topics, for example bugs we need to fix and implementation ideas for the patterns including more uml drafts.

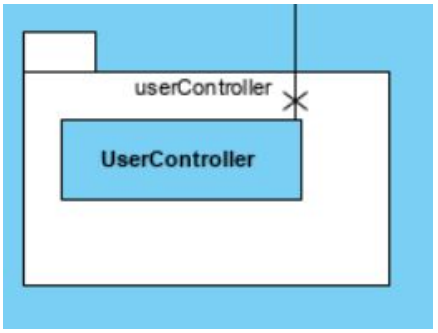
Most of the time developing was either spent together at university or by using discord to communicate changes or further implementations. This guaranteed fast support if someone hit a dead end.

1.1.1 Class Diagrams

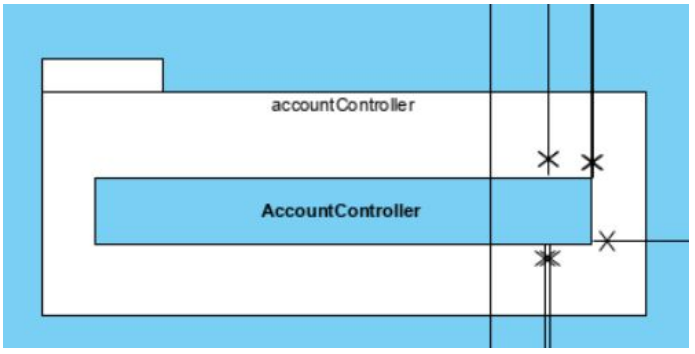
In the following section class diagrams of all modules will be shown. The main modules in this projects are “webController” and “webServices”. The class diagrams were modeled with Visual Paradigm 16 CE.

WebController

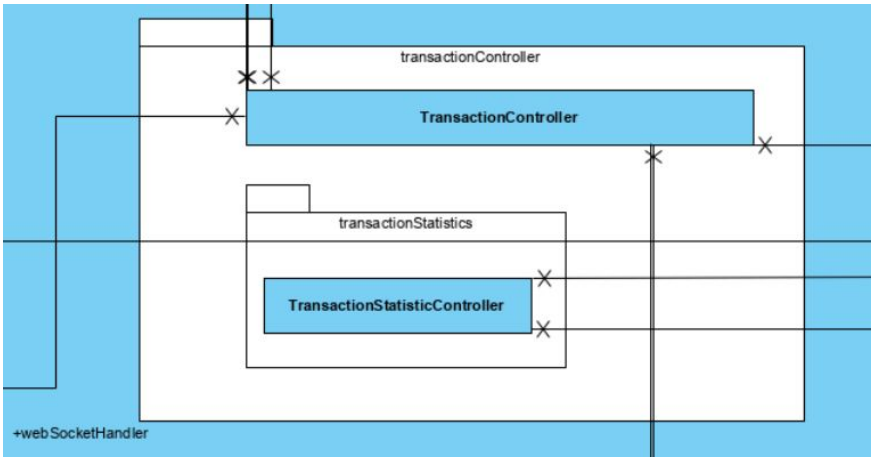
userController



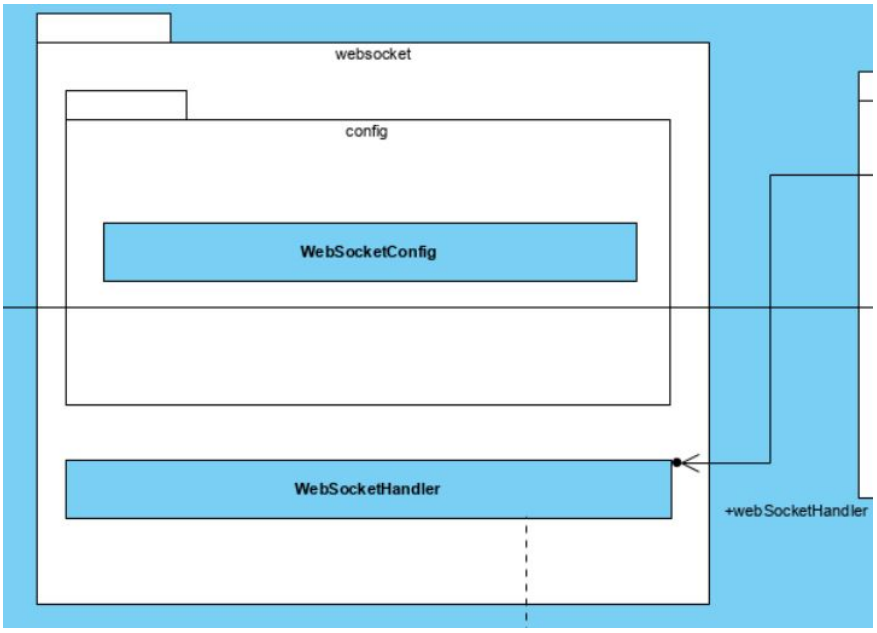
accountController



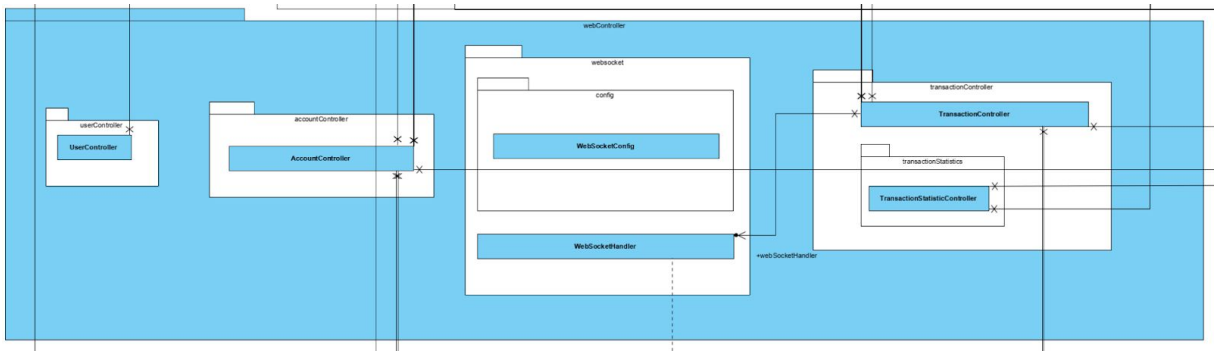
transactionController



websocket

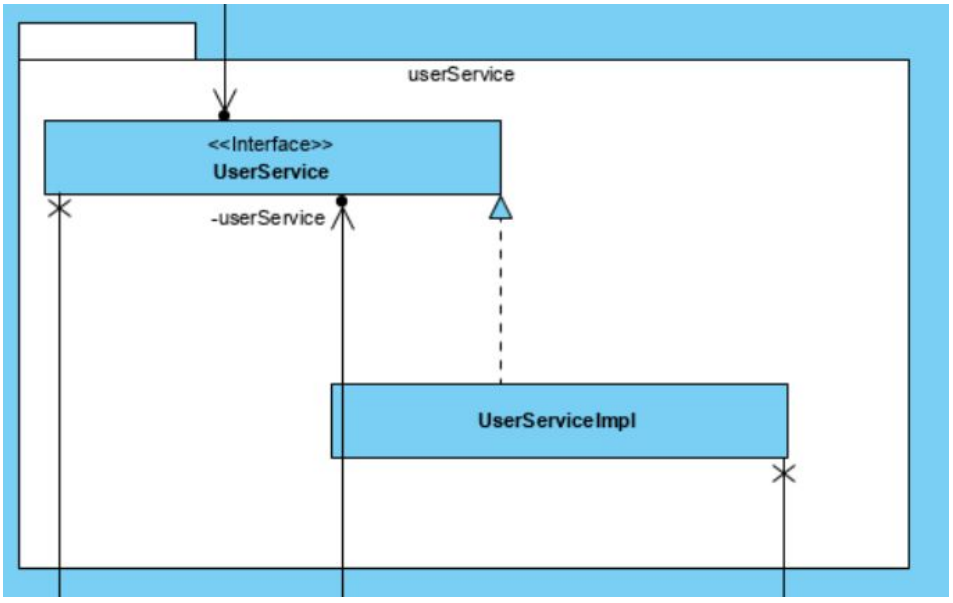


WebController(userController, accountController, transactionController, Websocket)

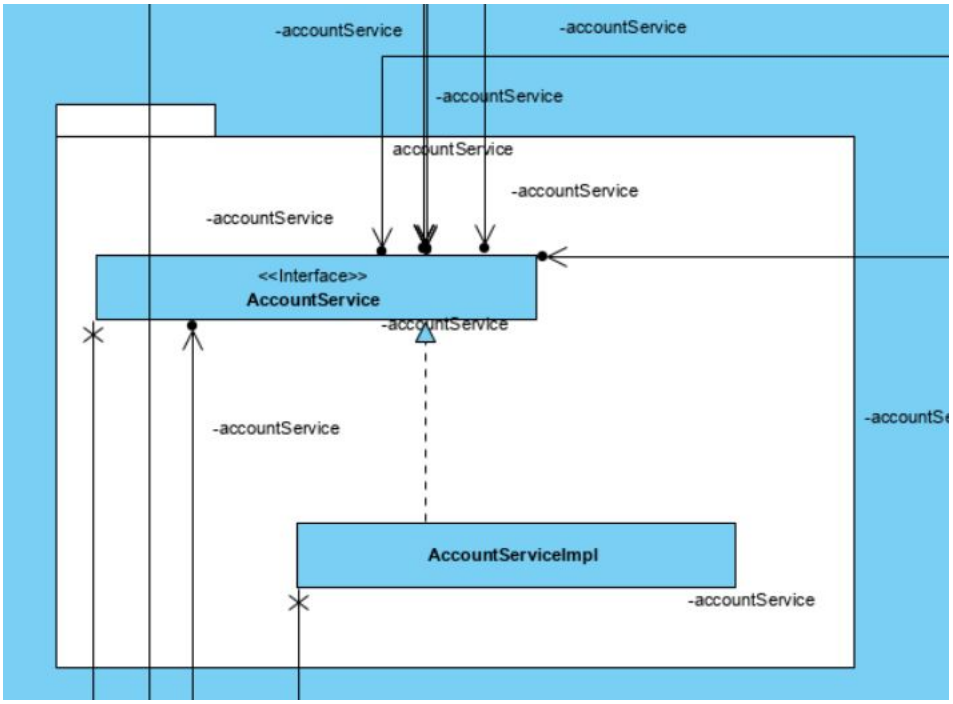


WebServices

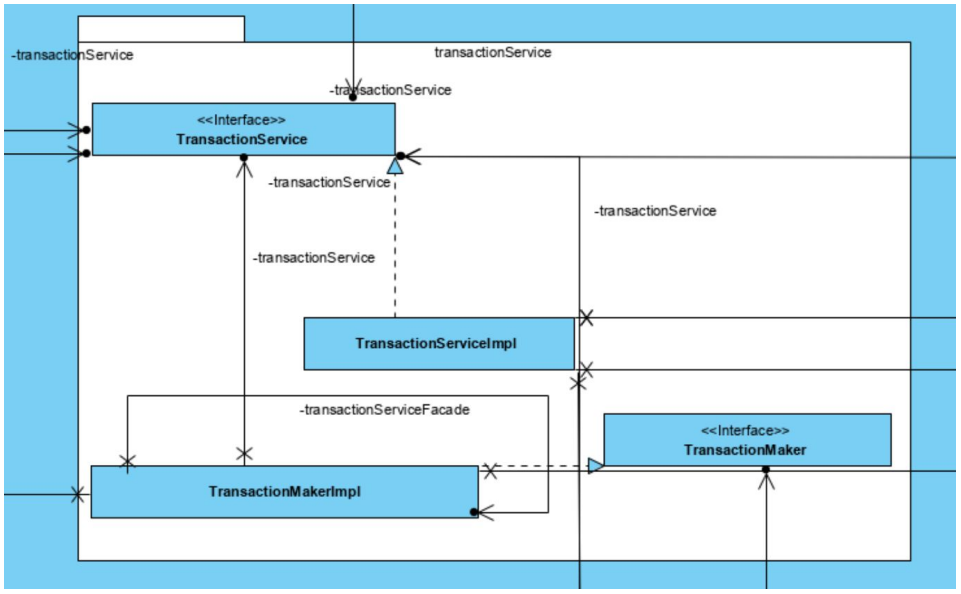
userService



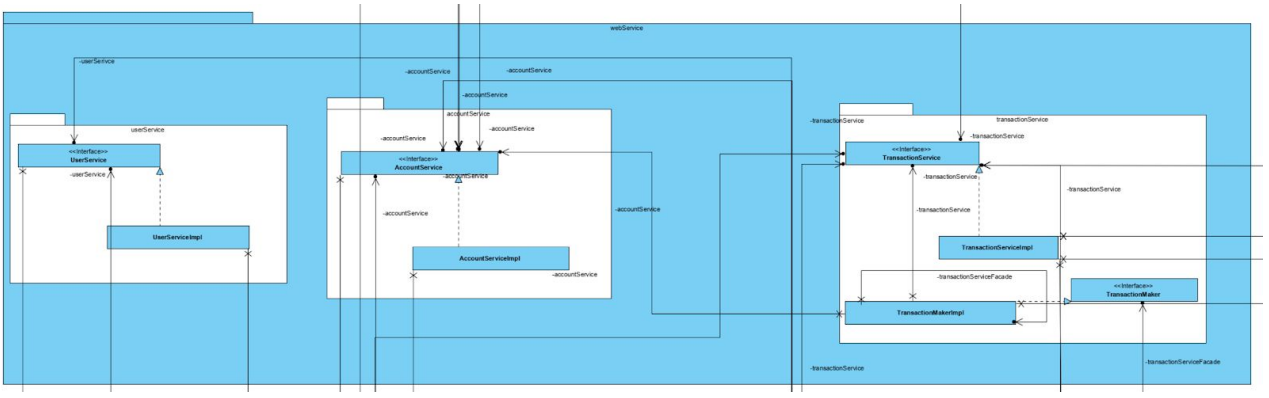
accountService



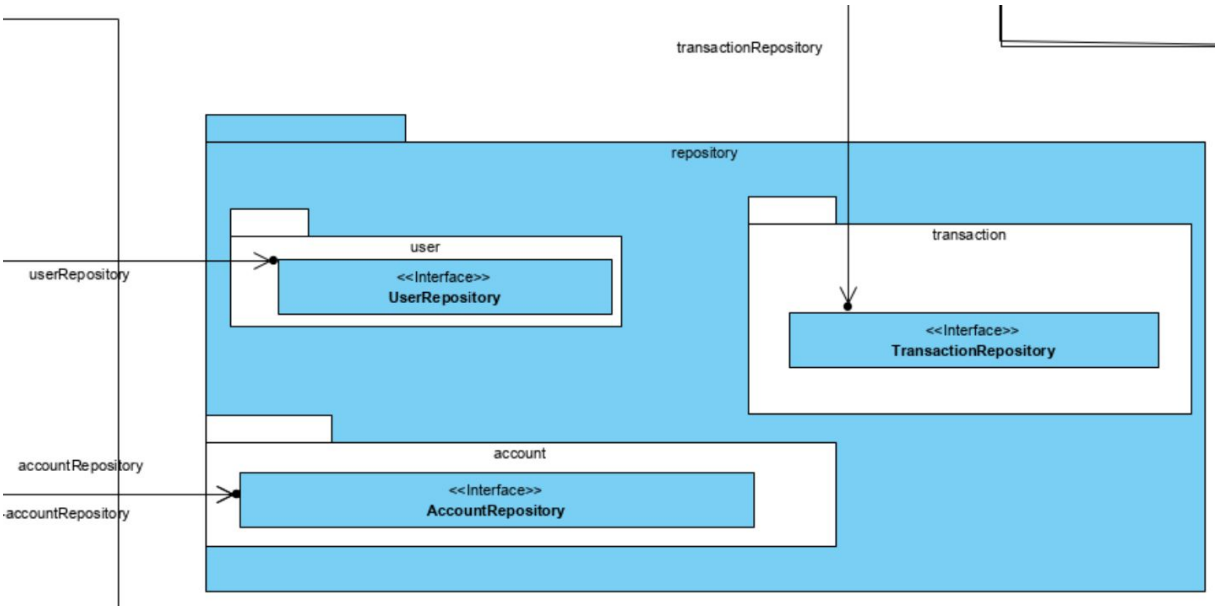
transactionService



webServices(transactionService, userService, accountService)

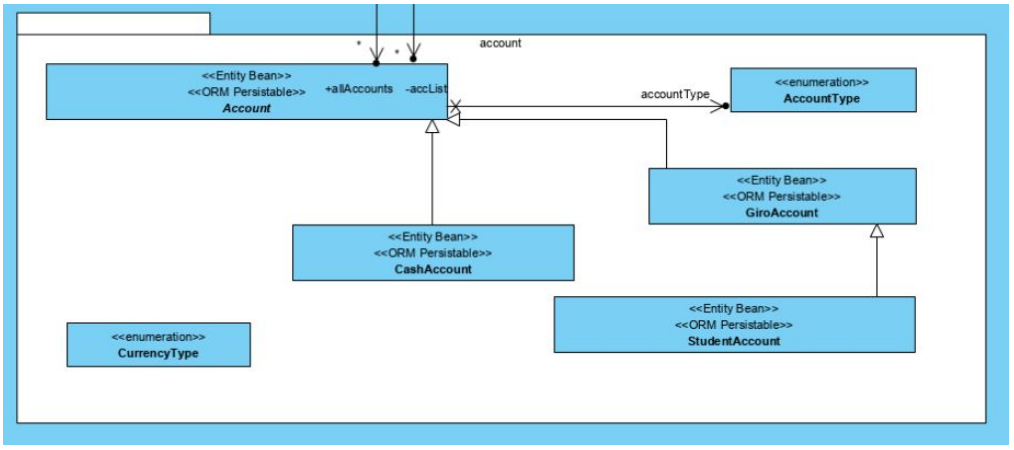


Repository

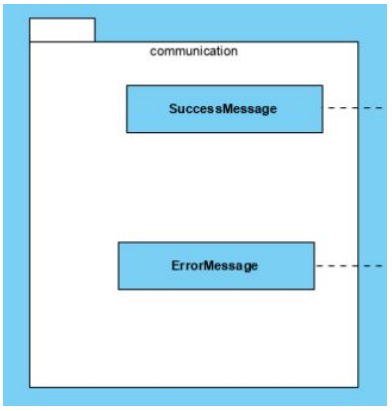

transactionRepository
repository
user
<<Interface>>
UserRepository
transaction
<<Interface>>
TransactionRepository
userRepository
account
accountRepository
<<Interface>>
AccountRepository
accountRepository

Model

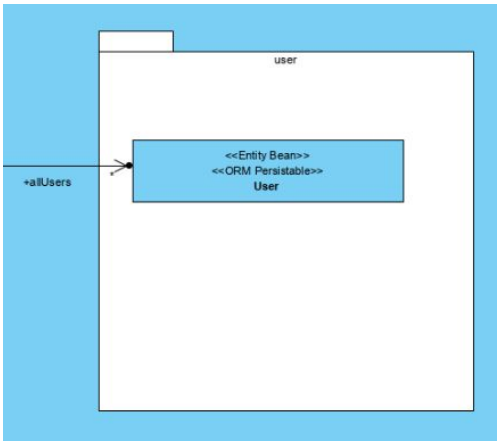
account



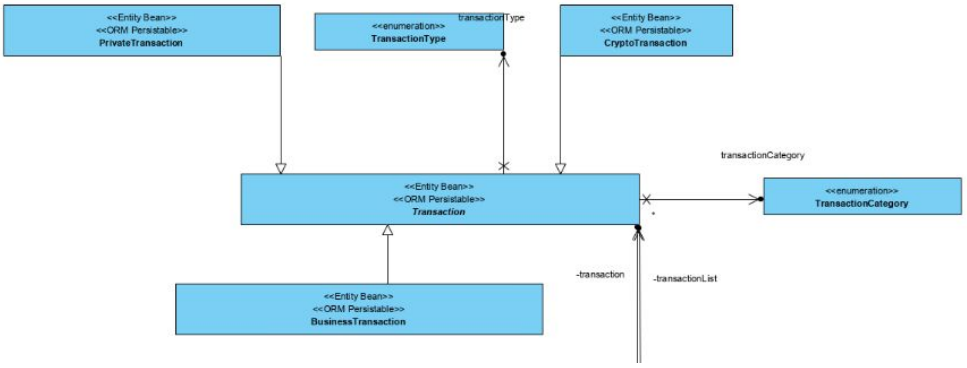
communication

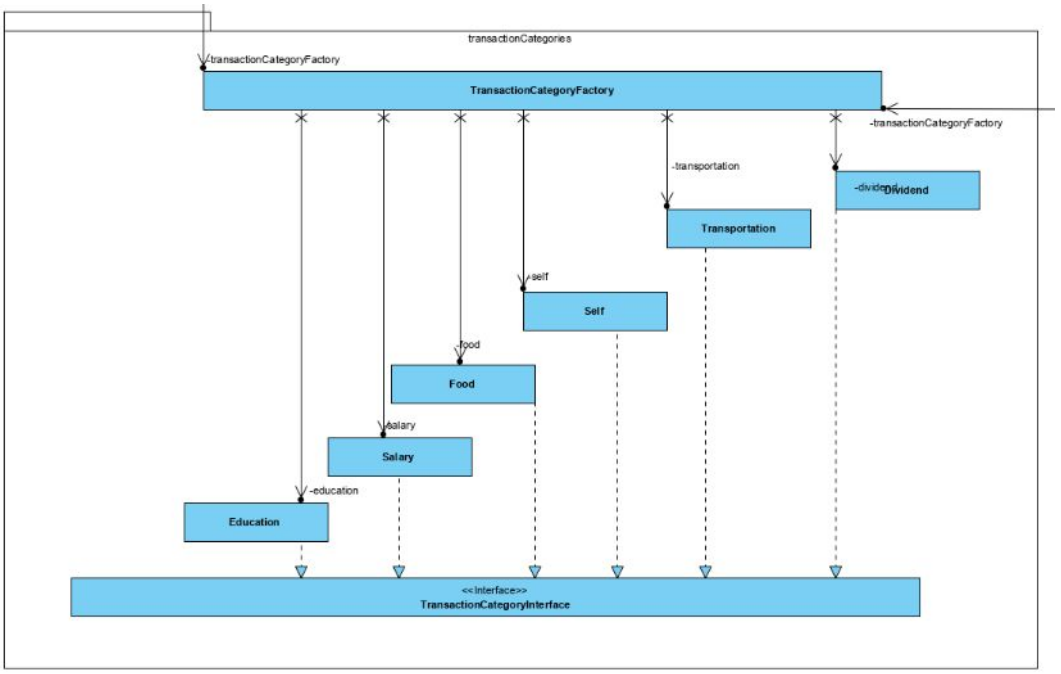


user

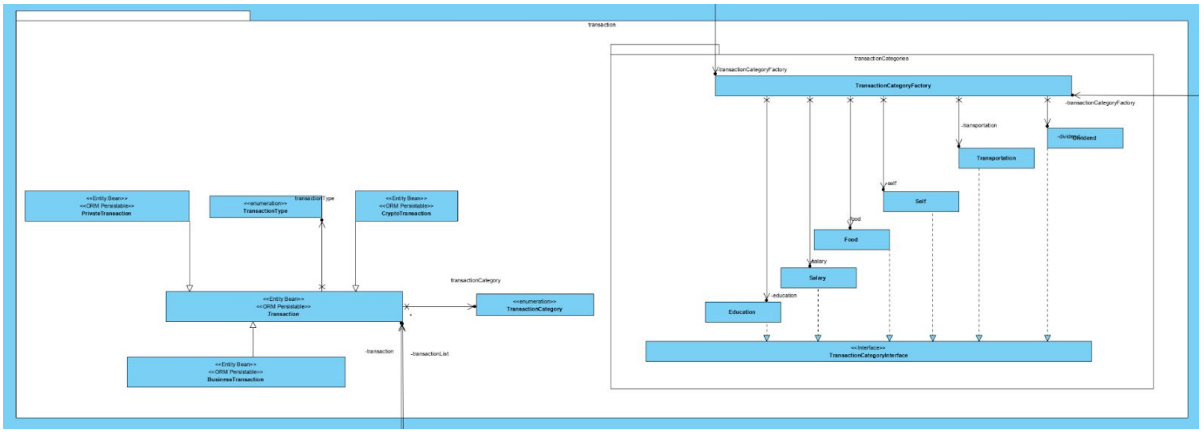


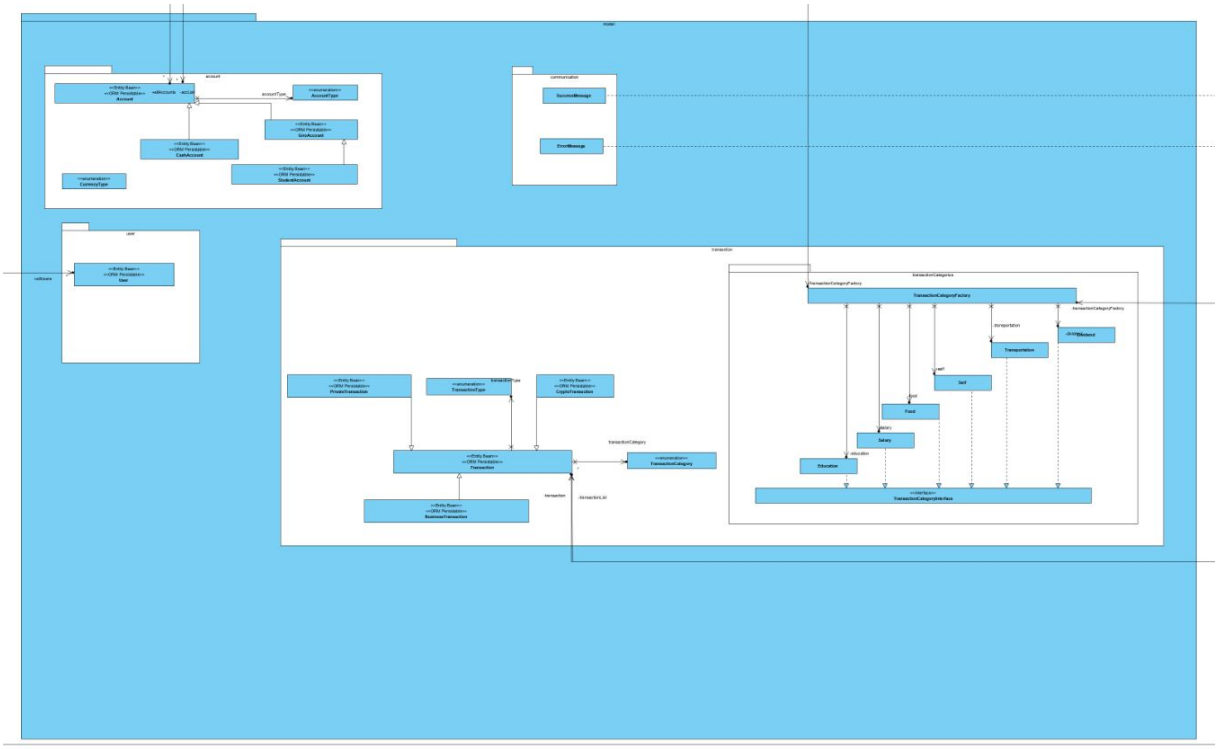
transaction



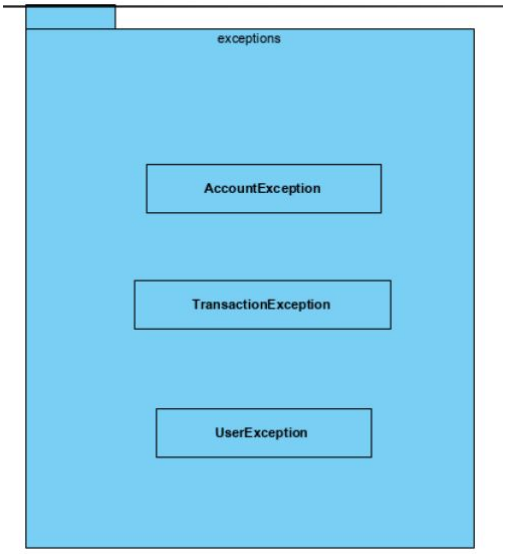


Complete transaction-module of Model



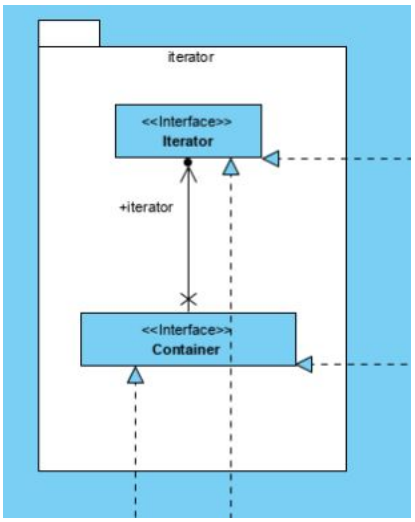


Exception

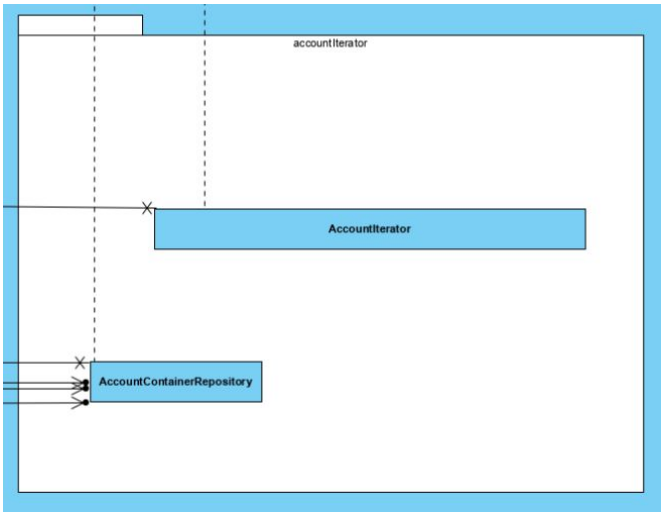

exceptions
AccountException
TransactionException
UserException

Logic

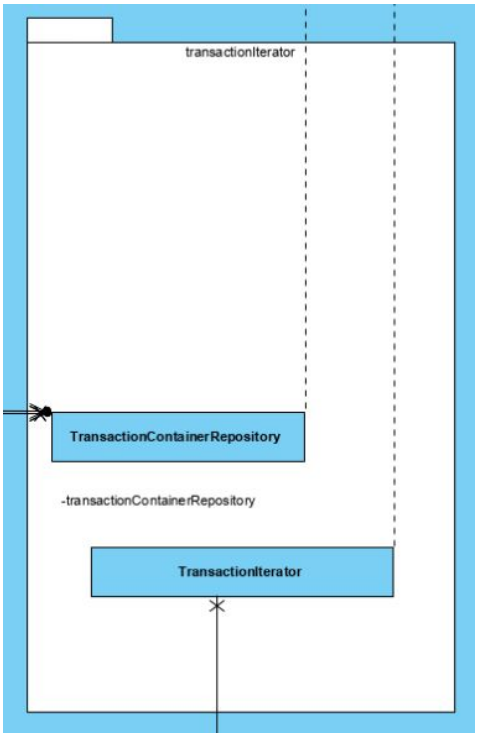
iterator



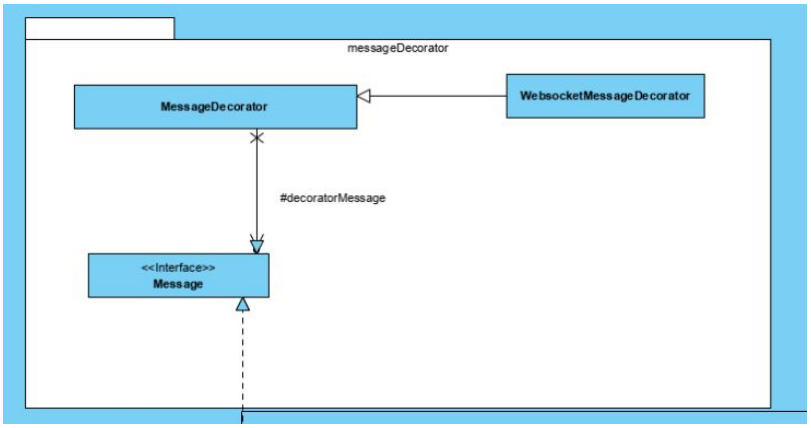
accountIterator



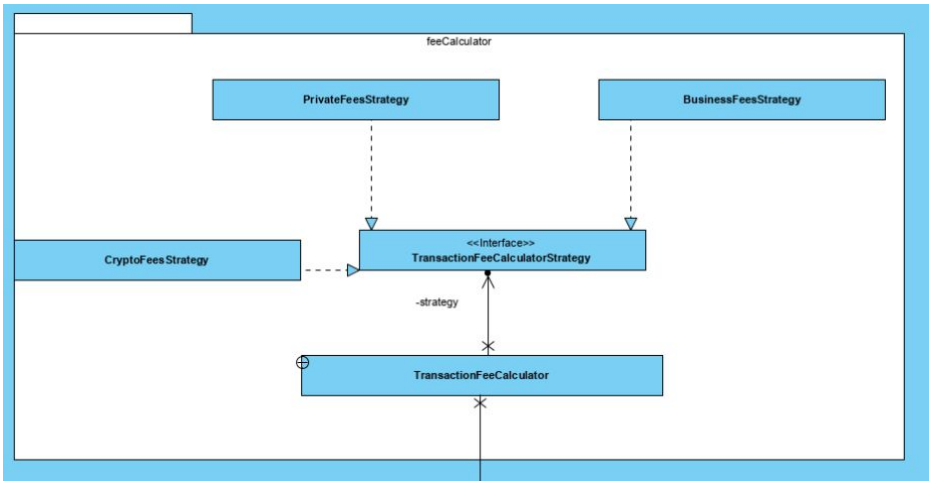
transactionIterator



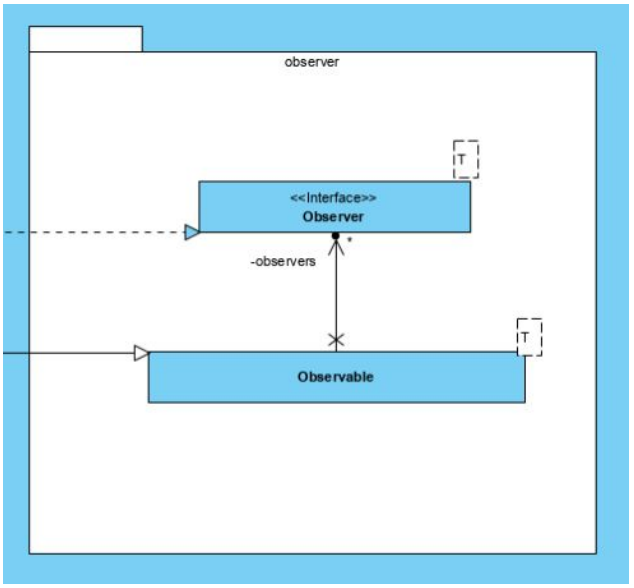
messageDecorator



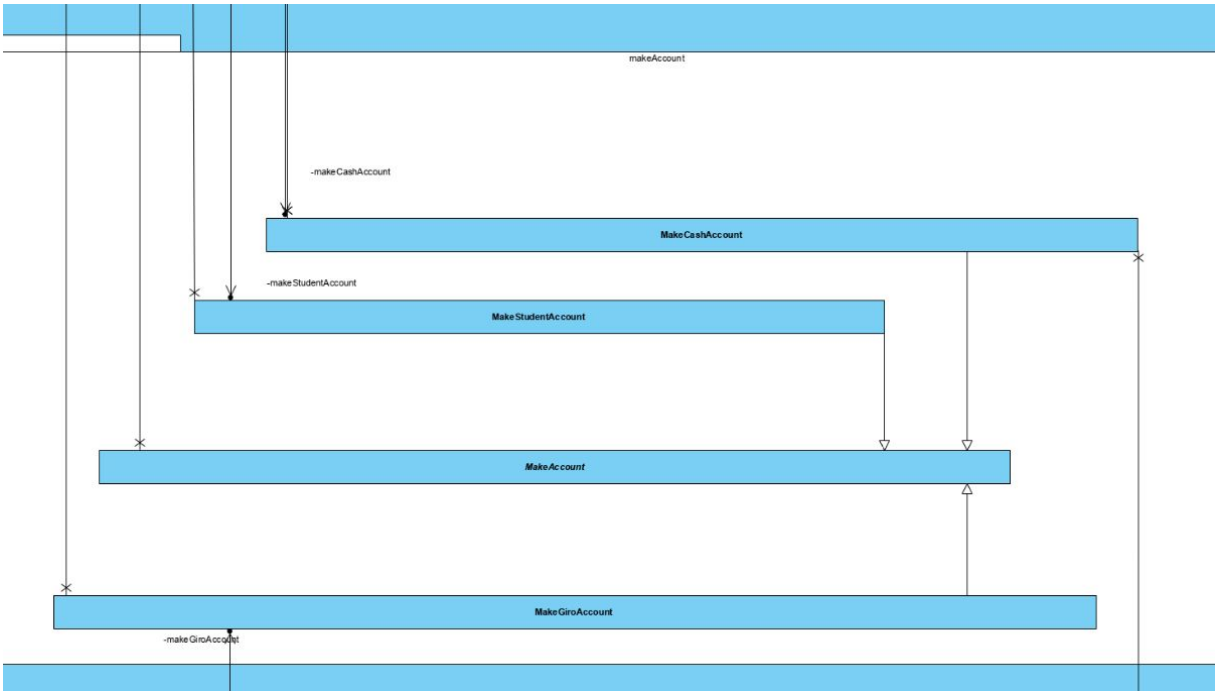
feeCalculator



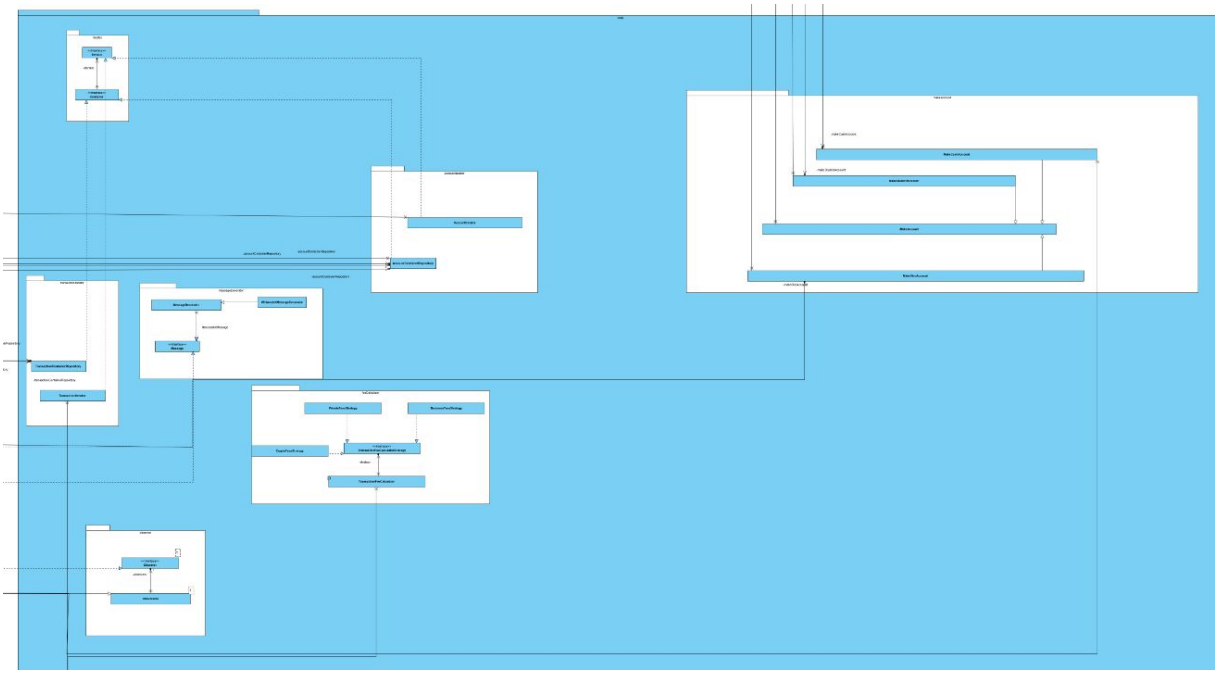
observer



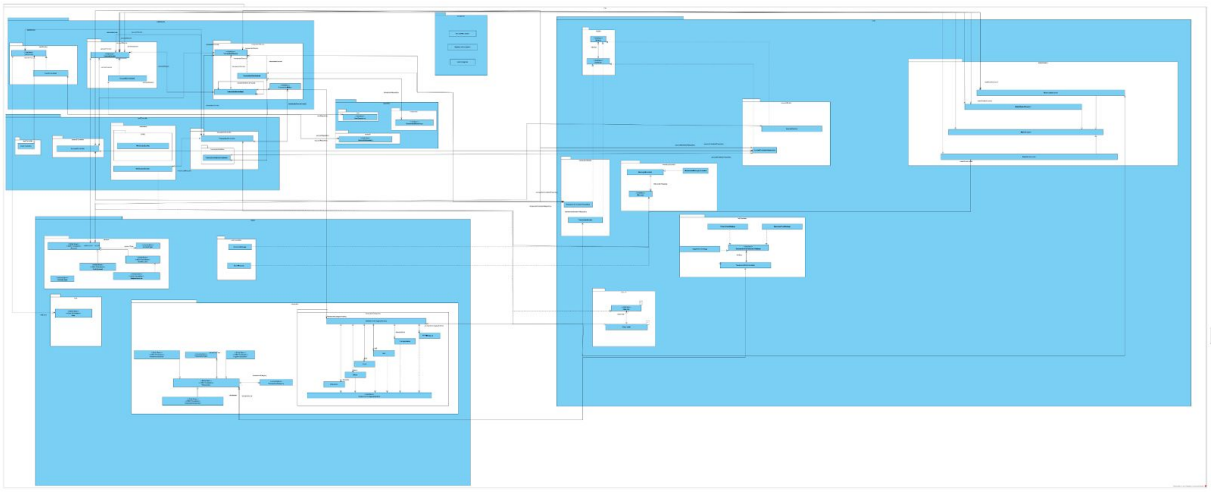
makeAccount



Complete logic module



Complete project class diagram



1.1.2 Technology Stack

As suggested for this course we use Java for our Project. We chose Java 11 because it's the latest version with Long Term Support. Because of that the documentation should already be solid.

https://www.oracle.com/technetwork/java/javase/downloads/jdk11-downloads-5066655.html

We are building a web based application using Rest(Spring Boot). This decision was made for multiple reasons. First and foremost all of our team members have experience with Rest. Nonetheless this decision wasn't merely made because of our prior knowledge but because of the advantages Rest brings to the table. It offers a uniform interface on top of being modular as a result of the separated Client and Server. This streamlines the process of implementing new features.

https://spring.io/

For version control, as suggested, Gitlab is used. It gives a good overview of who did what and makes it easy to revert back to an older version if bigger mistakes are made and the projects needs to be restructured from a certain point. Furthermore the Issues are a nice addition for coordinating tasks among the team members.

https://lab.swa.univie.ac.at/

Our building tool of choice is Maven. The reason for this is the use of XML for the configuration file to build the project. To run into less trouble and because of our prior experience from two of our team members with Maven we chose it over Gradle. We used the latest version 3.6.1 when we started our project and will stick with it unless there are major bugs fixed with newer versions.

https://maven.apache.org/

For the database we chose H2. We made this decision due to several reasons. Firstly because it's a SQL database which offers a nice transition from the created diagrams to the relational database, secondly it brings a browser based console application to the table thus making debugging and checking the database entries on the fly an easier process. Thirdly and most important for our project it is an in-memory database, meaning it stores the data locally ergo it's ideal for the requirements of the given application. Finally it has a really small footprint and therefore keeps our project from becoming too bloated as it only ways in with a 2MB jar file.

https://www.h2database.com/html/main.html

We discussed different IDEs to use. Finally we chose Intellij over the rest. All of us worked with Intellij and Eclipse for Java development prior to this project. Reasons why Intellij made the cut are that it is easy and straightforward to initialize and refactor the project. Code inspection and useful shortcuts make this IDE a blast to use. Because of GIT integration it's made easy to keep all the work up to date. In addition the code analysis and debugging tools help keep the implementation clean and hopefully keep bugs to a minimum.

https://www.jetbrains.com/idea/

During the first few hours of implementation we had to decide on how to keep our code uniformly. We decided to use the Google Java Style Guide as it seemed to be one of the, if not even the most commonly used style out there.

https://www.h2database.com/html/main.html

For the creation of the diagrams Visual Paradigm Community Edition is used. It has a wide variety of different diagram options to choose from. Especially the possibility to have a template for modeling Rest functionalities is a nice inclusion. In prior courses we had to work with BeeUp which didn't make the cut. On the one side Visual Paradigm offers better stability, as a result of that it is less likely to lose our progress. On the other hand it offers the possibility for collaboration which, for a group project, plays right into our hands.

https://www.visual-paradigm.com

For Testing JUNIT5 was used in combination with the Spring Boot Starter Test. This was mainly chosen because it has good integration with Intellij and is a common way to write tests in java. The Spring Boot Starter Test was necessary to work with the spring annotations.

https://junit.org/junit5/docs/current/user-guide/

https://spring.io/guides/gs/testing-web/

1.2 Major Changes Compared to SUPD

Most of the basic functionality was already implemented for SUPD. Major changes were made because we had to incorporate more design patterns. In retrospect we should have laid out where to use the patterns at the start of the project. This would have saved us from some of the refactoring we had to do.

Regarding the feedback the strategy pattern was adjusted to be in one package and be defensive by throwing an exception if the transaction is null. Furthermore the name was changed to better represent the used pattern. The implementation for FR2 was reworked to create new possibilities for patterns.

Some of the database tables were altered a little bit to better fit the additional functional requirements. Overall it was a goal to keep the code and implementation as structured and well defined as possible.

The recommendation regarding the big UML diagram was incorporated by splitting it up in smaller parts and using a “zoom-in” approach.

One major change is the usage of the decorator-pattern instead of composite pattern. The implementation of the composite-pattern was designated for the User to have a UserList of all other Users. Due to the Spring-Framework and Hibernate, some errors occurred when the User-Model got a list of all other users. Hibernate’s persistent storage manager led to a recursive deadlock when users were added to the userlist. Therefore, we decided to use the decorator-pattern for websocket messages instead of the composite-pattern.

Some of the package structure was rearranged to fit the underlying logic better. New packages were added to keep our structure clean and keep the project manageable.

The rest of the requirements were built up upon the existing SUPD implementation. New classes and modules were added for the remaining requirements.

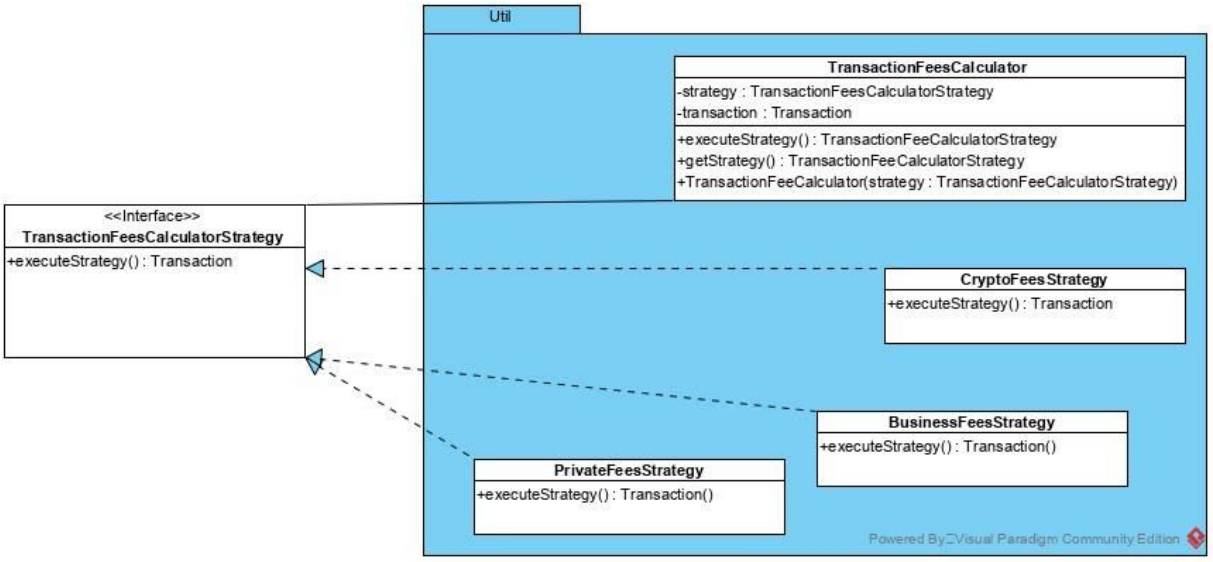
Regarding the Frontend there were no big functional changes, besides the introduction of the charts. Sorting functionality was extended. Regarding the Design we decided to keep a clean and professional design. The site was completely rebuilt, inspired by the W3 Social Media Template, using collapsable Cards.

1.3 Design Patterns

1.3.1 Strategy Pattern

The strategy pattern makes it possible to achieve what's called “encapsulate what varies” in object oriented programming. It helps keep the code modular. If for example we want to add another Strategy to calculate the fees differently without this pattern we would have every of those calculation algorithms in one file and one class. This violates the principle that new functionality should be added by writing new code instead of changing existing.

Using the strategy pattern it’s now possible to add a new class that implements the interface, write one new algorithm to calculate the fees and we are set.



Strategy Interface

```
public interface TransactionFeeCalculatorStrategy {
 Transaction executeStrategy(Transaction transaction);
}
```

Context

```
public class TransactionFeeCalculator {
 private TransactionFeeCalculatorStrategy strategy;
 private Transaction transaction;

 public TransactionFeeCalculator(TransactionFeeCalculatorStrategy strategy) {
   this.strategy = strategy;
 }
 public Transaction executeStrategy() {
   return strategy.executeStrategy(transaction);
 }

 public TransactionFeeCalculatorStrategy getStrategy() {
   return strategy;
 }
```

```
 public void setStrategy(TransactionFeeCalculatorStrategy strategy) {
   this.strategy = strategy;
 }

 public Transaction getTransaction() {
   return transaction;
 }
public void setTransaction(Transaction transaction) {
   try {
     if (transaction == null) {
       throw new Exception("Transaction is Null");
     }
     this.transaction = transaction;
   } catch (Exception e) {
     System.out.println(e.getMessage());
   }
 }
}
```

Concrete Strategy

```
public class CryptoFeesStrategy implements TransactionFeeCalculatorStrategy {
 @Override
 public Transaction executeStrategy(Transaction transaction) {
   System.out.println("Using CryptoFeesStrategy Calculation Method");

   float fees = 0.0025f;

   transaction.setAmount(transaction.getAmount() + transaction.getAmount() * fees);

   return transaction;
 }
}
```

Concrete Strategy

```
public class BusinessFeesStrategy implements TransactionFeeCalculatorStrategy {

 @Override
 public Transaction executeStrategy(Transaction transaction) {
   System.out.println("Using BusinessFeesStrategy Calculation Method");
   float fees = 0.015f;

   transaction.setAmount(transaction.getAmount() + transaction.getAmount() * fees);

   return transaction;
 }
}
```

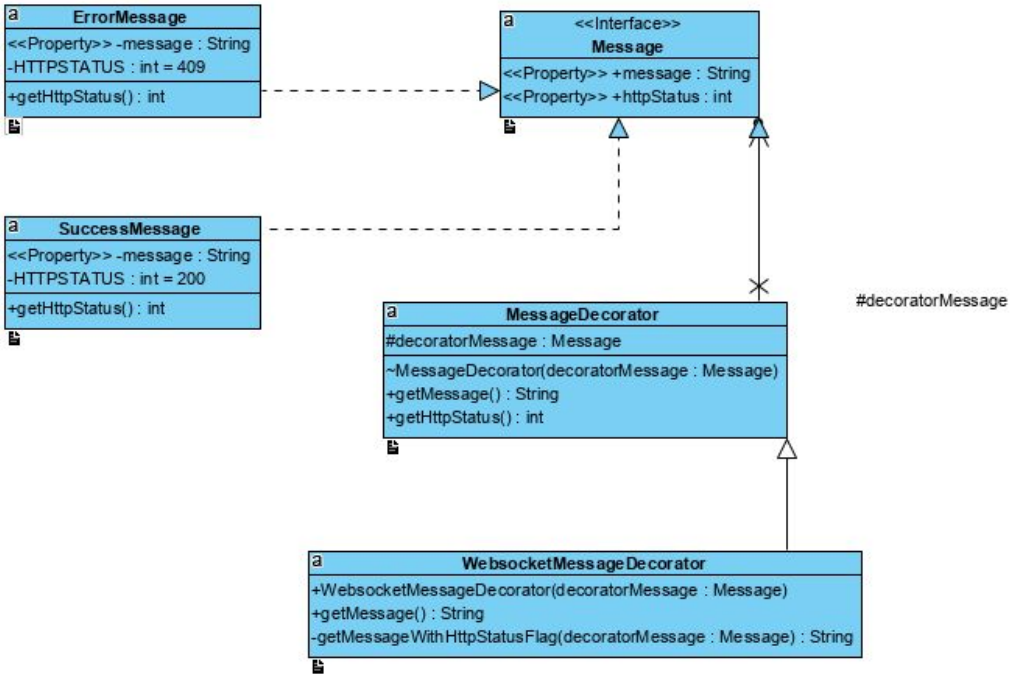
Concrete Strategy

```
public class PrivateFeesStrategy implements TransactionFeeCalculatorStrategy {


 @Override
 public Transaction executeStrategy(Transaction transaction) {
   System.out.println("Using noFees Calculation Method");
   return transaction;
 }
}
```

1.3.2 Decorator Pattern

The decorator pattern is used to attach new behaviour to objects by wrapping these objects which contain the behaviour. This pattern is taking usage of an interface which defines the behaviour to be implemented. In our project we wrapped messages for the communication channel through websockets.

With the usage of the decorator pattern it is possible to extend an object's behavior without creating a new subclass either it is possible to change the usage of objects during the runtime. Nevertheless, we need to mention that the order of the decorators is fixed and it is hard to remove a specific one from this stack. This pattern meets the single responsibility principle because it divides one big class into multiple smaller classes.



```
public interface Message {
 String getMessage();
 int getHttpStatus();
}
```

Therefore we created an interface “message” to define the behaviour of a message-object.

```
public class ErrorMessage implements Message {

 private String message;
 private final int HTTPSTATUS = 409;

 public void setMessage(String message) {
   this.message = message;
 }
 @Override
 public String getMessage() {
   return message;
 }
 @Override
 public int getHttpStatus() {
   return HTTPSTATUS;
 }
}
```

In the code-snipped above, we set the behaviour of an ErrorMessage by setting the message and the final HTTPStatusCode of the Message.

```
public class SuccessMessage implements Message {

 private String message;
 private final int HTTPSTATUS = 200;

 public void setMessage(String message) {
   this.message = message;
 }

 @Override
 public int getHttpStatus() {
   return HTTPSTATUS;
 }
 @Override
 public String getMessage() {
   return message;
 }
}
```

The same procedure is used to define the behaviour of the “SuccessMessage”.

```
public class MessageDecorator implements Message {

 protected Message decoratorMessage;
```

```
 MessageDecorator(Message decoratorMessage) {
   this.decoratorMessage = decoratorMessage;
 }

 @Override
 public String getMessage() {
   return decoratorMessage.getMessage();
 }
 @Override
 public int getHttpStatus() {
   return decoratorMessage.getHttpStatus();
 }
}
```

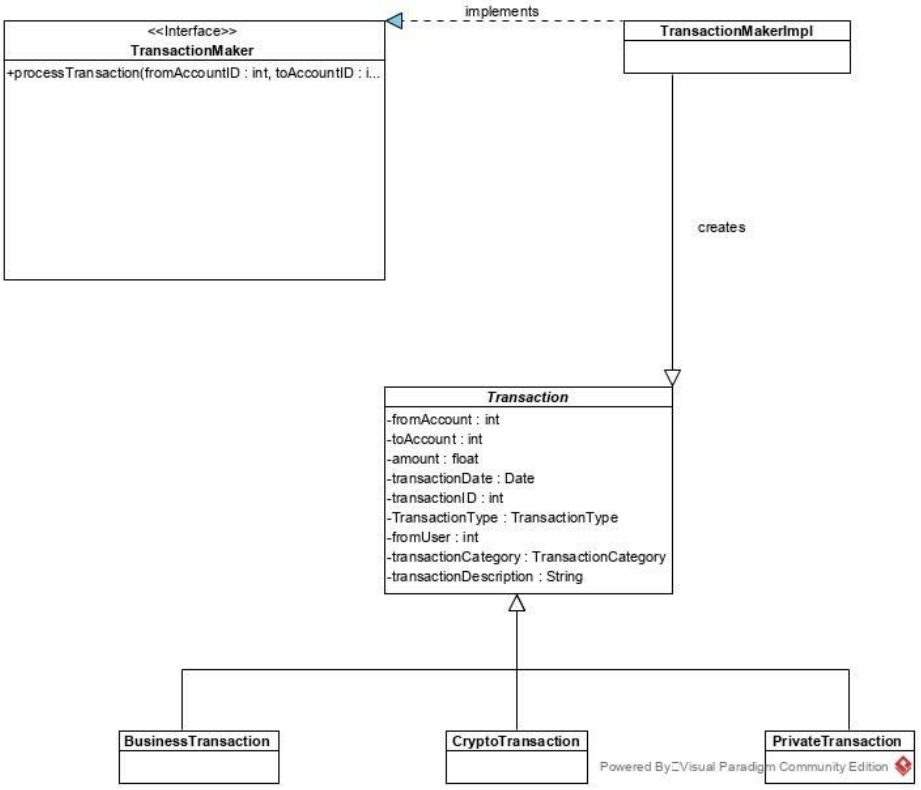
The MessageDecorator is used to build a Decorator-Class which defines the basic structure of the “decorations”.

```
public class WebsocketMessageDecorator extends MessageDecorator {

 public WebsocketMessageDecorator(Message decoratorMessage) {
   super(decoratorMessage);
 }

 @Override
 public String getMessage() {
   return getMessageWithHttpStatusFlag(decoratorMessage) + decoratorMessage.getMessage();
 }

 private String getMessageWithHttpStatusFlag(Message decoratorMessage) {
   return "WS-Message: " + decoratorMessage.getHttpStatus() + " | ";
 }
}
```

In a bigger project more different decorators could be created for example a session decorator.

1.3.3 Facade Pattern

The facade pattern is used to streamline certain tasks of a system. In theory the system is therefore easier to use for clients to access it. In our project the pattern was used to process different kinds of transactions. The TransactionMaker class redirects each call to make a transaction to the corresponding class. It was used to reduce the dependencies and get a well structured design.



Strategy Interface

```
public interface TransactionMaker {
 Transaction processTransaction(
     int fromAccountID,
     int toAccountID,
     int fromUserID,
     float amount,
     String transactionType,
     String transactionDate,
     String transactionDescription,
     String transactionCategory);
}
```

Concrete Strategy

```
@Component
 @Autowired private TransactionMakerImpl transactionServiceFacade;
 @Autowired private AccountService accountService;
 @Autowired private TransactionService transactionService;
 @Autowired private TransactionCategoryFactory transactionCategoryFactory;

 public TransactionMakerImpl() {}

 public Transaction processTransaction(
     int fromAccountID,
     int toAccountID,
     int fromUserID,
     float amount,
     String transactionType,
     String transactionDate,
     String transactionDescription,
     String transactionCategory) {

   if (accountService.accountExist(fromAccountID) && accountService.accountExist(toAccountID)) {
     TransactionCategory category = TransactionCategory.valueOf(transactionCategory);

     TransactionType type = TransactionType.valueOf(transactionType);

     java.sql.Date stringToDate = Date.valueOf(transactionDate);

     Transaction transaction = null;
     switch (transactionType) {
       case "PRIVATE":
         transaction =
             new PrivateTransaction(
                 fromAccountID,
                 toAccountID,
                 fromUserID,
                 amount,
                 type,
                 stringToDate,
                 transactionDescription,
                 category);
         TransactionFeeCalculatorStrategy privateStrategy = new PrivateFeesStrategy();
         TransactionFeeCalculator privateCalculator =
             new TransactionFeeCalculator(privateStrategy);
         privateCalculator.setTransaction(transaction);
         transaction = privateCalculator.executeStrategy();
         break;
       case "BUSINESS":
         transaction =
             new BusinessTransaction(
                 fromAccountID,
                 toAccountID,
                 fromUserID,
                 amount,
                 type,
                 stringToDate,
                 transactionDescription,
                 category);
         TransactionFeeCalculatorStrategy businessStrategy = new BusinessFeesStrategy();
         TransactionFeeCalculator businessCalculator =
             new TransactionFeeCalculator(businessStrategy);
         businessCalculator.setTransaction(transaction);
```

```
          transaction = businessCalculator.executeStrategy();
          break;
        case "CRYPTO":
          transaction =
              new CryptoTransaction(
                  fromAccountID,
                  toAccountID,
                  fromUserID,
                  amount,
                  type,
                  stringToDate,
                  transactionDescription,
                  category);
          TransactionFeeCalculatorStrategy cryptoStrategy = new CryptoFeesStrategy();
          TransactionFeeCalculator cryptoCalculator = new TransactionFeeCalculator(cryptoStrategy);
          cryptoCalculator.setTransaction(transaction);
          transaction = cryptoCalculator.executeStrategy();
          break;
        default:
          break;
      }

      transaction.setTransactionCategoryIcon(
          transactionCategoryFactory.getCategory(transactionCategory).drawCategoryIcon());

      Transaction temp_tran = transactionService.newTransaction(transaction);
      System.out.println("Save transaction");
      System.out.println(temp_tran);

      if (temp_tran != null) {
        System.out.println("returntype transaction not null " + temp_tran);
        return temp_tran;

      } else {
        System.out.println("transaction not successfull returned null");
      }
    }
    return null;
  }
}
```

Abstract Class

```
@Entity
public abstract class Transaction {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  int transactionID;

  float amount;
  int fromAccountID;
  int toAccountID;
  int fromUserID;
  TransactionType transactionType;
  java.sql.Date transactionDate;
  String transactionDescription;
  TransactionCategory transactionCategory;
  String transactionCategoryIcon;
}
```

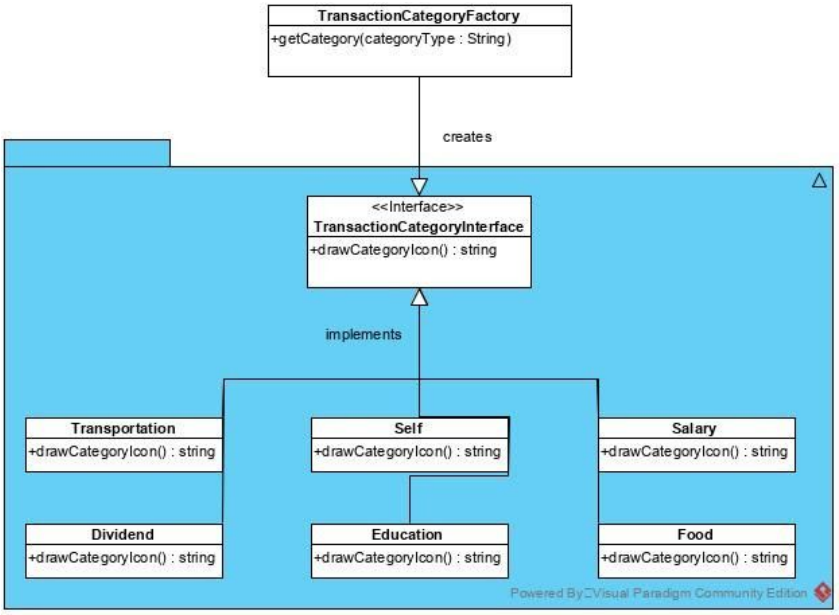
Concrete Class Example

```
@Entity
public class BusinessTransaction extends Transaction {
 public BusinessTransaction() {
   super();
 }
 public BusinessTransaction(
     int fromAccountID,
     int toAccountID,
     int fromUserID,
     float amount,
     TransactionType transactionType,
     Date transactionDate,
     String transactionDescription,
     TransactionCategory transactionCategory) {
   super(
       fromAccountID,
       toAccountID,
       fromUserID,
       amount,
       transactionType,
       transactionDate,
       transactionDescription,
       transactionCategory);
 }
}
```

As the classes CryptoTransaction, PrivateTransaction and BusinessTransaction are built in a similar manner, we decided to only show the BusinessTransaction as an example.

1.3.4 Factory Method Pattern

By using the factory method pattern, the creation of new objects is handled by a so-called factory. Therefore, the logic of the creation stays hidden from the client. It's easier to extend the factory to produce different products like in our case adding new categories independently from the rest of the code. With this pattern tight coupling is further avoided, the single responsibility principle and the open close principle is achieved because the code factory can be moved to a seperate part of the project and adding new categories is unlikely to break existing functionalities. We used this pattern in our project to mark transactions with a variety icons for specific categories. It's easy and safe to add or remove new categories. As mentioned above it should further improve the quality of our code by supporting these coding practices.



Interface

```
public interface TransactionCategoryInterface {
 String drawCategoryIcon();
}
```

Concrete Strategy

```
@Component
public class TransactionCategoryFactory {
 @Autowired private Dividend dividend;
 @Autowired private Education education;
 @Autowired private Food food;
 @Autowired private Salary salary;
 @Autowired private Self self;
 @Autowired private Transportation transportation;

 public TransactionCategoryInterface getCategory(String categoryType) {
   if (categoryType == null) {
     return null;
   }
   if (categoryType.equalsIgnoreCase("DIVIDEND")) {
     return dividend;
   } else if (categoryType.equalsIgnoreCase("EDUCATION")) {
     return education;
   } else if (categoryType.equalsIgnoreCase("FOOD")) {
     return food;
   } else if (categoryType.equalsIgnoreCase("SALARY")) {
     return salary;
   } else if (categoryType.equalsIgnoreCase("SELF")) {
     return self;
   } else if (categoryType.equalsIgnoreCase("TRANSPORTATION")) {
     return transportation;
   }
   return null;
 }
}
```

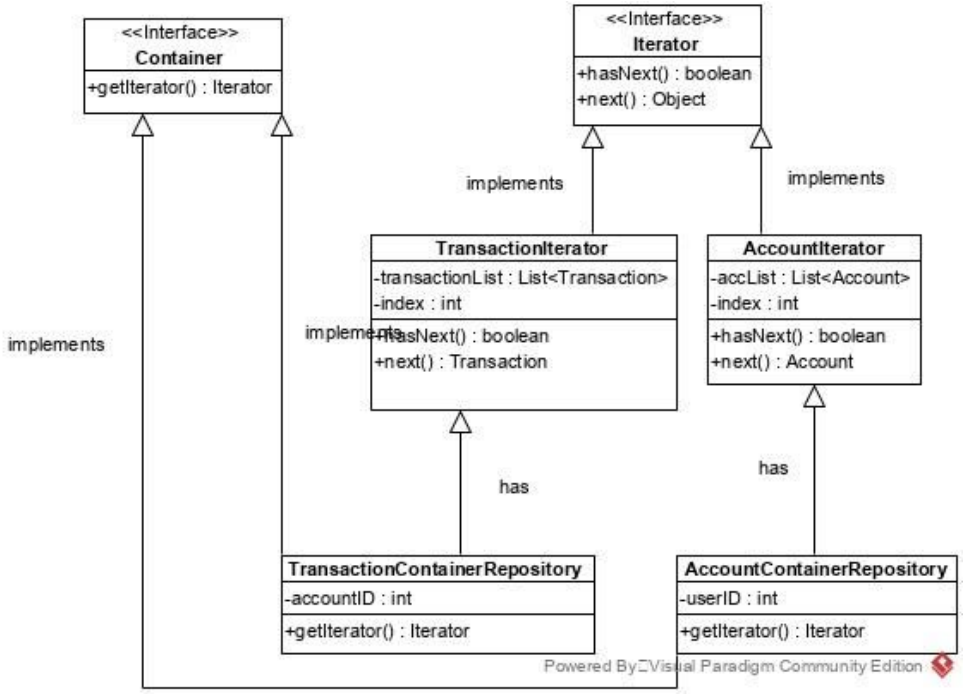
Concrete Class Example

```
@Component
public class Dividend implements TransactionCategoryInterface {
 @Override
 public String drawCategoryIcon() {
   return "<i class=\"fas fa-hand-holding-usd\"></i>";
 }
}
```

1.3.5 Iterator Pattern

The iterator pattern is used to provide a simple way to access the elements of a data structure. One benefit for using it is that a client doesn't have to know how the structure was built to iterate it's elements. Furthermore it's easy to change the sequence in which the elements are iterated. This Pattern achieves the single responsibility principle, and the open closed principle. The first one is supported because it only has the job to iterate and the second one is achieved because it doesn't break any existing code if a new type of iterator and collection is added.

For our project the iterator pattern was implemented for transactions and accounts. There are multiple instances when these data structures have to be iterated. This pattern was used to streamline this process and avoid rewriting loops all the time.



Iterator

```
public interface Iterator {
 public boolean hasNext();

 public Object next();
}
```

Container

```
public interface Container {
 Iterator getIterator();
}
```

TransactionContainerRepository

```
@Component
/** Implements Iterator Pattern Container */
public class TransactionContainerRepository implements Container {
 @Autowired private AccountService accountService;
 @Autowired private TransactionService transactionService;

 int accountID;

 public TransactionContainerRepository() {}

 @Override
 public Iterator getIterator() {
   return new TransactionIterator(transactionService.getAllTransactionsByAccountID(accountID));
 }

 public void setAccountID(int accountID) {
   this.accountID = accountID;
 }

 public int getAccountID() {
   return accountID;
 }
}
```

TransactionIterator

```
public class TransactionIterator implements Iterator {

 private List<Transaction> transactionList;
 private int index;

 public TransactionIterator(List<Transaction> accList) {
   this.transactionList = accList;
   this.index = 0;
 }

 @Override
```

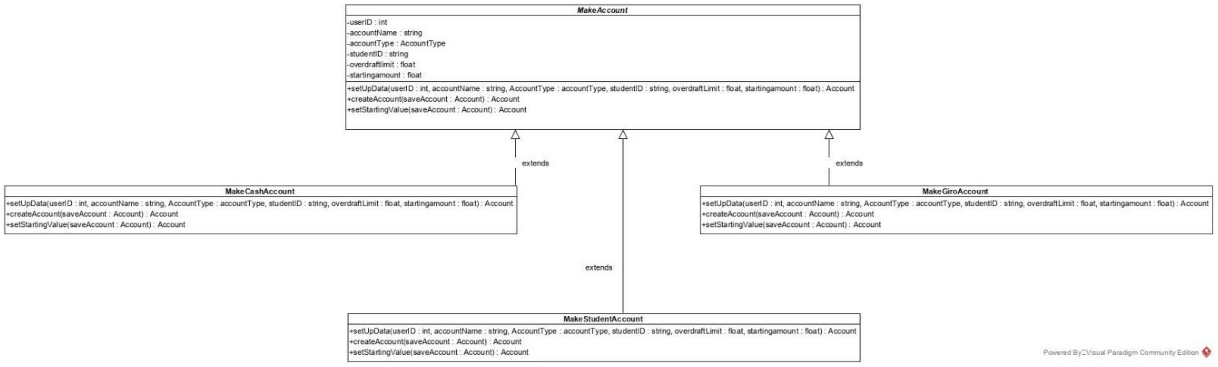
```
  public boolean hasNext() {
    if (index < transactionList.size()) return true;
    return false;
  }

  @Override
  public Transaction next() {
    if (hasNext()) {
      return transactionList.get(index++);
    }
    return null;
  }
}
```

For the purpose of this document only one iterator was shown as the structure of them is pretty similar.

1.3.6 Template Method Pattern

The template method pattern is used to determine an execution order for the functions of a class. This class is abstract and its methods can be overridden by its subclasses, nevertheless the execution order stays the same. It's also used to limit the methods which can be overridden by the subclasses and duplicate code can be handled in the superclass.

In our project this pattern was used for the creation of an account. The process order and most of the functions stays the same so it was handled in the superclass. Setting the starting amount was changed in the classes and therefore overwritten.



Abstract Class

```
@Component
public abstract class MakeAccount {
 @Autowired
 private AccountService accountService;

 protected abstract Account setUpData(
     int userID,
     String accountName,
     AccountType accountType,
     String studentid,
     float overdraftlimit,
```

```
    float startingamount);

 protected Account createAccount(Account saveAccount) throws AccountException{    /** if account
could not be created, throws AccountException */
   saveAccount = accountService.createAccount(saveAccount);
   if (saveAccount == null) {
     throw new AccountException("Account Creation Error");
   }
   return saveAccount;
 }

 protected abstract Account setStartingValue(Account saveAccount, float startingamount);

 public final Account makeAccount(
     int userID,
     String accountName,
     AccountType accountType,
     String studentid,
     float overdraftlimit,
     float startingamount)
     throws AccountException {
   Account account =
       setUpData(userID, accountName, accountType, studentid, overdraftlimit, startingamount);
   account = createAccount(account);
   account = setStartingValue(account, startingamount);
   return account;
 }
}
```

Concrete Class Example

```
 @Override
 public Account setStartingValue(Account account, float startingamount) {
   if (account != null) {
     long millis = System.currentTimeMillis();
     java.sql.Date date = new java.sql.Date(millis);
     Transaction startingAmountTrans =
         new PrivateTransaction(
             1,
             account.getAccountID(),
             account.getUserID(),
             startingamount,
             TransactionType.PRIVATE,
             date,
             "Cash StartAmount",
             TransactionCategory.SELF);
     startingAmountTrans.setTransactionCategoryIcon(
         transactionCategoryFactory.getCategory("Self").drawCategoryIcon());
     transactionService.newTransaction(startingAmountTrans);
   }
   return account;
 }
}
```

1.3.7 Proxy Pattern

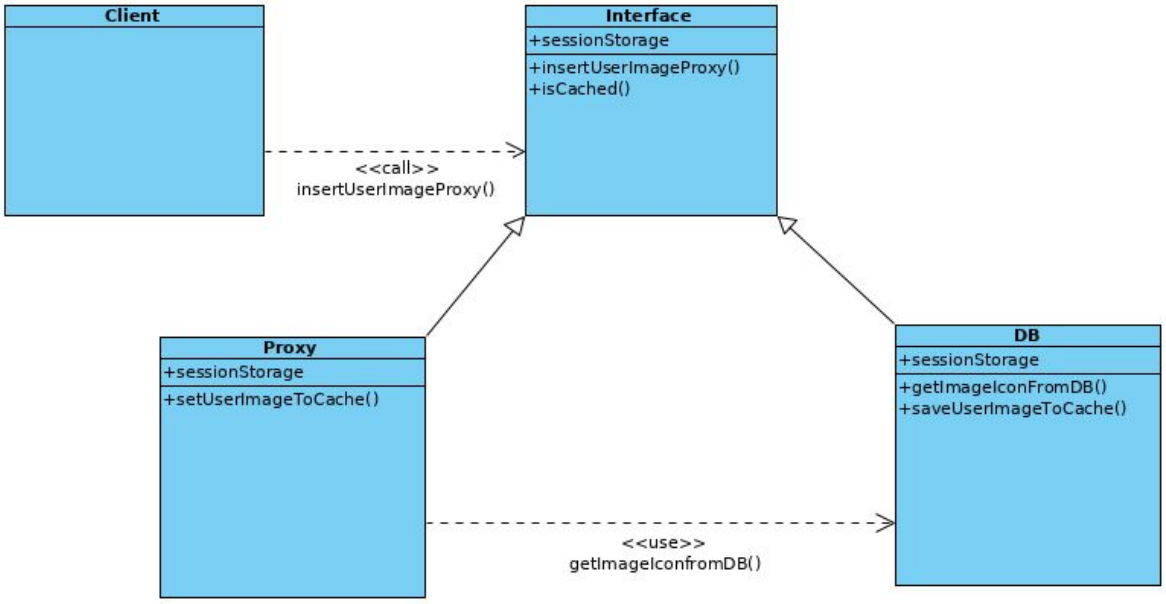
The Proxy Pattern helps to save resources by only loading the user image from the backend when necessary. The user image is only loaded once form the DataBase and set with the corresponding userID as a key. When the `insertUserImageProxy()` function is called again from the same User Account, the Image is loaded from the sessionStorage instead of the DB. When the function is called from another User Account the key does not match the ID, causing the proxy to reload the Image.

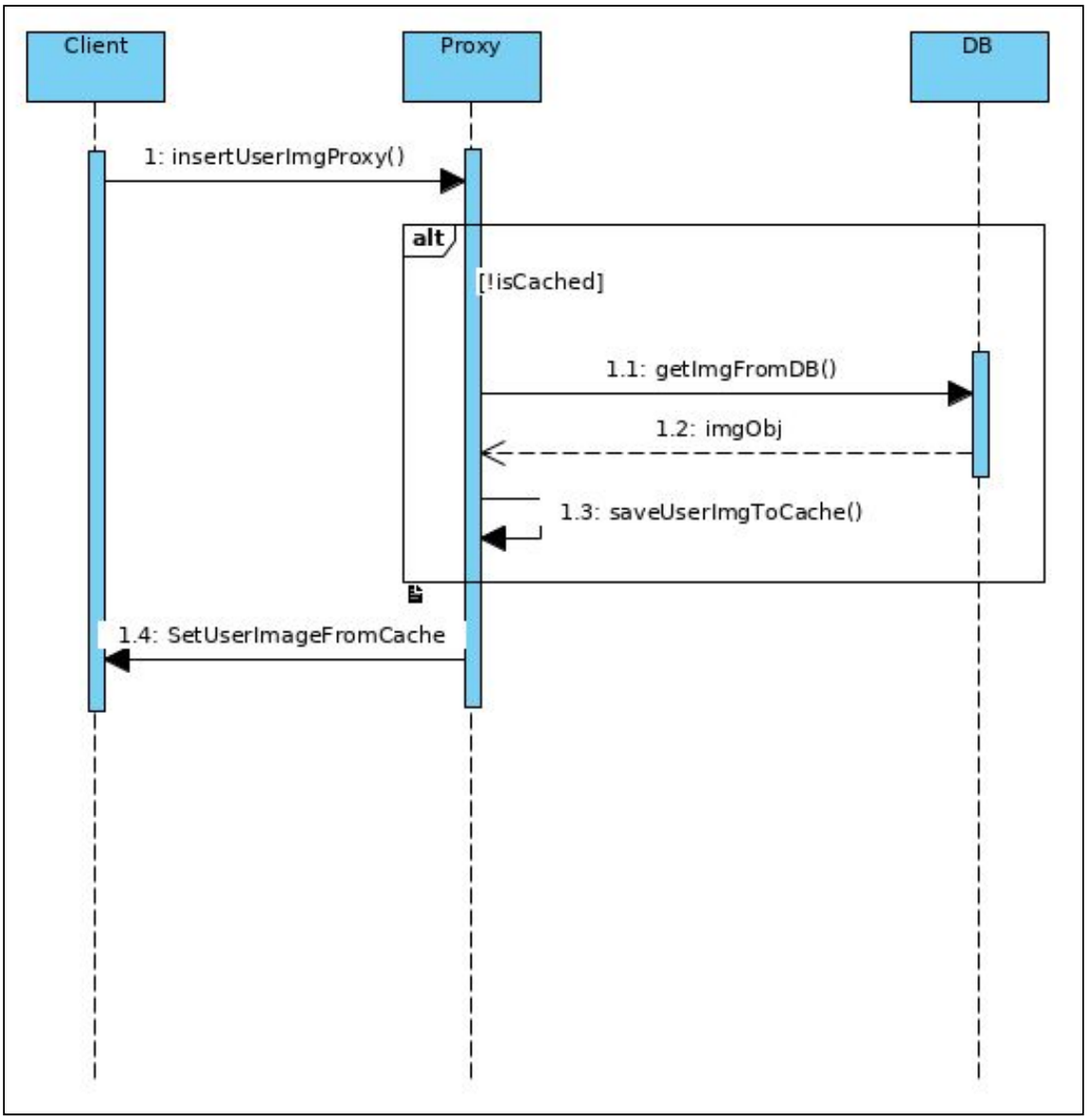
```
async function insertUserImageProxy() {
   function isCached () {
       return window.sessionStorage.getItem("imgUser")==getCookie("id");
   }
   if(!isCached()){
       console.log("about to load userImg from db");
       let imgObj=await getImageIconFromDB(getCookie("id"));
       saveUserImageToCache(imgObj);
   }
   console.log("setting userImg");
   setUserImageFromCache(window.sessionStorage.getItem("userImg"));
}
```

```
function saveUserImageToCache(userImgObject) {
   window.sessionStorage.setItem("userImg",userImgObject.userImg);
   window.sessionStorage.setItem("imgUser",userImgObject.imgUser);
   return 0;
}
```

```
function setUserImageFromCache(){
   document.getElementById("barUserIcon").src=window.sessionStorage.getItem("userImg");
   document.getElementById("bigAvatar").src=window.sessionStorage.getItem("userImg");
}
```

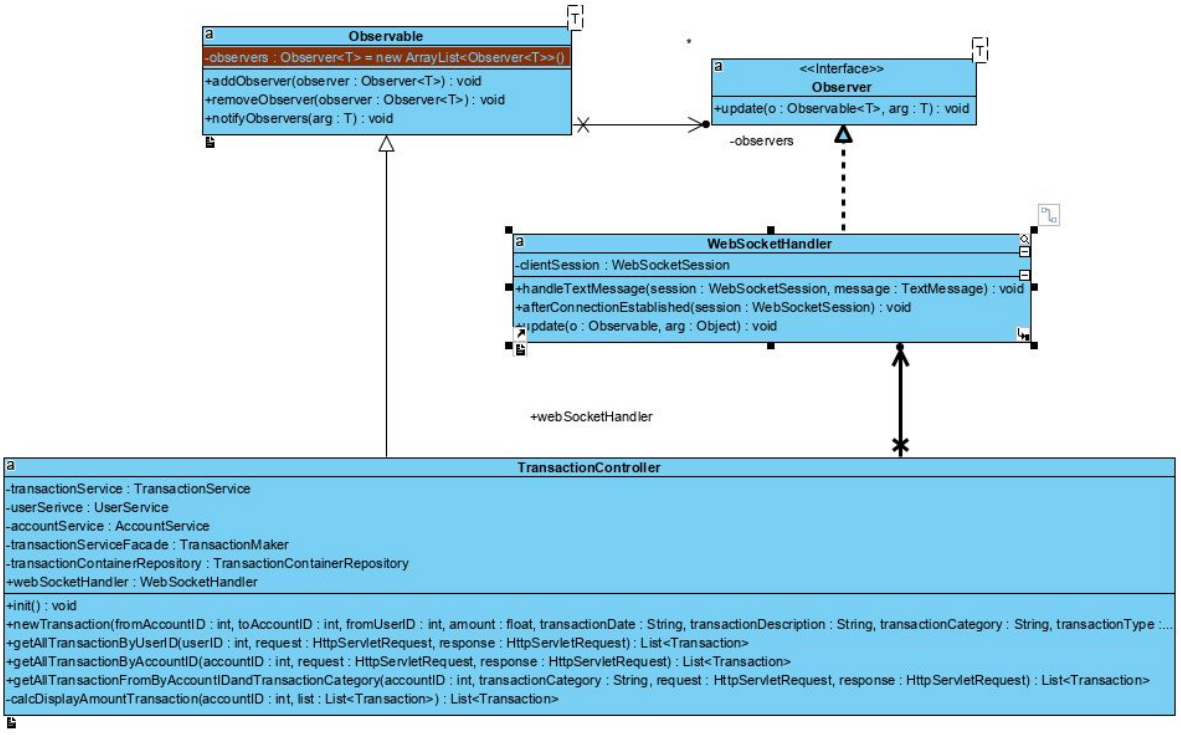
```
async function getImageIconFromDB(userID) {
   var path = "getAllUsers";
   var data_file = "./" + path;
   let response = await fetch(data_file);
   let jsonObj = await response.json();
   for (i in jsonObj) {
       if (jsonObj[i].id == userID) {
           return {"imgUser": userID, "userImg": jsonObj[i].imageIcon};
       }
   }
}
```


Client
Interface
+sessionStorage
+insertUserImageProxy()
+isCached()
<<call>>
insertUserImageProxy()
Proxy
+sessionStorage
+setUserImageToCache()
DB
+sessionStorage
+getImageIconFromDB()
+saveUserImageToCache()
<<use>>
getImageIconfromDB()


Client
Proxy
DB
1: insertUserImgProxy()
alt
[!isCached]
1.1: getImgFromDB()
1.2: imgObj
1.3: saveUserImgToCache()
1.4: SetUserImageFromCache

1.3.8 Observer Pattern

The Observer-pattern is realized by the usage of Websockets. This pattern is used to notify components if some data has changed. Similar to the publish and subscribe principle, the observer pattern uses two main classes like the observer, as the publisher, and an observable class as the subscriber. If a data change occurs, the observer will notify one or all observable classes that a change has been executed. The observable class will then execute another task with the changed data, like printing a message or drawing a new shape on the user interface. The open closed principle is achieved by adding new subscribers without changing the publishers code. On the one hand, relations between objects can be established during the runtime. On the other hand, the notification order of the observers is random generally.



Observer

```
public interface Observer<T> {
 void update(Observable<T> o, T arg);
}
```

Observable

```
public class Observable<T> {
 private final List<Observer<T>> observers = new ArrayList<Observer<T>>();
 public void addObserver(Observer<T> observer) {
   if (!observers.contains(observer)) {
     observers.add(observer);
   }
```

```
 }
 public void removeObserver(Observer<T> observer) {
   if (observers.contains(observer)) observers.remove(observer);
 }
 public void notifyObservers(T arg) {
   for (Observer<T> observer : observers) {
     observer.update(this, arg);
   }
 }
}
```

WebsocketConfig

```
@Configuration
@EnableWebSocket
public class WebSocketConfig implements WebSocketConfigurer {
 @Override
 public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {
   registry.addHandler(webSocketHandler(), "/notification").setAllowedOrigins("*");
 }
 @Bean
 public WebSocketHandler webSocketHandler() {
   return new com.group33.swa.webController.websocket.WebSocketHandler();
 }
}
```

WebsocketHandling

```
public class WebSocketHandler extends TextWebSocketHandler implements Observer {
 private WebSocketSession clientSession;
 @Override
 public void handleTextMessage(WebSocketSession session, TextMessage message) throws IOException {
   System.out.println(session);
   System.out.println(message.toString());

   if (message.getPayload().equals("hi")) {
     TextMessage send_message = new TextMessage("Hello back");
     session.sendMessage(send_message);
   }
 }
  * @param session
  */
 @Override
 public void afterConnectionEstablished(WebSocketSession session) {

   System.out.println(
       "New Client connection established! ClientID: "
           + session.getId()
           + " IP: "
           + session.getRemoteAddress().getHostName());
   this.clientSession = session;
 }
 @Override
 public void update(Observable o, Object arg) {
   System.out.println(this);
   System.out.println("ClientSession == null: " + (clientSession == null));
   System.out.println(clientSession);
   if (clientSession != null) {
     try {
       if (arg instanceof Message) {
         clientSession.sendMessage(new TextMessage(((Message) arg).getMessage()));
```

```
        }
      } catch (IOException e) {
        System.out.println(e);
      }

    } else {
      System.out.println("No clientsession available. Therefore no message will be send! ");
      System.out.println(o);
      System.out.println(arg);
    }
  }
}
```

```
var host = "localhost";
var port = 8080;
var endpoint = "notification";

let socket;

if (socket == null) {
    socket = new WebSocket("ws://" + host + ":" + port + "/" + endpoint);
    console.log("Websocket connection established!");
    console.log(socket);

} else {
    console.log("Websocket already connected!");
    console.log(socket);
}

socket.onopen = function (event) {

    console.log("Websocket is connected!");
    // openSnackbar("Websocket is connected!")

}

socket.onmessage = function (message) {

    openSnackbar(message.data)
    setTimeout(function () {
        location.reload();
    }, 3000);

    // openSnackbar(message)

}
```

The code snippet above shows up the implementation of a Websocket which receives the messages from the backend and shows them in the frontend as a snackbar.

2 Implementation

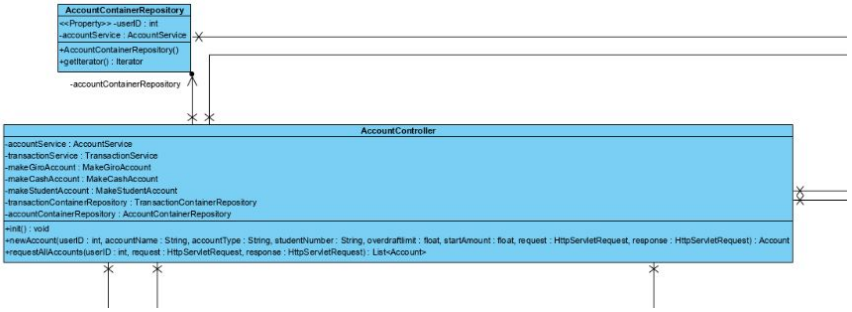
2.1 Overview of Main Modules and Components

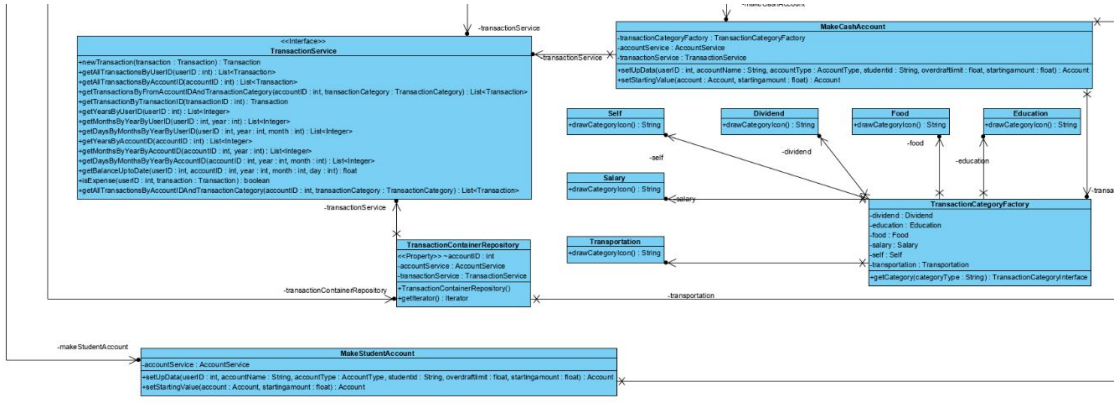
2.1.1 WebController

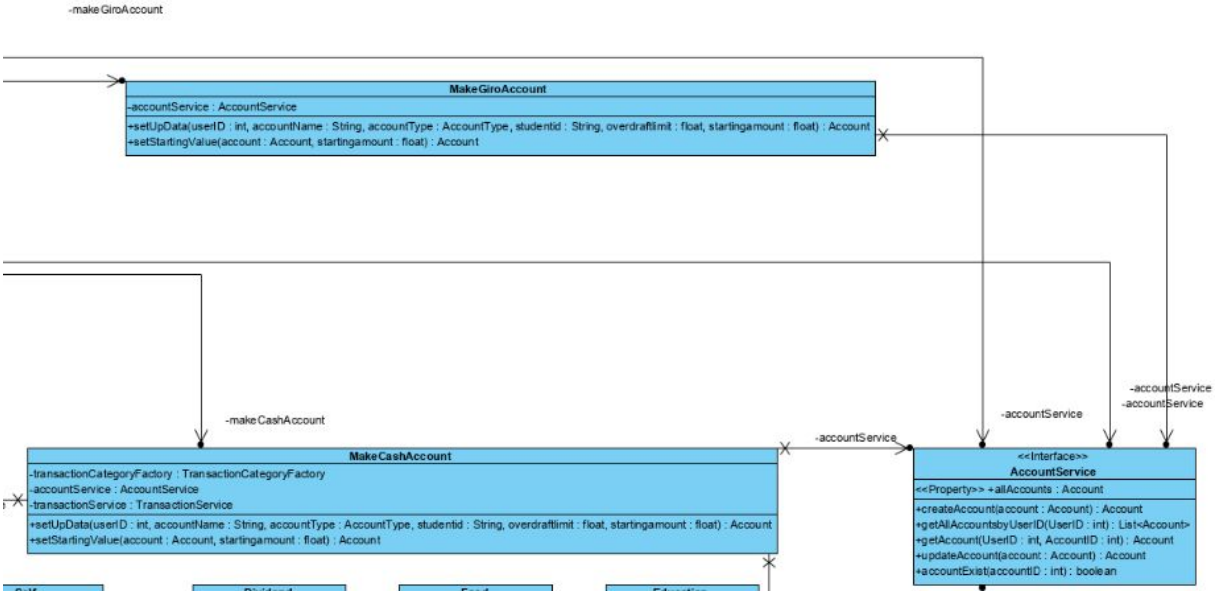
The WebController represents the REST-Interface where the communication between the Frontend and the Backend takes place. The WebController is divided into submodules like account, transaction user and websocket. All modules of the WebController except the WebSocket-Module are receiving GET- and POST-Requests with specific Query- or Body-Parameters for certain processes. For example, creating a new transaction, requesting all existing accounts as a list or getting some statistical or historical data. With the decoupling strategy of separating the REST-Interface and the Services, the frontend does not directly interact with the business logic of the backend.

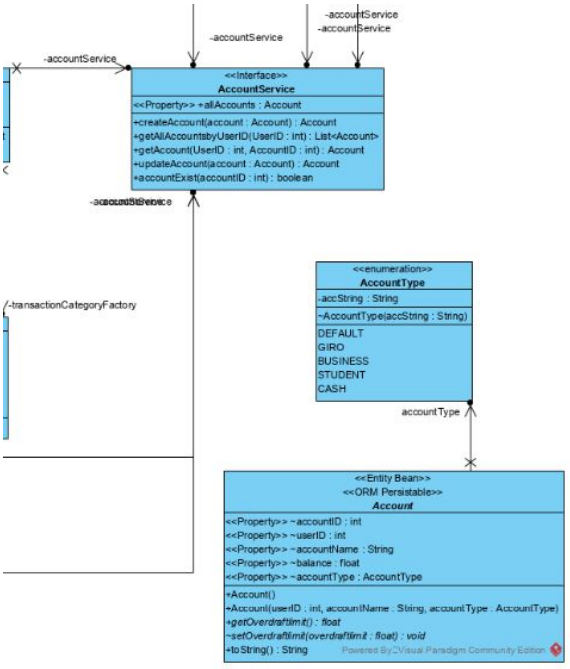
For a better overview of how these modules are structured, UML-Class diagrams of each module represent the implementations.

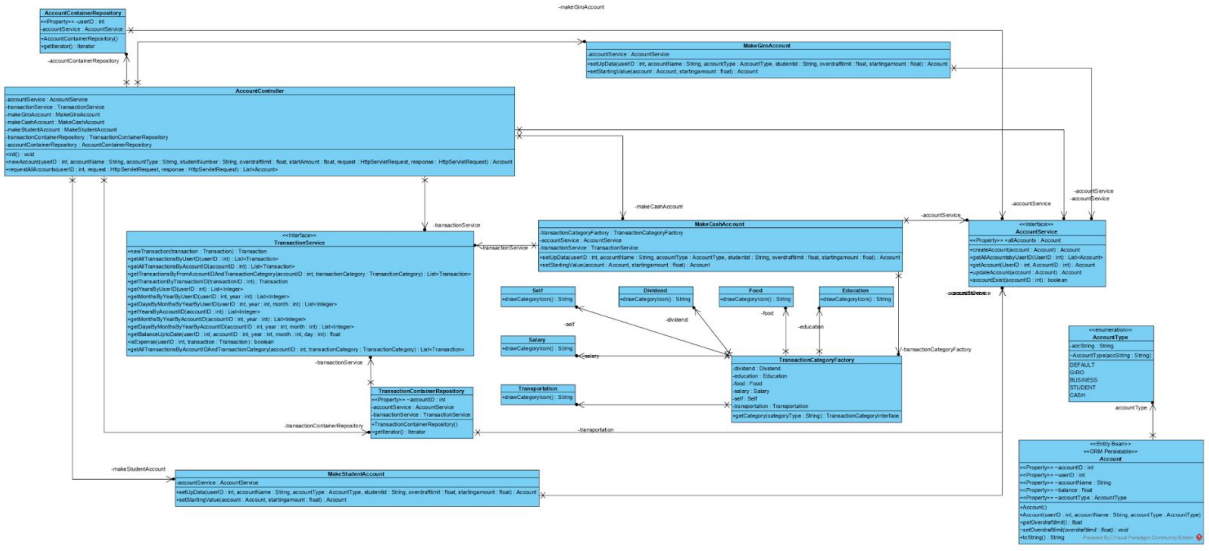
Account


AccountContainerRepository
<<Property>> -userID : int
-accountService : AccountService
+AccountContainerRepository()
+getIterator() : Iterator
-accountContainerRepository
AccountController
-accountService : AccountService
-transactionService : TransactionService
-makeGiroAccount : MakeGiroAccount
-makeCashAccount : MakeCashAccount
-makeStudentAccount : MakeStudentAccount
-transactionContainerRepository : TransactionContainerRepository
-accountContainerRepository : AccountContainerRepository
+init() : void
+newAccount(userID : int, accountName : String, accountType : String, studentNumber : String, overdraftlimit : float, startAmount : float, request : HttpServletRequest, response : HttpServletRequest) : Account
+requestAllAccounts(userID : int, request : HttpServletRequest, response : HttpServletRequest) : List<Account>

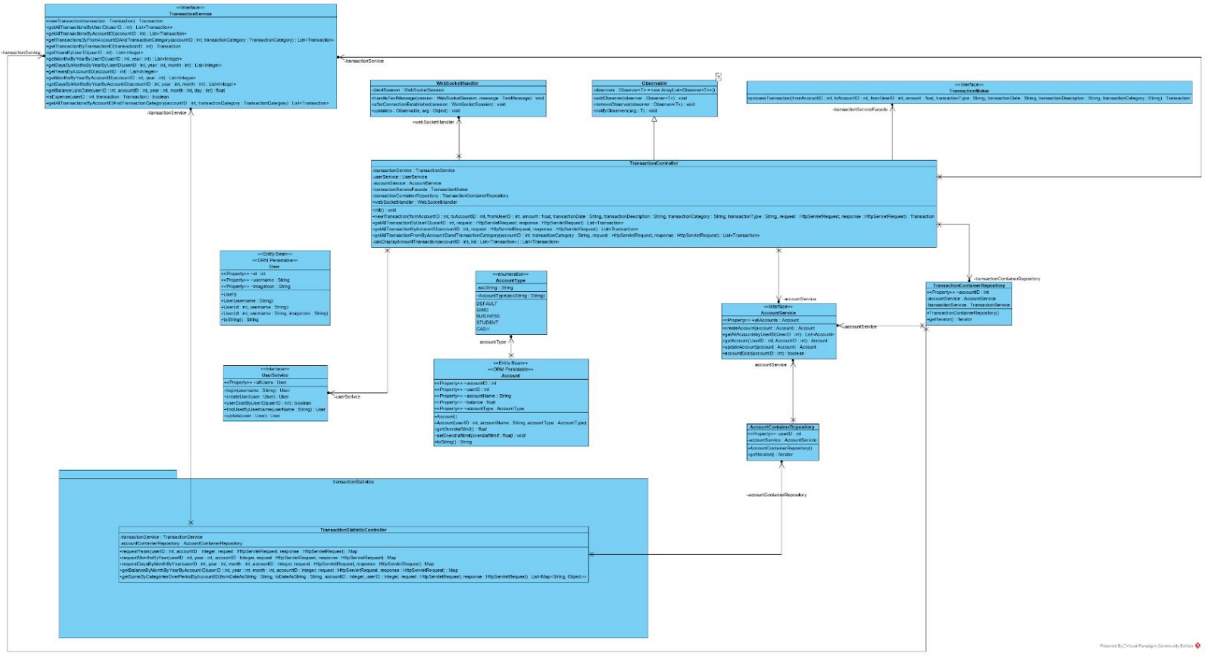

-transactionService
<<Interface>>
TransactionService
+newTransaction(transaction : Transaction) : Transaction
+getAllTransactionsByUserID(userID : int) : List<Transaction>
+getAllTransactionsByAccountID(accountID : int) : List<Transaction>
+getTransactionsByFromAccountIDAndTransactionCategory(accountID : int, transactionCategory : TransactionCategory) : List<Transaction>
+getTransactionByTransactionID(transactionID : int) : Transaction
+getYearsByUserID(userID : int) : List<Integer>
+getMonthsByYearByUserID(userID : int, year : int) : List<Integer>
+getDaysByMonthsByYearByUserID(userID : int, year : int, month : int) : List<Integer>
+getYearsByAccountID(accountID : int) : List<Integer>
+getMonthsByYearByAccountID(accountID : int, year : int) : List<Integer>
+getDaysByMonthsByYearByAccountID(accountID : int, year : int, month : int) : List<Integer>
+getBalanceUpToDate(userID : int, accountID : int, year : int, month : int, day : int) : float
+isExpense(userID : int, transaction : Transaction) : boolean
+getAllTransactionsByAccountIDAndTransactionCategory(accountID : int, transactionCategory : TransactionCategory) : List<Transaction>
-transactionService
TransactionContainerRepository
<<Property>> -accountID : int
-accountService : AccountService
-transactionService : TransactionService
+TransactionContainerRepository()
+getIterator() : Iterator
-transactionContainerRepository
MakeCashAccount
-transactionCategoryFactory : TransactionCategoryFactory
-accountService : AccountService
-transactionService : TransactionService
+setUpData(userID : int, accountName : String, accountType : AccountType, studentid : String, overdraftlimit : float, startingamount : float) : Account
+setStartingValue(account : Account, startingamount : float) : Account
Self
+drawCategoryIcon() : String
Dividend
+drawCategoryIcon() : String
Food
+drawCategoryIcon() : String
Education
+drawCategoryIcon() : String
Salary
+drawCategoryIcon() : String
Transportation
+drawCategoryIcon() : String
-self
-dividend
-food
-education
-salary
-transportation
TransactionCategoryFactory
-dividend : Dividend
-education : Education
-food : Food
-salary : Salary
-self : Self
-transportation : Transportation
+getCategory(categoryType : String) : TransactionCategoryInterface
-transportation
-makeStudentAccount
MakeStudentAccount
-accountService : AccountService
+setUpData(userID : int, accountName : String, accountType : AccountType, studentid : String, overdraftlimit : float, startingamount : float) : Account
+setStartingValue(account : Account, startingamount : float) : Account


-makeGiroAccount
MakeGiroAccount
-accountService : AccountService
+setUpData(userID : int, accountName : String, accountType : AccountType, studentid : String, overdraftlimit : float, startingamount : float) : Account
+setStartingValue(account : Account, startingamount : float) : Account
-makeCashAccount
MakeCashAccount
-transactionCategoryFactory : TransactionCategoryFactory
-accountService : AccountService
-transactionService : TransactionService
+setUpData(userID : int, accountName : String, accountType : AccountType, studentid : String, overdraftlimit : float, startingamount : float) : Account
+setStartingValue(account : Account, startingamount : float) : Account
-accountService
-accountService
-accountService
-accountService
<<Interface>>
AccountService
<<Property>> +allAccounts : Account
+createAccount(account : Account) : Account
+getAllAccountsbyUserID(UserID : int) : List<Account>
+getAccount(UserID : int, AccountID : int) : Account
+updateAccount(account : Account) : Account
+accountExist(accountID : int) : boolean

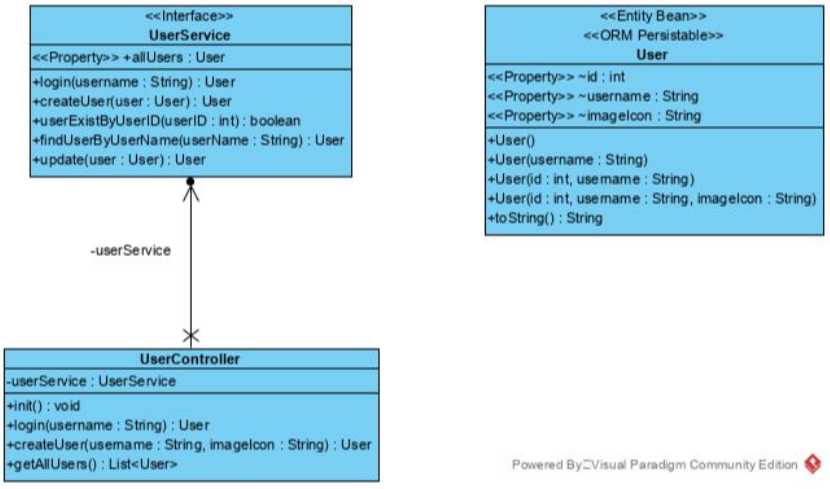

-accountService
-accountService
-accountService
-accountService
<<Interface>>
AccountService
<<Property>> +allAccounts : Account
+createAccount(account : Account) : Account
+getAllAccountsbyUserID(UserID : int) : List<Account>
+getAccount(UserID : int, AccountID : int) : Account
+updateAccount(account : Account) : Account
+accountExist(accountID : int) : boolean
-transactionCategoryFactory
<<enumeration>>
AccountType
-accString : String
~AccountType(accString : String)
DEFAULT
GIRO
BUSINESS
STUDENT
CASH
accountType
<<Entity Bean>>
<<ORM Persistable>>
Account
<<Property>> ~accountID : int
<<Property>> ~userID : int
<<Property>> ~accountName : String
<<Property>> ~balance : float
<<Property>> ~accountType : AccountType
+Account()
+Account(userID : int, accountName : String, accountType : AccountType)
+getOverdraftlimit() : float
~setOverdraftlimit(overdraftlimit : float) : void
+toString() : String
Powered By Visual Paradigm Community Edition



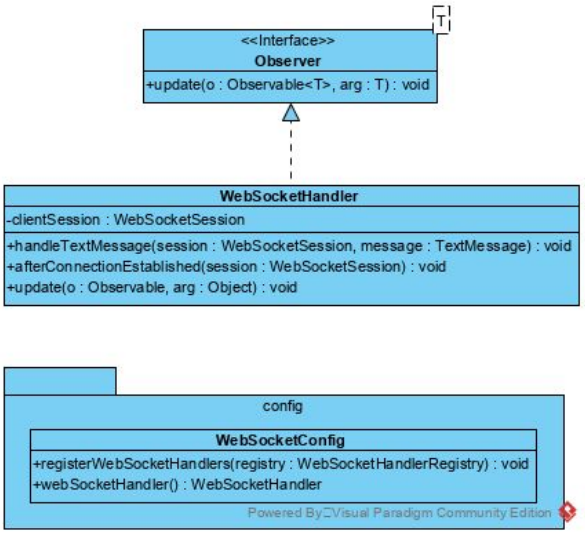
Transaction



User



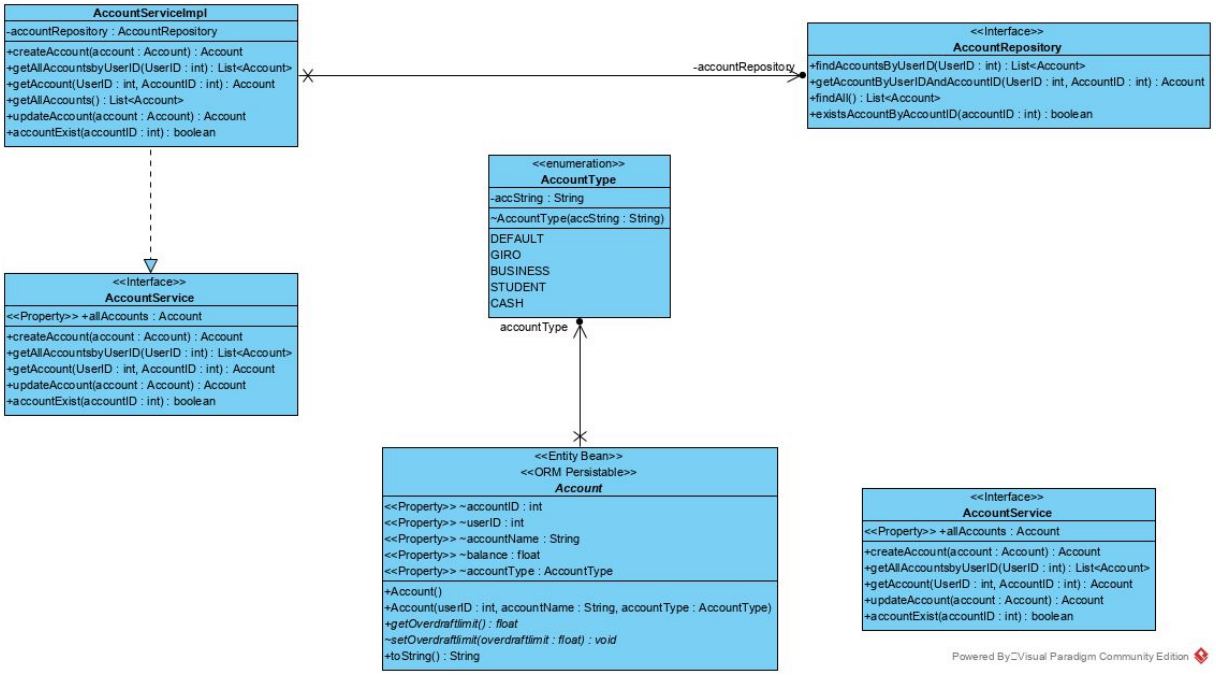
Websocket


<<Interface>>
Observer
+update(o : Observable<T>, arg : T) : void
T
WebSocketHandler
-clientSession : WebSocketSession
+handleTextMessage(session : WebSocketSession, message : TextMessage) : void
+afterConnectionEstablished(session : WebSocketSession) : void
+update(o : Observable, arg : Object) : void
config
WebSocketConfig
+registerWebSocketHandlers(registry : WebSocketHandlerRegistry) : void
+webSocketHandler() : WebSocketHandler
Powered By Visual Paradigm Community Edition

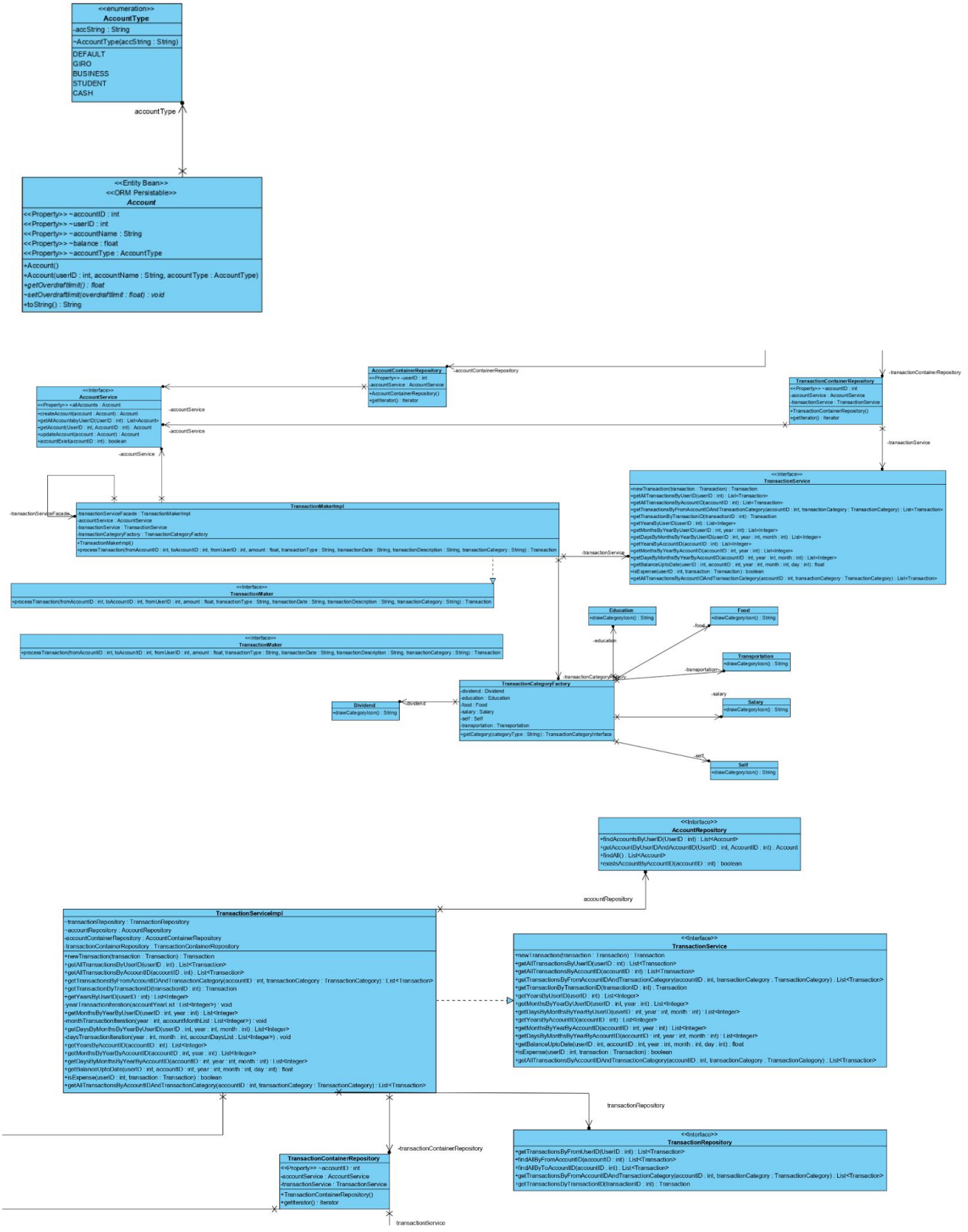
2.1.2 WebServices

The WebServices are submodules which handles the persistence storage of models like account, user and transactions. The submodules are also used to read data models from a database. To be able to store models in the database persistently, the spring-framework provides a generic JPARepository which enables the storing of data through several databases.

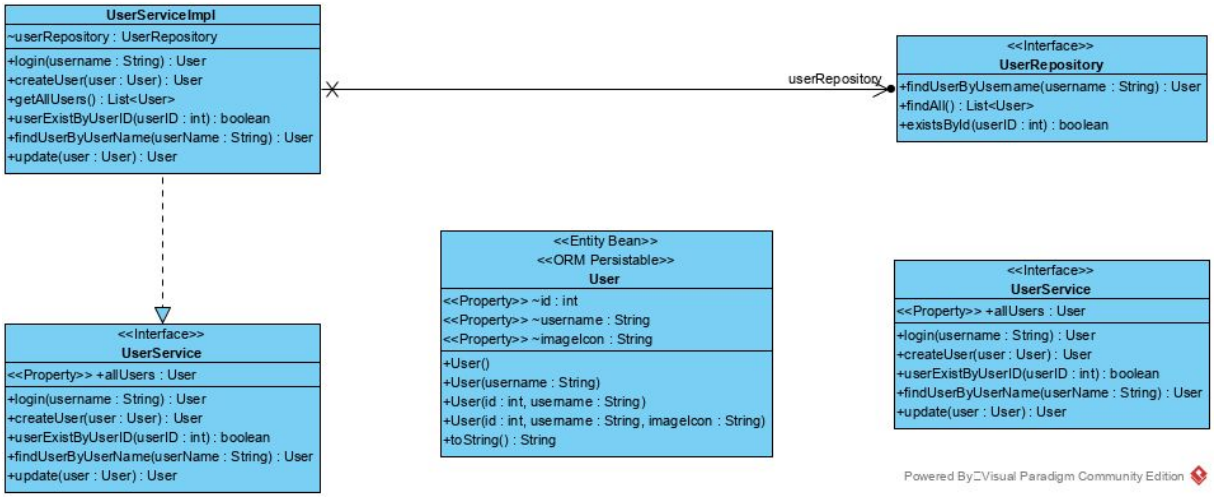
AccountService



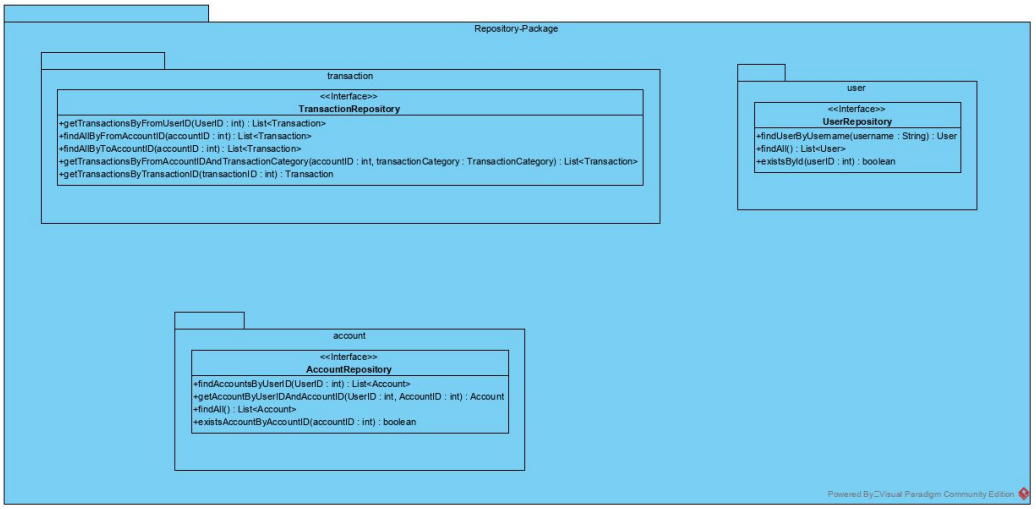
TransactionService



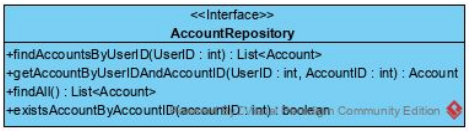
UserService



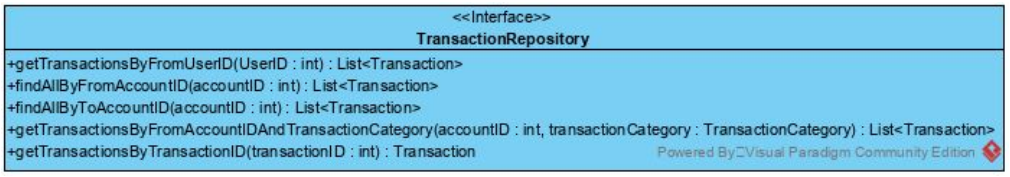
2.1.3 Repository



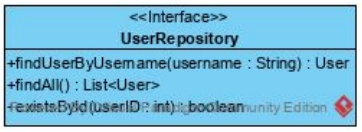
Account



Transaction

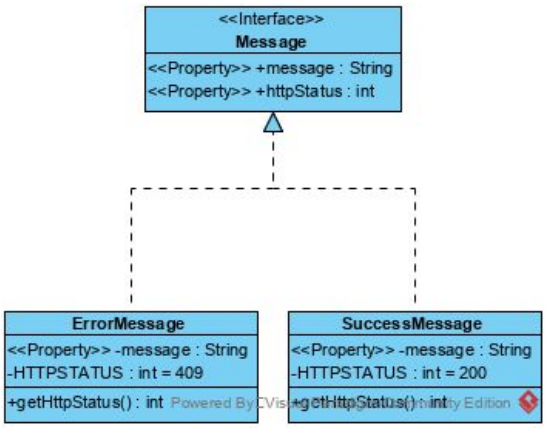


User

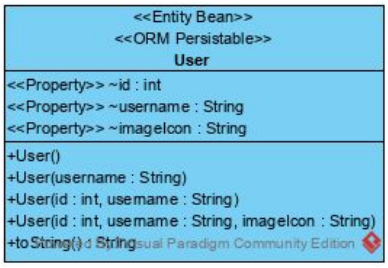


2.1.4 Model

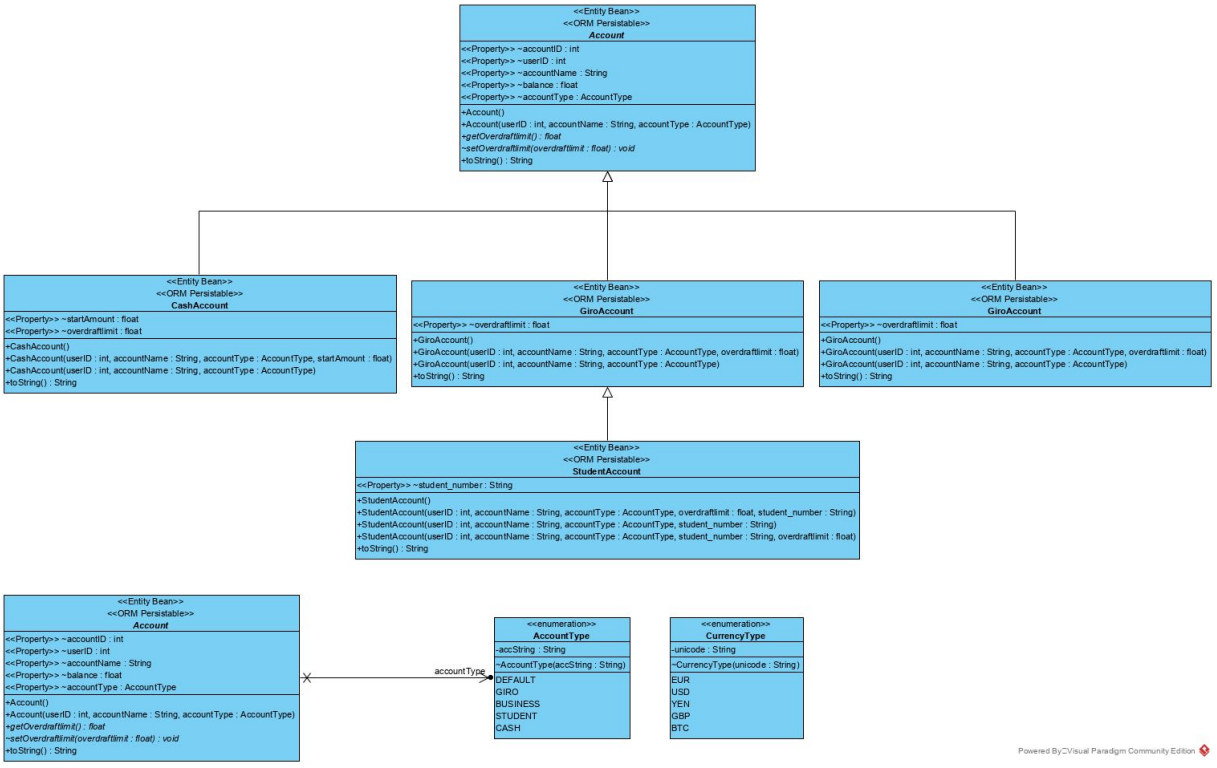
Communication



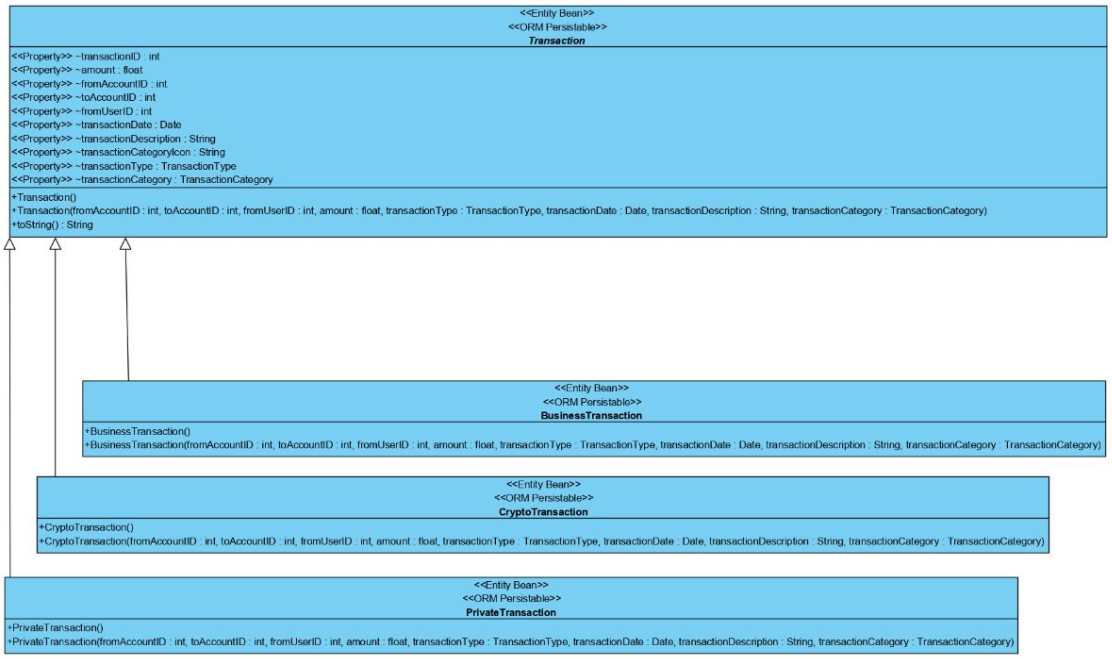
User

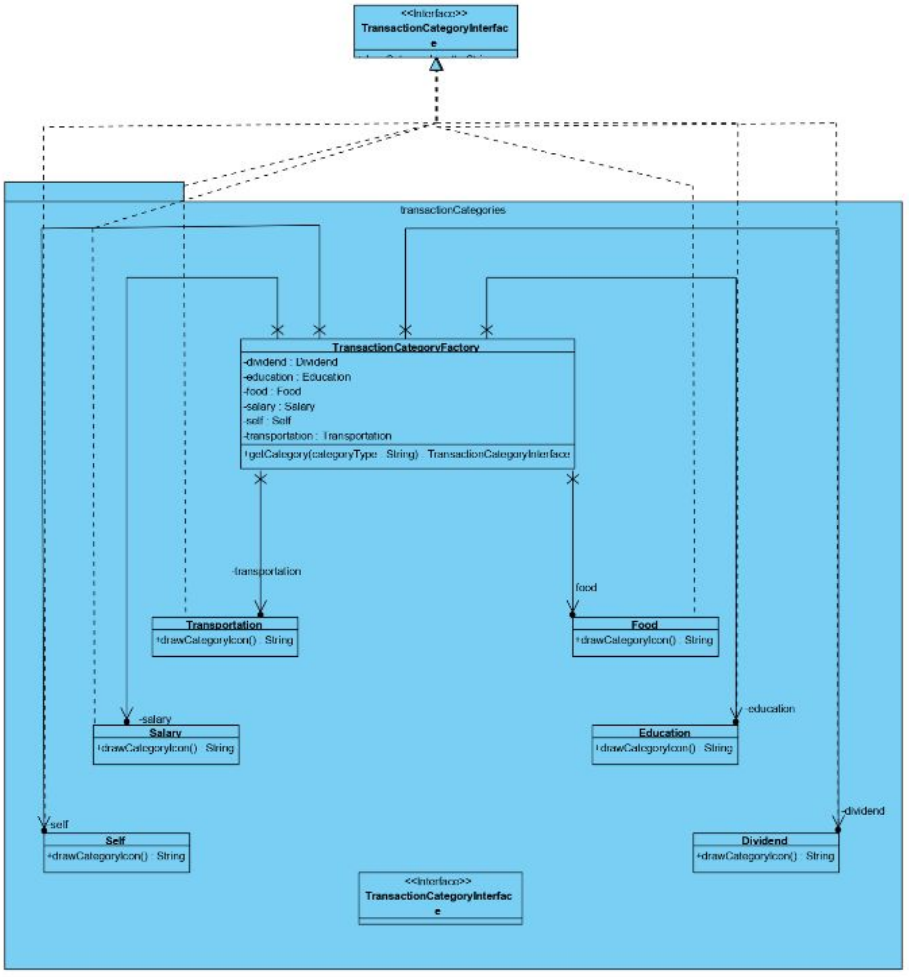


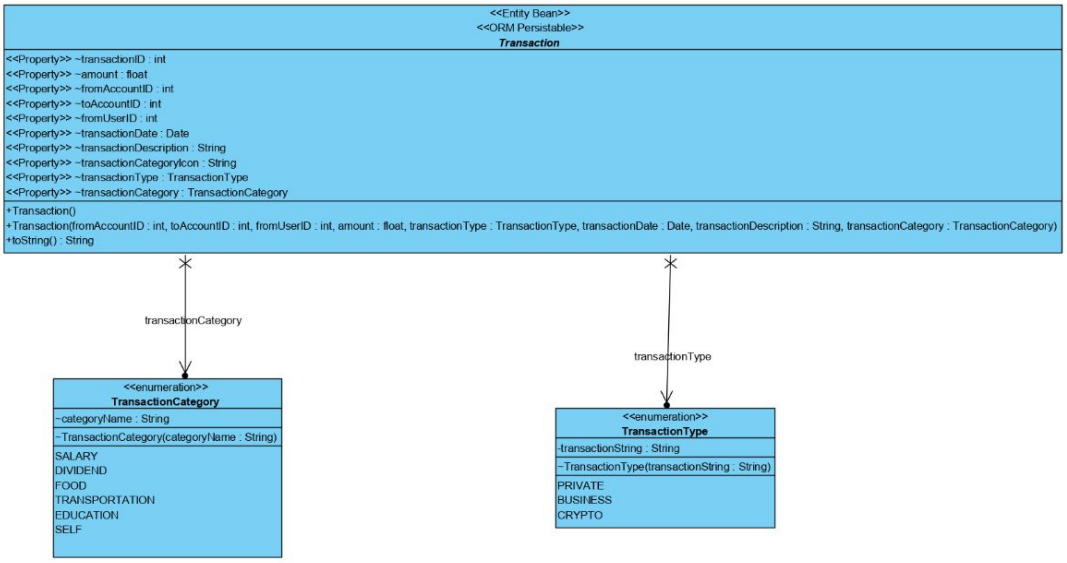
Account



Transaction

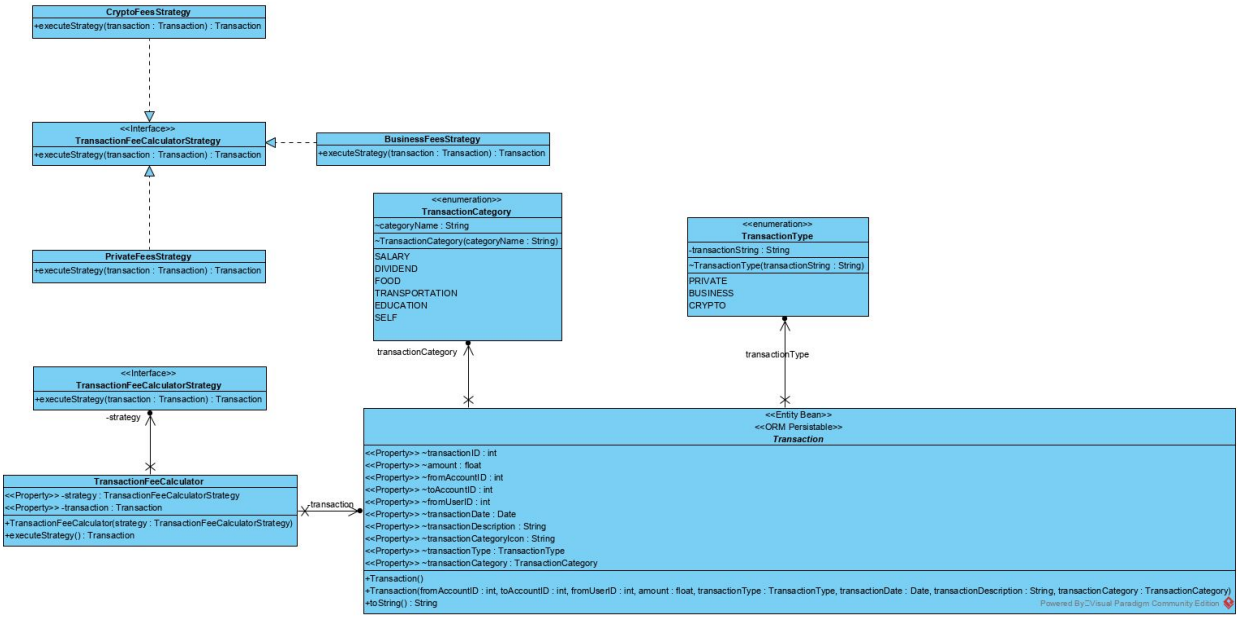




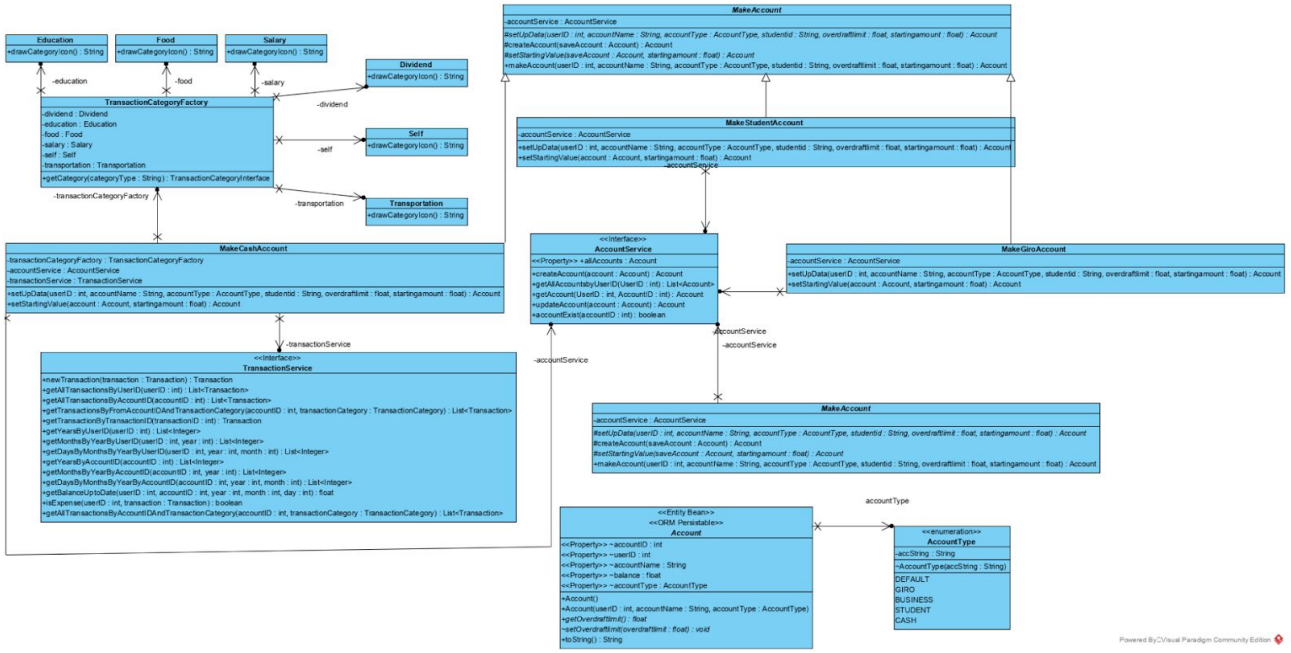

<<Entity Bean>>
<<ORM Persistable>>
Transaction
<<Property>> ~transactionID : int
<<Property>> ~amount : float
<<Property>> ~fromAccountID : int
<<Property>> ~toAccountID : int
<<Property>> ~fromUserID : int
<<Property>> ~transactionDate : Date
<<Property>> ~transactionDescription : String
<<Property>> ~transactionCategoryIcon : String
<<Property>> ~transactionType : TransactionType
<<Property>> ~transactionCategory : TransactionCategory
+Transaction()
+Transaction(fromAccountID : int, toAccountID : int, fromUserID : int, amount : float, transactionType : TransactionType, transactionDate : Date, transactionDescription : String, transactionCategory : TransactionCategory)
+toString() : String
transactionCategory
transactionType
<<enumeration>>
TransactionCategory
~categoryName : String
~TransactionCategory(categoryName : String)
SALARY
DIVIDEND
FOOD
TRANSPORTATION
EDUCATION
SELF
<<enumeration>>
TransactionType
-transactionString : String
~TransactionType(transactionString : String)
PRIVATE
BUSINESS
CRYPTO

2.1.5 Logic

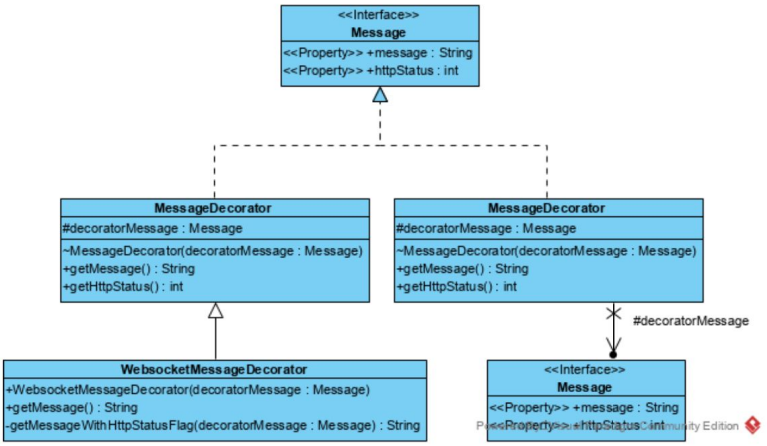
FeeCalculator



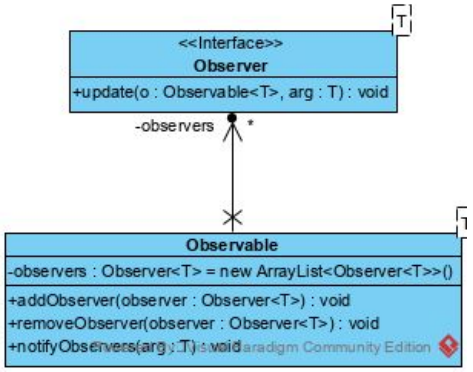
MakeAccount



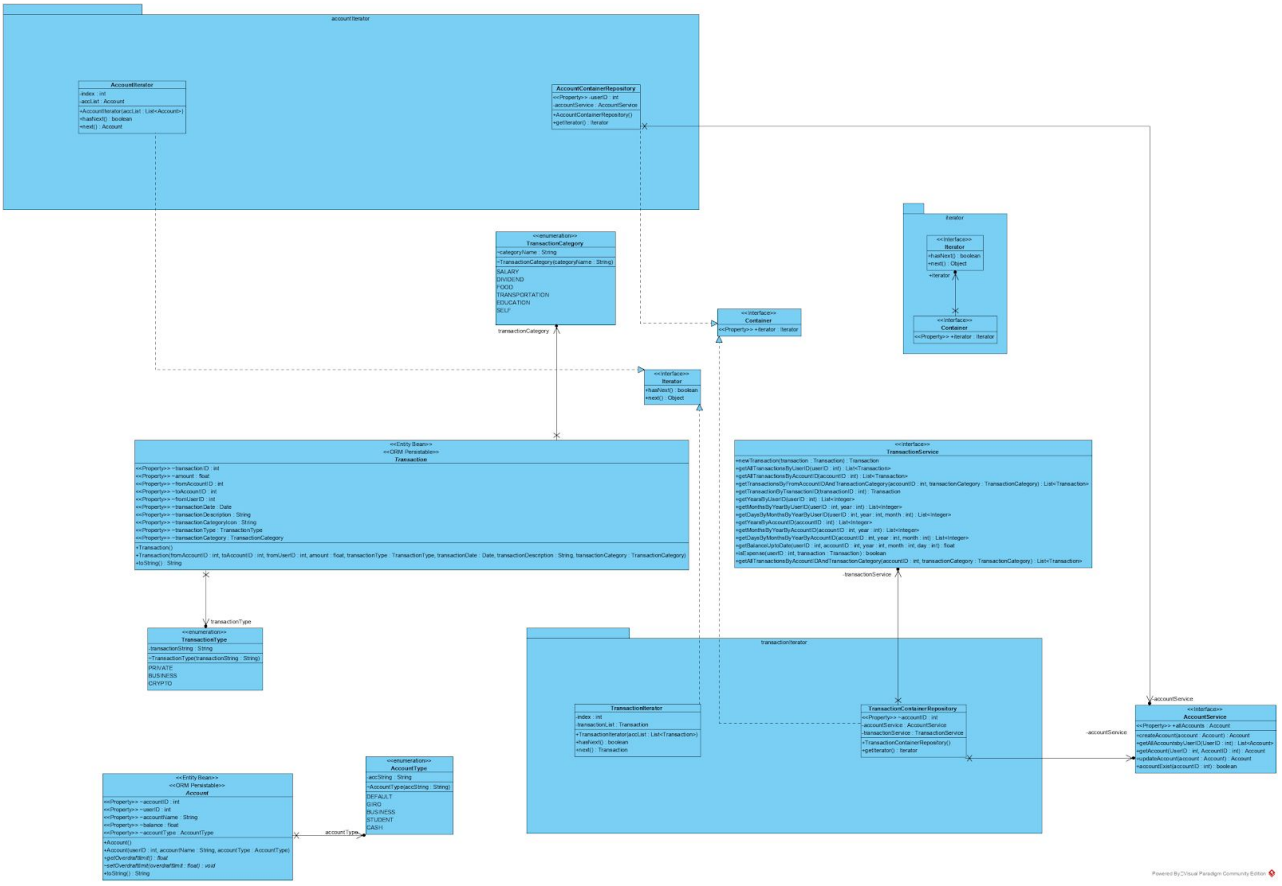
MessageDecorator



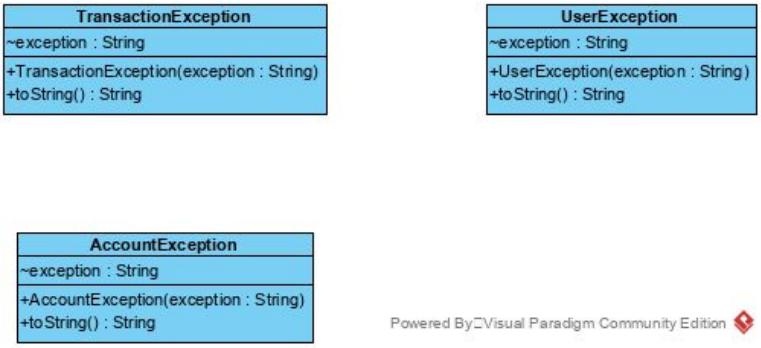
Observer



Iterator

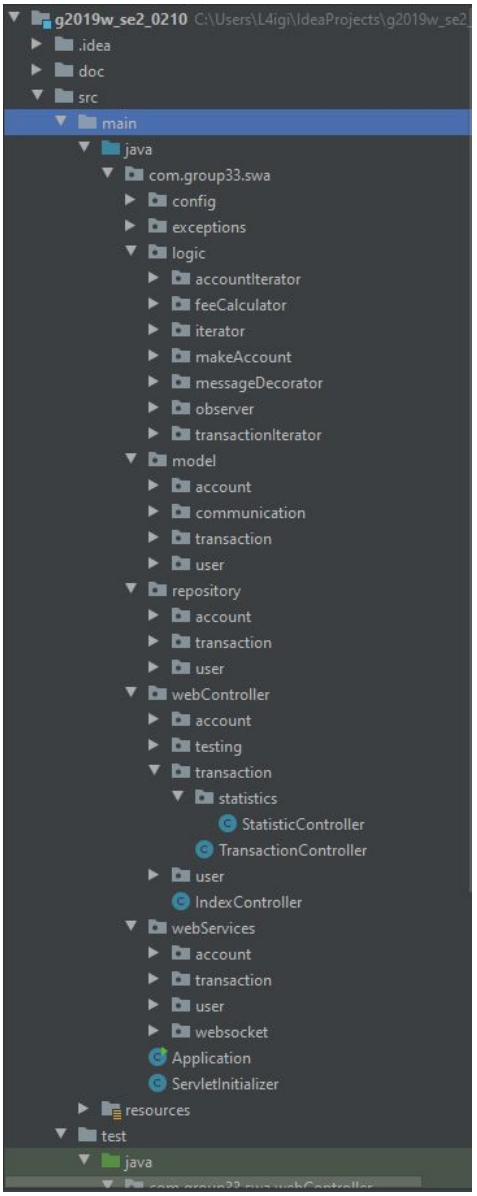


2.1.6 Exception



2.2 Coding Practices

We tried to keep a high cohesion between the different parts of our program. As seen in the screenshot below everything is split up in separate packages. An example showing this in the code would be the separation of the web-services from the web-controllers. The web-services handle the logic which is used by the web-controllers for communication. Both have a distinctive function and therefore are highly cohesive.



Most of the functions were created in a specific way, so that one function is responsible for one task. We tried to avoid conjoined operations as much as possible.

```
@RequestMapping(
        value = "/getBalanceByMonthByYearByAccountID",
        method = RequestMethod.GET,
        produces = "application/json")
@ResponseBody
public Map getBalanceByMonthByYearByAccountID(
        @RequestParam(value = "userID", required = true) int userID,
        @RequestParam(value = "year", required = true) int year,
        @RequestParam(value = "month", required = true) int month,
        @RequestParam(value = "accountID", required = true) Integer accountID,
        HttpServletRequest request,
        HttpServletRequest response) {

    Map<Integer, Float> returnMap = new TreeMap<Integer, Float>();

    int daysInMonth = 31;
    switch (month%2){
        case 0:
            daysInMonth = 30;
            break;
        case 1:
            daysInMonth = 31;
            break;
        default:
            break;
    }
    if(month == 2){
        daysInMonth = 29;
    }

    for(int i = 1; i <= daysInMonth; i++){
        returnMap.put(i,transactionService.getBalanceUptoDate(userID, accountID, year, month, i));
    }
    return returnMap;
}
```

The function “getBalanceByMonthByYearByAccountID” is shown as an example-function of the TransactionStatisticsController.

```
@Override
public float getBalanceUptoDate(int userID, int accountID, int year, int month, int day) {
 List<Transaction> allTransactions = getAllTransactionsByAccountID(accountID);

 float balanceUptoDate = 0;

 for (Transaction tempTran : allTransactions) {
   Calendar cal = Calendar.getInstance();
```

```
    cal.setTime(tempTran.getTransactionDate());
    int tempYear = cal.get(Calendar.YEAR);
    int tempMonth = cal.get(Calendar.MONTH);
    int tempDay = cal.get(Calendar.DAY_OF_MONTH);

    boolean isExpense =  isExpense(userID, tempTran);

    if (tempYear < year) {
      if(isExpense){
        balanceUptoDate -= tempTran.getAmount();
      } else {
        balanceUptoDate += tempTran.getAmount();
      }
    } else if (tempYear == year && (tempMonth+1) < month) {
      if(isExpense){
        balanceUptoDate -= tempTran.getAmount();
      } else {
        balanceUptoDate += tempTran.getAmount();
      }      } else if (tempYear == year && (tempMonth+1) == month && tempDay <= day) {
      if(isExpense){
        balanceUptoDate -= tempTran.getAmount();
      } else {
        balanceUptoDate += tempTran.getAmount();
      }      }
    else{
      //should never reach here
    }
  }
  return balanceUptoDate;
}
```

The function getBalanceUpToDate will return the balance till a specific date as shown in the code-snippet above.

```
@Override
public boolean isExpense(int userID, Transaction transaction) {
  return (accountRepository.getAccountByUserIDAndAccountID(userID, transaction.getFromAccountID())
      != null);
}
```

The separate help function “isExpense” is used multiple times to separate expenses from income.

```
protected Account createAccount(Account saveAccount) throws AccountException {
  /** if account could not be created, throws AccountException */
  saveAccount = accountService.createAccount(saveAccount);
  if (saveAccount == null) {
    throw new AccountException("Account Creation Error");
```

```
 }
 return saveAccount;
}
```

As shown in the code-snipped above, the logical part was handled in the service and the controller just manage the communication and collect the data.

Errors were handled locally and Exceptions were caught and processed in the corresponding exception class. In addition errors are logged.

```
@Test
void getAllUsers() throws Exception {
 mockMvc
     .perform(
         MockMvcRequestBuilders.get("/createUser")
             .param("username", "testUser1")
             .param("imageIcon", "Icon"))
     .andExpect(MockMvcResultMatchers.status().isOk());
 mockMvc
     .perform(
         MockMvcRequestBuilders.get("/createUser")
             .param("username", "testUser2")
             .param("imageIcon", "Icon"))
     .andExpect(MockMvcResultMatchers.status().isOk());
 mockMvc
     .perform(
         MockMvcRequestBuilders.get("/createUser")
             .param("username", "testUser3")
             .param("imageIcon", "Icon"))
     .andExpect(MockMvcResultMatchers.status().isOk());
```

Regarding naming conventions, the names were chosen so they are self explanatory easy to read. Sequential names were given in case needed. For example all the test data was created in such sequential manner as you can see in this example creating three test users.

2.3 Defensive Programming

From the start on we tried to incorporate defensive programming in our project. Our application was built with modularity in mind. It should be easy to add new components without breaking existing code and the existing code should stand on its own. Some bad code we produced while testing to introduce new features was refactored completely to improve the readability and keep the overall structure clean and simple.

Duplicate Code was split up in separate functions to enable resusage of code, minimize the code size and improve the mainability. As one can see in the

isExpense function example given in section 2.2. Comments were created where deemed necessary and to explain the intent of the underlying code. Most of the comments were created in such a way in order to export a JavaDoc. Commenting on obvious code like getter and setter-functions and simple loops was avoided.

```
/**
 * If amount is negativ, fromUser und toUser, fromAccount, toAccount will switch. Changes:
 * 12.11.2019 @RequestParam fromUser, required = false @RequestParam toUser, required = false
 *
 * <p>This Method represents a REST-Interface to create a new Transaction
 *
 * @param fromAccountID accountID where transaction is comming from
 * @param toAccountID accountID which the transaction is for
 * @param fromUserID userID from which user the transaction is beeing created
 * @param amount represents the actual amount of the specified currency
 * @param transactionDate represents the date or timestamp when the transaction was created
 * @param transactionDescription Description for the transaction
 * @param request REQ-Client informations
 * @param response RES-Client informations
 * @return {@link Transaction}
 */
```

As a requirement for the code formation the Google Java Style Guide was used.

Garbage was handled by returning nothing or an empty Object, List etc. In other cases Garbage is handled by an exception which is processed locally.

```
@RequestMapping(
   value = "/getAllTransactionByUserID",
   method = RequestMethod.GET,
   produces = "application/json")
@ResponseBody
public List<Transaction> getAllTransactionByUserID(
   @RequestParam(value = "userID", required = true) int userID
) {
 if (userSerivce.userExistByUserID(userID)) {
   List<Transaction> userTransactionList = new ArrayList<>();
   List<Account> accountList = accountService.getAllAccountsbyUserID(userID);
   for (Account tempacc : accountList) {
     List<Transaction> transactionList =
         transactionService.getAllTransactionsByAccountID(tempacc.getAccountID());
     for (Transaction temptrans : transactionList) {
       userTransactionList.add(temptrans);
     }
   }

   return userTransactionList;
 }
 return new ArrayList<>();
}
```

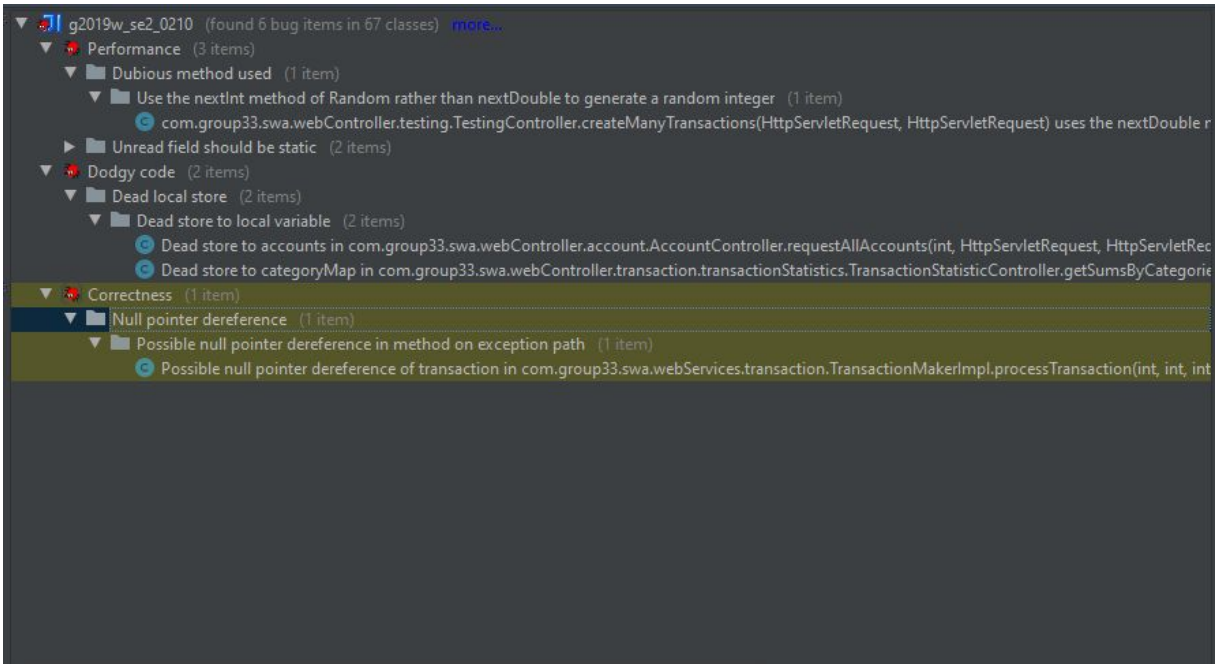
Assertion was used mainly for testing. By comparing the test data with the results it was assured, that our application handles different valid and invalid inputs accordingly.

Regarding robustness and correctness, our program keeps running even if invalid data is sent to the backend, because it is handled there accordingly. Nothing is returned instead of wrong data and changes will not be saved if the data was invalid. The previous correct data is not overwritten and kept instead of the new faulty data. We removed code for checking trivial errors after fixing them and also code that was functional but replaced with better, simpler and easier to understand code.

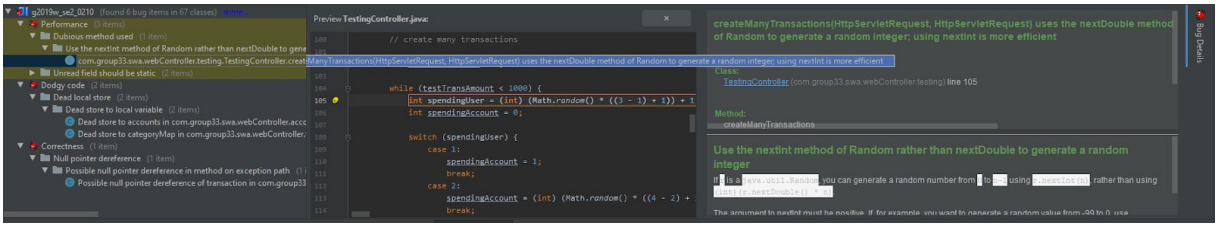
3 Software Quality

3.1 Code Metrics

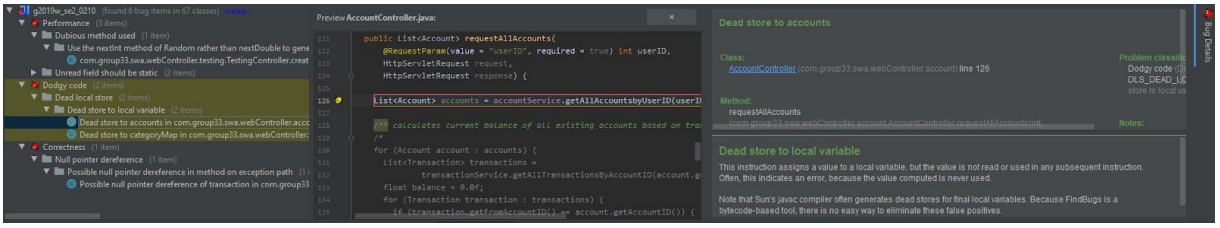
FindBugs was used to search for bugs in our project. Running it revealed a total of 6 bugs.

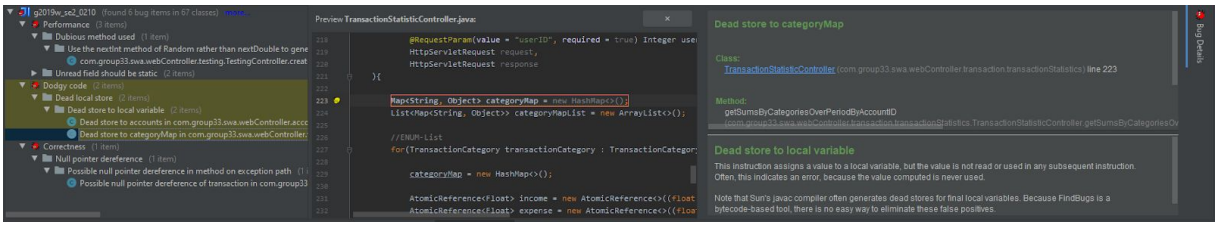


One of these bugs was related to the random data generation in the TestTransaction class. This bug was found because there was a faster way for this case of random data. This bug was removed, not by fixing it but by changing a logical error in this class which eliminated the bug too.

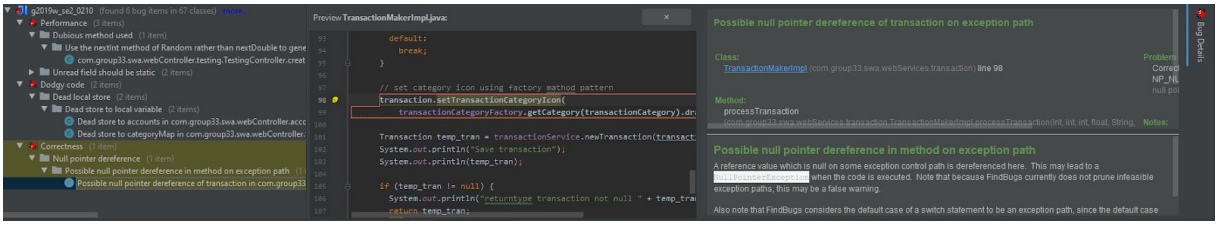


Two bugs were the result of dead local variables, meaning variables we created, never used and forgot to delete. This was an easy fix by just removing these variables.

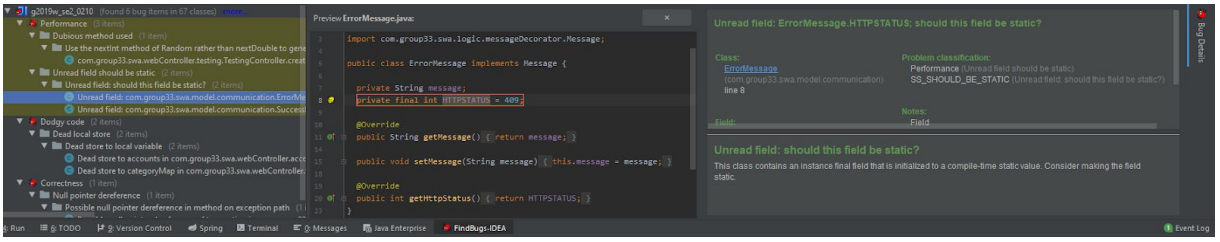




One bug was caused by an object that could have resulted in a null object. This was fixed by first checking if the object is null before working with it



In our opinion this bug is not warranted. These fields should not be static because they shouldn't be statically, accessible in other classes.



Java-Files without Testing-Classes, Total Lines of Code and commented lines

Source File	Total Lines	Source Code Lines	Comment Lines	Blank Lines
Total	**3642**	**2381**	**771**	**490**
Account.java	96	69	8	19
AccountContainerRepository.java	28	22	0	6
AccountController.java	177	113	52	12
AccountException.java	19	11	5	3
AccountIterator.java	31	24	0	7
AccountRepository.java	42	10	26	6
AccountService.java	58	12	38	8
AccountServiceImpl.java	77	42	26	9
AccountType.java	14	12	1	1
Application.java	13	10	1	2
BusinessFeesStrategy.java	20	11	5	4
BusinessTransaction.java	42	28	12	2
CashAccount.java	81	54	17	10
Container.java	5	4	0	1
CryptoFeesStrategy.java	21	11	5	5
CryptoTransaction.java	42	28	12	2
CurrencyType.java	15	12	1	2
Dividend.java	11	9	0	2
Education.java	11	9	0	2
ErrorMessage.java	21	17	0	4
Food.java	11	9	0	2
GiroAccount.java	66	42	16	8
IndexController.java	19	11	6	2
Iterator.java	7	5	0	2
MakeAccount.java	61	40	13	8
MakeCashAccount.java	62	51	5	6
MakeGiroAccount.java	31	26	0	5
MakeStudentAccount.java	32	27	0	5
Message.java	6	5	0	1
MessageDecorator.java	19	15	0	4
Observable.java	45	19	20	6
Observer.java	12	4	6	2
PrivateFeesStrategy.java	20	9	8	3
PrivateTransaction.java	43	27	13	3
Salary.java	11	9	0	2
Self.java	11	9	0	2
ServletInitializer.java	12	4	5	3
StudentAccount.java	86	58	18	10
SuccessMessage.java	22	17	0	5
TestingController.java	220	187	6	27
Transaction.java	170	124	21	25
TransactionCategory.java	16	13	1	2
TransactionCategoryFactory.java	34	31	0	3

TransactionCategoryInterface.java	5	4	0	1
TransactionContainerRepository.java	32	24	1	7
TransactionController.java	240	161	52	27
TransactionException.java	18	11	5	2
TransactionFeeCalculator.java	56	32	16	8
TransactionFeeCalculatorStrategy.java	14	5	7	2
TransactionIterator.java	32	24	1	7
TransactionMaker.java	15	13	0	2
TransactionMakerImpl.java	122	105	5	12
TransactionRepository.java	54	13	32	9
TransactionService.java	123	21	86	16
TransactionServiceImpl.java	273	207	26	40
TransactionStatisticController.java	277	224	15	38
TransactionType.java	17	10	5	2
Transportation.java	11	9	0	2
User.java	97	50	30	17
UserController.java	104	65	27	12
UserException.java	18	11	5	2
UserRepository.java	33	11	18	4
UserService.java	56	12	37	7
UserServiceImpl.java	74	39	25	10
WebSocketConfig.java	34	19	11	4
WebSocketHandler.java	80	48	20	12
WebsocketMessageDecorator.java	17	13	0	4

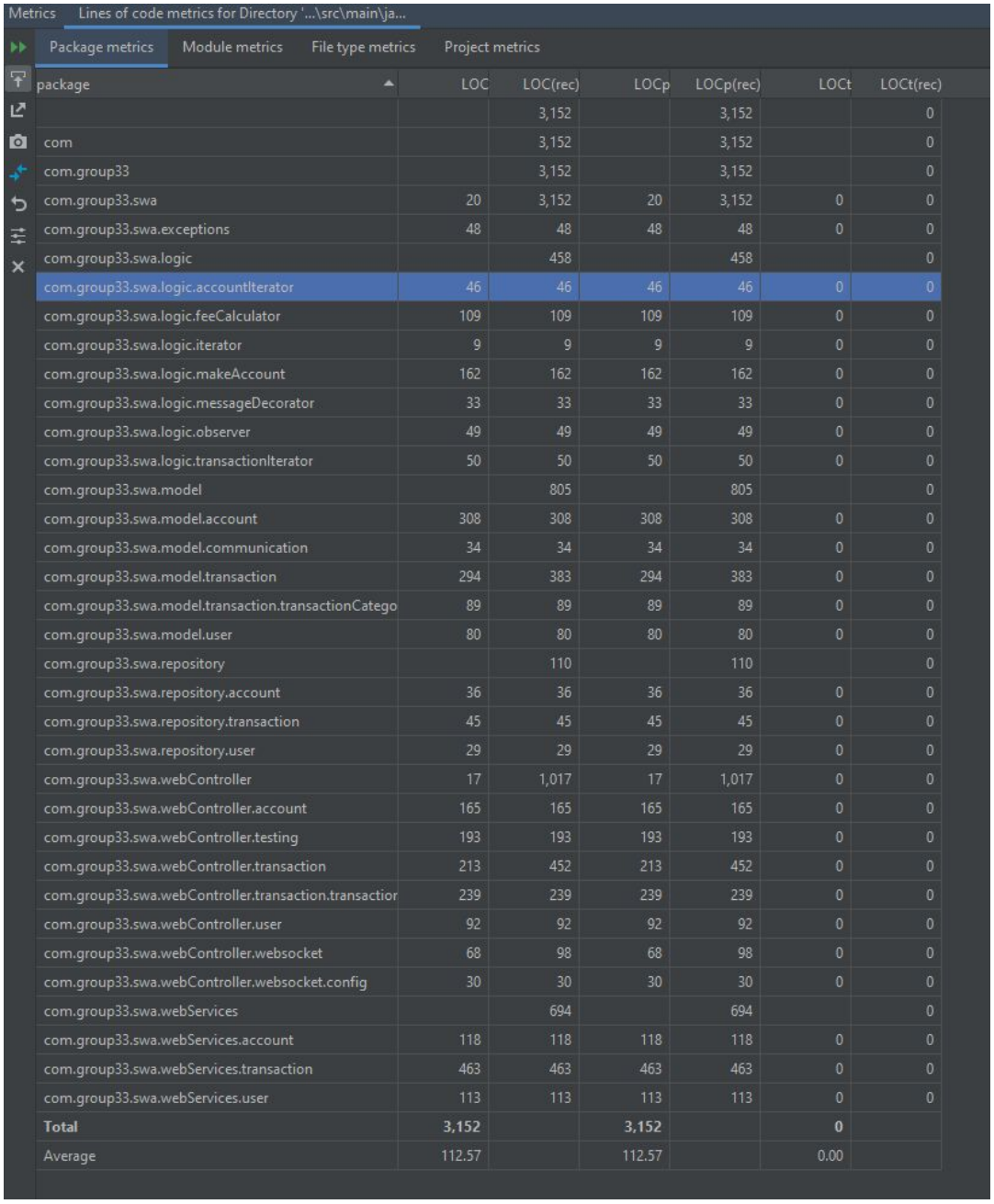
TestClasses

Source File	Total Lines	Source Code Lines	Comment Lines	Blank Lines
Total	**990**	**781**	**113**	**96**
AccountControllerTest.java	213	166	27	20
TransactionControllerTest.java	604	495	66	43
UserControllerTest.java	114	85	14	15
WebSocketTest.java	59	35	6	18

All in All

Source File	Total Lines	Source Code Lines	Comment Lines	Blank Lines
Total	**4632**	**3152**	**884**	**586**

Name of Packages and Lines of code in every package

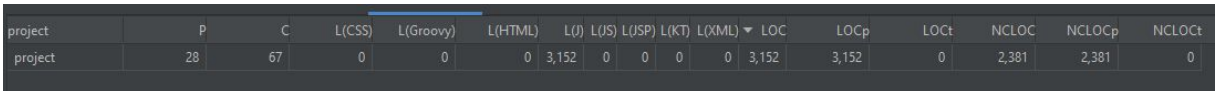


Metrics | Lines of code metrics for Directory '...\src\main\ja...

Package metrics | Module metrics | File type metrics | Project metrics

package	LOC	LOC(rec)	LOCp	LOCp(rec)	LOCt	LOCt(rec)
		3,152		3,152		0
com		3,152		3,152		0
com.group33		3,152		3,152		0
com.group33.swa	20	3,152	20	3,152	0	0
com.group33.swa.exceptions	48	48	48	48	0	0
com.group33.swa.logic		458		458		0
com.group33.swa.logic.accountIterator	46	46	46	46	0	0
com.group33.swa.logic.feeCalculator	109	109	109	109	0	0
com.group33.swa.logic.iterator	9	9	9	9	0	0
com.group33.swa.logic.makeAccount	162	162	162	162	0	0
com.group33.swa.logic.messageDecorator	33	33	33	33	0	0
com.group33.swa.logic.observer	49	49	49	49	0	0
com.group33.swa.logic.transactionIterator	50	50	50	50	0	0
com.group33.swa.model		805		805		0
com.group33.swa.model.account	308	308	308	308	0	0
com.group33.swa.model.communication	34	34	34	34	0	0
com.group33.swa.model.transaction	294	383	294	383	0	0
com.group33.swa.model.transaction.transactionCatego	89	89	89	89	0	0
com.group33.swa.model.user	80	80	80	80	0	0
com.group33.swa.repository		110		110		0
com.group33.swa.repository.account	36	36	36	36	0	0
com.group33.swa.repository.transaction	45	45	45	45	0	0
com.group33.swa.repository.user	29	29	29	29	0	0
com.group33.swa.webController	17	1,017	17	1,017	0	0
com.group33.swa.webController.account	165	165	165	165	0	0
com.group33.swa.webController.testing	193	193	193	193	0	0
com.group33.swa.webController.transaction	213	452	213	452	0	0
com.group33.swa.webController.transaction.transactior	239	239	239	239	0	0
com.group33.swa.webController.user	92	92	92	92	0	0
com.group33.swa.webController.websocket	68	98	68	98	0	0
com.group33.swa.webController.websocket.config	30	30	30	30	0	0
com.group33.swa.webServices		694		694		0
com.group33.swa.webServices.account	118	118	118	118	0	0
com.group33.swa.webServices.transaction	463	463	463	463	0	0
com.group33.swa.webServices.user	113	113	113	113	0	0
Total	**3,152**		**3,152**		**0**	
Average	112.57		112.57		0.00	

This my friend are the total number of packages

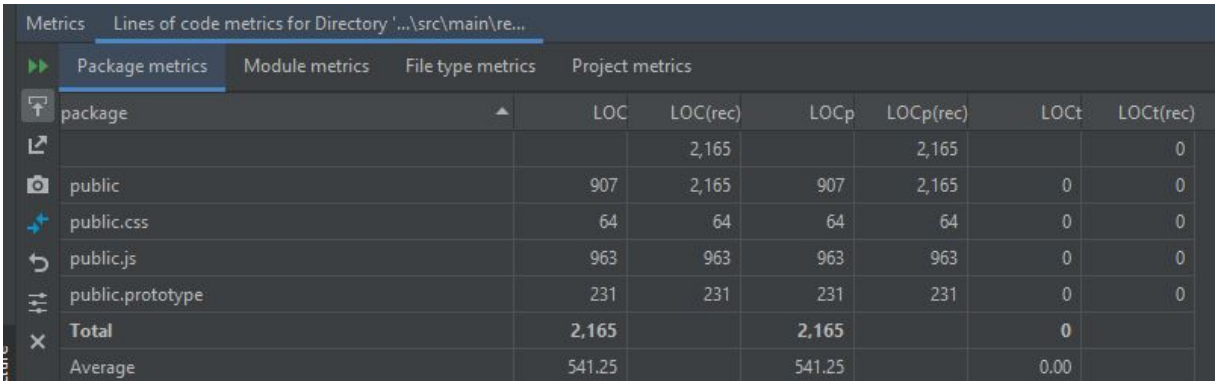


project	P	C	L(CSS)	L(Groovy)	L(HTML)	L(I)	L(JS)	L(JSP)	L(KT)	L(XML)	LOC	LOCp	LOCt	NCLOC	NCLOCp	NCLOCt
project	28	67	0	0	0	3,152	0	0	0	0	3,152	3,152	0	2,381	2,381	0

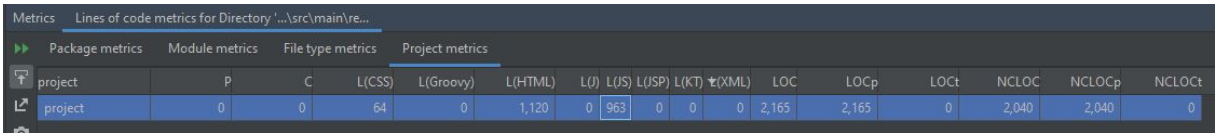
P…Number of Packages

C…Number of Classes

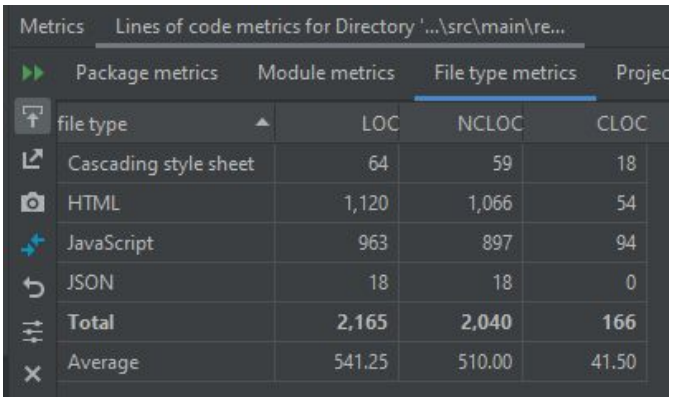
Frontend CodeMetrics



package	LOC	LOC(rec)	LOCp	LOCp(rec)	LOCt	LOCt(rec)
		2,165		2,165		0
public	907	2,165	907	2,165	0	0
public.css	64	64	64	64	0	0
public.js	963	963	963	963	0	0
public.prototype	231	231	231	231	0	0
Total	**2,165**		**2,165**		**0**	
Average	541.25		541.25		0.00	



project	P	C	L(CSS)	L(Groovy)	L(HTML)	L(J)	L(JS)	L(JSP)	L(KT)	L(XML)	LOC	LOCp	LOCt	NCLOC	NCLOCp	NCLOCt
project	0	0	64	0	1,120	0	963	0	0	0	2,165	2,165	0	2,040	2,040	0



file type	LOC	NCLOC	CLOC
Cascading style sheet	64	59	18
HTML	1,120	1,066	54
JavaScript	963	897	94
JSON	18	18	0
Total	**2,165**	**2,040**	**166**
Average	541.25	510.00	41.50

3.2 Test Cases for Functional Requirements

For FR1 through FR4 test classes for the controllers were created. It tests all the Spring requests and responses by using MockMVC and checking the expected result for a Http status code (mostly 200 or 400)

If the mock results in a database change, the test continues by trying to get the result from the database. With asserts it checks if the right amount and in the next step the right data was passed and saved.

We introduced users to our implementation which are not required for the FRs but nonetheless tested because a user is required for an account which is required for transactions.

For the User three functionalities are tested always including the database. Firstly the login method is checked. When the username exists it logs the user into his user-account. If the user does not exist an exception is thrown.

Secondly, the createNewUser function is tested by creating a new user and checking if an entry was made in the database and if the data was correctly stored.

Thirdly, getAllUsers was checked by creating 4 different Users and checking if all of them stored correctly in the database including the default user created on program initialization.

FR1 was tested by first setting up different users which are necessary to create accounts. Multiple accounts for multiple users were created with the different account types as parameters. It was checked if the right username and accounttype was saved to the database. Furthermore, it was tested if an exception is thrown, if an account with the same name as an existing would be created. It’s also tested if the default values work if no argument is passed for certain fields.

FR2 and FR3 go hand in hand for testing. Once again the transactions, users and different accounts had to be set up beforehand. On a side note to get unbiased test results the @DirtiesContext annotation was used to get a fresh program start and data for every test. This might not be the most performant solution but for such small scaled testing it seemed appropriate. Multiple transactions with different categories and all in all different data were created. Every time it was checked if the number of transactions in the database was equal to the number created. Furthermore, the equality of the saved data and the send data was checked. Different ways of getting transactions, for example by User ID or by Account ID were also checked including exceptions which should be thrown if creating a transaction was unsuccessful.

The different transaction strategies were also tested in this process. It was checked if the fee was calculated and saved correctly in the database in relation to the amount transferred.

Expenses and income are differentiated by a (-)minus in front of the amount transferred. We calculate the total account balance anew every time it is requested by summing up all transactions associated with the active account.

FR4 is tested during testing FR1-FR3 by always checking if the data in the database is equal to the data sent and if after calculations the right amount is saved. Therefore, it also checks that no nonsensical data is saved in the database.

The first part from FR5 was also tested in the TransactionController. Different accounts were set up and different transactions with different dates from and to differentiate accounts are processed. It is then tested if the functions return the correct years, months or days which were requested. It’s also checked what happens if a transaction or user from which data is requested does not exist.

In addition, a function was created to produce random transaction data from and to different user accounts and from and to different users. With this test data it was easier to test different functions of our implementation and saved some time by not having to create those manually all the time.

Some limitations for our test cases are that they are not designed to test the correctness of big data sets. The test cases were created for a smaller scope which made it easier and faster to test the functionalities implemented in our project. For example, writing tests for websockets is more initial or setup coding of the websocket than testing the actual cases.

3.3 Quality Requirements Coverage

QR1 - Comments and Documentation.

Comments were created, where deemed necessary for the whole Java project. A code documentation was exported as a JavaDoc. Comments were written in with Defensive Programming and Programming Practices in mind as already described in the corresponding part (2.2 & 2.3) in the document.

QR2 - Style Guide.

As a Style Guide Google Java Style Guide was chosen. It is one of the most commonly used Style Guides with very good documentation. The whole Java Project was formatted by this standard.

QR3 - Common coding practices.

We tried to use common coding practices as goods as possible. More about the usage can be found in section 2.2.

QR4 - defensive programming.

We tried to use defensive programming throughout our whole implementation. More about the usage can be found in section 2.3

QR5 - key design principles.

We tried to have a high level of abstraction through separating all necessary functions like creating and saving new transactions into separate classes. Furthermore, we tried to take advantage of the practice for developing one function for one task. The maximum hierarchy depth in our project is settled by 3. On the service-layer, each service, uses the implementation for reading and writing serialized models like User, Account, Transaction etc. These models need to be saved to a database, where only the services have the possibility to read and write values. The modularity is given, by separating the REST-API, which is communicating with the frontend whereas the services, are communicating with a database.

QR6 - testing.

The FRs were tested using Junit tests and by testing manually. JUnit tests and how the implementation process can be found in section 3.2. Additionally we created a function to create random test data to make it easier to manually test different functionalities.

QR7 - pattern implementation.

All patterns were considered for our implementation. We tried to find a use case in our implementation for each pattern. The patterns we used can be found in section 1.3.

QR8 - build tool.

Maven was used as a build tool. Why we chose maven over gradel can be found in section 1.1.2.

QR9 - launching.

Maven is used for building the jar file of the final solution. Maven provides a simple solution to create an executable jar file, just by defining the packaging process in the pom.xml

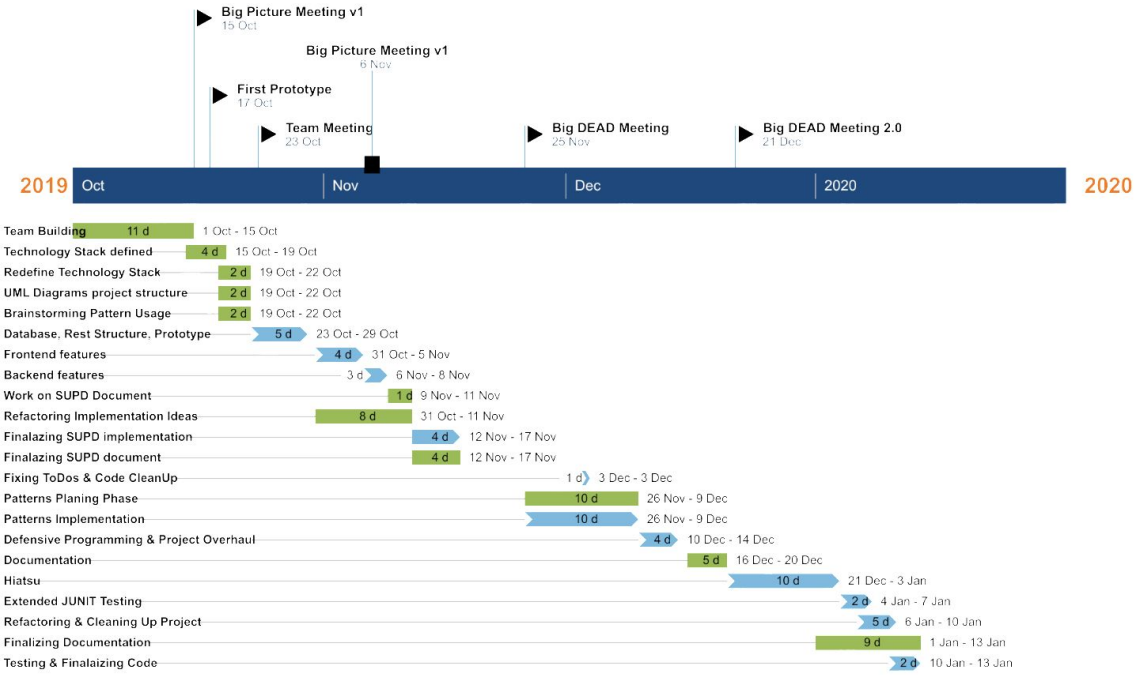
```
<project>
    <packaging>jar</packaging>
</project>
```

To take advantage of the maven configuration, execute the command in the project directory in the command line interface:

```
mvn package
```

4 Team Contribution

4.1 Project Tasks and Schedule



4.2 Distribution of Work and Efforts

Phases	**Alex**		**Lukas**		**Vincent**		
	SUPD	**DEAD**	*SUPD*	**DEAD**	*SUPD*	**DEAD**	**Total Time**
Planning	*8h*	**7h**	*4h*	**4h**	*5h*	**5h**	**33h**
Meetings	*10h*	**15h**	*10h*	**15h**	*10h*	**15h**	**75h**
Documentation	*8h*	**22h**	*8h*	**27h**	*4h*	**5h**	**70h**
Project init	*2h*	**0h**	*0:30h*	**0h**	*0h*	**0h**	**2:30h**
Database-init	*0h*	**0h**	*1h*	**0h**	*0h*	**0h**	**1h**
Prototype-frontend	*0:30h*	**0h**	*0:30h*	**0h**	*3h*	**0h**	**4h**
Backend	*37h*	**45h**	*37h*	**50h**	*5h*	**10h**	**184h**
Frontend	*1h*	**3h**	*1h*	**1h**	*8h*	**70h**	**84h**
Patterns	*4h*	**15h**	*4h*	**25h**	*10h*	**7h**	**65h**
Testing	*5h*	**8h**	*5h*	**15h**	*5h*	**7h**	**45h**
Total Time	*75,5h*	**115h**	*71h*	**137h**	*50h*	**119**	**567.5h**

A1 HowTo

Step by Step Guide build, start, initialize:

1. Clone the Git Repository

```
console> git clone https://lab.swa.univie.ac.at/submission/g2019w_se2_0210.git
```

2. Navigate to the subdirectory

```
console> cd /g2019w_se2_0210
```

3. Checkout SUPD Branch

```
console/g2019w_se2_0210>  git checkout 2019w_se2_dead
```

4. Maven clean to prevent building errors

```
console/g2019w_se2_0210>  mvn clean
```

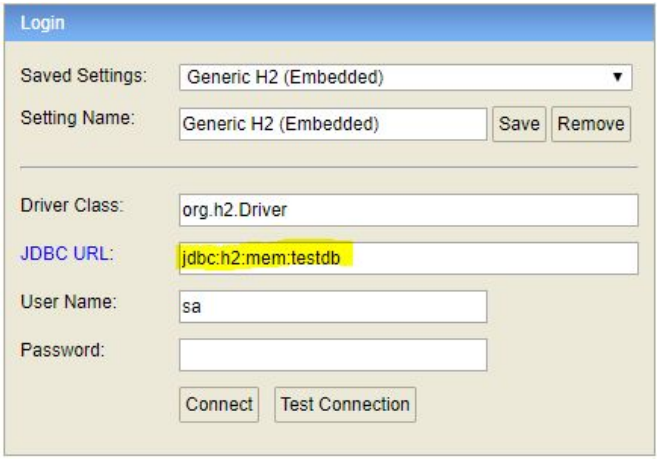
5. Build the jar file

```
console/g2019w_se2_0210>  mvn package
```

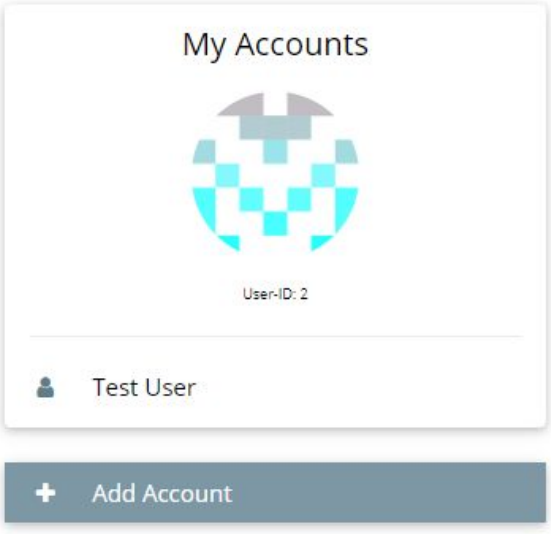
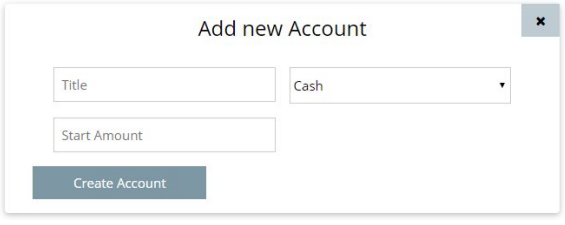
6. Run the jar file

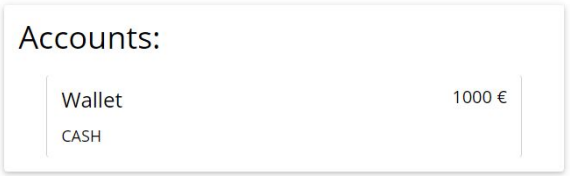
```
console/g2019w_se2_0210>  java -jar ./g2019w_se2_2010-0.0.1-SNAPSHOT.jar
```

7. to access the expense tracker open http://localhost:8080 in your browser
8. <Optional> For Database Access navigate to http://localhost:8080/h2 and change the JDBC URL to that shown in picture

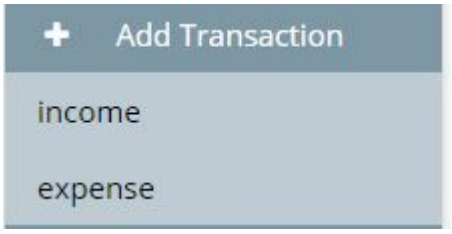


Login Screen

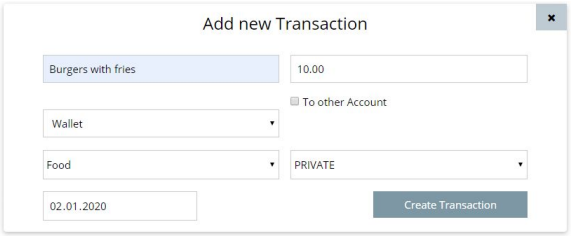
Login Screen - Just type in a username to create an account.
Overview

Overview 1 - When you login for the first time, there isn't much to see. Add an Account.

Select an Account Type and fill in all Fields.



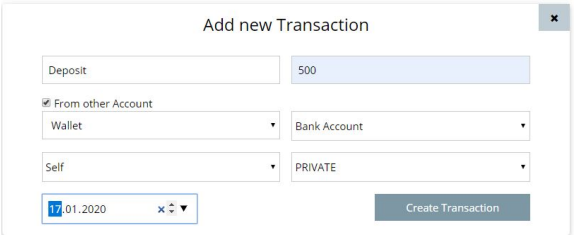
After submitting there is a new Account in your Accountlist



Now we can add a Transaction. Click on Add Transaction, then select if you recieve or spend money.

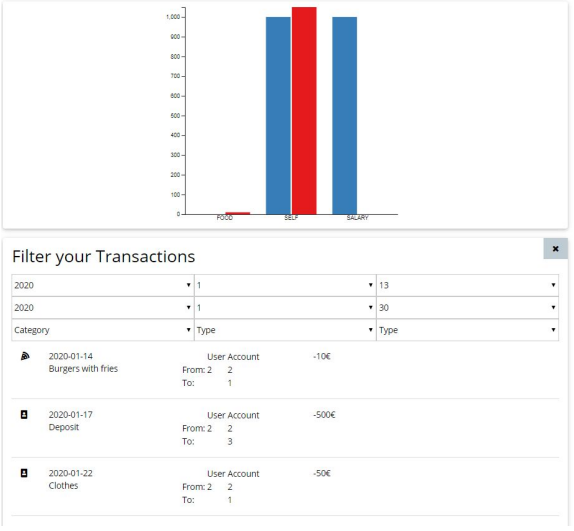


Fill in all Fields.



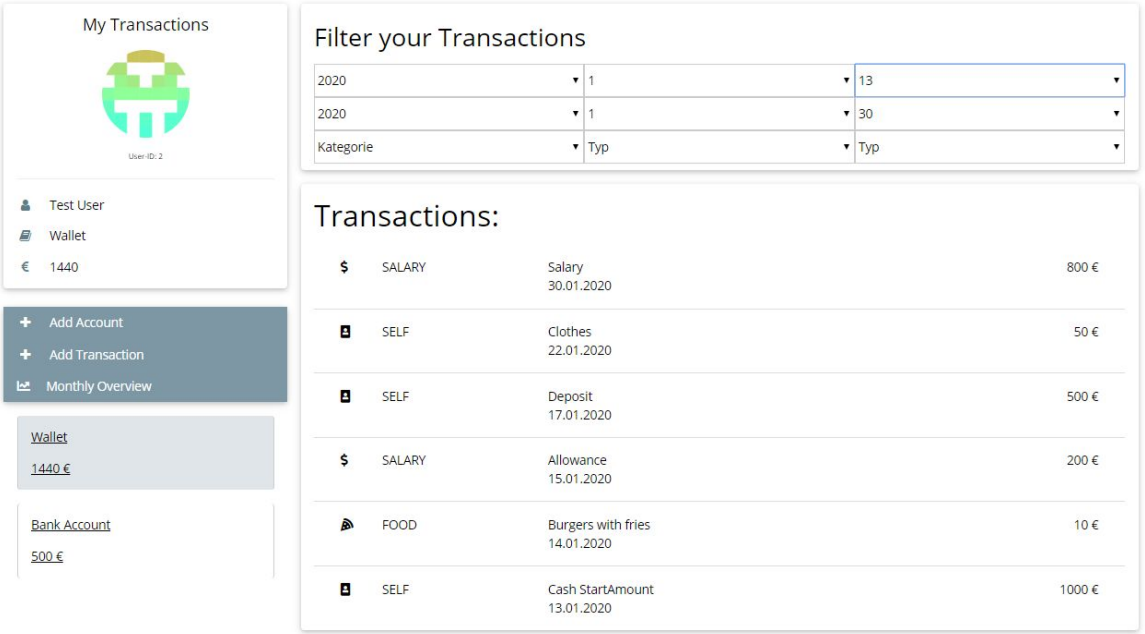
If you have multiple Accounts you can move money from one to another

Click “Filter all Transactions” to filter Transactions from all Accounts

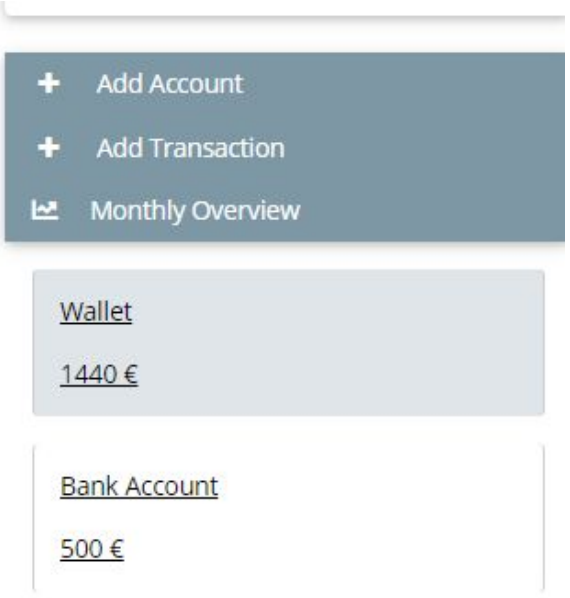


By Clicking on one of the Accounts you get the transactions for that account

Account



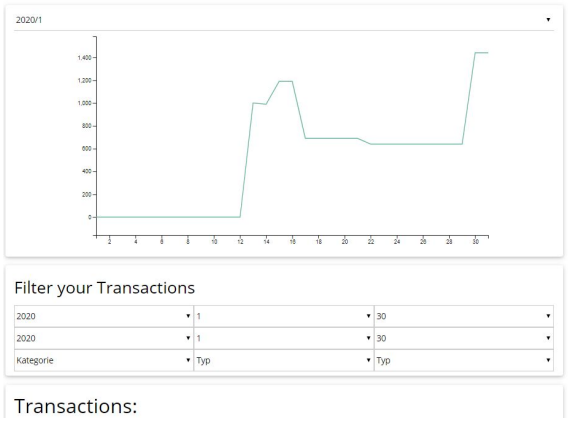
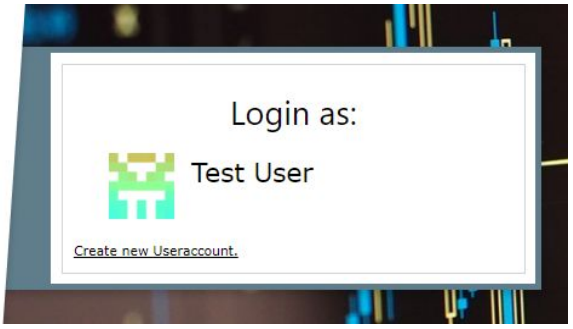
On the Account Page you can filter your Transactions, add new Transactions, get Monthly Overviews,



Add Account and Add transaction work just as in the Overview site.

Monthly Overview gives you a Line Chart over a selected month

At the bottom of the left bar you can select a different Account.

	Select a month from the dropdown Bar at the top, and you get a nice chart.
	You can get back to the overview by clicking Home in the top left corner.
	By hovering over the Image in the top-right corner you can click the Logout button select another user account.
	When you come back, just select your user account or create a new one