

Software Engineering 2

DEAD REPORT

Status Update Report

Team number:	5
--------------	---

Team member 1	
Name:	Oleksandra Lylak
Student ID:	01400128
E-mail address:	a01400128@unet.univie.ac.at

Team member 2	
Name:	Antonov Nikita
Student ID:	01348746
E-mail address:	a01348746@unet.univie.ac.at

Team member 3	
Name:	Redzic Majda
Student ID:	01552937
E-mail address:	a01552937@unet.univie.ac.at

Team member 4	
Name:	Barloga Aneta Katarzyna
Student ID:	11711138
E-mail address:	a11711138@unet.univie.ac.at

1 Final Design

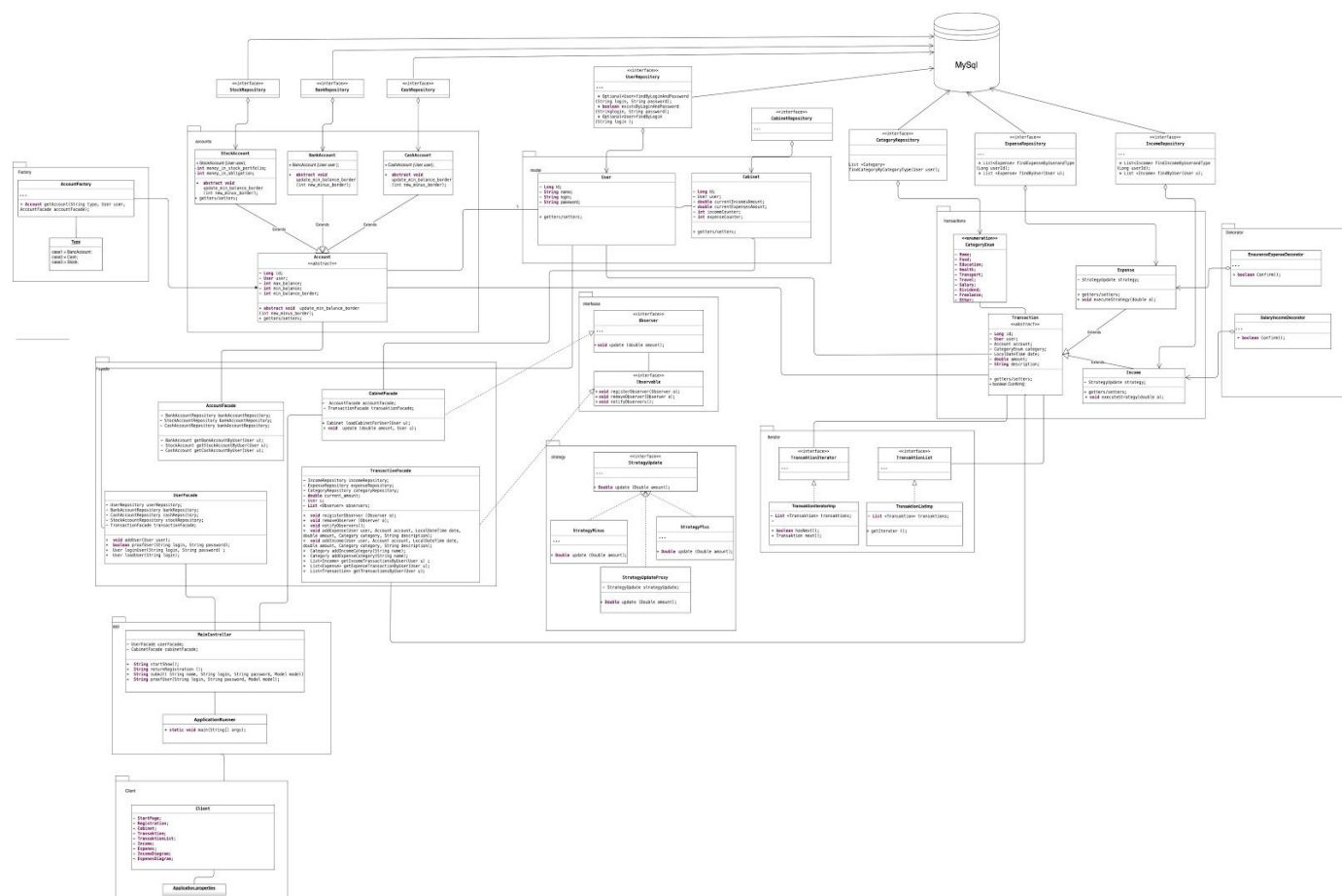
1.1 Design Approach and Overview

In this semester we implemented a personal expense tracker for end users. The main idea for implementation was to perform income / expense transactions and display those actions in a graphical form.

An ExpenseTracker is a computer program that allows a user to store electronic money in different accounts and manage income/expense in different categories.

1.1.1 Class Diagrams

We took the necessary parts from the SUPD design and created a final class diagram for the initial implementation.



See uploaded file DEAD Classdiagram.svg

1.1.2 Technology Stack

For the implementation use we the project features:

Environment	IntelliJ IDEA
Application framework	Spring Framework
Language	Java
Build tool	Maven
Database	MySql/Hibernate
User Interface	HTML

1.3 Design Patterns

1.3.1 Observer pattern

We used Observer Pattern to define the relationship between CabinetFacade and TransactionFacade, so when we have new Income/Expense, the Cabinet through CabinetFacade notified and updated automatically. The object that is being viewed, the subject, is Observable and the object that are viewing the state changes is Observer.

The Observable class refers to the Observer interface that contains update() method for updating state.

```
package expenseTracker.app.interfaces;

public interface Observable {
    void registerObserver(Observer o);
    void removeObserver(Observer o);
    void notifyObservers();
}
```

- *registerObserver(o)* and *removeObserver(o)* are methods to add and remove observers respectively. *notifyObservers()* is called when the amount is changed or modified and the observers need to be supplied with the latest data.

```
package expenseTracker.app.interfaces;

import expenseTracker.app.model.User;

public interface Observer {
    void update (double amount, User u);
}
```

- So every time changes happen in TransactionFacade, the state (the money amount) in the CabinetFacade class will be updated.

```
@Component
public class TransactionFacade implements Observable {

    @Autowired
    private IncomeRepository incomeRepository;

    @Autowired
    private ExpenseRepository expenseRepository;

    @Autowired
    private CategoryRepository categoryRepository;

    private double current_amount;

    private User u;

    private List<Observer> observers;

    public TransactionFacade() { this.observers = new ArrayList<>(); }

    @Override
    public void registerObserver(Observer o) { observers.add(o); }

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

    @Override
    public void notifyObservers() {
        for (Observer observer : observers)
            observer.update(current_amount, u);
    }
}
```

1.3.2 Strategy pattern

Strategy pattern is implemented to show the changes of the subclasses Income and Expense which extend the class Transaction. Changes in Transaction will be added and saved in the repositories and returned to the user - Income/StrategyPlus or Expense/StrategyMinus will be updated (update() method) and observed in the StrategyUpdate.

```
1 package expenseTracker.app.interfaces.strategy;
2
3 import expenseTracker.app.interfaces.StrategyUpdate;
4
5 public class StrategyPlus implements StrategyUpdate {
6
7
8     public Double update( Double amount) {
9         return amount;
10    }
11 }
```

```
1 package expenseTracker.app.interfaces.strategy;
2
3 import expenseTracker.app.interfaces.StrategyUpdate;
4
5 public class StrategyMinus implements StrategyUpdate {
6
7
8     public Double update( Double amount) {
9         return -amount;
10    }
11 }
```

1.3.3 Facade pattern

We used the facade pattern in order to reduce complexity of the program. The facades provide a convenient way to access all methods for the MainController without revealing the inner logic of the classes.

Example usage:

Method `getStockAccountByUser()` is implemented in `TransactionFacade` and retrieves the stock account saved in `StockAccountRepository`.

```
StockAccount getStockAccountByUser(User u) throws NoAccountFoundException {
    Optional<StockAccount> stockAccount = stockAccountRepository.findStockAccountByUserAndType(u);
    if(!stockAccount.isPresent())
        throw new NoAccountFoundException();

    return stockAccount.get();
}
```

This method provides simplified functionality for `CabinetFacade` where the method `loadCabinetForUser()` loads all data regarding the user.

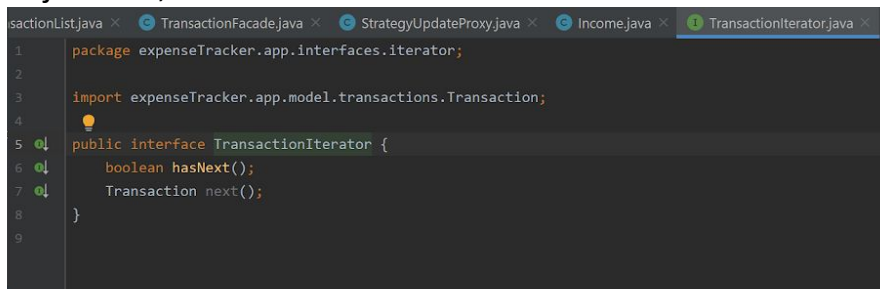
```
public Cabinet loadCabinetForUser(User u){

    try {
        BankAccount bankAccount = accountFacade.getBankAccountByUser(u);
        CashAccount cashAccount = accountFacade.getCashAccountByUser(u);
        StockAccount stockAccount = accountFacade.getStockAccountByUser(u);

        Cabinet cabinet = new Cabinet(
```

1.3.4 Iterator pattern

TransactionIterator defines the methods as described in the Iterator pattern. It has a “hasNext” and a “next” method. The “hasNext” method is used for checking if there is a next object. If there is an object left in the iteration it returns true. It returns false if there is no more object for iteration. The “next” method returns the object of the current iteration index. If there is not any object left, then it returns null.



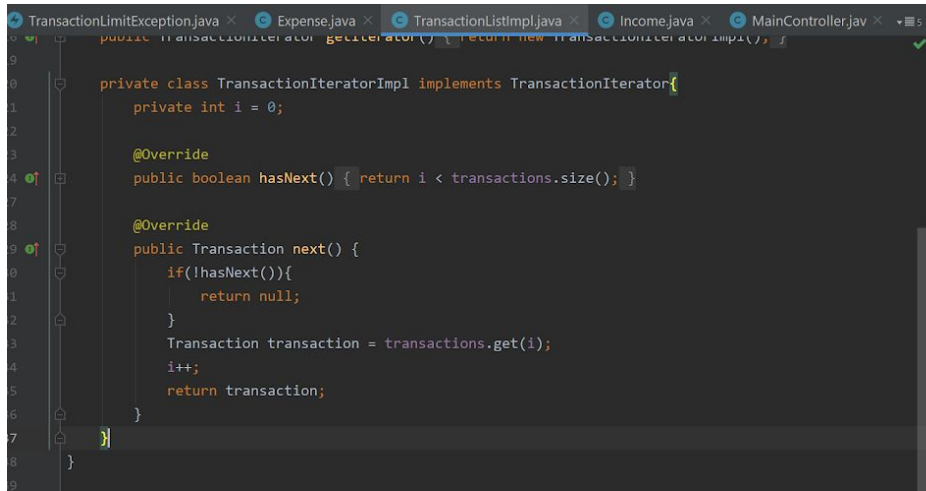
```
1 package expenseTracker.app.interfaces.iterator;
2
3 import expenseTracker.app.model.transactions.Transaction;
4
5 public interface TransactionIterator {
6     boolean hasNext();
7     Transaction next();
8 }
9
```

TransactionList interfaces defines a container that returns an iterator (in our case TransactionListImpl), so the caller can iterate over the containing elements.



```
1 package expenseTracker.app.interfaces.iterator;
2
3 public interface TransactionList {
4     TransactionIterator getIterator();
5 }
6
```

The TransactionListImpl is a concrete class of the TransactionList interface. Internally it holds a collection of elements of the type “List” that is used to iteration. The TransactionIteratorImpl is the implementation for the TransactionIterator and is not visible for callers (so no one can make wrong use of this implementation and it is tightly coupled to the TransactionList). Basically it uses the list and an index variable that is incremented every time next() is called. In the “hasNext” method the index variable is used to calculate if there is another object in the list (returns true) or it is already at the end (returns false).



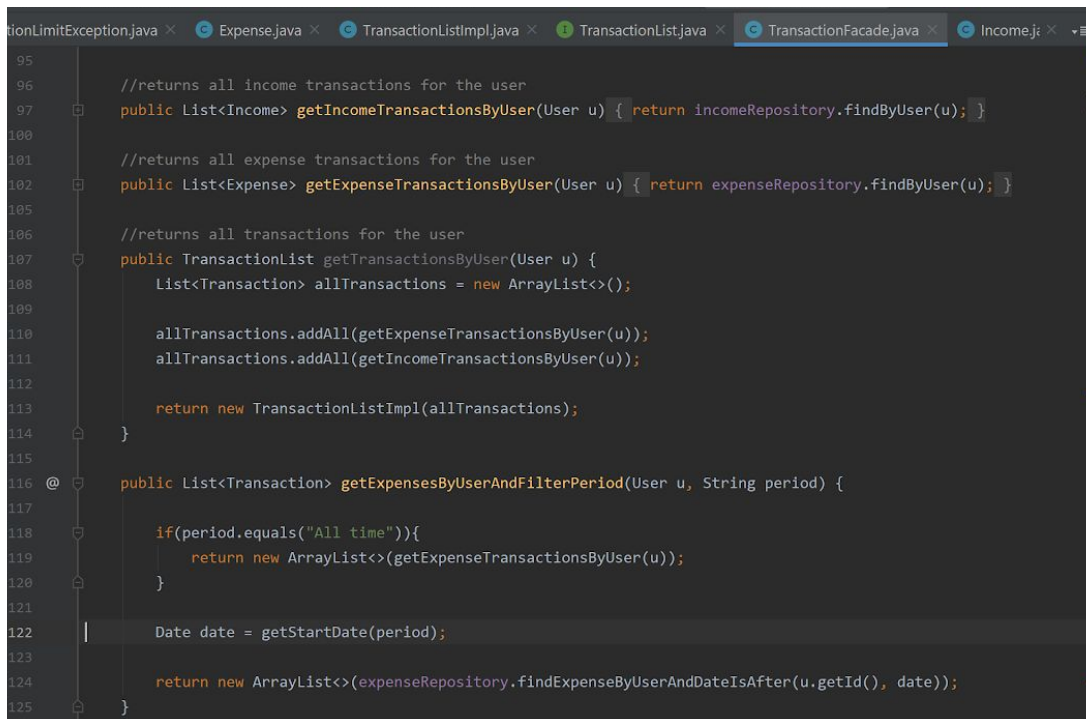
```
TransactionLimitException.java × Expense.java × TransactionListImpl.java × Income.java × MainController.java ×
public TransactionIterator getIterator() { return new TransactionIteratorImpl(); }

private class TransactionIteratorImpl implements TransactionIterator {
    private int i = 0;

    @Override
    public boolean hasNext() { return i < transactions.size(); }

    @Override
    public Transaction next() {
        if (!hasNext()) {
            return null;
        }
        Transaction transaction = transactions.get(i);
        i++;
        return transaction;
    }
}
```

The TransactionList is used in the TransactionFacade class and is initialized with all income and expenses (both of type Transaction) of the customer.



```
TransactionLimitException.java × Expense.java × TransactionListImpl.java × TransactionList.java × TransactionFacade.java × Income.java ×
//returns all income transactions for the user
public List<Income> getIncomeTransactionsByUser(User u) { return incomeRepository.findByUser(u); }

//returns all expense transactions for the user
public List<Expense> getExpenseTransactionsByUser(User u) { return expenseRepository.findByUser(u); }

//returns all transactions for the user
public TransactionList getTransactionsByUser(User u) {
    List<Transaction> allTransactions = new ArrayList<>();
    allTransactions.addAll(getExpenseTransactionsByUser(u));
    allTransactions.addAll(getIncomeTransactionsByUser(u));
    return new TransactionListImpl(allTransactions);
}

public List<Transaction> getExpensesByUserAndFilterPeriod(User u, String period) {
    if (period.equals("All time")) {
        return new ArrayList<>(getExpenseTransactionsByUser(u));
    }
    Date date = getStartDate(period);
    return new ArrayList<>(expenseRepository.findExpenseByUserAndDateIsAfter(u.getId(), date));
}
```

1.3.5 Proxy pattern

We used the Proxy pattern to enhance the different StrategyUpdate classes with limit checking. The StrategyUpdateProxy class is the proxy. The proxy class implements the common StrategyUpdate interface to look like a StrategyUpdate class for its caller. The only field in this class is the to-be-proxied StrategyUpdate instance that is used for delegating the “update” method call. Before the call is delegated to the strategyUpdate field, a limit

check is executed. If the amount is out of the set boundaries the update fails with a TransactionLimitException.

```
package expenseTracker.app.interfaces.strategy;

import expenseTracker.app.interfaces.StrategyUpdate;
import expenseTracker.app.model.transactions.TransactionLimits;
import expenseTracker.app.model.transactions.exceptions.TransactionLimitException;

public class StrategyUpdateProxy implements StrategyUpdate {

    private StrategyUpdate strategyUpdate;

    public StrategyUpdateProxy(StrategyUpdate strategyUpdate) { this.strategyUpdate = strategyUpdate; }

    @Override
    public Double update(Double amount) {
        if(TransactionLimits.isLimitReached(amount)){
            throw new TransactionLimitException("Transaction limit for one transaction is reached. amount is too high");
        }
        return strategyUpdate.update(amount);
    }
}
```

The Income and Expense class make use of this proxy for their strategy.

```
package expenseTracker.app.model.transactions;

import ...

@Entity
public class Expense extends Transaction {

    @Transient
    private StrategyUpdate strategy;

    public Expense() {
        super();
        strategy = new StrategyUpdateProxy(new StrategyMinus());
    }

    public void executeStrategy(Double a) { super.setAmount(strategy.update(a)); }

    @Override
    public boolean Confirm() { return false; }
}
```



```
sactionListImpl.java x TransactionList.java x TransactionFacade.java x StrategyUpdateProxy.java x Income.java x
1 package expenseTracker.app.model.transactions;
2
3 import ...
4
5
6
7
8
9
10
11
12 @Entity
13 public class Income extends Transaction {
14
15     @Transient
16     private StrategyUpdate strategy;
17
18     public Income() {
19         super();
20         strategy = new StrategyUpdateProxy(new StrategyPlus());
21     }
22
23     public void executeStrategy(Double a) { super.setAmount(strategy.update(a)); }
24
25
26
27     @Override
28     public boolean Confirm() { return false; }
29
30 }
31
32
```

1.3.6 Factory pattern

With the help of Factory pattern we simplified declaration of account class. The types of Account would be declared in a switch expression using a String so that the correct account type is returned each time Account is called.

```
StrategyUpdateProxy.java x AccountFactory.java x
package expenseTracker.app.model.factory;
import ...
public class AccountFactory {
    public static Account getAccount(String type, User user, AccountFacade accountFacade) throws NoAcc
    {
        switch(type) {
            case "Bank account":
                return accountFacade.getBankAccountByUser(user);
            case "Cash":
                return accountFacade.getCashAccountByUser(user);
            case "Stock":
                return accountFacade.getStockAccountByUser(user);
        }
        return null;
    }
    public static boolean isTransactionFromAccount(String type, Transaction t) {
        switch (type) {
            case "Bank account":
                return (t.getAccount() instanceof BankAccount);
            case "Cash":
                return (t.getAccount() instanceof CashAccount);
            case "Stock":
                return (t.getAccount() instanceof StockAccount);
        }
        return false;
    }
}
```

1.3.7 Decorator pattern

Decorator pattern is implemented for two constant issues / receipts that take place every month with the same parameters.

The class `EnsuranceExpenseDecorator` implements the monthly expense for health insurance. Decorator class `SalaryIncomeDecorator` implements monthly salary income.

```
import expenseTracker.app.model.transactions.Expense;

public class EnsuranceExpenseDecorator extends Expense {

    @Override
    public boolean Confirm(){

        // Monthly debit for insurance
        //last day of the month

        return super.Confirm();
    }
}
```

With the help of decorator classes, the classes for constant tasks do not have to be changed but only the method required for this.

2 Implementation

2.1 Overview of Main Modules and Components

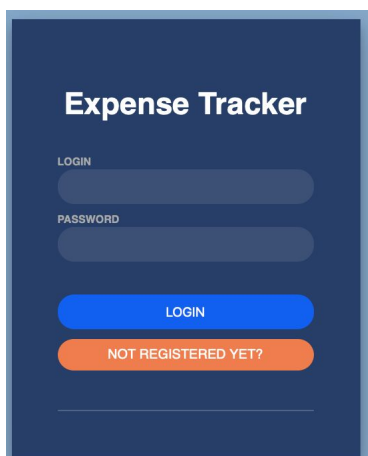
To start and test the project, the DB MySQL must run and start the program. We have realized our requests using `@RequestMapping`.

To run the project use this link:

<http://localhost:8080/>

The first `RequestMapping` is the "StartPage" with the login

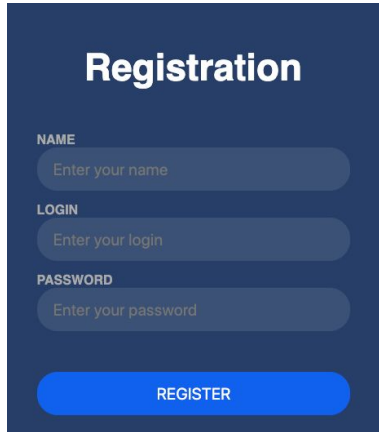
<http://localhost:8080/start>

A screenshot of a web application's login page. The page has a dark blue background. At the top, the text "Expense Tracker" is displayed in white. Below it, the word "LOGIN" is shown in small white capital letters. There are two input fields: the first is labeled "PASSWORD" in small white capital letters and contains a white dot, indicating a password field. Below the input fields are two buttons: a blue button with the text "LOGIN" in white, and an orange button with the text "NOT REGISTERED YET?" in white. At the bottom, there is a thin white horizontal line.

Here you can enter the user login and password which will load the Cabinet for the user

The second RequestMapping is the "Registration" where the registration information is located. This window can be reached in two ways: via Login and the button "Not registered yet?" Or via the link:

<http://localhost:8080/get/registration>

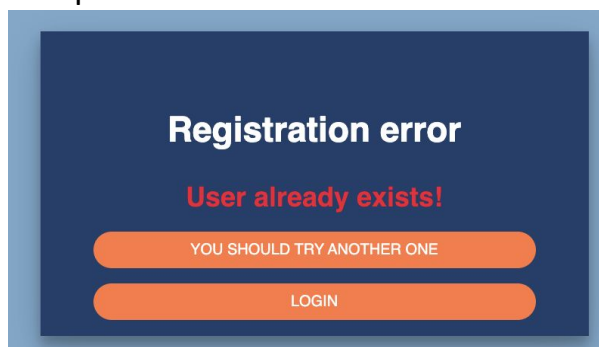
A registration form with a dark blue background. At the top, the word "Registration" is written in white. Below it, there are three input fields: "NAME" with the placeholder "Enter your name", "LOGIN" with the placeholder "Enter your login", and "PASSWORD" with the placeholder "Enter your password". At the bottom, there is a blue button labeled "REGISTER".

If the user is not yet registered, he needs to by following the “not registered yet?” button and move on to the registration Window.



Upon registration, a new user is inserted into the DB. If the registration is completed correctly, the user is immediately sent to LoginPage so that they could continue to log in to the program.

A duplicate registration is not allowed, should the user try to register with a username that exists in the database, the program will respond with an exception and as well as inform the user on the screen.



The next request is responsible for checking if the user has a valid login and password and throws an exception, should either of them be incorrect. If the user data is correct, the cabinet is returned.

By having an account, the user will have a simple and clean preview of all transactions (also shown as a list), account balances and have control over his expenses, get warnings if he reaches expense limits, salary information ect:

<http://localhost:8080/get/validate>

Cabinet

Bank Account Bank saldo: 4305,00€	Cash Account Cash saldo: 0,00€	Stock Account Stock saldo: 2000,00€
---	--	---

Total saldo: 8695,00€

Select period

Income List All income: 7500,00€	Expenses List All expenses: -1195,00€
--	---

+ Income Transaction	+ Expense Transaction
-----------------------------	------------------------------

Transactions like Income and Expense can be executed:

<http://localhost:8080/transaction?type=expense>

<http://localhost:8080/transaction?type=income>

New Transaction

200

Education ▼

Bank account ▼

14.01.2020

New Skripten

Cancel Insurance Confirm

New Transaction

300

Other ▼

Cash ▼

15.01.2020

Private tutoring

Cancel Salary Confirm

Here u can see the Salary and Insurance being executed (Decorator Pattern):

New Transaction

65

Health

Bank account

31.01.2020

Current month insurance

Cancel Insurance Confirm

New Transaction

1000

Salary

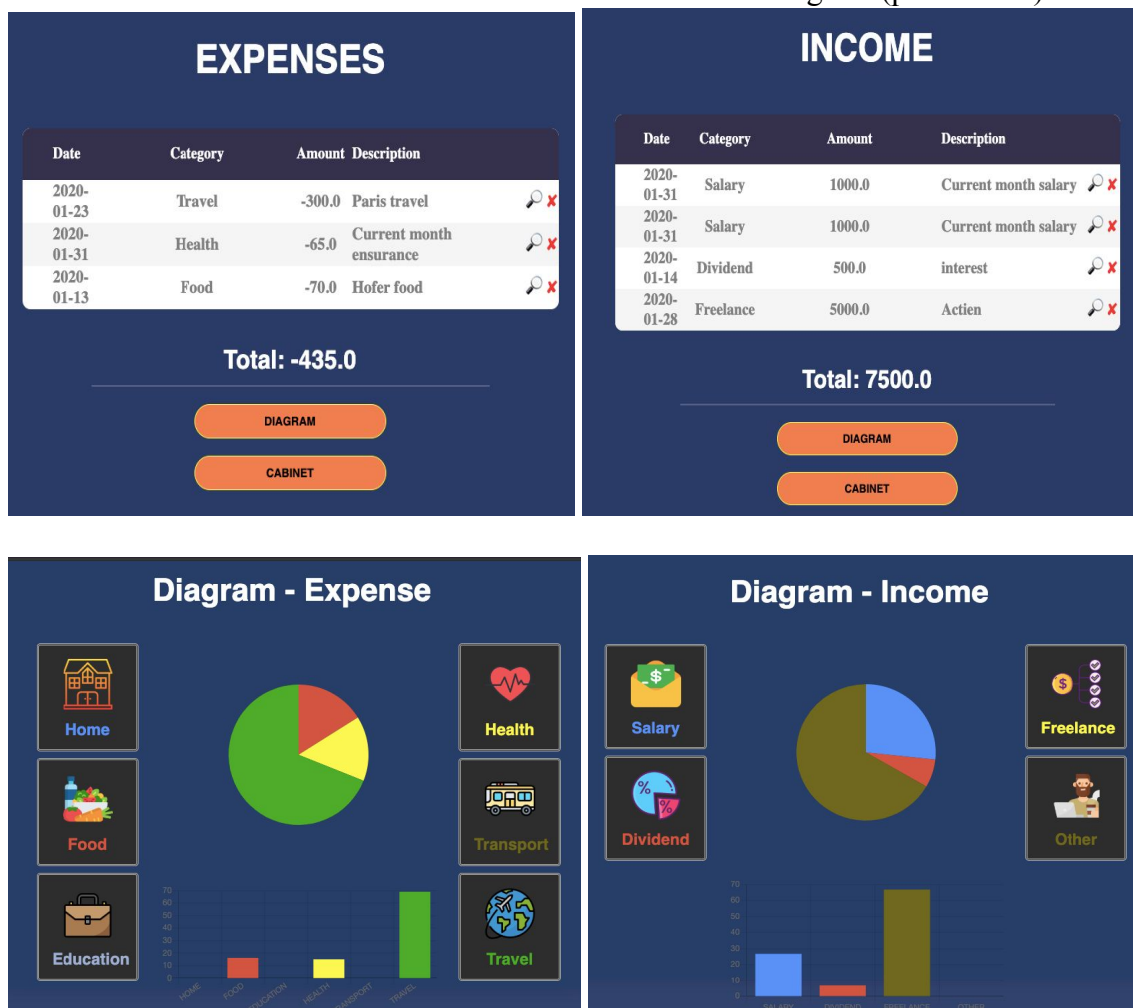
Bank account

31.01.2020

Current month salary

Cancel Salary Confirm

The executed transactions can be shown as a list and also a diagram (pie and bar):

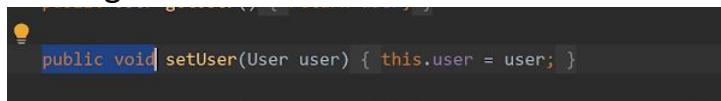


2.2 Coding Practices

For the project implementation we tried to make the code as readable and easy as possible so that the other developers and users can easily analyze and understand it. It was of high importance for us that we establish a main structure from the beginning so that it won't require major changes as the project expands. Based on this principle, it is also possible to avoid errors and improve the quality of the code.

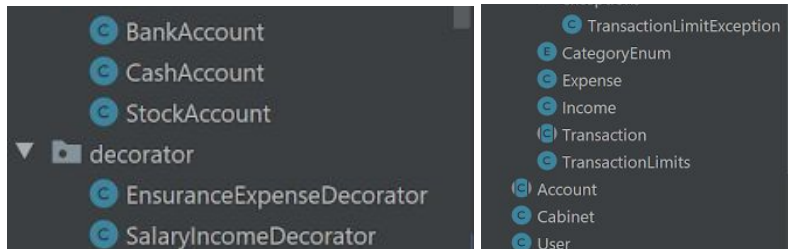
We agreed on using self-explanatory class/methods/variable names and also comment the code snippets in order for anyone to understand why the code works/is written that way. We achieved that by using Google Java Style Guide (@Override, Class names are written in UpperCamelCase, Package names are all lowercase, method names are written in lowerCamelCase ect). Examples:

Naming the methods:



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

Naming our classes:

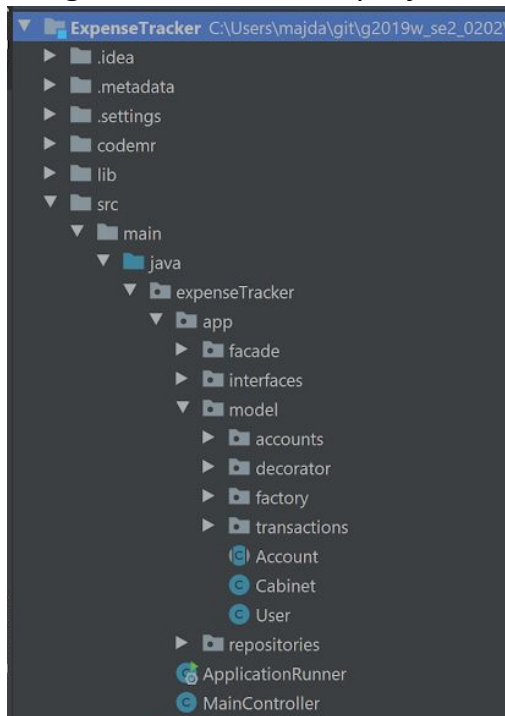


We also focused on maintaining a clean format and a logical organization of the code, therefore readability issues shouldn't emerge.

We used @Override to allow our subclasses or child classes to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes. It also improves the readability of the code. So if we change the signature of overridden method then all the sub classes that overrides the particular method would throw a compilation error, which would eventually help us to change the signature in the sub classes.

```
@Override
public void update_min_balance_border(int new_minus_border) {
    if (new_minus_border < 0) {
        super.setMin_balance_border(0);
    } else
        super.setMin_balance_border(new_minus_border);
}
```

Rough structure of the project:



At the end we focused on:

1. Know what the code block must perform
2. Maintain naming conventions which are uniform throughout.
3. Correct errors as they occur.
4. Keeping our code simple

2.3 Defensive Programming

Defensive programming is a coding practice that ensures the timely recognition of bugs by anticipating scenarios in which something could go wrong. The importance of implementing those techniques/requirements often depends on the level of security needed for the project.

Using the principles of defensive programming explicitly we handle errors by throwing the corresponding exceptions in case if some block of the code fails. We have Exceptions for the following Classes: AccountFacade, CabinetFacade, TransactionFacade, UserFacade

Example for AccountFacade :

```
import ***

@Component
public class AccountFacade {

    @Autowired
    BankAccountRepository bankAccountRepository;

    @Autowired
    StockAccountRepository stockAccountRepository;

    @Autowired
    CashAccountRepository cashAccountRepository;

    //get methods for user accounts
    public BankAccount getBankAccountByUser(User u) throws NoAccountFoundException {
        //Optional<> avoids nullpointer exceptions and will throw NoAccountFoundException if isPresent() false.
        Optional<BankAccount> bankAccount = bankAccountRepository.findBankAccountByUserAndType(u);
        if(!bankAccount.isPresent())
            throw new NoAccountFoundException();

        //Optional<> needs to be fetched back with get()
        return bankAccount.get();
    }

    public StockAccount getStockAccountByUser(User u) throws NoAccountFoundException {
        Optional<StockAccount> stockAccount = stockAccountRepository.findStockAccountByUserAndType(u);
        if(!stockAccount.isPresent())
            throw new NoAccountFoundException();

        return stockAccount.get();
    }

    public CashAccount getCashAccountByUser(User u) throws NoAccountFoundException {
        Optional<CashAccount> cashAccount = cashAccountRepository.findCashAccountByUserAndType(u);
        if(!cashAccount.isPresent())
            throw new NoAccountFoundException();

        return cashAccount.get();
    }
}
```

3. Software Quality

3.1 Code Metrics

The key to quality assessment ultimately lies in the compliance and stability of our code. It is very important to us that we always validate the goal of our project and determine if we program the right product. We have analyzed our code and the software with the tool “CodeMR”.

This is a plugin for the IDE in IntelliJ IDEA and it represents all the necessary information that we need.

The important information:

Total lines of code: 724

Number of classes: 43

Number of packages: 14

Number of external packages: 11

Number of external classes: 34

Number of problematic classes: 0

Number of highly problematic classes: 0

The last phase of programming the project, we solved the problems we had in the SUPD phase. Showing that we have no problematic classes this also determines the reliability, reusability and understandability of the project and the list of all the classes (shown below).

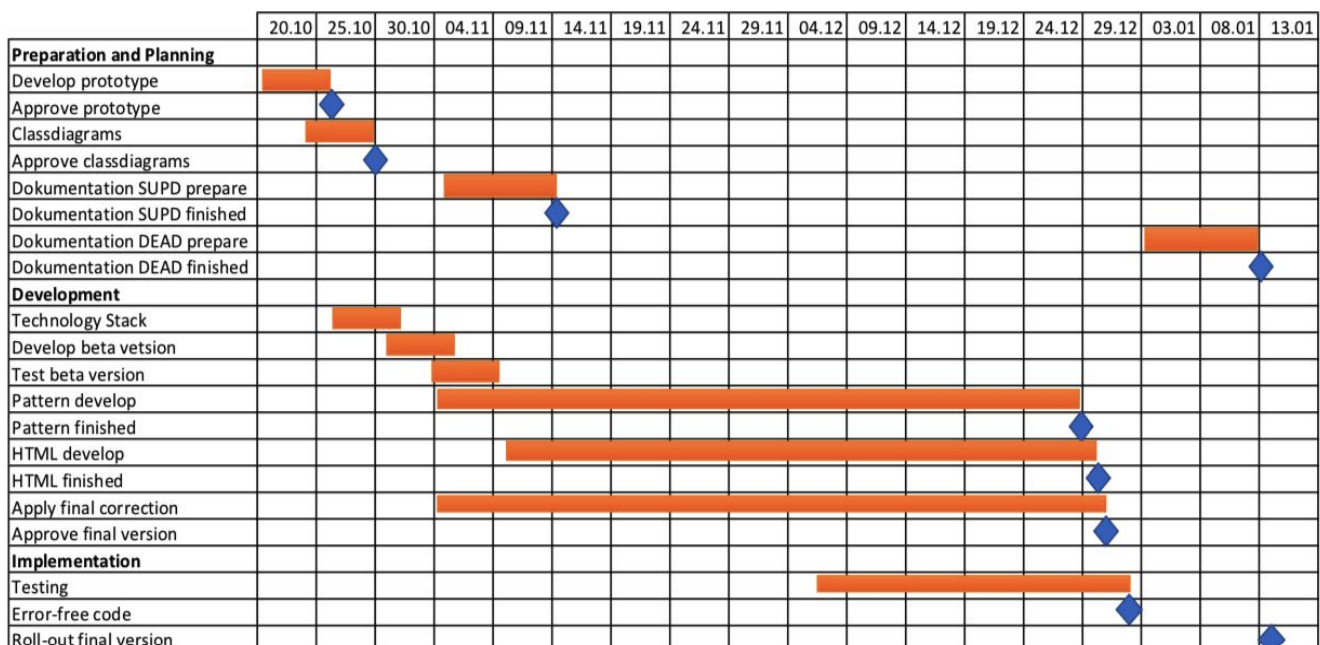
4.Team Contribution

4.1 Project Tasks and Schedule

Planning of our project is characterized by regular rectification, plan updates and iterative refactorization, not only at the stage of development, but also in the implementation process. The schedule, presented in the form of a Gantt chart, provides an instrumental opportunity to better assess the resource composition and the relationship of work. We've defined our diagrams for the total project so we can more easily visualize the complete work process.

Some processes are based or dependent on others, and some require the previous process to finish first to start the next phase (See picture).

Thanks to the Gantt diagram, we have received all appointments and have finished implementing them according to the plan.



Distribution of work and efforts

Name	Contributions
Antonov Nikita	Technology stack selection. Strategy und Observer pattern. SQL konfiguration. Some methods in controller konfiguration.
Redzic Majda	The implementation of the Proxy pattern and Iterator pattern. Documentation
Barloga Aneta Katarzyna	Implementation of the patterns: Facade, Factory. Setup and development of Project. Communication between Client and Server.
Lylak Oleksandra	Decorator und Factory Pattern. Some methods in controller konfiguration. HTML pages. Documentation. UML class diagrams and Gantt diagram.

How to start project:

1. clone repository.
2. open project in IDE (Eclipse or IntelliJ).
3. run "ApplicationRun"
ExpenseTracker/src/main/java/expenseTracker/ApplicationRunner.java
4. open "localhost:8080/start" in browser

JAR-File: .\JarFile\ROOT.jar

Our implementation can be found on gitlab in the ExpenseTracker directory.
Of the top in DEAD directory contains

- report_DEAD
- UML class diagrams
- jar ROOT.jar