Contract for Group E - Banking system

Design based contract

Created by:

Created by:

Kasper Karstensen

Nicklas Hemmingsen

Mathias Valling

Mads Heckmann

E-mail address

nhh777@gmail.com

Table of contents

1. Introduction

The design based contract is based on an application for a newly formed bank, which needs a system to manage the clients' accounts.

The application should be accessed as a local Java client. The application is meant to be a simplified system to manage the clients of a bank and their accounts. The user of the system is a bank teller or another employee of the bank who has the rights to create accounts and transfer money between them.

The use case model describes the functionality needed for the system and the actors of the functionality. For the implementation of a use case we have made a fully dressed use case which in detail describes all the needed information about the particular transaction.

2. Vision

It should be possible for the user of the application to create new clients and accounts, and manage currency transactions, i.e. making withdrawals, deposits and money transfers between accounts. The system must implement the specified rules for transactions, which are a maximum withdrawal of 1000 DKK on an account which has a positive balance, that means that no accounts can have more debt than -999 DKK.

A client is able to have several accounts, and all transactions should be registered on their respective accounts, so the bank employee can see a brief history of transactions on a certain account.

3. Use case model

3.1 Actor descriptions

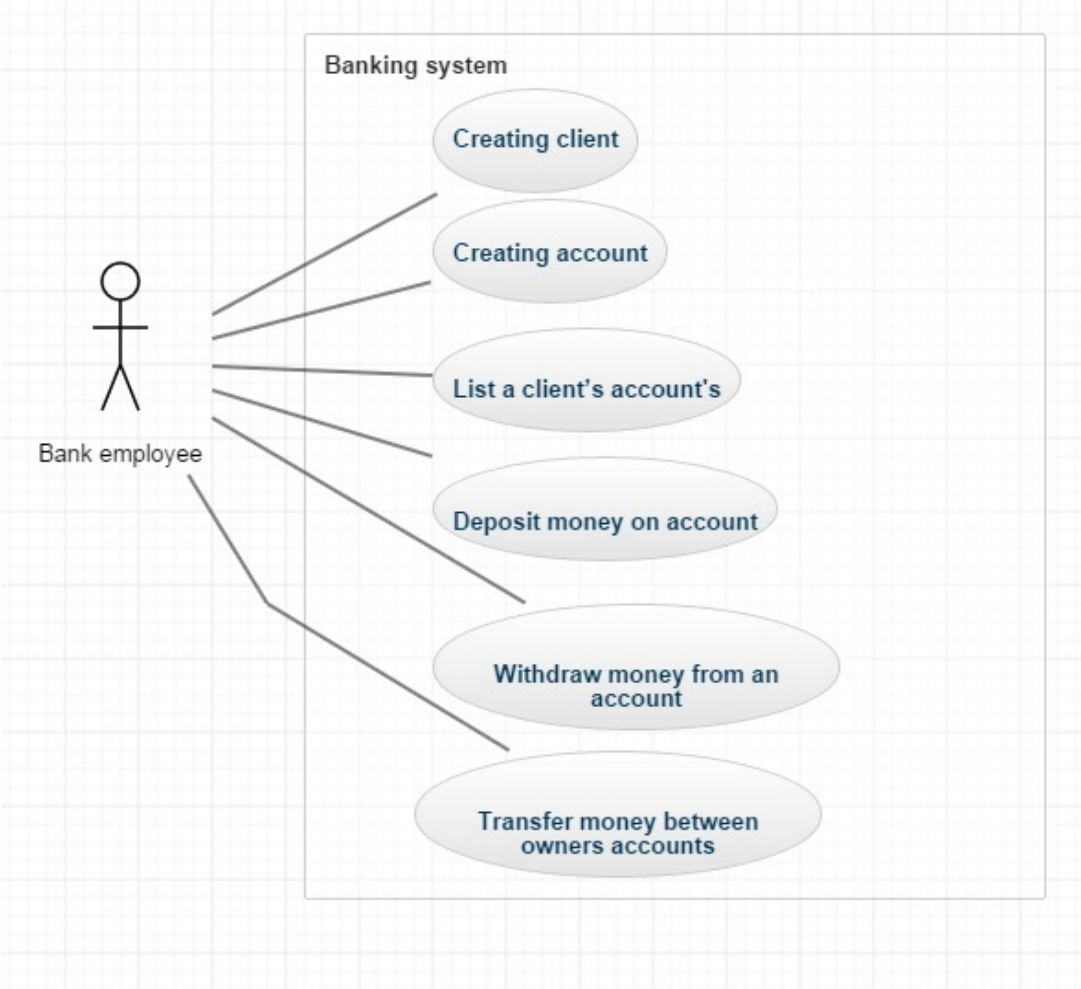
Primary actor

Bank employee
A bank employee handles transaction between information regarding customers and the banking system.

Stakeholder

Customer
A customer is a person who has or wants an account in the bank.

Use-case diagram



3.2 Brief use-case descriptions

UC 1: Creating client

- Actor: Bank Employee
- Brief description: Employee creates a new client in the banking system.

UC 2: Creating account

- Actor: Bank Employee
- Brief description: Employee create a new account for an existing client in the banking system.

UC 3: List a client's accounts

- Actor: Bank Employee
- Brief description: The employee get a list of a clients accounts from the banking system.

UC 4: Deposit money to account

- Actor: Bank Employee
- Brief description: The employee make a deposit for a customer on a given account

UC 5: Withdraw money from an account

- Actor: Bank Employee
- Brief description: The employee withdraw money on a given account for a customer

UC 6: Transfer money between owners accounts

- Actor: Bank Employee
- Brief description: The employee transfer money from on account to another account on the same clint

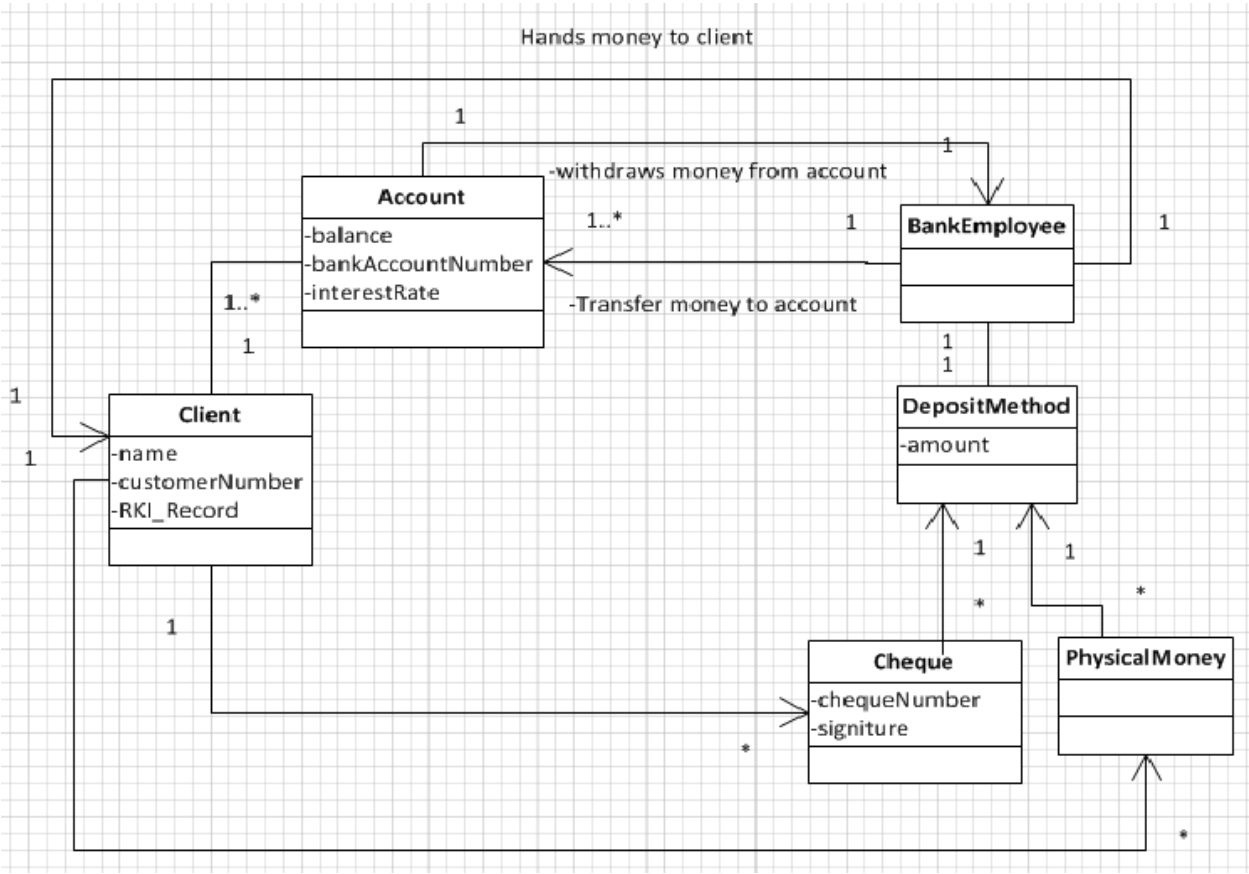
3.3 Fully dressed use-case

UC:5 Withdraw money from an account

Use case section	**Comment**
Use case name	Withdraw money from an account
Scope	System
Level	User
Primary Actor	Employee
Stakeholder and interests	Customer
Preconditions	Customer has a client and a account with a positive balance.
Main success scenario	1. Customer provides accountnumber, and a amount he/she want to withdraw. 2. Employee inputs given information in the banking system. 3. The bank system check if the account balance is greater than 0. 4. The system withdraw the amount from the account. 5. Employee gets the money from the banking system. 6. Customer gets the money from the employee.
Postconditions	The account is updated with a new balance.
Extensions	3a. Employee withdraw more than 1000: 1. System make an error msg.

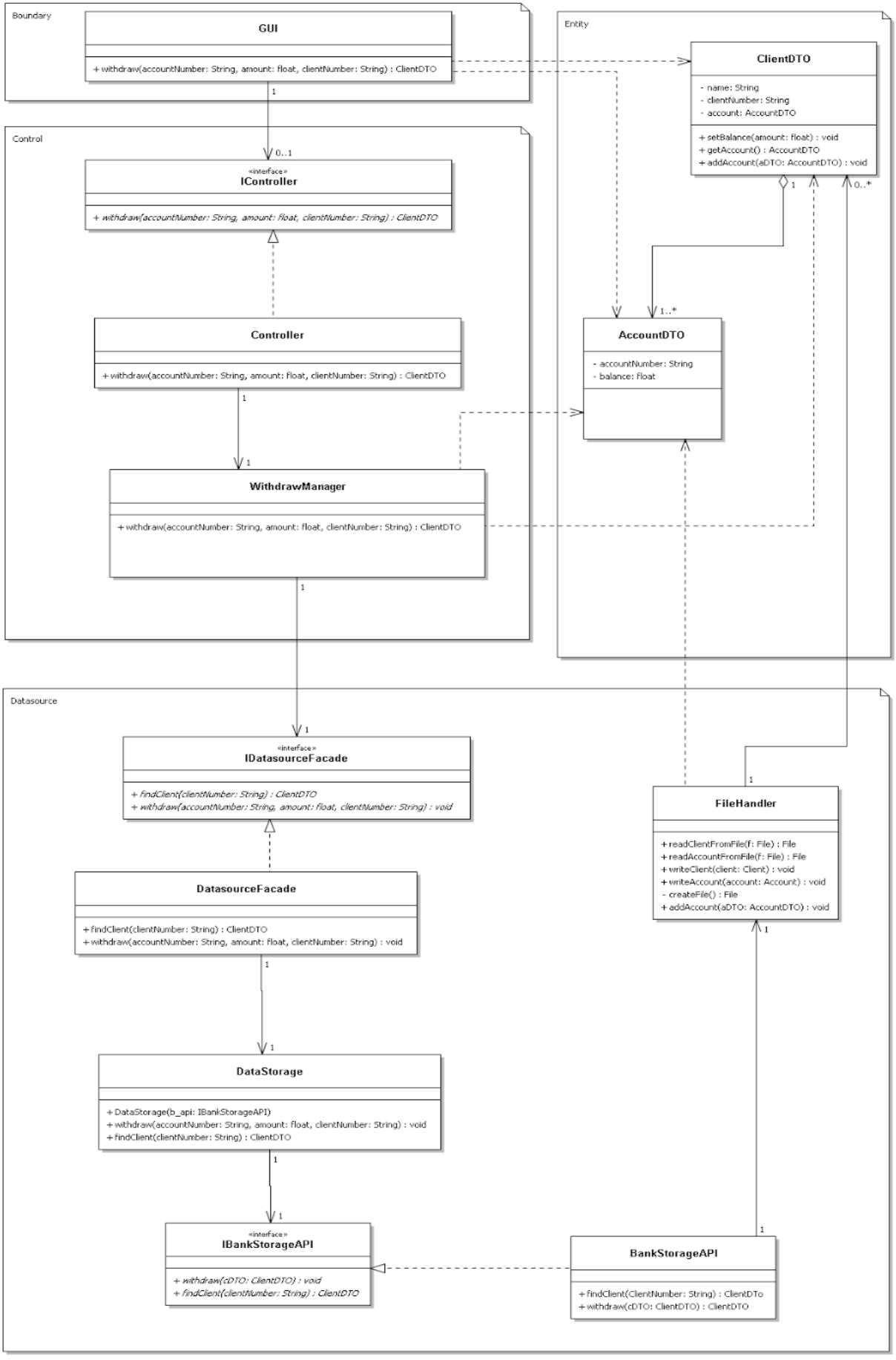
	2. Employee asks customer of a new amount less than a 1000. 3. Employee withdraw the new amount. 4. Return to step 4 in Main Success Scenario. 4a. System error while withdrawing money: 1. The system make a rollback 2. The system give a error msg 3. Return to step 2 in Main Success Scenario.
Special Requirements	● The customer is identified by an accountnumber. ● The transaction has an identified ID.

4. Domain Model



In this domain model we have a BankEmployee that act like a controller since he/she is the only thing that interactor with all other components. it is important to understand that in this system the client is a physical person and that we handle physical money in the system. DepositMethod has a one to many relation with both Cheque and PhysicalMoney since it is accepted that a client want to deposit more than one physicalMoney or Cheque. the Client hold a one to many relation with cheque and physicalMoney as well, and with Account since the client may have more than one account and the client can have more then one cheque/physicalmoney he/she want to deposit. The bank employee can only handle one depositmethod at a time but the employee many handle more than one account if needed.

5. Design Class Diagram


Boundary
GUI
+ withdraw(accountNumber: String, amount: float, clientNumber: String) : ClientDTO
1
Control
0..1
«interface»
IController
+ withdraw(accountNumber: String, amount: float, clientNumber: String) : ClientDTO
Controller
+ withdraw(accountNumber: String, amount: float, clientNumber: String) : ClientDTO
1
1
WithdrawManager
+ withdraw(accountNumber: String, amount: float, clientNumber: String) : ClientDTO
1
Entity
ClientDTO
- name: String
- clientNumber: String
- account: AccountDTO
+ setBalance(amount: float) : void
+ getAccount() : AccountDTO
+ addAccount(aDTO: AccountDTO) : void
1
0..*
1..*
AccountDTO
- accountNumber: String
- balance: float
Datasource
1
«interface»
IDatasourceFacade
+ findClient(clientNumber: String) : ClientDTO
+ withdraw(accountNumber: String, amount: float, clientNumber: String) : void
DatasourceFacade
+ findClient(clientNumber: String) : ClientDTO
+ withdraw(accountNumber: String, amount: float, clientNumber: String) : void
1
1
DataStorage
+ DataStorage(b_api: IBankStorageAPI)
+ withdraw(accountNumber: String, amount: float, clientNumber: String) : void
+ findClient(clientNumber: String) : ClientDTO
1
1
«interface»
IBankStorageAPI
+ withdraw(cDTO: ClientDTO) : void
+ findClient(clientNumber: String) : ClientDTO
BankStorageAPI
+ findClient(ClientNumber: String) : ClientDTo
+ withdraw(cDTO: ClientDTO) : ClientDTO
1
FileHandler
+ readClientFromFile(f: File) : File
+ readAccountFromFile(f: File) : File
+ writeClient(client: Client) : void
+ writeAccount(account: Account) : void
- createFile() : File
+ addAccount(aDTO: AccountDTO) : void
1
1

5.1 Description of Design Class Diagram

We follow a BCE pattern with a added datasource layer to handle the persistence of data. We use interfaces in the program, to give the possibility of change. If we decide later on we want a relational database all we would need to do would be to change out the API with another api that implements the same interface.

5.2 Object Constraint Language

GUI

Invariance: Not defined

withdraw()

Preconditions: User has entered the information

Postconditions: Balance has been updated

Controller

Invariance: Not defined

withdraw()

Preconditions: accountNumber and clientNumber must be a valid String, amount <= 1000 && amount > 0

Postconditions: Balance has been updated

WithdrawManager

Invariance: Not defined

withdraw()

Preconditions: accountNumber and clientNumber must be a valid String, amount <= 1000 && amount > 0

Postconditions: Balance has been updated

DatasourceFacade

Invariance: Not defined

findClient()

Preconditions: The client with the clientNumber must exist in the file

Postconditions: client has been found

FileHandler

Invariance: Not defined

readClientFromFile()

Preconditions: file exist

Postconditions: client found

readAccountFromFile()

Preconditions: file exist

Postconditions: account found

WriteClient()

Preconditions: client exist

Postconditions: client updated in file

ClientDTO

Invariance: accountDTO is an AccountDTO, name must be a String, clientNumber must be a String

setBalance()

Preconditions: amount <= 1000

Postconditions: balance = balance - amount

addAccount()

Preconditions: ClientDTO must exist

PostConditions: Account added to file

AccountDTO

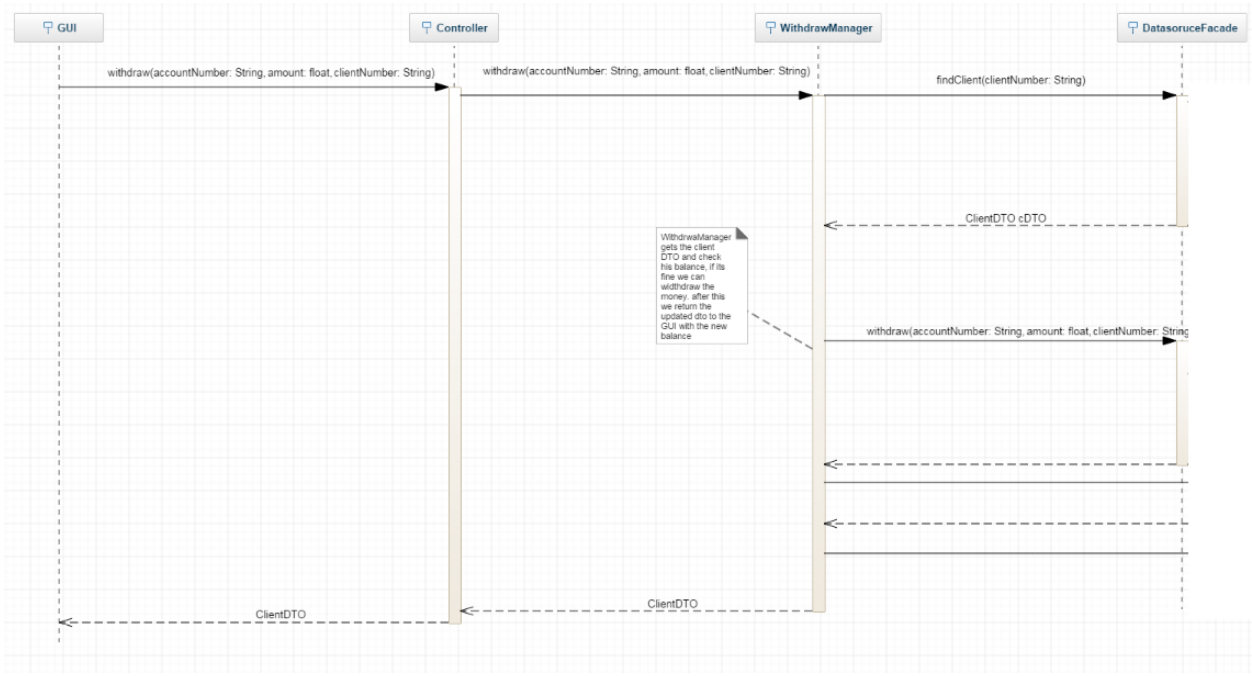
Invariance: balance >= -999

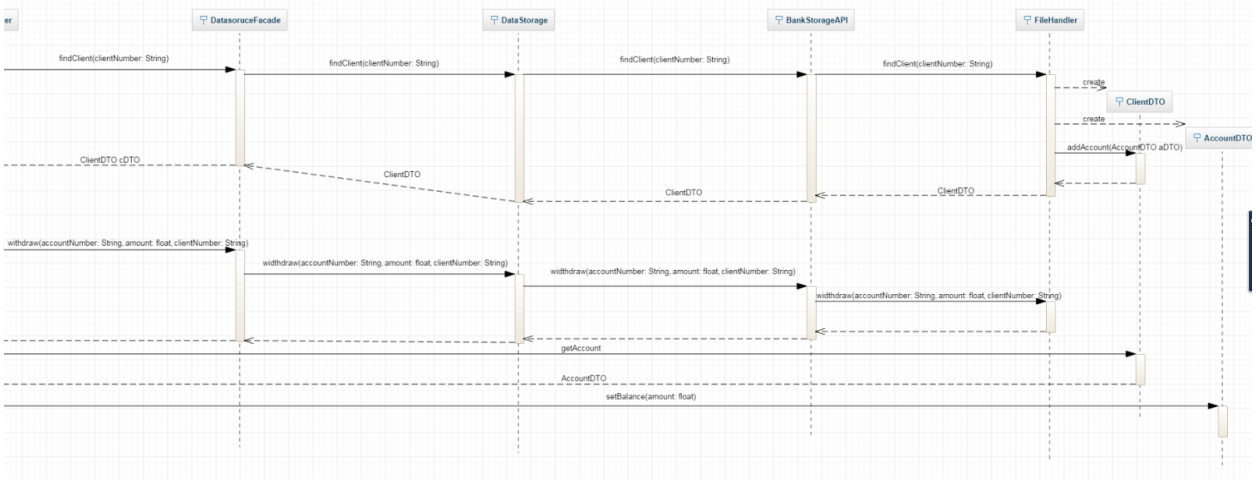
Preconditions: Not defined

Postconditions: Not defined

**Some classes do not show all methods or itself, as it is redundant because the method is just called through the system*

6. Sequence Diagram





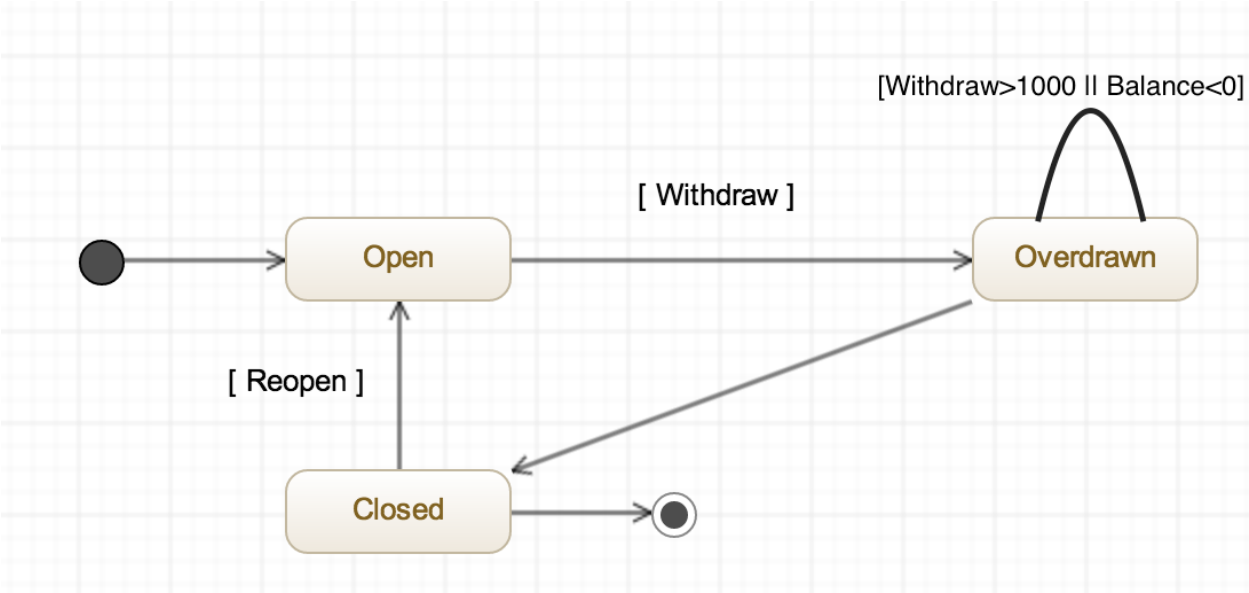
The sequence Diagram show the main success scenario described for UC:5 Withdraw money from an account. The bank employee types in the user information and how much money the client want to withdraw. Then we call the controller which delegate the job to a withdrawManager. The withdrawManager calls the datasoruceFacade to get the stored client and account information. The filehandler reads the information and return a DTO object. The withdraw

manager checks the information and if the balance is positive and the amount is not higher than 1000 dkr then we will withdraw the money by saving the new amount on the users account that will be stored in the file. Here there should had been an if statement in case of an error when doing the check so we return a message to the GUI, but for some reason our UML tool would not allow us to make one. We update the new balance in the clientDTO's accountDTO and send it to the GUI so it can be used to show on a result screen and maybe be printed for the customer. Normaly a clientDTO can have more accounts but to limit the codding a littel a clientDTO will just have one account in this example.

We have decided not to show setters when adding the stored data to the DTO's and getters when the withdrawManager check the balance.

If you want to be able to have a better look at the diagram it can be found at the following link: https://repository.genmymodel.com/kaboka/BankSystemSequenceDiagram/defaultDiagram/_QiNk8VqvEeS18qaWE5MM-A

7. State Diagram



State diagram over the full dressed use-case scenario

8. Contracted Party's work

The contracted party work is to make the fully dressed use-case: UC5, it is important that the code follow the given diagrams. The code made for the withdraw money method has to save any balance in the DTO object before using the withdrawManager to check the balance and store the new data in the file. It is also important to validate user input. This validation needs to check if the balance on an account is positive and that the amount wanted to withdraw is less than a 1000 DKK. Any account in the system may at most have a negative value of 999 DKK. The contracted party must deliver the code for the withdraw functionality of the application.

9. Glossary

Account number - the number used to ID a given account, the number is made of 12 chars.

Client - is the customer.

Cheque - Technically, a cheque is a negotiable instrument instructing a financial institution to pay a specific amount of a specific currency from a specified transactional account held in the drawer's name with that institution.

UC+Number: Use case and which number it is.