Software Engineering 2
SUPD REPORT
Status Update 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:	Kleinl Lukas
Student ID:	01546221
E-mail address:	a01546221@unet.univie.ac.at

Team member 4	
Name:	Michael Watholowitsch
Student ID:	01613664
E-mail address:	a01613664@unet.univie.ac.at

1 Design Draft

1.1 Design Approach and Overview

1.1.1 Class Diagrams

Figure 1 shows a first sketch of the whole project, which me made when we met the first time to discuss the project as a group. Please note that there are few things missing, and we also did put any attention on the syntactical correctness. This was just for a first overview for us.

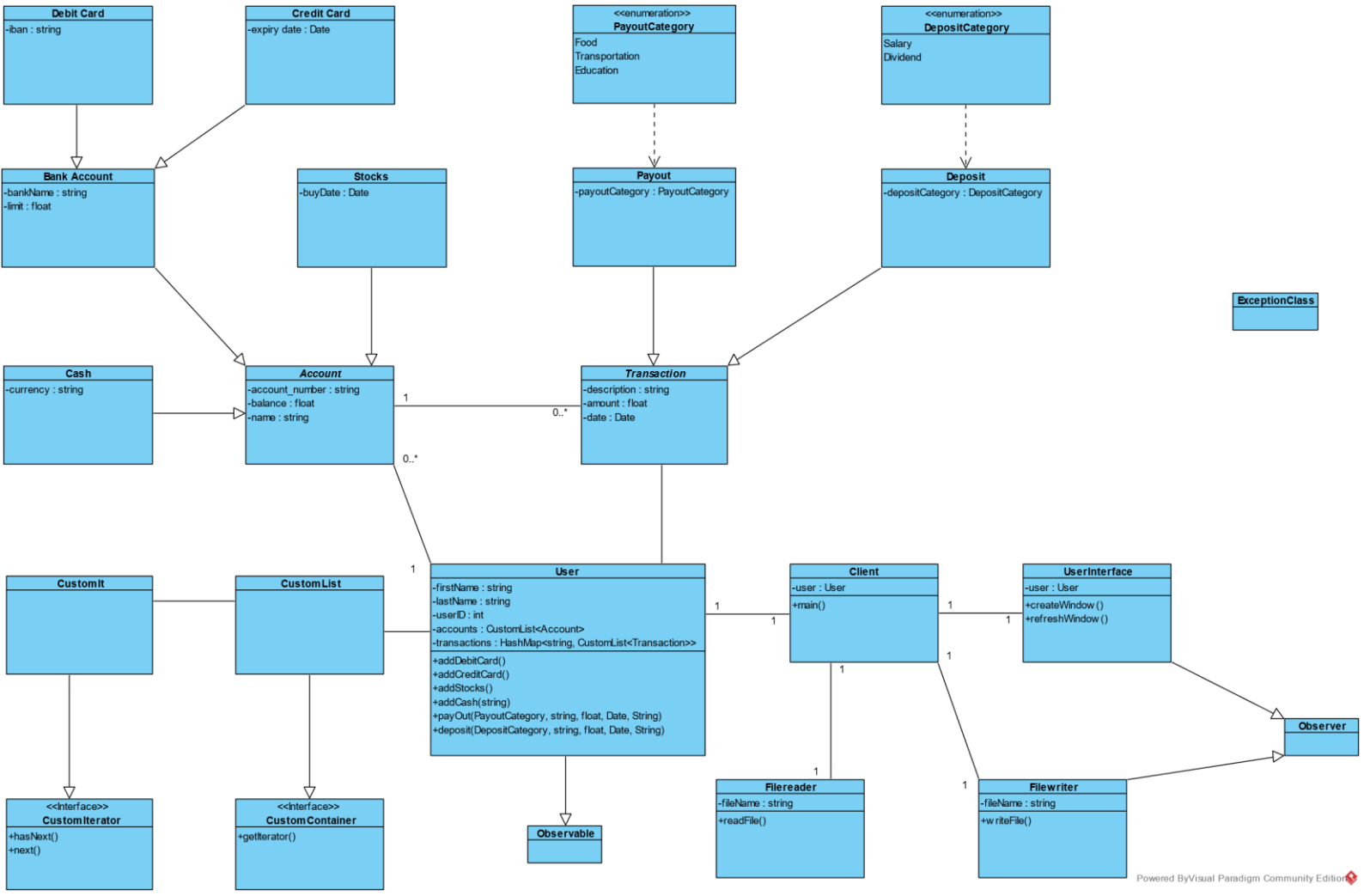

Figure 1: First sketch of the whole project

Figure 2 shows the final class diagram of the whole project. You can see a lot of changes regarding major design decisions, especially in regard of persistency (change from saving users to a file to using MongoDB). Also, the needed design patterns are easy to spot and shown in detail.

Note that the package for the user interface, in figure 2 called “ui”, is just a cut-out for the sake of clarity. For a detailed overview of the user interface, please see figure3.

Also, if you want an even better overview, we uploaded the .svg files of the diagrams in gitlab in the SUPD branch in a folder called “diagrams”.

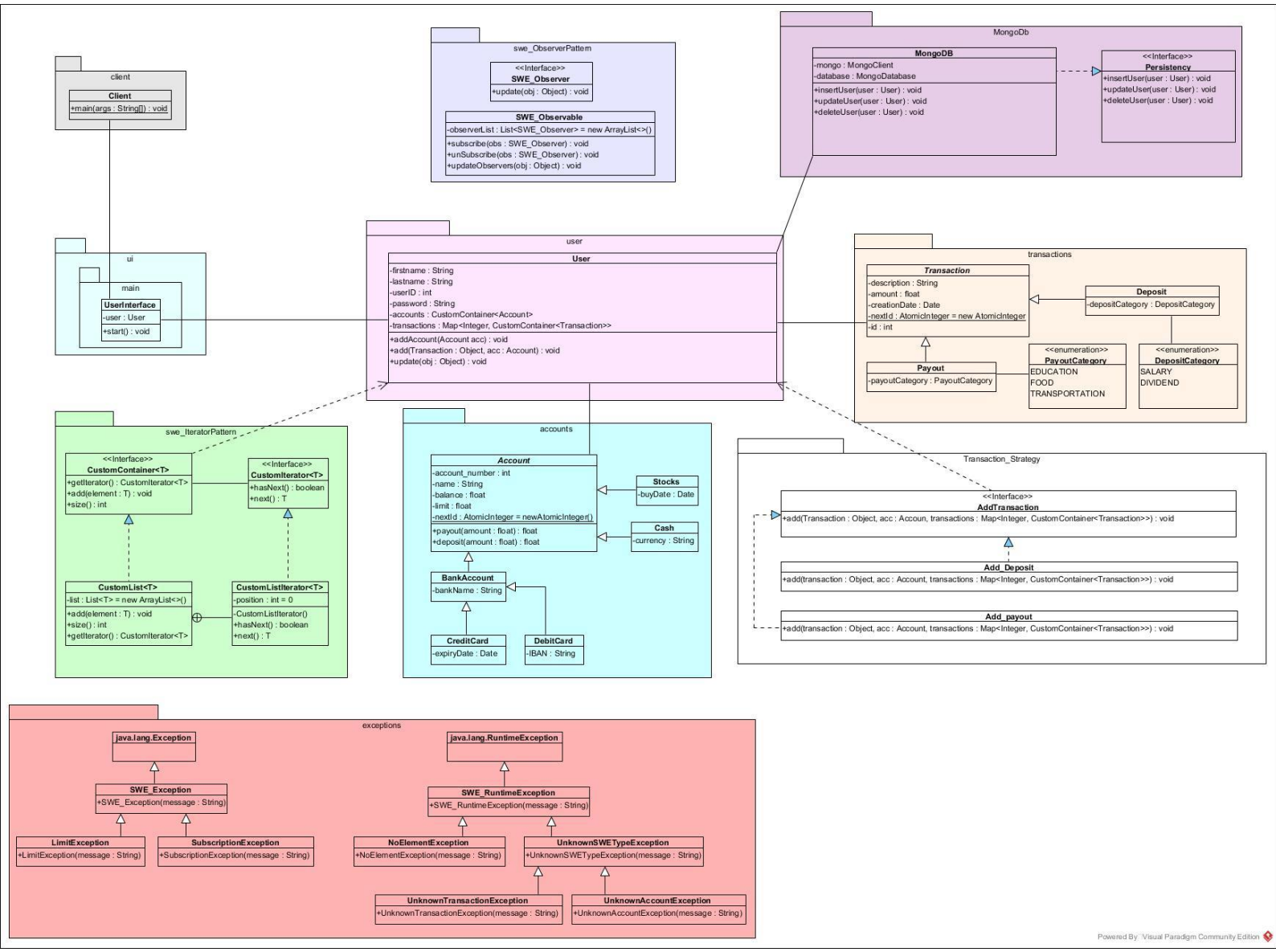

Figure 2: Final class diagram of the whole project

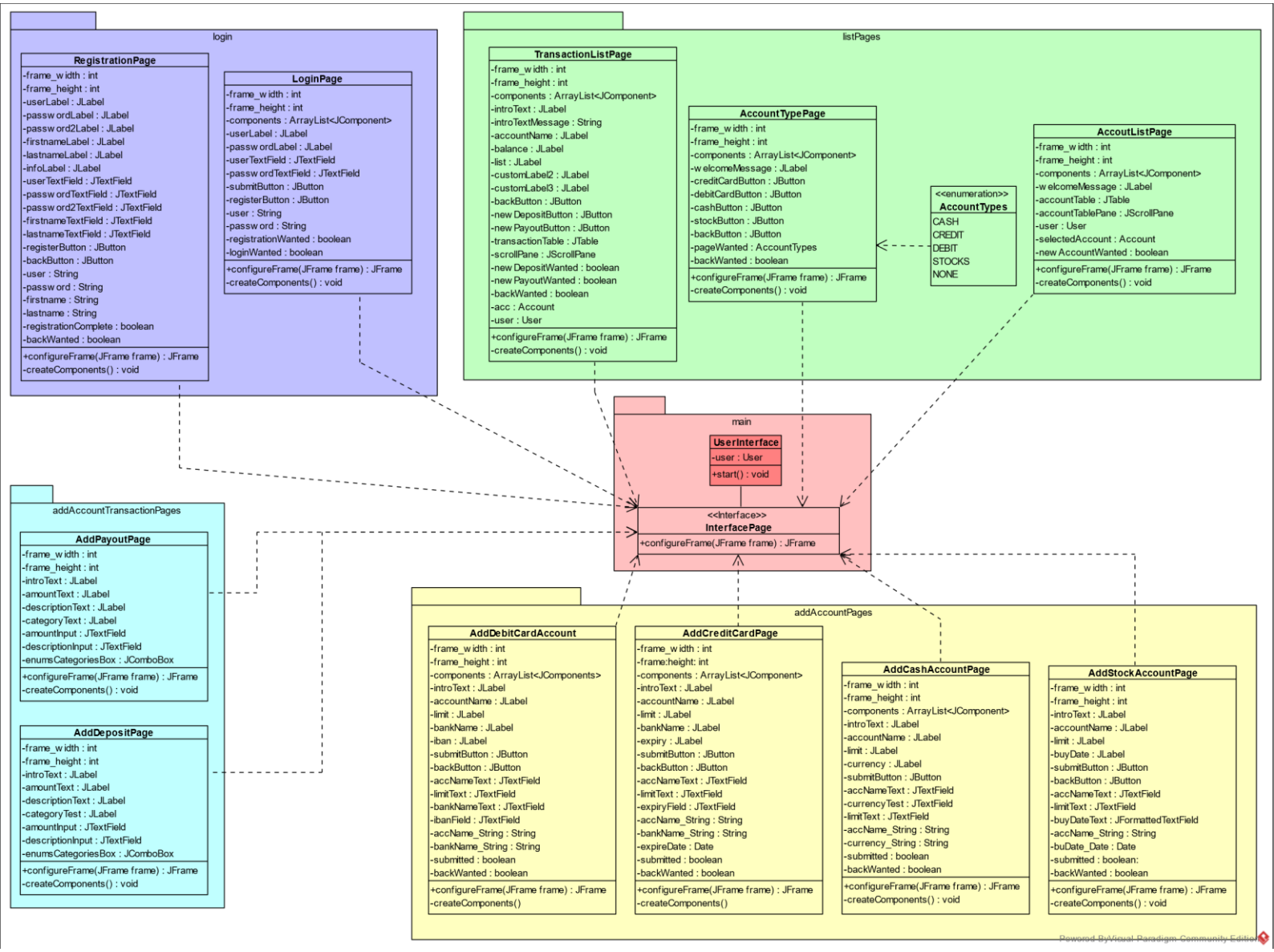

Figure 3: User Interface in detail

1.1.2 Technology Stack

As build tool we decided to use Gradle. For unit tests we use JUnit 5.4.0.

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.

For the user interface we use Java Swing. We agreed on using Swing because we chose to keep the GUI simple. Another reason we chose this API was because two of our team members had some basic experience from the course "Software Engineering 1". Swing has also a pretty good documentation (https://docs.oracle.com/javase/8/docs/api/javax/swing/package-summary.html).
An alternative would have been JavaFX, but because no one had experience with it and a simple UI is easy to build with Swing, we agreed to choose Swing over JavaFX.
For graphical representation within the user interface we will use "JFreeChart" (jfree.org/jfreechart/).

1.2 Design Patterns

1.2.1 Iterator Pattern

General outline of the pattern: In order to ensure that the saved data cannot be messed with (like removed from the storage), the classes involved in the pattern only provide operations that do not alter the data like addition and retrieval of data. The CustomList class supports adding new elements and getting the number of saved elements. 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. Depending on the context, one can add much more functionality to the interfaces (and therefore the implementing classes) but for this project, adding, getting the size of the container and traversing by an iterator will suffice. The CustomListIterator is implemented as private class of the CustomList since these two classes are directly related.

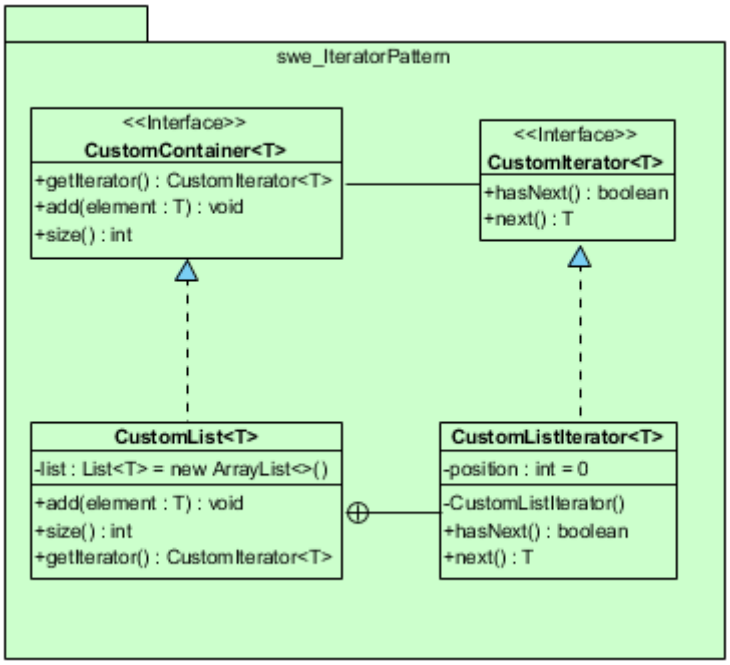

Figure 4: Class diagram of the iterator pattern

Since the interfaces only define the required methods, we just provide the implementations of said methods by the CustomList and the CustomListIterator:

```java
public class CustomList<T> implements CustomContainer<T> {
  private List<T> list = new ArrayList<>();

  /** Creates an empty CustomList. */
  public CustomList() {}

   * Creates a CustomList from an existing List.
  public CustomList(List<T> list) {

   * Adds a new element in this {@code CustomList}.
  @Override
  public void add(T element) {
    this.list.add(element);
  }

   * Returns the number of elements in this {@code CustomList}.
  @Override
  public int size() {
    return this.list.size();
  }

   * Returns a {@linkplain CustomIterator} pointing to the first element in the {@code CustomList}.
  @Override
  public CustomIterator<T> getIterator() {
    return new CustomListIterator<T>();
  }

  /**
   * The Iterator working on instances of CustomList. By default, this Iterator always points at the
   * first element of the CustomList. Implemented as a private nested class inside the
   * CustomList since this Iterator directly belongs to the CustomList.
   */
  private class CustomListIterator<T> implements CustomIterator<T> {
    private int position = 0;

    private CustomListIterator() {}

    /** {@inheritDoc} */
    @Override
    public boolean hasNext() {
      return this.position < CustomList.this.list.size();
    }

    /** {@inheritDoc} */
    @Override
    public T next() throws NoElementException {
      if (!hasNext()) throw new NoElementException("This iterator has already processed all elements !");
      T elem = (T) CustomList.this.list.get(this.position);
      this.position++;
      return elem;
    }
  }
}
```

Figure 5: Overview of the implementation of the iterator pattern

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. The datastructure can be used for any kind of data where we want to be sure that they cannot be removed though.

How is it used:

1. The CustomContainer of transactions of a user is used to create a table inside a UI-Page that displays the transactions done by a specific Account. For now, this table shows all Transactions, constraints on which Transactions to display are not yet included. Usage of the CustomContainer guarantees that the order, in which the transactions were originally performed, is maintained.

```
String[] TransactionDescription = {"Type", "Descriptions", "Amount", "Creation-Date",
    "Category"};
CustomContainer<Transaction> transactionlist = user.getTransactions().get(acc.getAccount_number());
int listSize = transactionlist == null ? 0 : transactionlist.size();
String[][] TransactionList_VISU = new String[listSize][6];

if (listSize > 0) {

  CustomIterator<Transaction> it = transactionlist.getIterator();

  int i = 0;

  while (it.hasNext()) {
    Transaction transtemp = it.next();

    if (transtemp instanceof Payout) {
      TransactionList_VISU[i][0] = "Payout";
      TransactionList_VISU[i][4] = ((Payout) transtemp).getPayoutCategory()
          .toString();
    } else {
      TransactionList_VISU[i][0] = "Deposit";
      TransactionList_VISU[i][4] = ((Deposit) transtemp).getDepositCategory()
          .toString();
    }

    TransactionList_VISU[i][1] = transtemp.getDescription();
    TransactionList_VISU[i][2] = "" + transtemp.getAmount();
    TransactionList_VISU[i][5] = "" + transtemp.getID();
    TransactionList_VISU[i++][3] = transtemp.getCreationDate().toString();


  }

}

transactionTable = new JTable(TransactionList_VISU, TransactionDescription);
```

Figure 6: Usage of the iterator pattern

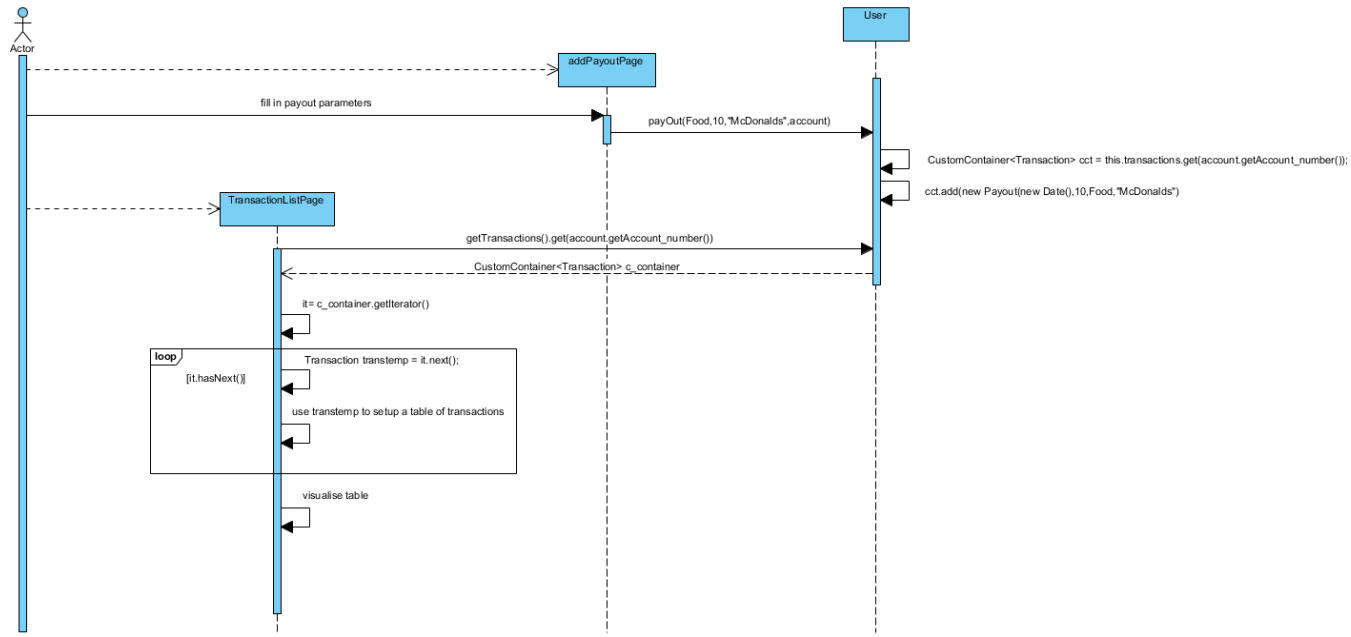

Figure 7: Sequence diagram for iterator pattern

2. In MongoDB the Iterator pattern is used to iterate over all accounts and transactions of a user. All this information is then saved in a Document which is then inserted in our database.

```
public void insertUser(User user)
{
  Map<Integer, CustomContainer<Transaction>>  Transactions =user.getTransactions();
  CustomContainer<Account>  accounts =user.getAccounts();
  List<Document> accounts_array = new ArrayList<>();
  List<Document> trans_array = new ArrayList<>();
  CustomIterator<Account> iter=accounts.getIterator();

  while (iter.hasNext()) {
    Document doc=Account(iter.next());
    accounts_array.add(doc);
  }

  for (Entry e : Transactions.entrySet())
  {
    CustomContainer<Object> list = (CustomList<Object>) e.getValue();
    CustomIterator<Object> iterator = list.getIterator();
    Integer account_number = (Integer) e.getKey();
    while (iterator.hasNext()) {
      Document doc=getTrans(iterator.next(),account_number);
      trans_array.add(doc);
    }
  }

  Document dep = new Document("_id", user.getUserID())
      .append("First Name",user.getFirstname())
      .append("Last Name", user.getLastname())
      .append("Transactions",trans_array )
      .append("Accounts", accounts_array);


  collection = database.getCollection( collectionName: "User");
  collection.insertOne(dep);
}
```

Figure 8: Usage of iterator in MongoDB

1.2.2 Strategy Pattern

General outline of the pattern: Following the principle of 'programming to an interface not an implementation' this pattern is used to decouple methods from another class and because of that the dependency to other classes is less. This pattern can also help in choosing the wanted behaviour dynamically instead of statically because we can change the behaviour on the fly.

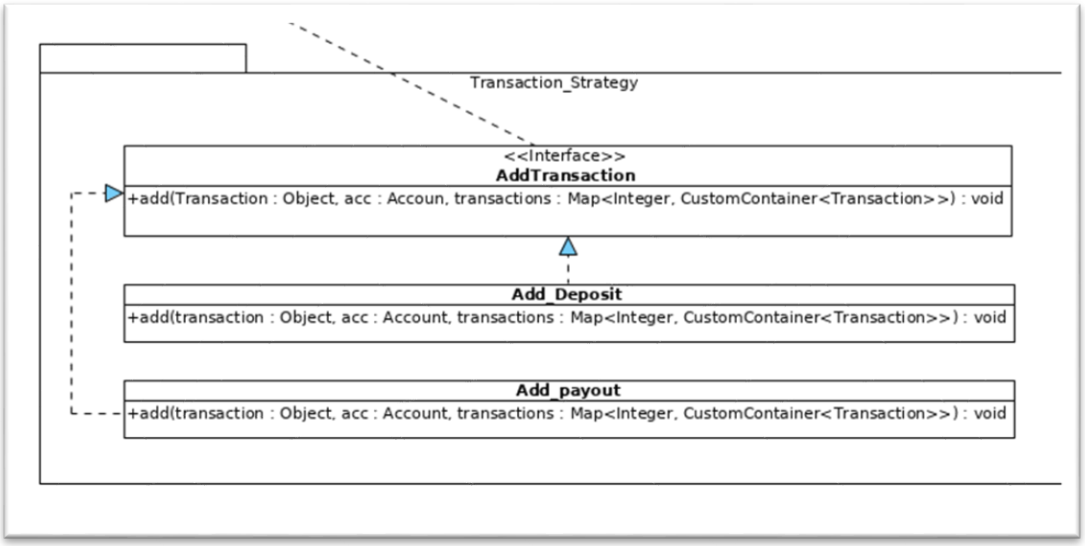

Figure 9: Class diagram of the strategy pattern

How does the pattern relate to one of the FR: With the help of this pattern we are able to decouple the transaction methods from the user and because of that it is also possible for us to choose transaction method dynamically instead of statically. So we don't have to write add_deposit or add_payout anymore. Now we can choose the method based on the given object.

How is it used: Based on the transaction type which is given to the method, the program chooses, if it should perform a deposit or a payout. If the given object is neither a deposit or a payout class an exception is thrown. The methods add Deposit and add payout are implementing the interface addTransaction so they are interchangeable.

```
public void addTransaction(Object transaction,Account account)
{
  AddTransaction add;
  if(transaction.getClass().equals(Deposit.class))
  {
    add=new Add_Deposit();
    try
    {
      add.add(transaction,transactions,account);
    }
    catch (Exception e)
    {
      System.out.println(e);
    }
  }
  else if(transaction.getClass().equals(Payout.class))
  {
    add=new Add_Payout();
    try
    {
      add.add(transaction,transactions,account);
    }
    catch (Exception e)
    {
      System.out.println(e);
    }
  }
  else
  {
    throw new UnknownTransactionException("Unknown Transaction");
  }

}
```

Figure 10: Use of the different strategies

```
@Override
public void add(Object Transaction,
    Map<Integer, CustomContainer<Transaction>> transactions, Account account)
    throws LimitException {
  Date date = new Date();
  Payout payout =(Payout) Transaction;
  if (account.getBalance() - payout.getAmount()  < account.getLimit())
    throw new LimitException("Limit exceeded!");

  account.payout(payout.getAmount());

  transactions.putIfAbsent(account.getAccount_number(), new CustomList<>());
  transactions.get(account.getAccount_number()).add(payout);
}
```

Figure 11: Strategy payout

2 Code Metrics

For a getting information about our code metrics, we used a plugin for IntelliJ called “Statistics”, version 3.5.

Number of packages: 15

Lines of source code: 2590

Lines of comments: 1096

Number of classes: 60

Please not that all these numbers also include tests. We have 45 classes without tests.

We also used a static code analysis tool, called “FindBugs” version 1.0.1, also a plugin for IntelliJ. In figure 12, you can see a screenshot of found bugs.

Most of the bugs where because of unfinished implementations, especially in regard of the persistency. Some other bugs, like “Unread fields should be static” quickly helped us improving our code.

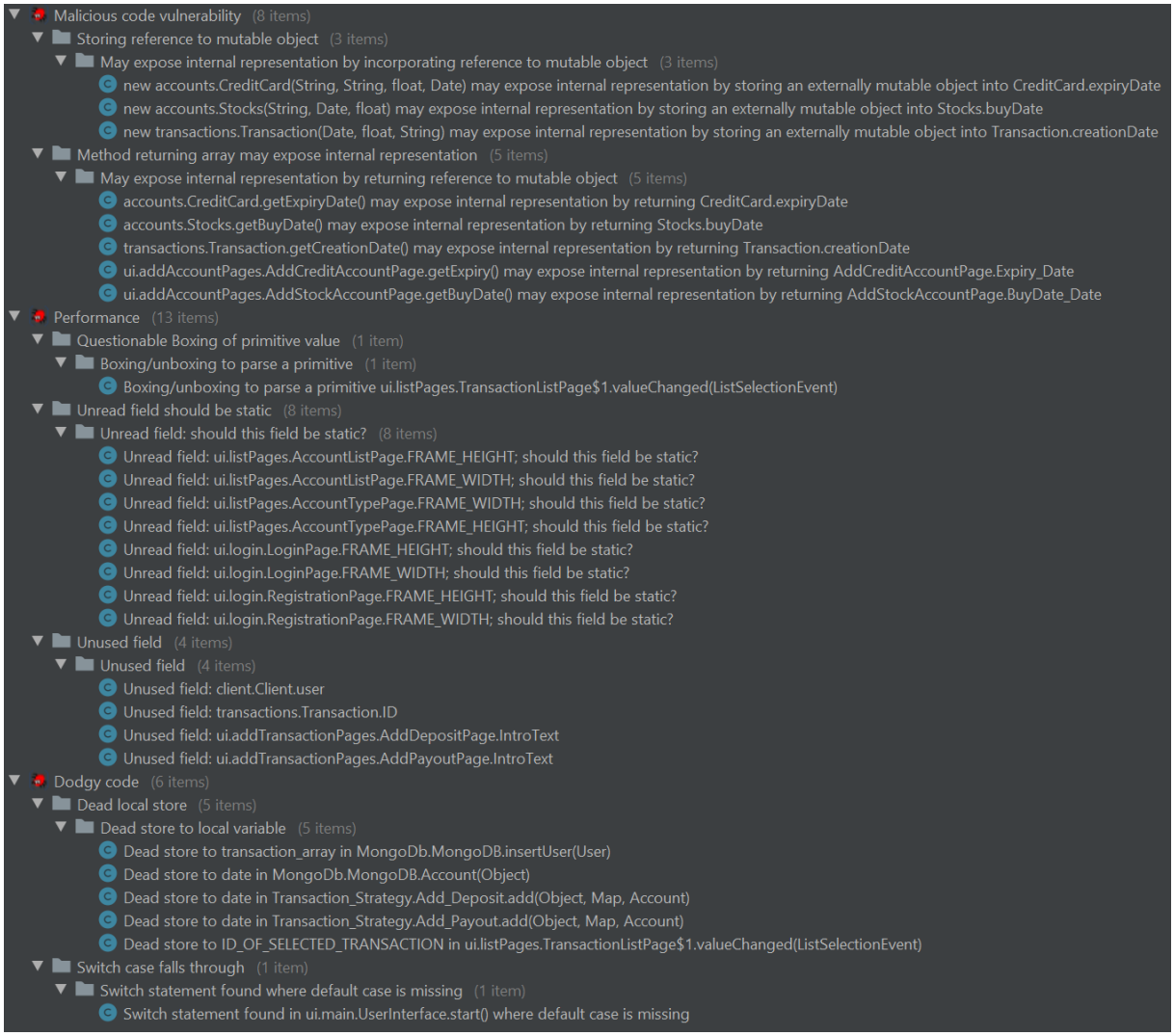

Figure 12: Screenshot of found bugs using FindBugs

3 Team Contribution

3.1 Project Tasks and Schedule

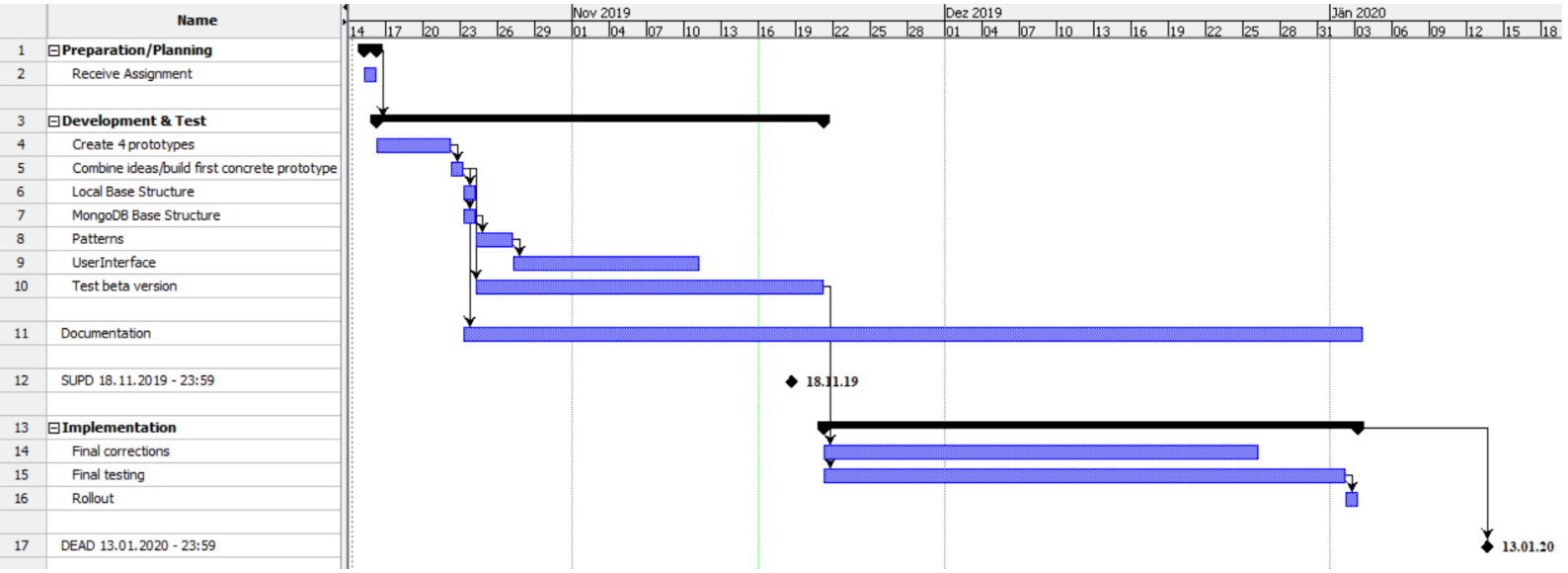

Figure 13: Gantt Diagram of the project

3.2 Distribution of Work and Efforts

Categories:	Paul	Michael	Patrick	Lukas
Planning:	3h	3h	3h	3h
Commuting–allowance:	1h	1h	1h	1h
Reading Assignment:	2h	2h	2h	2h
UI:	6h	0h	8h	0h
Class-Diagram:	1h	3h	2h	2h
Strategy-Pattern:	0h	0h	0h	3h
Iterator Pattern:	0h	4h	0h	0h
Observer-Patter:	2h	0h	0h	0h
Debugging UI:	1h	0h	1,5h	0h
Gantt-Diagram:	0h	2h	0h	0h
MongoDB:	0h	0h	0h	4h
Writing Report	1h	2h	2h	1,5h
Total:(in hours.. h)	17h	17h	19,5h	16,5h

A1 HowTo

Start the Jar file. Registration page doesn’t create new users for now. There is just 1 User Object (hardcoded). To login enter ‘admin’,’admin’ on the Login-Page.

Adding Parameters in the New-Accounts and New-Transactions Page’s does not check for correct values (e.g. empty, negative values etc..) for now.

New account: Login->CreateAcc ->select your desired acc -> enter values, submit or back!

New Transaction: Login -> select the Acc that u want the transaction to happen from(double click in List) -> Payout or deposit at bottom -> enter values, back or submit!