

Software Engineering 2

DEAD REPORT

Deadline Report

Team number:	0309
---------------------	------

Team member 1	
Name:	Patrick Gmasz
Student ID:	11708954
E-mail address:	gmaszp95@unet.univie.ac.at

Team member 2	
Name:	Paul Kraft
Student ID:	11708991
E-mail address:	a11708991@unet.univie.ac.at

Team member 3	
Name:	Michael Watholowitsch
Student ID:	01613664
E-mail address:	a01613664@unet.univie.ac.at

Team member 4	
Name:	Kleinl Lukas
Student ID:	01546221
E-mail address:	A01546221@unet.univie.ac.at

1 Final Design

1.1 Design Approach and Overview

All of the diagrams are in the “diagrams” folder.

1.1.1 Class Diagrams

A rough overview of the important parts, detailed diagrams are added afterwards:

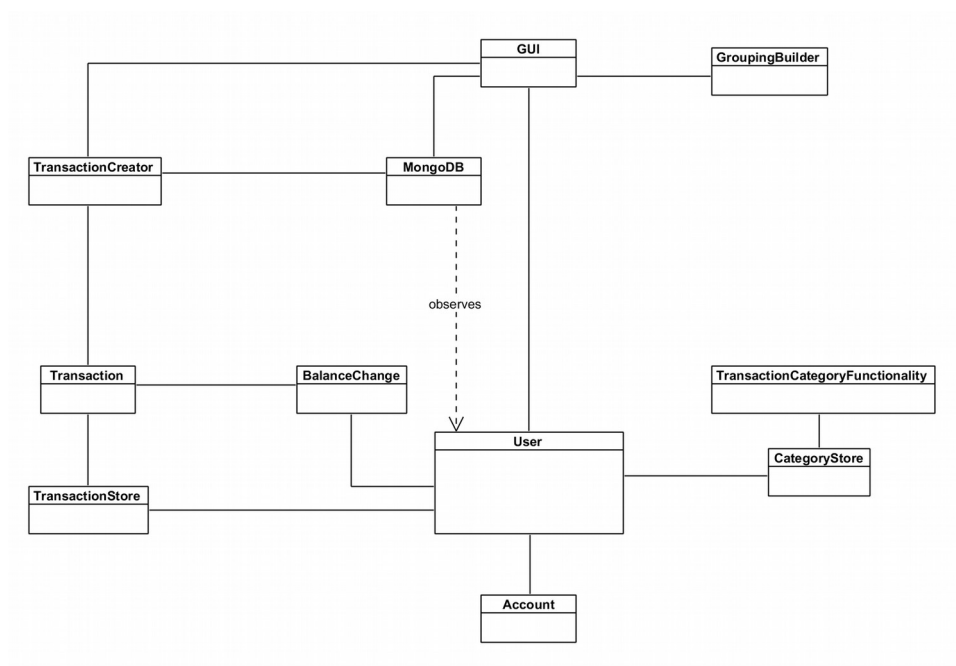


Figure 1: Overview over the most important parts

Accounts:

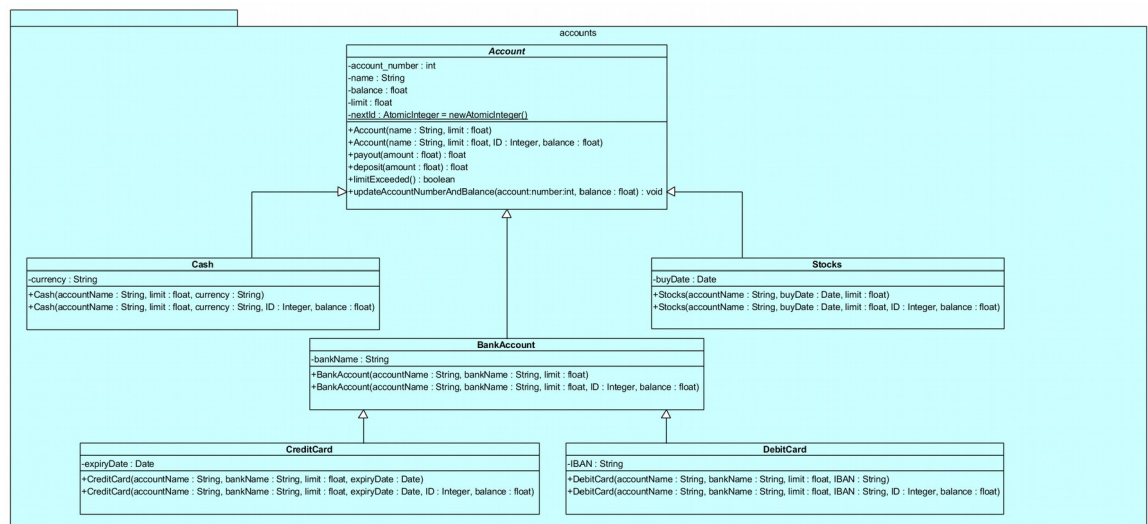


Figure 2 : Overview over the different accounts

Client:

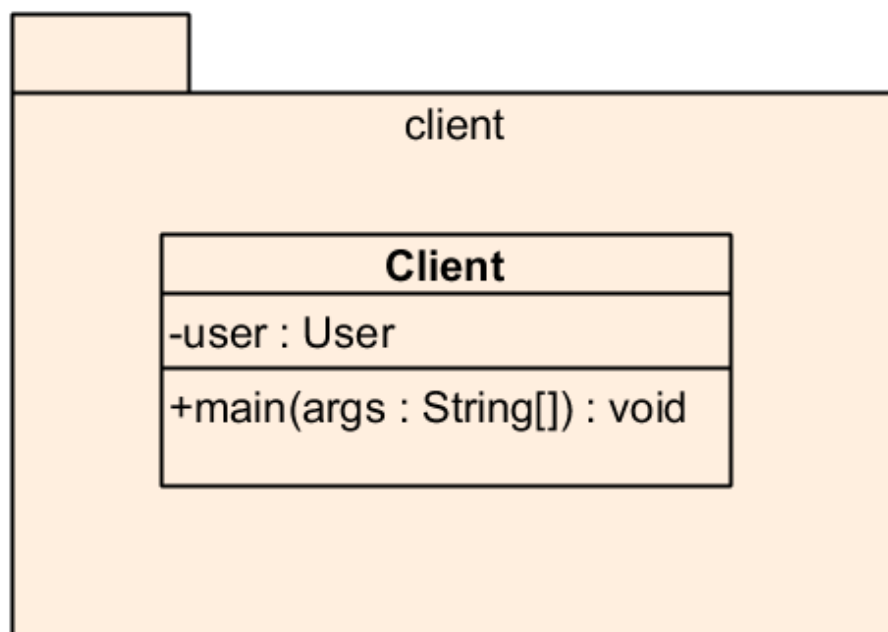


Figure 3 : Executable main class client

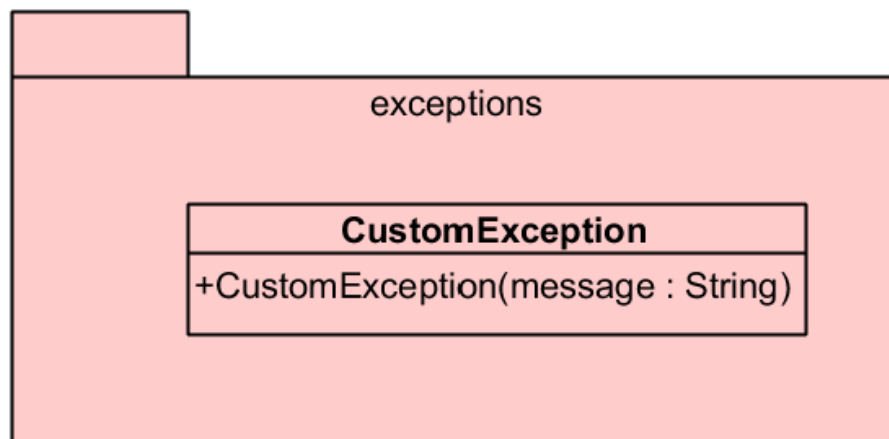
Exceptions:

Figure 4 : Exception class

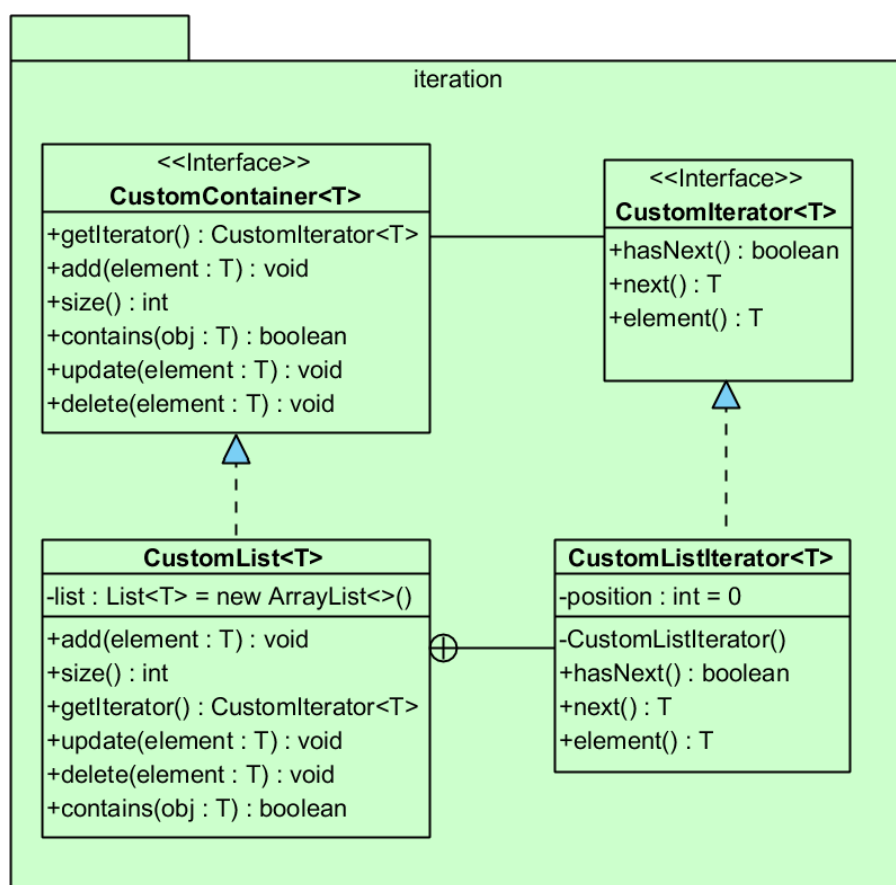
Iteration:

Figure 5: Overview Iterator Pattern

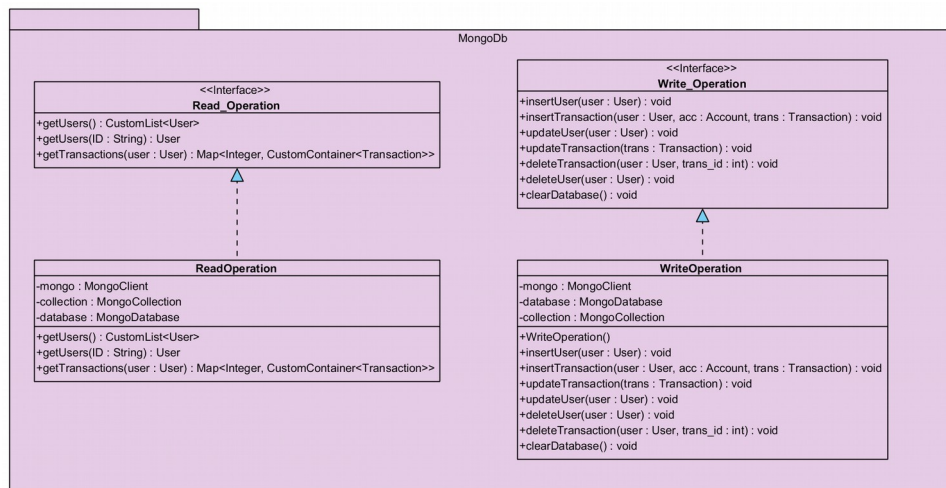
MongoDb:

Figure 6 : Overview over CRUD operations to and from database

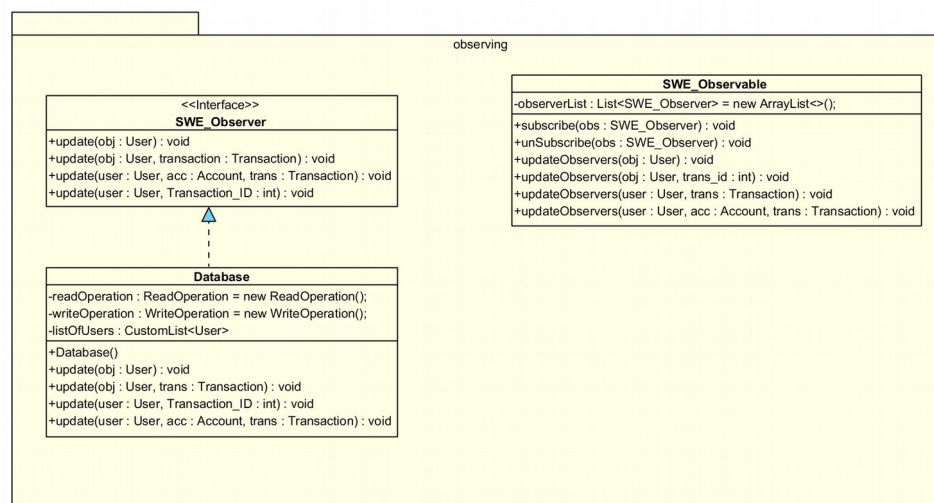
Observing:

Figure 7 : Overview Observer Pattern

Transactions:

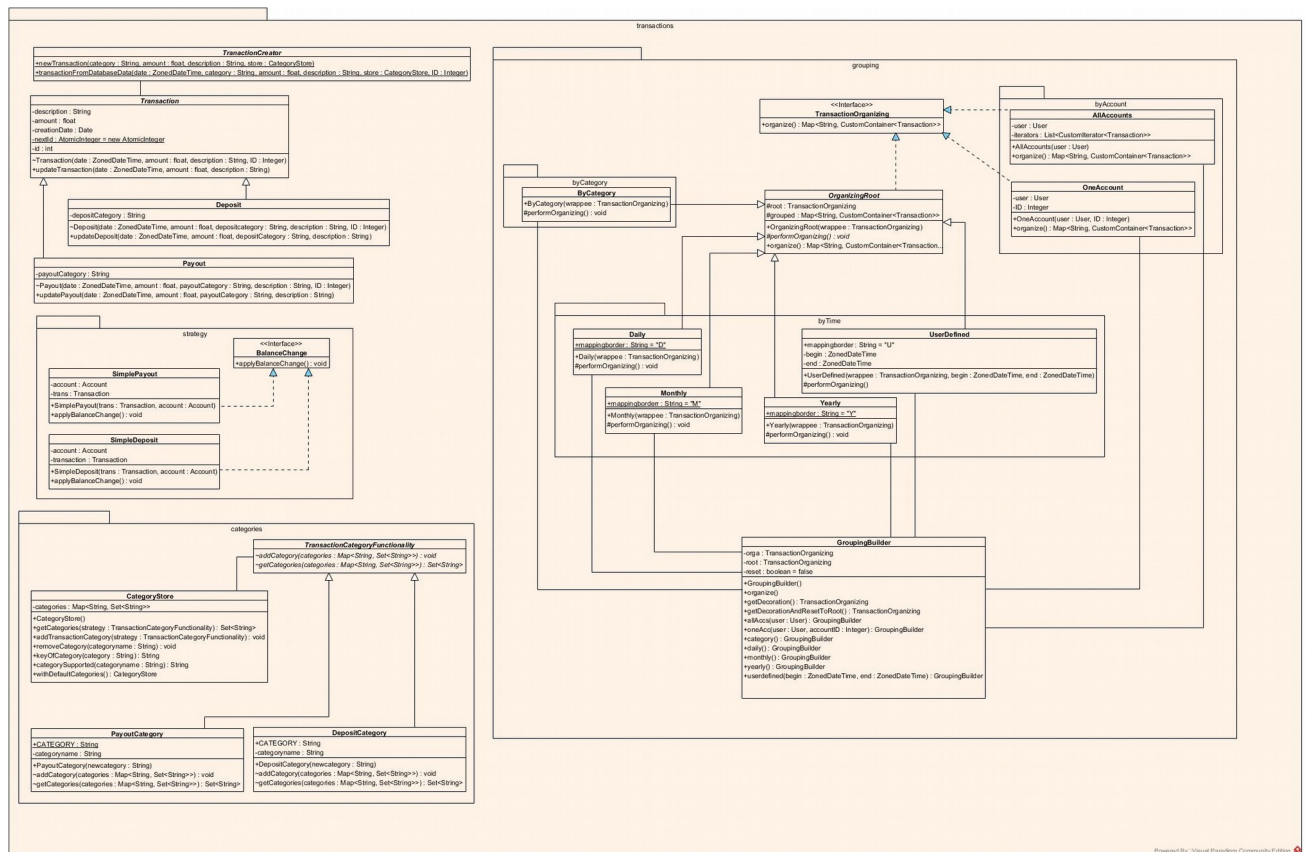
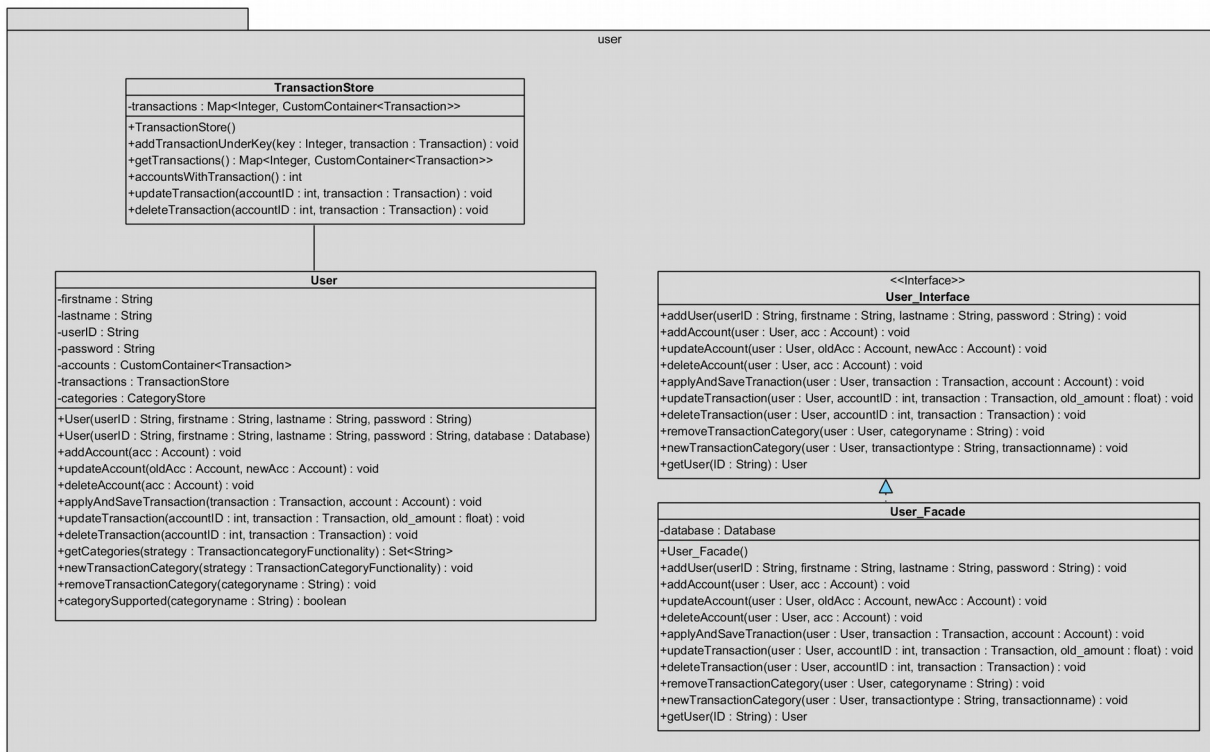


Figure 8 : Overview over Transactions

User:*Figure 9 : Overview of the user*

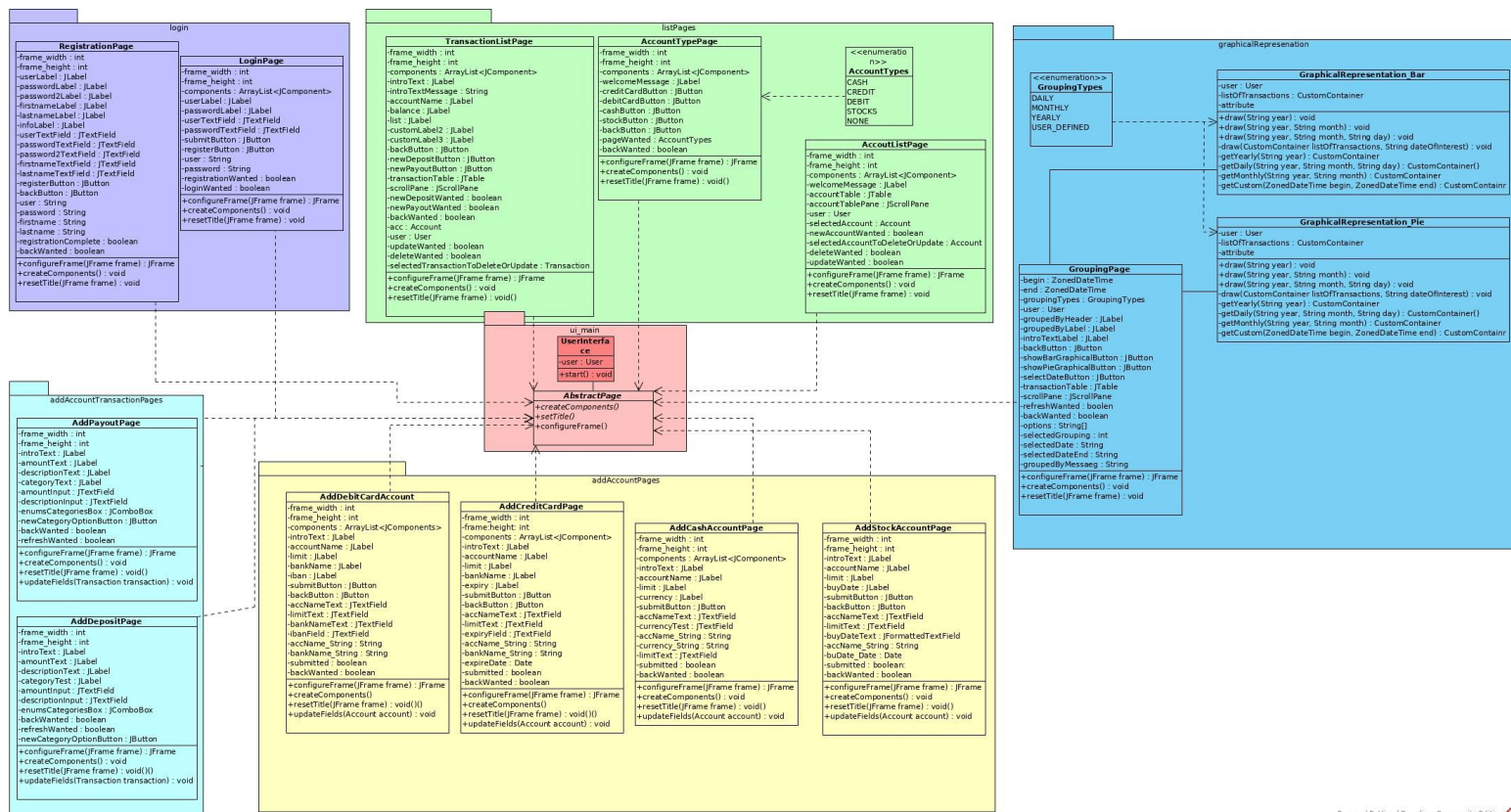
User Interface:

Figure 10 : Overview of the User Interface

1.1.2 Technology StackMongoDB:

For the persistency we are using MongoDB. First we wanted to use file writer and file reader to store our data, but this approach had a huge downside because if we used this technology we would have to always read or write the whole data. So we decided to switch to another technology called MongoDB. MongoDB is a NoSQL database. In this database the data is stored in a format similar to JSON-format. It is also very easy to connect to the database.

We are using the version 3.11.2
[\(https://mongodb.github.io/mongo-java-driver/\)](https://mongodb.github.io/mongo-java-driver/)

JSON

We needed to convert the data from mongodb back to the original format. So we needed a library to do that. With the help of this library it is very easy to convert the data back to its original form.

We are using the latest version of JSON: 20190722
(<https://mvnrepository.com/artifact/org.json/json>)

Build _____ Tool:
Gradle

JavaSwing:

An easy way to realise the user interface was to use Java Swing, since it offers a good documentation on the Internet. We were thinking about using Html and Javascript to display it on a webpage, or JavaFX, but since we already knew Swing from SWE1 a little and also didn't want to bother with communicating with a webpage we decided to use Java SWING instead.
(<https://docs.oracle.com/javase/8/docs/api/javax/swing/package-summary.html>)

JFreeChart:

We wanted to display the data in a nice-looking fashion, a quick google search for a Swing compatible chart library led to J4FreeChart.
(<http://www.jfree.org/jfreechart/>)

1.2 Major Changes Compared to SUPD

-Added the FR5 functionality, and therefore Created a lot of new Pages in the UI.
(Pie Chart, Bar Chart)

-Capability to update existing Objects.(changed the Existing Pages)

-Overall Prettier UI.

-Persistence in Storing.

-users may now modify their wanted categories, this adding is supported by the strategy pattern as suggested by SUPD feedback

-reduced the number of custom exceptions and switched to more standard exceptions like suggested in the SUPD feedback

-packages were renamed and nested according to their functionality

1.3 Design Patterns

1.3.1 Observer Pattern:

General outline of the pattern: The pattern ensures that if the user object is changed within The program(e.g New transaction from within the UI) the database (MongoDB) gets notified and can therefore immediately update the object and thereby guarantee CRUD functionality.

In our case we decided to use the observer pattern since it was a easy way of making sure that in case of changing important objects they automatically notify the corresponding database, if nothing has changed the database does not need to “spam” the user objects with requests if they have changed.

How does the pattern relate to one of the FR: Goal is definitely to achieve CRUD persistence functionality in a smooth, easy to understandable way. The UI can handle the user as if it was a local object.

How it works in our Code:

There is a class database that implements the observer interface.

Which has a write and a read operation class that can access the MongoDB client. The user extends the observable.

On start the database creates a local list of all the users that exist on the MongoDB client and provides it to the actual client and user interface.

If a new user gets created it is request from the database and automatically subscribed to the observer. If a user object then is changed anywhere in the program it informs the database to update, which then updates it on the MongoDB client.

```

public class SWE_Observable
{
    /** List of all Observers that are Interested in the Data of the Observer!
     */
    private final List<SWE_Observer> observerList = new ArrayList<>();

    /** Adds a certain Observer to the observerList
     * @param obs the Observer that should be added.
     * @throws Exception if Observer already subscribed.
     */
    public void subscribe(final SWE_Observer obs) throws CustomException {
        if(observerList.contains(obs))
            throw new CustomException("Observable ALREADY SUBSCRIBED to this Observer -");
        else {
            observerList.add(obs);
        }
    }

    /** Removes a certain Observer from the observerList
     * @param obs the Observer that should be removed.
     * @throws Exception if Observer not subscribed at this time.
     */
    public void unsubscribe(final SWE_Observer obs) throws CustomException {
        if(!observerList.contains(obs))
            throw new CustomException("Observable was NOT SUBSCRIBED to this Observer -");
        else {
            observerList.remove(obs);
        }
    }
}

```

SWE_Observable > observerList

Implementation [Client.main()] x

Figure 11 : Observeable class

```

/**
 * DIY Observer Pattern - defines the Observer.
 * @author Paul Kraft
 * @author Lukas Kleinl
 */
public interface SWE_Observer {
    /** Method to implement, what to do with the new given Object.
     * @param obj Object that changed, which interests the observer.
     */
    void update(User obj);
    void update(User obj, Transaction transaction);
    void update(User user, Account acc, Transaction trans);
    void update(User user, int Transaction_ID);
}

```

Figure 12 : Observer Interface

1.3.2 Iterator Pattern

General outline of the pattern: The pattern ensures that the order of the elements (in which they were added) is kept. Accessing the data is only possible by using a CustomListIterator. This iterator allows to traverse the list in a specified way, namely the order in which the elements were added. The CustomListIterator is implemented as private class of the CustomList since these two classes are directly related.

How does the pattern relate to one of the FR: Since we need to visualise transactions of a user, using a type of datastructure that only allows traversal in a specified way is beneficial. It ensures that the order in which they were created is maintained. You can still skip some entries if you were to display only certain types of transactions (like only display food expenses) but are still forced to keep their order.

```
public interface CustomIterator<T> {
    * Return {@code true} if there is another element after the current one, else {@code false}.
    public boolean hasNext();

    * Return the current Object if there is one and move to the next element.
    public T next() throws RuntimeException;

    * Return the current Object if there is one.
    public T element() throws RuntimeException;
}
```

Figure 13 : Iterator class with its methods

```
public interface CustomContainer<T> {
    /** Creates and returns a {@link CustomIterator} for this {@code CustomContainer}. */
    CustomIterator<T> getIterator();

    * Adds a new element in this {@code CustomContainer}.
    void add(T element);

    /** Returns the number of elements in this {@code CustomContainer}. */
    int size();

    * Checks if the passed {@code T} is stored in this {@code CustomContainer}
    boolean contains(T obj);

    * Updates the element in this {@code CustomContainer} that is equal to the passed element.
    void update(T element);

    * Removes the element in this {@code CustomContainer} that is equal to the passed element.
    void delete(T element);
}
```

Figure 14 : Container class with methods

1.3.3 Decorator Pattern

General outline of the pattern: This pattern is used to add further functionality to a type of object by wrapping the base-class with other classes. By doing so, you can keep the required functionality separated from the improved/specified functionality and combine different functionality easily.

How does the pattern relate to one of the FR: Displaying transactions can be done in different ways. Some users want to see what their cash flow by category is, while some want an overview of their monthly money flow. Having multiple methods to group creates a lot of combinations and a lot of duplicate code. Keeping every sorting method separately enabled easier maintainability. In order to simplify the usage and readability, we provided an extra class for wrapping, the GroupingBuilder.

All classes related to the pattern implement this interface.

```
package transactions.grouping;

import java.util.Map;

/**
 * Base-Interface for grouping decorators.
 *
 * @author Michael Watholowitsch
 */
public interface TransactionOrganizing {
    /**
     * Implementations decorate the grouping.
     *
     * @return the grouped transactions
     */
    Map<String, CustomContainer<Transaction>> organize();
}
```

Figure 15 : Interface for transaction organizer

One of the 2 possible base classes is shown in figure 16

```

1 package transactions.grouping.byAccount;
2
3 import java.util.HashMap;
4
5 /**
6  * Base class for decorating the view of the transactions of a single account of the passed user.
7  *
8  * @author Michael Watholowitsch
9  */
10 public class OneAccount implements TransactionOrganizing {
11
12     private final User user;
13     private final Integer ID;
14
15     /**
16      * @param user the user whose transactions will be grouped
17      * @param accountID the ID of the account which's transactions will be grouped
18      */
19     public OneAccount(final User user, final Integer accountID) {
20         this.user = user;
21         this.ID = accountID;
22     }
23
24     @Override
25     public Map<String, CustomContainer<Transaction>> organize() {
26         Map<String, CustomContainer<Transaction>> organized = new HashMap<>();
27         organized.put("", this.user.getTransactionStore().getTransactions().get(this.ID));
28         return organized;
29     }
30 }

```

Figure 16 : Base class for organizing

All classes related to the pattern:

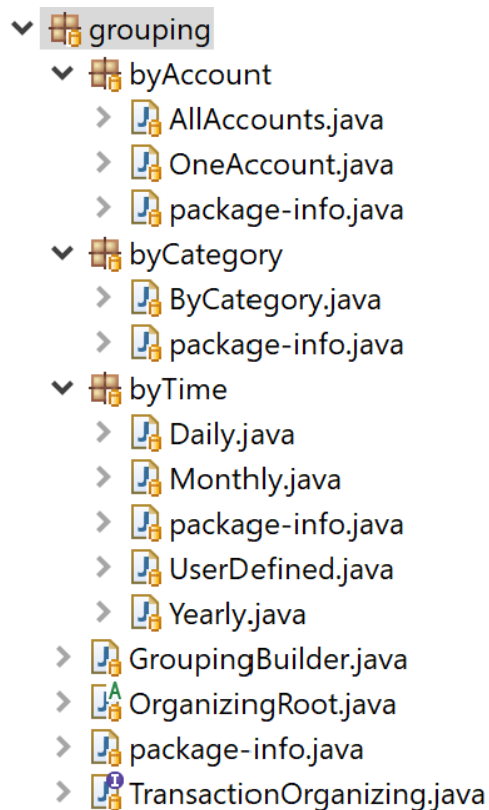


Figure 17 : Related classes

1.3.4 Template Method Pattern

General outline of the pattern: Using this pattern made creating different User Interface pages much easier. Every time when changing a page in the UI, the needed JComponents need to get updated in the JFrame.

Independently from the type of page, this is always the same procedure: Remove old components from the JFrame, reset the appropriate title of the frame, create the needed components of the page and finally repaint the JFrame. Creating the components for the different pages is obviously different for every page, same with setting the title page. That is why there are abstract methods for resetting the title and creating the components. These methods are used in the final configureFrame method, which does the whole procedure of loading a new page.

How does the pattern relate to one of the FR: This pattern does not relate directly to one of the FR's. It more defines, how we create different UI pages in our project.

Abstract Page implementation:

```
package ui.main;

import javax.swing.*;
import java.awt.*;
import java.util.ArrayList;

/**
 * Abstract class for the User Interface
 *
 * @author Patrick Gmasz
 */
public abstract class AbstractPage {

    protected ArrayList<JComponent> components;
    protected static final int FRAME_WIDTH = 1200;
    protected static final int FRAME_HEIGHT = 800;
    protected static final Font LABEL_FONT = new Font("Serif", Font.BOLD, 11: 25);
    protected static final Font TEXTFIELD_FONT = new Font("Serif", Font.PLAIN, 11: 20);
    protected static final Font HEADER_FONT = new Font("Serif", Font.BOLD, 11: 19);
    protected static final Font BUTTON_FONT = new Font("Serif", Font.BOLD, 11: 20);

    /**
     * This method updates the given JFrame with all of the components.
     *
     * @param frame The JFrame, which components will be updated
     */
    public final void configureFrame(JFrame frame) {

        frame.getContentPane().removeAll();
        frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);

        resetTitle(frame);

        createComponents();

        for (JComponent comp : components) {
            frame.add(comp);
        }

        frame.revalidate();
        frame.repaint();
    }

    protected abstract void resetTitle(JFrame frame);

    protected abstract void createComponents();
}
```

Figure 18 : Abstract page

Example of a usage of the pattern:

```
/** This page will be the start page, when the application is started. The user can input his user id ...*/
public class LoginPage extends AbstractPage {

    private JLabel userLabel;
    private JLabel passwordLabel;
    private JTextField userTextField;
    private JTextField passwordTextField;
    private JButton submitButton;
    private JButton registerButton;

    private JLabel message;

    private String user;
    private String password;
    private volatile boolean registrationWanted;
    private volatile boolean loginWanted;

    /** Creates a new LoginPage, which will load all needed components in a list. */
    public LoginPage() { createComponents(); }

    /** Returns the input, which is in the user textfield. ...*/
    public String getUser() { return user; }

    /** Returns the input, which is in the password textfield. ...*/
    public String getPassword() { return password; }

    /** The page has a button, which the user can press if he wants to register a new user account. If ...*/
    public boolean isRegistrationWanted() { return registrationWanted; }

    /** The page has a button, which the user can press if he wants to log in. If the button gets ...*/
    public boolean isLoginWanted() { return loginWanted; }

    /** This method creates all components, such as buttons and text fields, and adds it to a list. It ...*/
    @Override
    protected void createComponents() {...}

    /** This method resets the title of the JFrame to "Login". ...*/
    @Override
    protected void resetTitle(JFrame frame) {
        frame.setTitle("Login");
    }
}
```

Figure 19 : Usage of abstract page

1.3.5 Factory Method Pattern

General outline of the pattern: This pattern extracts the creation of objects from the rest of the code. This makes sure that the underlying structure can be modified without worrying to change the creation elsewhere. Additionally, creating new types of objects later on becomes more flexible. Another benefit of using this process to create objects is explicitly telling the programmer which type of creation he should apply for his/her usecase since method names are more declarative than multiple constructors with different parameters.

How does the pattern relate to one of the FR: The UI and the database should not need to worry how the transactions are created. By providing factory methods, both UI and database can simply pass the required parameters to create a new transaction and the actual creation is handled inside the method depending on these parameters.

```
/**
 * Factory for creating transactions.
 */
public abstract class TransactionCreator {
    * Creates new transactions. The primary way to create completely new transactions.
    public static Transaction newTransaction(final String category, final float amount,
        final String description, final CategoryStore store) {

        if ((store == null) || (category == null))
            throw new RuntimeException("Could not check if the category is known !");

        if (store.categorySupported(category)) {
            String storedcategory = store.keyOfCategory(category);

            if (storedcategory.equalsIgnoreCase(Deposit.getSimpleName()))
                return new Deposit(null, amount, category, description, null);
            else if (storedcategory.equalsIgnoreCase(Payout.getSimpleName()))
                return new Payout(null, amount, category, description, null);
        }

        throw new RuntimeException("Cannot create this transaction, the category is unknown !");
    }

    * Creates new transactions. This method should only be used to create transactions from
    public static Transaction transactionFromDatabaseData(final ZonedDateTime date,
        final String category, final float amount, final String description,
        final CategoryStore store, final Integer ID) {

        if ((store == null) || (category == null))
            throw new RuntimeException("Could not check if the category is known !");

        if (store.categorySupported(category)) {
            String storedcategory = store.keyOfCategory(category);

            if (storedcategory.equalsIgnoreCase(Deposit.getSimpleName()))
                return new Deposit(date, amount, category, description, ID);
            else if (storedcategory.equalsIgnoreCase(Payout.getSimpleName()))
                return new Payout(date, amount, category, description, ID);
        }

        throw new RuntimeException("Cannot create this transaction, the category is unknown !");
    }
}
```

Figure 20: Factory Pattern

1.3.6 Strategy Pattern

General outline of the pattern: This pattern allows applying a different kind of algorithm depending on the calling class. By changing the object that calls a method, one can dynamically change behaviour which makes adding new kinds of algorithms easier and more flexible.

How does the pattern relate to one of the FR:

This tracker should update account balances when a transaction is performed. Using this pattern allows us to only have one method in the user to apply balance changes of an account because this method determines which kind of changes it needs to perform itself depending on the kind of transaction.

```

1  /**
2   * Used to allow flexible ways of conducting balance changes on accounts.
3   *
4   * @author Michael Watholowitsch
5   */
6  public interface BalanceChange {
7
8  }
9
10 /**
11  * Changes the balance depending on the implementing class.
12  */
13 void applyBalanceChange();
14 }

```

Figure 21 : Interface for Strategy pattern

The implementing classes are SimpleDeposit and SimplePayout. As their name suggests, they only call the respective deposit(...) and payout(...) methods of the account to change the balance. When a new transaction is added, the method checks how the balance should change:

```

public void applyAndSaveTransaction(final Transaction transaction, final Account account)
    throws RuntimeException {
    BalanceChange strategy;

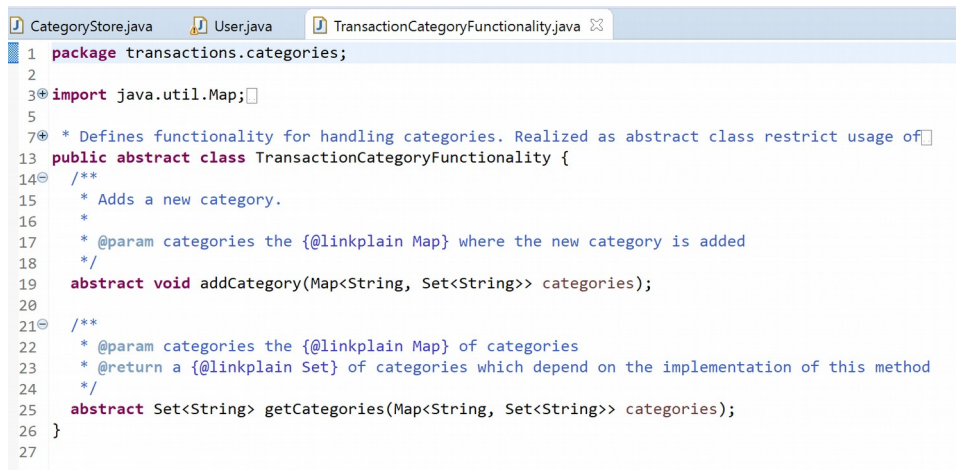
    if (transaction instanceof Deposit) {
        strategy = new SimpleDeposit(transaction, account);
    } else if (transaction instanceof Payout) {
        strategy = new SimplePayout(transaction, account);
    } else
        throw new RuntimeException("Unknown Transaction !");

    strategy.applyBalanceChange();
    this.transactions.addTransactionUnderKey(account.getAccount_number(), transaction);
    updateObservers(this, account, transaction); // VON PAUL fÄrs observer
}

```

Figure 22 : First usage of strategy pattern

The pattern is also applied for adding new categories. When a user creates a new category, the name of the category is wrapped by a class. This class is then passed to the local representation of a user which then passes it to its CategoryStore. This store then lets the wrapping class add the category depending on the implementation.

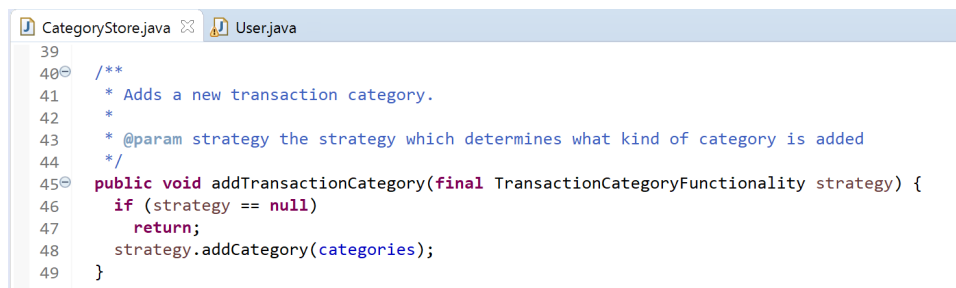


```

1 package transactions.categories;
2
3 import java.util.Map;
4
5
6 * Defines functionality for handling categories. Realized as abstract class restrict usage of
7 public abstract class TransactionCategoryFunctionality {
8     /**
9      * Adds a new category.
10      * @param categories the {@link Map} where the new category is added
11      */
12     abstract void addCategory(Map<String, Set<String>> categories);
13
14     /**
15      * @param categories the {@link Map} of categories
16      * @return a {@link Set} of categories which depend on the implementation of this method
17      */
18     abstract Set<String> getCategories(Map<String, Set<String>> categories);
19 }
20
21
22
23
24
25
26
27

```

Figure 23 : Class of transaction functionality



```

39
40 /**
41  * Adds a new transaction category.
42  *
43  * @param strategy the strategy which determines what kind of category is added
44  */
45 public void addTransactionCategory(final TransactionCategoryFunctionality strategy) {
46     if (strategy == null)
47         return;
48     strategy.addCategory(categories);
49 }
50

```

Figure 24 : Usage of strategy in add transaction categories

```

/**
 * Adds a new transaction category.
 *
 * @param strategy the strategy which determines what kind of category is added
 */
public void newTransactionCategory(final TransactionCategoryFunctionality strategy) {
    this.categories.addTransactionCategory(strategy);
    updateObservers(this); // VON PAUL fÃ¼rs observer
}

```

Figure 25 : Usage of strategy in new transaction category

1.3.7 Facade Pattern

General outline of the pattern: This pattern is most suitable for simplifying actions for the client. With the help of this pattern it is possible to improve the readability and to decouple the client from the implementation in the backend.

How does the pattern relate to one of the FR:

For our program we have lots of different classes and our user is the façade for all those classes. If we wouldn't have the user we would have to create all classes (account, transaction, category) within the client and that would be bad for the readability and also the cohesion between the classes and so also the dependency of the client to the different classes would be greater. With the help of the user as façade we can limit the interaction between the backend and the client and improve the readability. Also changes within the backend doesn't have such a big impact onto the client.

2 Implementation

2.1 Overview of Main Modules and Components

We split the project into 3 rough parts: the UI, the backend and the database. The UI itself communicates with the backend and any changes there will be automatically performed in the database too thanks to the observer pattern. Communication directly from the UI to the database is therefore not possible.

2.2 Coding Practices

We tried to keep the code in a formal consistent way through the whole project. So, the intention is that the style looks the same way throughout the project so that a reader would think that the code was written by one person. We also tried to keep the code as simple as possible. Regarding naming conventions, we focused to be consistent throughout the project and kept in mind to always name the variables properly.

2.3 Defensive Programming

In the User Interface, if “Bad-Data” was submitted by the user, we tried to inform the user that his data was incorrect by displaying it with JOptionPane's.

The bad data was then either never used, or for example in the GroupingPage the Data is resetted to a neutral time value being the time now.

3 Software Quality

3.1 Code Metrics

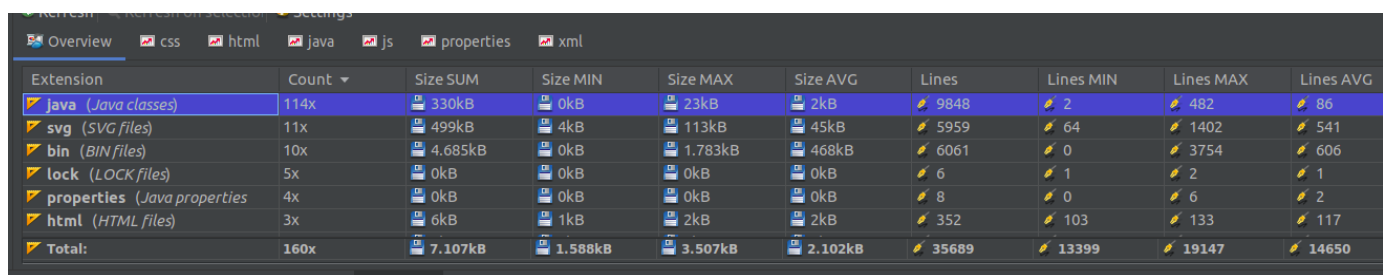
For a getting information about our code metrics, we used a plugin for IntelliJ called “Statistic” . As seen in the following figures(figures 26 and 27) we have:

Lines of source code: 6396

Lines of comments: 1998

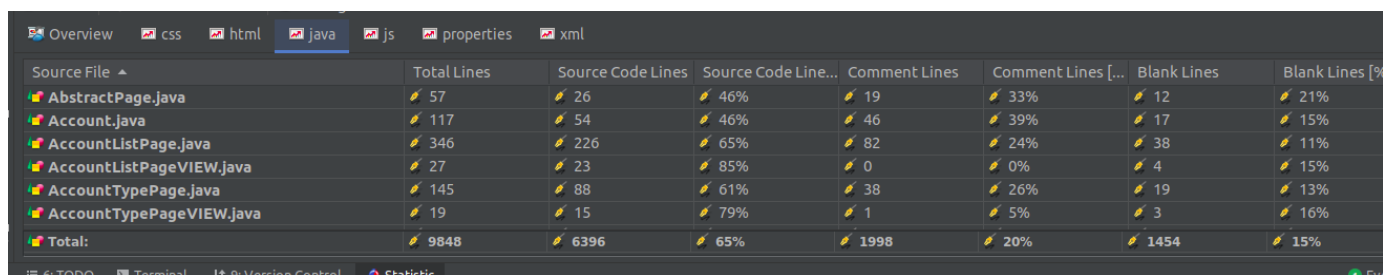
Number of classes: 114

Compared to SUPD we have nearly twice as much classes and the lines of source code are nearly three times bigger. This is because we added a lot of functionality to the old program.



Extension	Count	Size SUM	Size MIN	Size MAX	Size AVG	Lines	Lines MIN	Lines MAX	Lines AVG
java (Java classes)	114x	330kB	0kB	23kB	2kB	9848	2	482	86
svg (SVG files)	11x	499kB	4kB	113kB	45kB	5959	64	1402	541
bin (BIN files)	10x	4.685kB	0kB	1.783kB	468kB	6061	0	3754	606
lock (LOCK files)	5x	0kB	0kB	0kB	0kB	6	1	2	1
properties (Java properties)	4x	0kB	0kB	0kB	0kB	8	0	6	2
html (HTML files)	3x	6kB	1kB	2kB	2kB	352	103	133	117
Total:	160x	7.107kB	1.588kB	3.507kB	2.102kB	35689	13399	19147	14650

Figure 26 : Metrics overview



Source File	Total Lines	Source Code Lines	Source Code Line...	Comment Lines	Comment Lines [...]	Blank Lines	Blank Lines [%]
AbstractPage.java	57	26	46%	19	33%	12	21%
Account.java	117	54	46%	46	39%	17	15%
AccountListPage.java	346	226	65%	82	24%	38	11%
AccountListPageVIEW.java	27	23	85%	0	0%	4	15%
AccountTypePage.java	145	88	61%	38	26%	19	13%
AccountTypePageVIEW.java	19	15	79%	1	5%	3	16%
Total:	9848	6396	65%	1998	20%	1454	15%

Figure 27 : Metrics for java code

In regard of finding bugs we used the the plugin calles “FindBugs”. In figure 28 you can see an overview of the found bugs. In SUPD we found 31 bugs and in the final version we have 43 bugs. In regard to number of growth of our program the number of bugs didn’t grow that much. We still have a malicious code vulnerability which means that our internal representation could be exposed because of the getters.

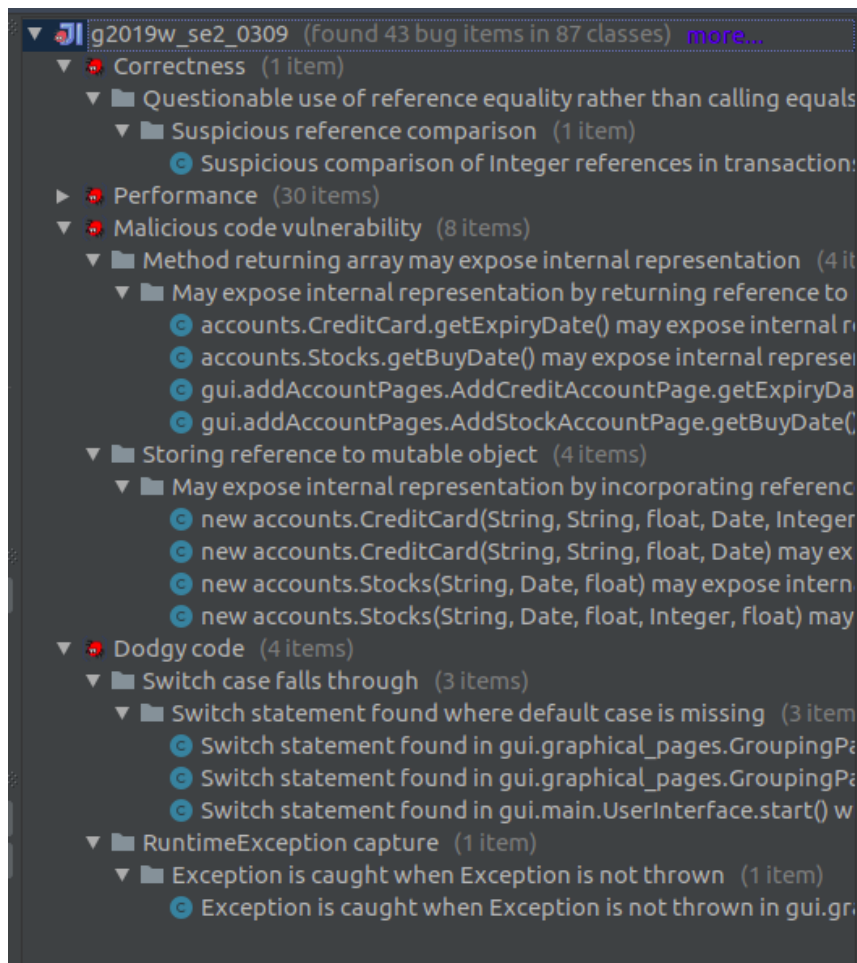


Figure 28 : Bugs in our project

3.2 Testcases for Functional Requirements

FR1: No tests for accounts, the only non-getter methods perform simple set operations.

FR2: Realised by a CategoryStore and classes which add new categories for income and expenses. The CategoryStore has some UnitTests and some tests with these extra classes.

FR3: No tests for transactions since the only methods here are for updates (which simply set new values) and getters.

FR4: Was tested with a serperate main within the testcases and was also tested in use with the userinterface. After the inserts we had to check if everything was stored correctly within the mongoDB.

FR5: The grouping is tested by using a testuser (GroupingTestUser). This testuser simulates a user and provides helper methods to make the tests shorter. It also provides a method to generate data for the parameterized tests. The tests check if a sequence of characters is found in the keys of the grouped map. The grouping classes change the keys by adding a specific character sequence and the tests check if these match with the content of the transaction.

FR6: The spending limit warning was tested in praxis: by creating all different kinds of accounts, and then created multiple transactions: pay-outs as well as deposits.

3.3 Quality Requirements Coverage

QR1: Created a JavaDoc, which you can access here: <http://wwwlab.cs.univie.ac.at/~gmaszp95/se2/>

QR2: As recommended, we used Google Java Style Guide. We used the following IntelliJ Plugin: <https://plugins.jetbrains.com/plugin/8527-google-java-format>

QR3: Used Google Java Style Guide
Used intention revealing, meaningful names for variables and methods.

QR4: The creation of the transactions checks for illegal arguments and throws exceptions.

QR5: Abstraction: Tried to focus on essential features in UML Diagrams (e.g. didn't add getters/setters)
Modularity: Tried to split functionalities in different packages and more classes.

QR6: The test directory of the project contains Unit Tests for most of the backend functionality. There are some classes with mains to test small behaviour-chains or to simply visualise output for the programmer.

QR7: See 1.3 in this document.

QR8: Gradle

QR9: FatJar

4Team Contribution

4.1Project Tasks and Schedule

As with the class diagrams, the gantt-chart was added to the diagrams directory on gitlab too.

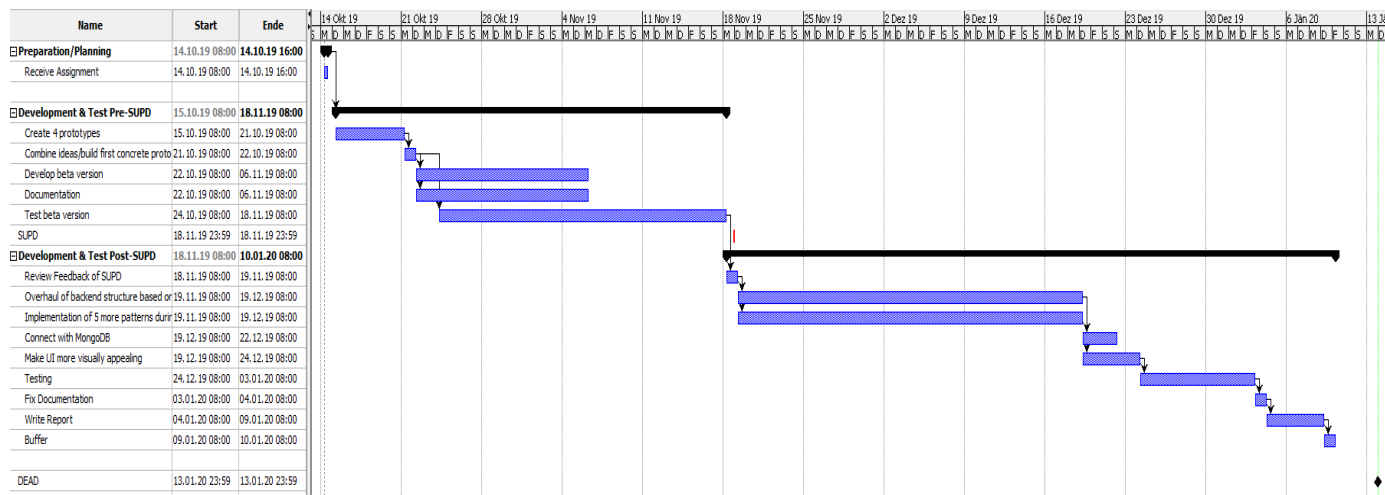


Figure 29 : Gantt-Chart

4.2 Distribution of Work and Efforts

Work	Paul	Michael	Patrick	Lukas
Planning	4h	4h	4h	4h
Reading Assignment and Feedback	3h	3h	3h	3h
UI:	26h	0h	32h	2h
Class-Diagram	2h	6h	2h	2h
Strategy-Pattern:	0h	1h	0h	3h
Iterator-Pattern:	0h	6h	0h	0h
Observer-Pattern:	3h	0h	0h	3h
Debugging UI:	2h	0h	4h	3h
Gantt-Diagram:	0h	3h	0h	0h
MongoDB:	0h	0h	0h	20h
Writing Report:	3h	6h	4h	3h
Decorator-Pattern:	0h	20h	0h	0h
Factory-Method-Pattern:	0h	4h	0h	0h
Template-Method Pattern:	0h	0h	1h	0h
Total:	43 hours	38,5 hours	50 hours	44 hours

Figure 30 : Distribution of work

A1 HowTo

The Project is to be git cloned.

Project Import:

You open the project in an IDE (tested with IntelliJ) as a Gradle project (inside the implementation folder open the build.gradle). Main class is the Client.java class (containing a main method)

The Main Entry Point for the Application:

The .jar of the project is in the "ExpenseTracker-jar" folder.

There is a test user whose login credentials are 'test' 'test'.

You can create your own user though.

It is also required that you are running a MongoDB service.

After starting the .jar a Java Swing window will open.

We provide a test user with the login credentials:

User ID=test and password = test

LoginPage:

If you are already registered, you can log in with your username and password. By pressing the submit button (leads to AccountListPage).

If you are not yet registered you can create a new user by clicking on the bottom right 'Register' -button.

RegisterPage:

In the RegisterPage you will be prompted to enter a username (needed for login later) your first and last name and a password.

After submitting you are back to the LoginPage and can log into your new created user with the user-id and the password.

AccountListPage:

You will then see a list of all your accounts. By clicking on an account once you select it and can then update (leads to CreateAccountPage's) or delete it(double checks if you want to delete). On the bottom of that page there are 2 more buttons to go the summaries of the accounts (leads to GroupingPage), or to create a new account (leads to Create Account)

By double clicking you go to the TransactionListPage of the account you just clicked on.

CreateAccountPage:

In the create account you can select between 4 types of accounts. After clicking the wanted you will be asked to enter the relevant Information and can then submit. The account will then be created and you will be sent back to AccountListPage.

GroupingPage:

In the GroupingPage you see all the transactions grouped. On the top right you can change the date and the style of Grouping(e.g monthly).

There are also 2 buttons to either display the distribution of the transactions as a pie chart or as a bar chart (note: If there are no transactions in the given time period, the charts won't open and instead a window will pop up telling you the date is invalid)

Important: The pie charts overlap there are 2!

TransactionListPage:

In the TransactionListPage relevant account information will be displayed on the right (e.g. Balance). The transactions related to this account will also be displayed in the middle of the page. On the bottom are again 2 buttons to create a new transaction (1 that leads to the newDeposit, 1 that leads to newPayout –page)

You can select a transaction by clicking it once, after selecting you can then update or delete it with the 2 buttons provided on the top right.

NewDepositPage:

In the newDepositPage you can select the category of the deposit, the amount and the description. You can also create a new category.

NewPayoutPage:

In the newPayoutPage you can select the category of the deposit, the amount and the description. You can also create a new category.

(If the Balance is 0, this will not work)

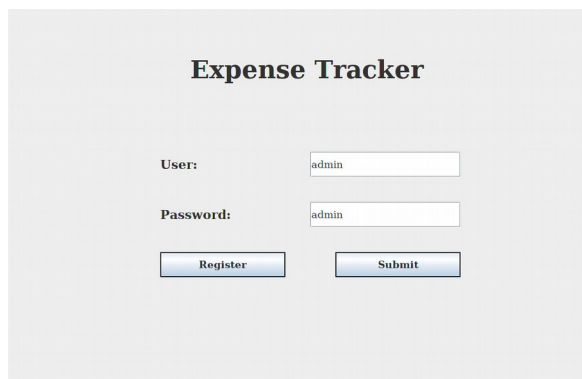
Use Case 1:

I have an account with UserId ='admin', password ='admin'.

I want to payout money from my creditcard cccount to buy my dog 'Rupert' a new dress.

1 First I will open the Jar

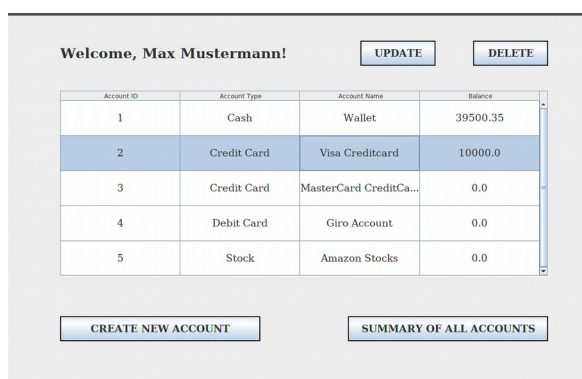
2 Then I will log in using my user credentials, and press the button 'Submit'



The image shows a login page for an "Expense Tracker". It has a title "Expense Tracker" at the top. Below it, there are two input fields: "User:" with the value "admin" and "Password:" with the value "admin". At the bottom, there are two buttons: "Register" and "Submit".

Figure 31 : Login page

3 I will then select the right account, and double click it.



The image shows a page titled "Welcome, Max Mustermann!". It has two buttons at the top: "UPDATE" and "DELETE". Below them is a table with four columns: "Account ID", "Account Type", "Account Name", and "Balance". The table contains five rows of data. The second row is highlighted. At the bottom, there are two buttons: "CREATE NEW ACCOUNT" and "SUMMARY OF ALL ACCOUNTS".

Account ID	Account Type	Account Name	Balance
1	Cash	Wallet	39500.35
2	Credit Card	Visa Creditcard	10000.0
3	Credit Card	MasterCard CreditCa...	0.0
4	Debit Card	Giro Account	0.0
5	Stock	Amazon Stocks	0.0

Figure 32 : List of accounts

4 I will press the 'NEW PAYOUT' button

BACK

Currently logged in as: Max Mustermann

UPDATE DELETE

Account Name:
Visa Creditcard

Balance:
10000.0

Limit:
1500.0

Expiry date:
1.1.2021

Account Type:
Credit Card

ID	Type	Category	Description	Creation Date	Amount
9	Deposit	DIVIDEND	Drugs	2020-01-12 ...	10000.0

NEW PAYOUT NEW DEPOSIT

Figure 33 : List of transactions

5 I will enter the amount, and the description of the payout and press submit

BACK

Currently logged in as: a a.

Payout category: DOG

Create category

Amount: 2999

Description: Versace Nr 7.

SUBMIT

Figure 34 : New Transaction

6 Have dinner with Rupert. Done!