

Title: A bank account management system using Object Oriented design and principle, and adopting design patterns and framework

Author: Usman Basharat

Date: 17<sup>th</sup> December 2018

Course: COMP-1605 Enterprise Patterns and Framework

Word Count: 3154

School of Computing and Mathematical Sciences



## Table of Contents

Table of Contents.....	2
Introduction .....	3
Statement of Functionality .....	4
Declaring any bugs .....	8
Entity Relationship Diagram .....	9
Class Diagram of Task 1.....	11
Final Class Diagram .....	13
Task 2 .....	15
List of Files.....	15
Screenshots.....	15
Advantages of Approach.....	15
Disadvantage of Approach.....	15
Task 3 .....	16
List of Files.....	16
Screenshots.....	16
Advantages of Approach.....	16
Disadvantage of Approach.....	16
Task 4 .....	17
List of Files.....	17
Screenshots.....	17
Advantages of Approach.....	17
Disadvantage of Approach.....	17
Task 5 .....	18
Future Enhancements.....	19
SQL Database Examples .....	20
Conclusion.....	20
References .....	21

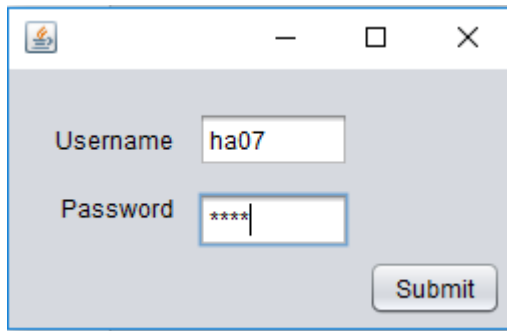
## Introduction

In this report, I am going to present a bank account management system using Object Oriented design and principle for this application. Adopting a design and framework within this application is given much importance as many patterns have been inserted.

This coursework has been split within 5 Tasks that needs to be completed. Task 1 are to complete the creation of customer, implement all payments and all views. Task 2, Task 3 and Task 4 are all to update from Task 1. And Task 5 is to reflect upon what has been completed. All of these tasks that have been completed will be given in much greater detail in the report. A statement of functionality will explain these tasks of what has been completed in each task and what has been completed. All software application has bugs that need to be explained.

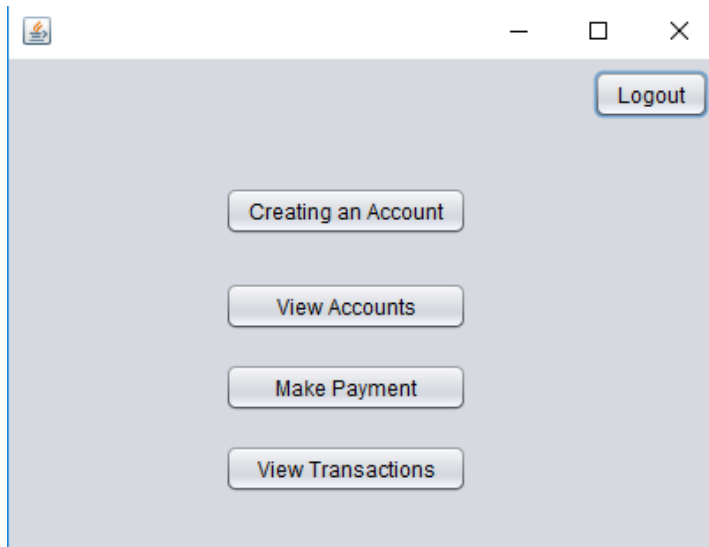
In this application, I will be explaining any bugs that have been missed. Entity Relationship Diagram and two Class Diagrams are going to be shown. One Class Diagram will show it for only Task 1 and one final Class Diagram for the whole system.

## Statement of Functionality



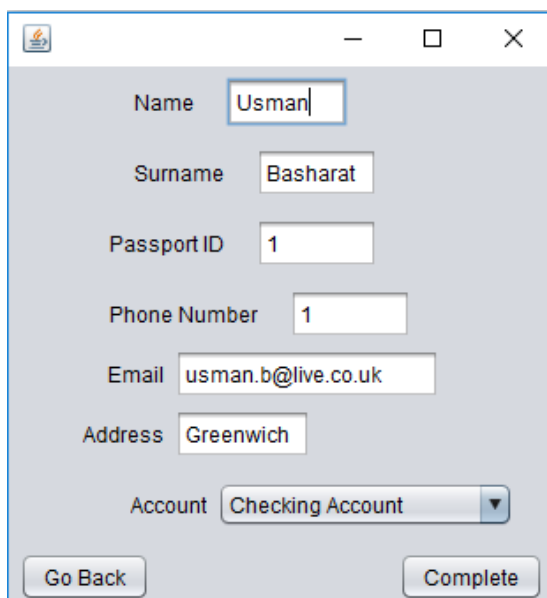
A screenshot of a login window. It has a title bar with a small icon, a minus button, a maximize button, and a close button. The window contains two text input fields: 'Username' with the text 'ha07' and 'Password' with four asterisks '\*\*\*\*'. Below the password field is a 'Submit' button.

Figure 1 shows the login for the application



A screenshot of a user application menu. It has a title bar with a small icon, a minus button, a maximize button, and a close button. In the top right corner is a 'Logout' button. In the center, there are four buttons stacked vertically: 'Creating an Account', 'View Accounts', 'Make Payment', and 'View Transactions'.

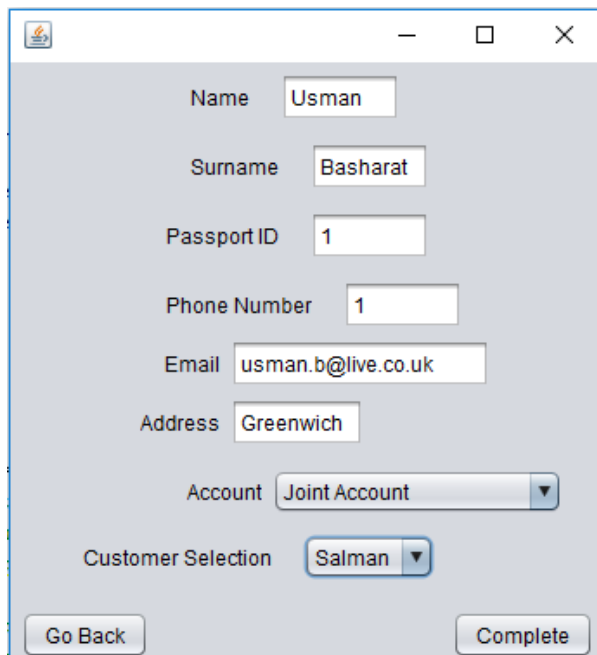
Figure 3 shows the options for the user's application.



A screenshot of a customer account creation form. It has a title bar with a small icon, a minus button, a maximize button, and a close button. The form contains several text input fields: 'Name' with 'Usman', 'Surname' with 'Basharat', 'Passport ID' with '1', 'Phone Number' with '1', 'Email' with 'usman.b@live.co.uk', and 'Address' with 'Greenwich'. Below the address field is a dropdown menu for 'Account' with 'Checking Account' selected. At the bottom are two buttons: 'Go Back' and 'Complete'.

Figure 2 shows the creation for customer account.

Figure 2 shows how customer creation is created. The sort code and account number are automatically generated. A random number generated are created for this to be completed. This makes it easier so that unique sort code and account numbers are not similar with each other that have been created. Also, Customer and Accounts have an account link with each other. This is to identify which customer links with each account. This makes it easier.

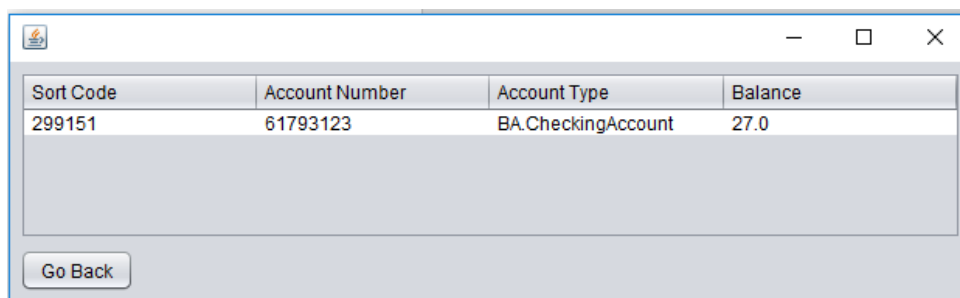


A screenshot of a web application window titled "Customer Creation" showing a form for creating a joint account. The form contains the following fields and controls:

- Name:
- Surname:
- Passport ID:
- Phone Number:
- Email:
- Address:
- Account: - Customer Selection:

At the bottom of the form are two buttons: "Go Back" and "Complete".

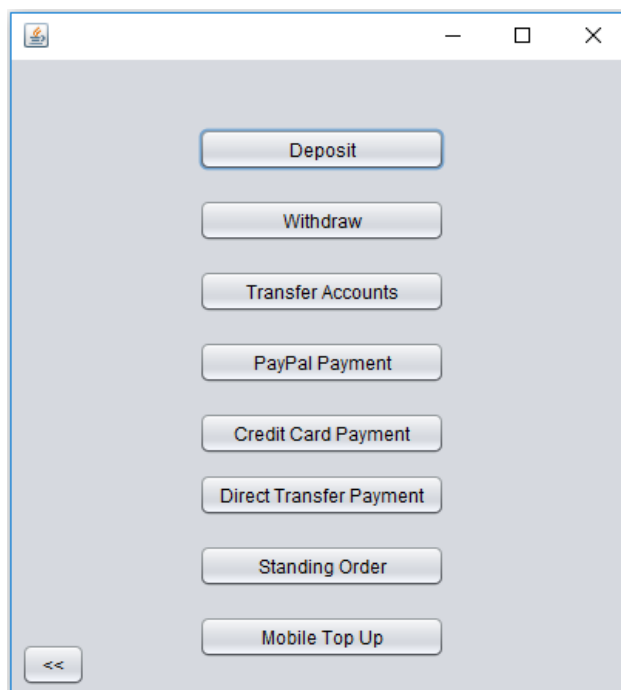
Figure 4 shows the joint account for customer creation.



A screenshot of a web application window titled "View Account" showing a table with account details. The table has four columns: Sort Code, Account Number, Account Type, and Balance. Below the table is a "Go Back" button.

Sort Code	Account Number	Account Type	Balance
299151	61793123	BA.CheckingAccount	27.0

Figure 5 shows the view account for the user.

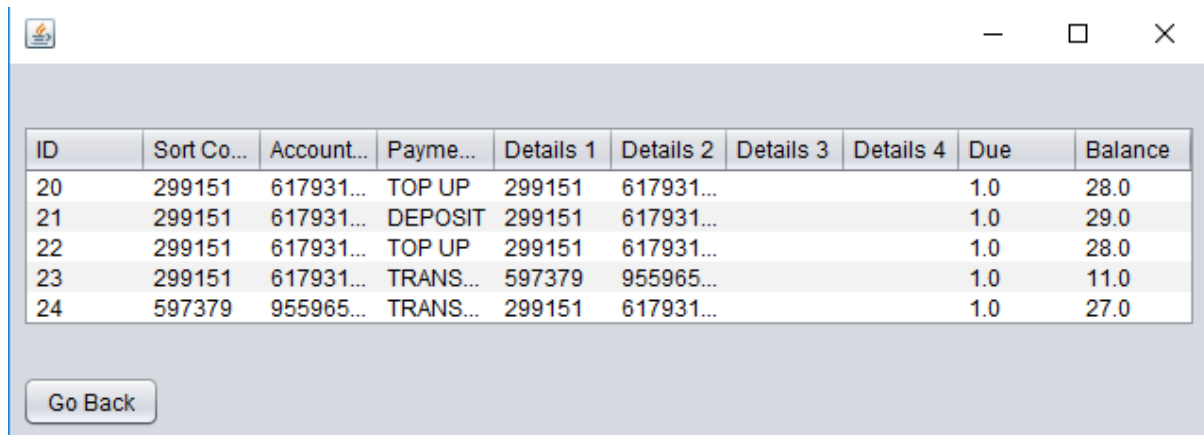


A screenshot of a web application window titled "Payment Options" showing a list of payment methods. The options are displayed as buttons in a vertical list:

- Deposit
- Withdraw
- Transfer Accounts
- PayPal Payment
- Credit Card Payment
- Direct Transfer Payment
- Standing Order
- Mobile Top Up

At the bottom left of the window is a "<<" button.

Figure 6 shows the all the payment options.

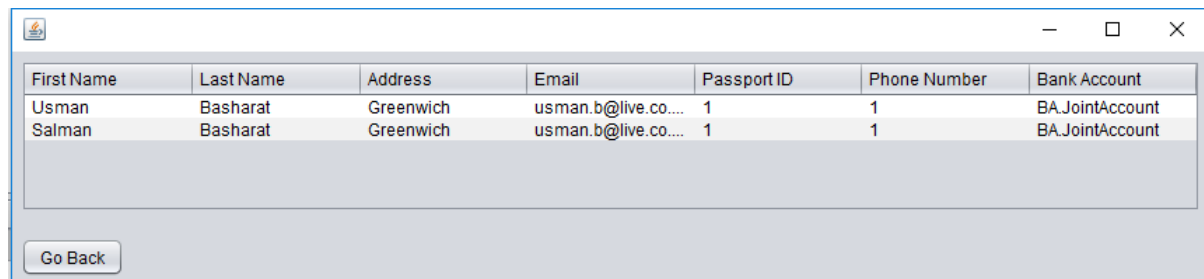


ID	Sort Co...	Account...	Payme...	Details 1	Details 2	Details 3	Details 4	Due	Balance
20	299151	617931...	TOP UP	299151	617931...			1.0	28.0
21	299151	617931...	DEPOSIT	299151	617931...			1.0	29.0
22	299151	617931...	TOP UP	299151	617931...			1.0	28.0
23	299151	617931...	TRANS...	597379	955965...			1.0	11.0
24	597379	955965...	TRANS...	299151	617931...			1.0	27.0

Go Back

Figure 7 shows the transaction history based upon the current month.

Referring to Figure 7, this shows the current transactions are only up to a month. Last month one transaction was complete. However, the current transaction history is according to the current month.



First Name	Last Name	Address	Email	Passport ID	Phone Number	Bank Account
Usman	Basharat	Greenwich	usman.b@live.co....	1	1	BA.JointAccount
Salman	Basharat	Greenwich	usman.b@live.co....	1	1	BA.JointAccount

Go Back

Figure 8 shows the view customers that are associated.

	Features – Functional Requirement	Statement of Functionality	Evidence
<b>Task 1</b>	Customer can open three different types of accounts, e.g., savings, current/checking, and ISA (individual savings account)	Function has been implemented.	Please refer yourself to Class Diagram to see this implemented as a screenshot.
	A customer should be able to transfer or move his/her money from one account to another via the application besides the usual deposit, withdraw and view balance operations.	Function has been implemented	Please refer yourself to Figure 7 where this shows this.
	You need to keep provisions for a customer to pay via multiple ways from his/her account.	Function has been implemented.	Please refer yourself to Figure 6 to see this.
	A number of views are primarily required: 1) a summarised view of number of accounts for a particular customer and the balance of each account	Function has been implemented	Please refer yourself to Figure 5 to see this.
	2) A detailed view of a particular account with transactions for the current month,	Function has been implemented.	Please see Figure 7 for this.

<b>Task 2</b>	Suppose a new type of payment method (e.g., standing order) needs to be added on top of the existing one	Function has been implemented.	Please refer yourself to Figure 6.
<b>Task 3</b>	Suppose a new type of operation (e.g., top-up mobile phone) needs to be added on top of the existing ones for the account as discussed in (ii)	Function has been implemented.	Please refer yourself to Figure 6.
<b>Task 4</b>	Suppose the bank came back to you again to add the provisions for joint accounts [another type of account] for customers	Function has been implemented.	Please refer yourself to Figure 4
	they required another view which should list all associated customers for a particular account which a customer can view from his/her application	Function has been implemented.	Please refer yourself to Figure 8.
<b>MVC Pattern</b>	This design pattern has been used within deposit, withdraw and mobile top up.	Function has been implemented.	Please refer yourself to code to see the evidence.
<b>Singleton</b>	This design pattern has been used within Model.java.	Function has been implemented.	Please refer yourself to Figure 11.
<b>JUnit Testing</b>	This type of testing has been used and included under the testing package for CustomerAndAccountTest.java, DepositPaymentTest.java, TransactionsTest.java and WithdrawPaymentTest.java.	Function has been implemented.	Please refer yourself over to the code to see evidence of this test.
<b>Hibernate</b>	This type of framework has been implemented under Hibernate.java	Function has been implemented	Please refer yourself over to the code to see evidence of this test.
<b>Spring Framework</b>	This type of framework has been implemented under Beans.xml	Function has been implemented	Please refer yourself over to the code to see evidence of this test.
<b><i>Please if you want to see in more detail any of these implementations, you can refer yourself over to database approach where this will be discussed.</i></b>			

## Declaring any bugs

We strive for perfection in any software. However, Steel Kiwi discusses on the website about how bugs have classifications. These are critical, major, moderate, minor and cosmetic. These are the levels that emphasizes the importance of bugs (Kiwi, 2017). One bug that has given much importance is that when mismatching sort code and account number. This could be sorted by validating this error before the submit button reads the sort code and account number. However, I did not manage to sort this error out due to the time-frame of this coursework. I would have sorted this out if more time was given. This could have been an enhancement for future references.

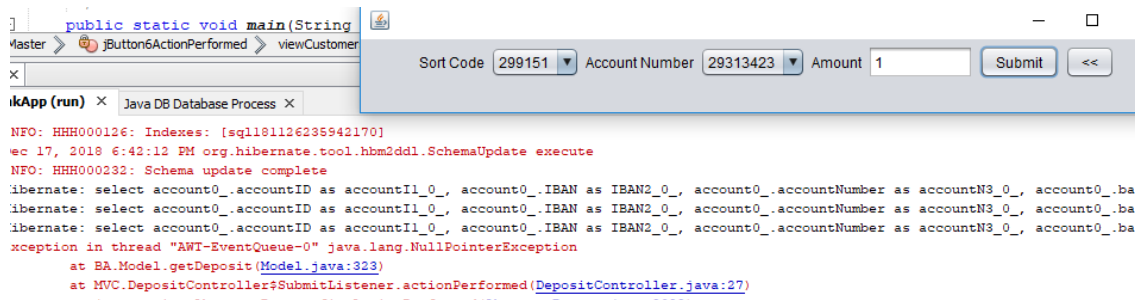


Figure 9 shows the error that mismatches sort code and account number.

Referring to Figure 5, this shows the bank account view. This account that has been implemented as a joint account. However, I managed to update the two accounts that link to the each other. However, I did not manage to get update the account that has been viewed on the accounts. This has still stayed the same as Checking Account. This should have been updated. However, I did not manage to sort this error out due to the time-frame of this coursework. I would have sorted this out if more time was given. This could have been an enhancement for future references.

One that may consider as a bug is that the balance goes to minus. As I do know that accounts may go into overdraft, but there are no restrictions on this. As said for the previous bugs, this could be the one that can be enhancement for future references.

Another bug that can be considered is that makes the application longer is that adding a new account, the user has to add a new type of account manually into the database. I do realise that this is a longer way and a new type can be added to make get rid of this bug. I do understand how long this would make creation of new account type longer. However, this was one of the major improvements that can be made for the application too.



## Entity Relationship Diagram

ERD is presented below to demonstrate the outlook of the database. As this is a small database, connections for each entity small. The main relationship that are important in the ERD is between the **Account** and **Customers**. The main relationship is **One to Many** as Customers have many accounts between them. Others have a small relationship between each other. Referring to Figure 10, this shows the relationship between each other.

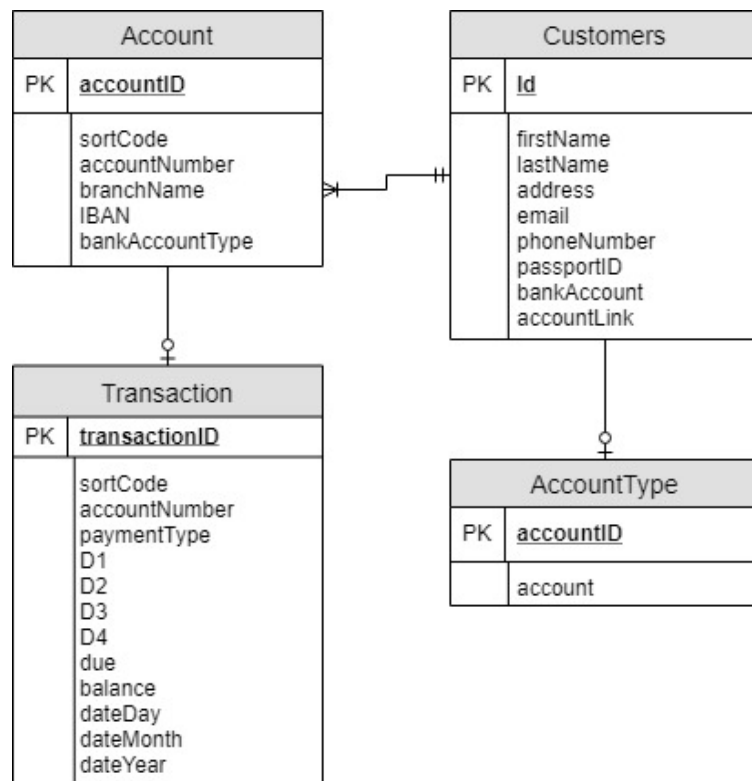
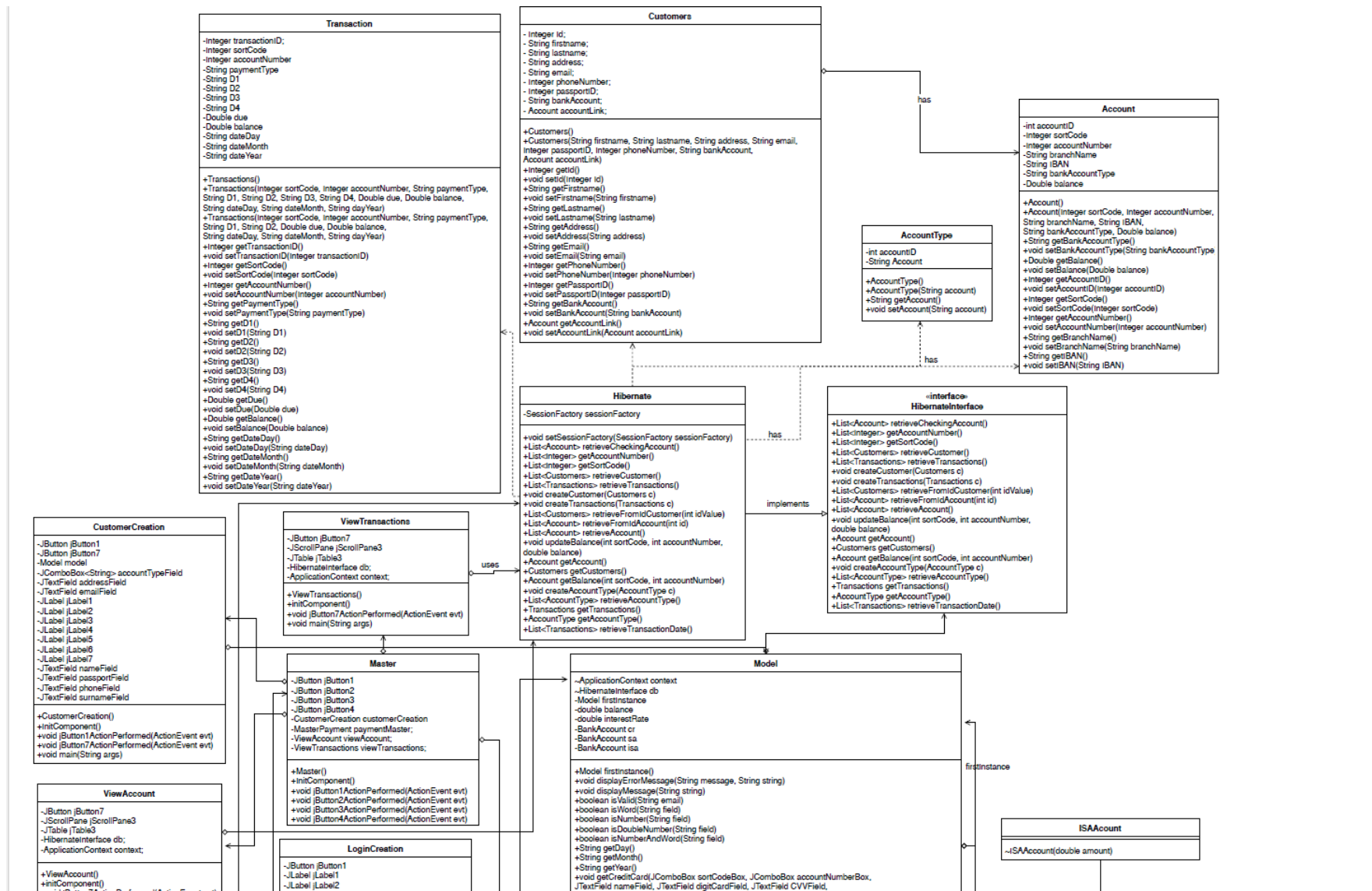
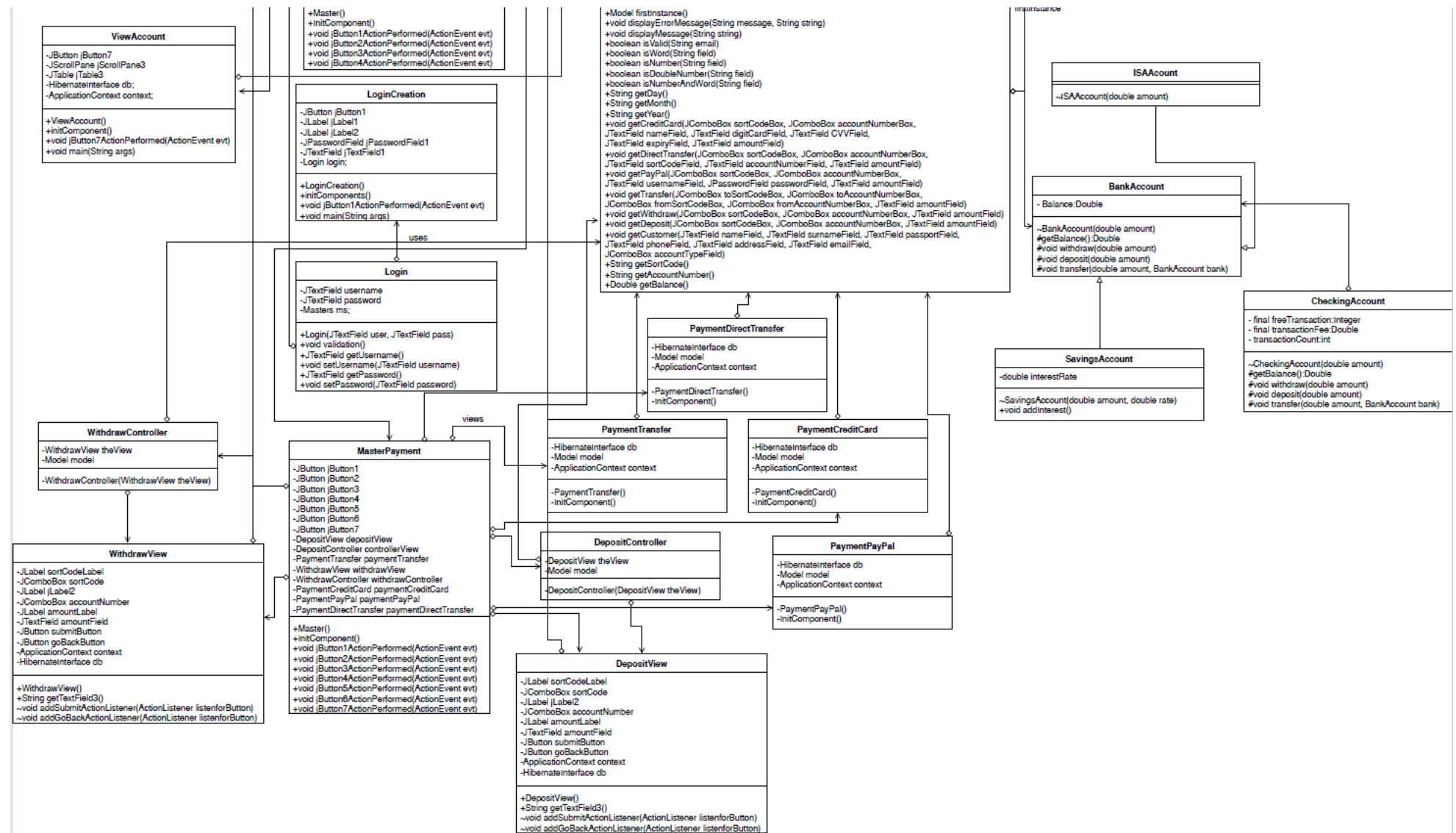


Figure 10 shows the Entity Relationship Diagram for my prototype

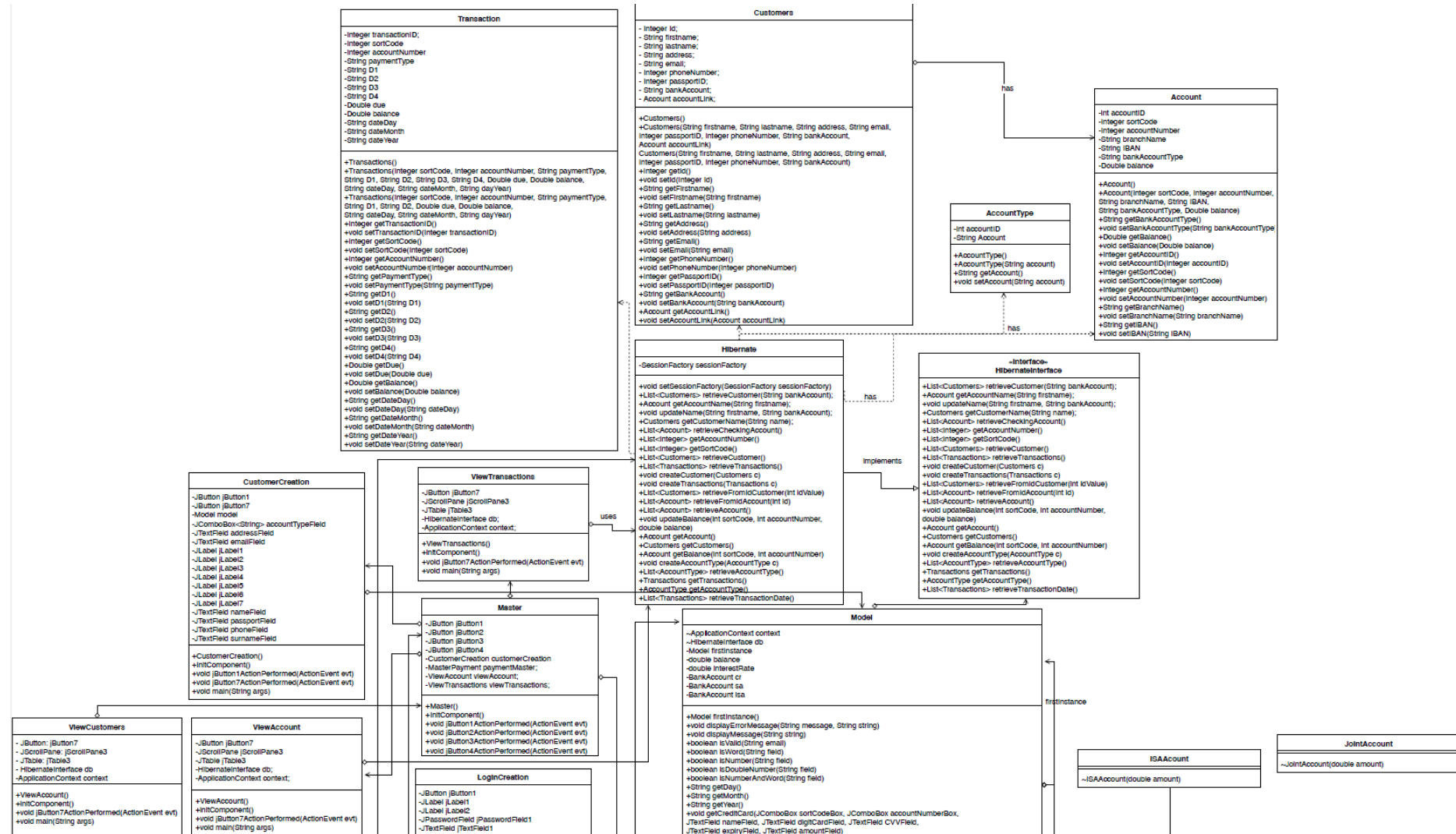


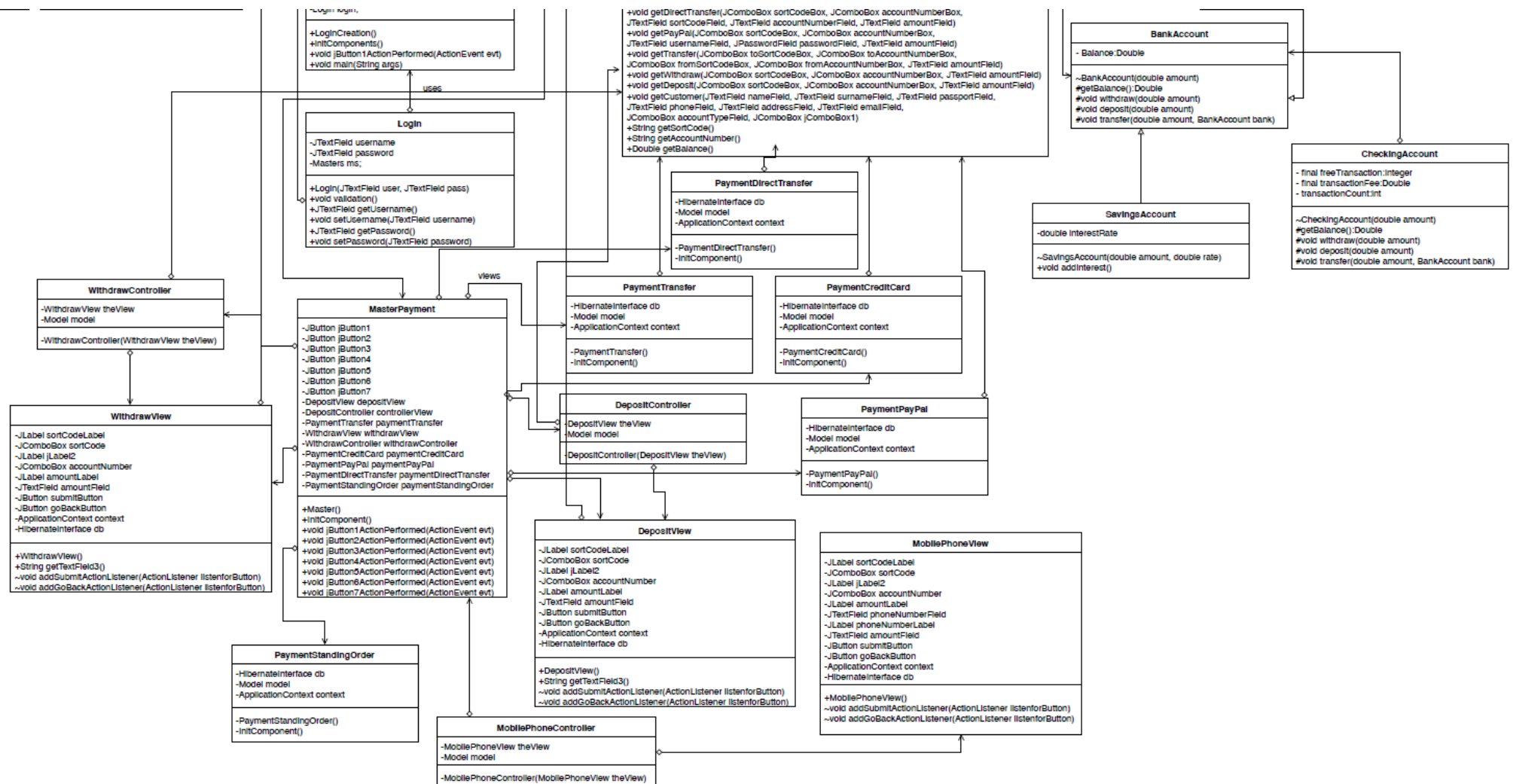
## Class Diagram of Task 1





## Final Class Diagram





Please find the attached ZIP file to view the Class Diagram in full detail.

## Task 2

### List of Files

Here is the list of files that were changed and incorporated.

- PaymentStandingOrder.java [Incorporated]
- MasterPayment.java [Changed]
- Model.java [Changed]

### Screenshots

```
public void getStandingOrder(JComboBox sortCodeBox, JComboBox accountNumberBox, JTextField numberField, JPasswordField CVVField,
    JTextField amountField) {
    String sortCode = sortCodeBox.getSelectedItem().toString();
    String accountNumber = accountNumberBox.getSelectedItem().toString();
    String number = numberField.getText();
    String CVV = CVVField.getText();
    String amount = amountField.getText();

    if (sortCode.isEmpty() && accountNumber.isEmpty()
        && number.isEmpty() && CVV.isEmpty() && amount.isEmpty()) {
        displayMessage("Please fill in the rest of the details to submit a payment");
    } else if (!(isNumber(sortCode) && isNumber(accountNumber))) {
        displayMessage("Please enter a valid number");
    } else if (!(isDoubleNumber(amount))) {
        displayMessage("This is not correct format.");
    } else {
        int resultSortCode = Integer.parseInt(sortCode);
        int resultAccountNumber = Integer.parseInt(accountNumber);
        double resultInput = Double.parseDouble(amount);
        if (db.getSortCode().contains(resultSortCode) && db.getAccountNumber().contains(resultAccountNumber)) {
            double result = (db.getBalance(resultSortCode, resultAccountNumber).getBalance() - resultInput);
            db.updateBalance(resultSortCode, resultAccountNumber, result);

            System.out.println(amount + " " + "using Standing Order");

            Transactions nPT = new Transactions(resultSortCode, resultAccountNumber, "STANDING ORDER", number,
                CVV, Double.parseDouble(amountField.getText()), result, getDay(), getMonth(), getYear());
            db.createTransactions(nPT);
            System.out.println("Transaction has been complete");
        }
    }
}
```

```
private void jButton8ActionPerformed(java.awt.event.ActionEvent evt) {
    PaymentStandingOrder paymentStandingOrder = new PaymentStandingOrder();
    paymentStandingOrder.setVisible(true);
    this.setVisible(false);
    paymentStandingOrder.setLocationRelativeTo(null);
}
```

### Advantages of Approach

- A few advantages for this approach are that a very few classes were changed in order to complete this task.
- Data that already was written in the code was reused.
- Data from the user are separated.

### Disadvantage of Approach

- A new view for JFrame had to be created to make standing order work by implementing the all the buttons and fields. This should be changed so that a similar view, such as withdraw can be used as a base to recreate a standing order view.
- A better approach could have been to introduce more of polymorphism within this change. A template of view could have been implemented. And, all that can be included is another class that overrides this. This could have improved the code drastically.



### Task 3

Here is the list of files that were changed and incorporated.

#### List of Files

- MobilePhoneController.java [Incorporated]
- MobilePhoneView.java [Incorporated]
- Model.java [Changed]
- MasterPayment.java [Changed]

#### Screenshots

```
public void getMobilePhone(JComboBox sortCodeBox, JComboBox accountNumberBox, JTextField phoneNumber, JTextField amountField) {
    String sortCodeString = sortCodeBox.getSelectedItem().toString();
    String accountNumberString = accountNumberBox.getSelectedItem().toString();
    String amountString = amountField.getText();

    if (sortCodeString.isEmpty() && accountNumberString.isEmpty() && amountString.isEmpty()) {
        displayMessage("Please fill in the rest of the details to submit a payment");
    } else if (!isDoubleNumber(amountString) && isNumber(sortCodeString) && isNumber(accountNumberString)) {
        displayMessage("This is not correct format.");
    } else {
        int sortCodeInt = Integer.parseInt(sortCodeString);
        int accountNumberInt = Integer.parseInt(accountNumberString);
        double amountDouble = Double.parseDouble(amountString);

        if (db.getSortCode().contains(sortCodeInt) && db.getAccountNumber().contains(accountNumberInt)) {
            double result = (db.getBalance(sortCodeInt, accountNumberInt).getBalance() - amountDouble);
            System.out.println("BEFORE BALANCE:" + db.getBalance(sortCodeInt, accountNumberInt).getBalance());
            db.updateBalance(sortCodeInt, accountNumberInt, result);
            System.out.println("AFTER BALANCE:" + db.getBalance(sortCodeInt, accountNumberInt).getBalance());
            System.out.println(amountField.getText() + " " + " Mobile Top Up");

            Transactions wTransaction = new Transactions(sortCodeInt, accountNumberInt, "TOP UP",
                sortCodeString, accountNumberString, amountDouble, result, getDay(), getMonth(), getYear());
            db.createTransactions(wTransaction);
        }
    }
}
```

```
private void jButton9ActionPerformed(java.awt.event.ActionEvent evt) {
    MobilePhoneView depositView = new MobilePhoneView();
    MobilePhoneController controllerView = new MobilePhoneController(depositView);
    depositView.setVisible(true);
    this.setVisible(false);
    depositView.setLocationRelativeTo(null);
}
```

#### Advantages of Approach

- MVC design pattern has been used with this approach to separate the view, model and controller.
- Very few classes needed to change for this to be implemented.
- Model is merged in one

#### Disadvantage of Approach

- View and Controller has separate controller for each MVC payment that has been implemented. This should have been changed to be merged like Model has been implemented.



## Task 4

Here is the list of files that were changed and incorporated.

### List of Files

- Model.java [Changed]
- CustomerCreation.java [Changed]
- ViewCustomers.java [Incorporated]
- Hibernate.java [Changed]

### Screenshots

```
} else if ("Joint Account".equals(accountTypeField.getSelectedItem().toString())) {
    Customers newCustomers = new Customers(name, surname, address, email, phoneNumber, passportID, jt.getClass().getName().toString());
    db.createCustomer(newCustomers);
    db.updateName(jComboBox1.getSelectedItem().toString(), jt.getClass().getName());
}

@Override
public void updateName(String firstname, String bankAccount) {
    Session session = this.sessionFactory.openSession();

    session.beginTransaction();
    String queryString = "UPDATE from Customers SET bankAccount = :bankAccount where firstname = :firstname";
    //UPDATE entity SET attribute='value' WHERE anotherAttribute IN (val1, val2);
    Query query = session.createQuery(queryString);
    query.setString("firstname", firstname);
    query.setString("bankAccount", bankAccount);
    query.executeUpdate();
    session.getTransaction().commit();
}
```

### Advantages of Approach

- A few lines execute the joint account code.
- Separation of Model and View has been implemented.
- Database was separated and was used from one class.

### Disadvantage of Approach

- Database change for update Name method was added for joint account to be executed.
- Too many changes were made to incorporate this task.

## Task 5

Abstraction is the act of representing *essential* data without including the essential explanation for this. Abstraction was used within this application in *HibernateInterface.java* as the essential data are representation of Figure 11. Referring to Figure 11, this demonstration shows the interface of *Hibernate.java* class. As soon as Figure 11 class has been implemented, the abstraction shown is a must that has to be implemented in the *Hibernate.java* class. Figure 11 only shows one example of where Abstraction has been used out of the four pillars. However, in the *HibernateInterface.java* class, there are many more of these that have been used and 'explained' in *Hibernate.java* class.

```
public interface HibernateInterface {
    public abstract List<Account> retrieveCheckingAccount();
}
```

Figure 11 shows the example of where Abstraction of the four pillars have been used.

Encapsulation simply hides the implementation from the users. Chaitayna Singh states on her Beginners guide of how Encapsulation *"improves maintainability and flexibility and re-usability"* (Singh, 2018). Encapsulation was used as a separation from the users in the *Model.java* class. This stores all the data of from all the other classes. Both Class Diagrams represent this case later on in the report. Hibernation also uses constant encapsulation when entities' getters and setters were called back and forth. Encapsulation is known as *"data hiding"*. Making sure that users are kept away from the what is going behind the scenes are the best sense of security too. One way that this application separates data by keeping the database (*Hibernate.java*), data (*Model.java*) and each view away from each other.

Inheritance is the process of acquiring properties and functionalities of another class. The parent class is where these methods and behaviours are all used for its child class (Singh, 2018). Main reason of inheritance its ability to reuse code and this can be changed in its child class if any adjustments are needed can be made. One example in the application is when the *BankAccount.java* is the main parent class. *CheckingAccount.java*, *SavingsAccount.java* and *ISAAccount.java* are all inherited from the *BankAccount.java* class. Adjustments are made to each of these child classes to adjust to its own account. This is where Polymorphism comes in more detail. This is explained next.

Polymorphism uses inheritance as a base to improve the class. As mentioned above about Bank Account that was used in the application. Polymorphism was used in this application as *BankAccount.java* has a general withdraw, but *CheckingAccount.java* had used this but adjusted as an override of the withdraw method. This is essentially the principle of how Polymorphism works.

Model View Controller is a type of design pattern Shubham states in her tutorial that MVC *"used to separate the logic of different layers in a program in independent units."* (Shubham, 2018). Within this application, I have used this is Deposit Payment, Withdraw Payment and Mobile Phone Top Up. These are only selected classes that have been used as an example of how MVC Pattern can be used. Model is where the data is all stored, View is where the JFrame and actual 'view' has been implemented and Controller puts all of this together. All of the Controllers could have been merged together to create one. However, I created separate for each for simplicity MVC reasons. This is one of the improvements that can be made for this application that is later discussed in the report too.

Singleton is a type of design pattern that has been used within this application that restricts the instantiation of the class and ensures only one instance of the class has been used for each class (Pankaj, 2018). Restricting the instantiation of the classes makes it easier. However, if another is called and used, this would not work. This is the pure reason why I chose to put Singleton in as this

makes it simply easier to instance only once. “Model model = Model.firstInstance();” is how the Singleton class has been called from other classes for the *Model.java* class.

Hibernate is an Object Relational Mapping framework that as Fei, Jingjing and Xiao Luo mention that Hibernate acts as a “*the bridge between Java Applications and Relational Databases*” (Guo, Lui, Luo, 2012). Hibernate maps the classes entities to convert them to database tables. Hibernate has its edge as this process reduces time development by increasing development efficiency compared to SQL and JDBC. One main reason that users may choose to avoid Hibernate is the complexity of understanding the connection between the layer of database and application. I used the original Hibernate, however, Hibernate with Spring framework are the two combinations that were used. As Mahtab said, “*it’s an upgrade*”. Hibernate with Spring framework uses the *Beans.xml* file does all the settings for the database. Application Context is used to access this information for other classes. Aswani discusses in his flexibility of using different framework alongside Hibernate such as logging framework, JEE, JDK and many more (Aswani, 2014). Main reason for the use of Hibernate with Spring Framework was to make the database access and creation easier for this application. Spring Framework was later introduced and easy and flexible to implement. This made the code easier and rid of *HiberateUtil.java* and did the same job with a few adjustments. One adjustment that can be made for *Hibernate.java* access is the entities can be improved.

JUnit Testing is another type of framework that has been used within this application that simply has a structure of testing the code. Making sure that testing is important as this debugs errors within the application and this has certainly helped. For this application, I tested from using JUnit Testing is the creation of customer and accounts, withdraw and deposit payment testing and transaction test. More test could have been implemented. However, these were the main classes of the application that needed this. As I adopted to use JUnit Testing, I came across one error where the database was only reading the first sort code and account number. I came across this error whilst testing the customer creation for JUnit Testing. This is where JUnit Testing helps improve the code as this way certainly helped me.

## Future Enhancements

I feel that for this application, many future enhancements can be made for this application. One improvement that can be made for this application is the MVC design pattern. MVC was implemented and I made sure that the model was all in one to make it much more efficient. However, the controller for withdraw, mobile phone and deposit payment have each controller separately. This is the same for the view too. I managed to combine only the model. However, the controller and view could be improved for future enhancements.

Hibernate is that the Hibernate Example 3 was used as a base to implement Hibernate within my application. Hibernate with Spring Framework was later updated. However, no enhancement of Inheritance was used to extend the Hibernate. This could enhance the Hibernate completely and make it better. This was not implemented as an option due to the time frame that was given.

## SQL Database Examples

SELECT \* FROM HA07.CUSTOM... X

Max. rows: 100 | Fetched Rows: 3 | Matching Rows:

#	ID	ADDRESS	BANKACCOUNT	EMAIL	FIRSTNAME	LASTNAME	PASSPORTID	PHONENUMBER	ACCOUNTLINK_ACCOUNTID
1		1 Greenwich	BA_JointAccount	usman.b@live.co.uk	Usman	Basharat	1	1	1
2		111 Greenwich	BA_JointAccount	usman.b@live.co.uk	Salman	Basharat	1	1	<NULL>
3		112 Greenwich	BA_CheckingAccount	usman.b@live.co.uk	Usman	Basharat	1	1	111

SELECT \* FROM HA07.ACCOUN... X

Max. rows: 100 | Fetched Rows: 2 | Matching Rows:

#	ACCOUNTID	IBAN	ACCOUNTNUMBER	BALANCE	BRANCHNAME	SORTCODE	BANKACCOUNTTYPE
1		1 GW015	61793123		27.0 Greenwich		299151 BA,CheckingAccount
2		111 GW015	29313423		0.0 Greenwich		209821 BA,CheckingAccount

TRANSACTIONS

- TRANSACTIONID
  - D1
  - D2
  - D3
  - D4
- ACCOUNTNUMBER
- BALANCE
- DATEDAY
- DATEMONTH
- DATEYEAR
- DUE
- PAYMENTTYPE
- SORTCODE
- Indexes
- Foreign Keys
- Views
- Procedures
- Metadata

SELECT \* FROM HA07.TRANS... X

Max. rows: 100 | Fetched Rows: 6 | Matching Rows:

#	ACCOUNTNUMBER	BALANCE	DATEDAY	DATEMONTH	DATEYEAR	DUE	PAYMENTTYPE	SORTCODE
1	36774752		10.0 27	11	2018		10.0 DEPOSIT	127472
2	61793123		28.0 12	12	2018		1.0 TOP UP	299151
3	61793123		29.0 12	12	2018		1.0 DEPOSIT	299151
4	61793123		28.0 12	12	2018		1.0 TOP UP	299151
5	61793123		11.0 12	12	2018		1.0 TRANSFER	299151
6	95596533		27.0 12	12	2018		1.0 TRANSFER	597379

## Conclusion

In conclusion, I feel that this course has given me the extra improvement from the start to the end. I feel that I will continue to improve and use this course as a base of the principles of programming. I always feel that the current prototype will have enhancements to be made. However, these improvements can be learnt from and made better for the future.

## References

Aswani (2014). *Exploring spring framework advantages and disadvantages*. [Online] One Stop Blog. Available at: <http://www.aksindiblog.com/spring-framework-advantages-disadvantages.html> [Accessed 16 Dec. 2018].

Guo, F., Liu, J. and Luo, X., 2012. A Method of Creating and Accessing Dynamical Database Table Based on Hibernate Framework. In *Frontiers in Computer Education* (pp. 777-783). Springer, Berlin, Heidelberg.

Kiwi, S. (2017). *Is There Such a Thing as Bug-free Software? – Hacker Noon*. [online] Hacker Noon. Available at: <https://hackernoon.com/is-there-such-a-thing-as-bug-free-software-320cd862af17> [Accessed 17 Dec. 2018].

Pankaj (2018). *Java Singleton Design Pattern Example Best Practices - JournalDev*. [Online] JournalDev. Available at: <https://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples> [Accessed 16 Dec. 2018].

Shubham (2018). *MVC Design Pattern - JournalDev*. [Online] JournalDev. Available at: <https://www.journaldev.com/16974/mvc-design-pattern> [Accessed 16 Dec. 2018].

Singh, C. (2018). *OOP Concepts*. [Online] Beginners Book. Available at: <https://beginnersbook.com/2013/05/encapsulation-in-java/> [Accessed 16 Dec. 2018].

Tutorialspoint (2018). *JUnit Test Framework*. [Online] Available at: [https://www.tutorialspoint.com/junit/junit\\_test\\_framework.htm](https://www.tutorialspoint.com/junit/junit_test_framework.htm) [Accessed 16 Dec. 2018].

Vasavi, B., 2011. Hibernate Technology for an efficient business application extension. *Journal of Global Research in Computer Science*, 2(6), pp.118-125.