Exercise Series 2

Banking example: Banking layer

11 May 2012

The objective of this exercise series is to develop the banking layer (application-independent business logic) of the bank example using the results of series 2 and Java RMI for the interaction between system parts. *All questions below have to be answered explicitly in your report*, with your own words. Pieces of text literally copied from the lectures notes or books do not qualify for marking. The report with answers to the questions below should clearly describe the solutions produced, making it unnecessary for the teachers to dive into the code. Clarity and readability will be important for the marking of the reports.

The results of this exercise have to be submitted to Blackboard as a *zip file* containing the report in PDF format, all the files of the code you have produced (Java classes, JSP files, description files, Ant scripts, etc.) and the WAR files of the web applications. The WAR files should be generated using Ant scripts.

1 Banking example: architecture

In Figure 1 we recall the architecture of the banking system. The banking system has been structured according to the three tiers *presentation*, *business logic* and *resources*. The business logic has been split in two layers (application and banking layers), and the resources tier includes the database access. In the business logic we separated the parts that are present in any banking application, from the parts that are specific for the applications that have to be built. The banking system has two applications, namely *cash dispensing* and *home banking*.

The banking layer comprises a single subsystem for each of the following generic banking functions: transaction processing and authentication. The interfaces supported by these subsystems are given below, together with the interface of the database layer and the InterBank registry. These are all Java RMI interfaces and they have to be implemented exactly like this in order to guarantee interoperability between the groups.

In this exercise you are asked to build the banking layer, more specifically to implement the Transaction processing and Authentication services of your bank, and integrate it with your implementation of the presentation and application specific logic layers. The database layer service should be simulated by local mock up objects, so that the implementations can be tested. The InterBank Registry and the interoperability between banks will be addressed in series 3.

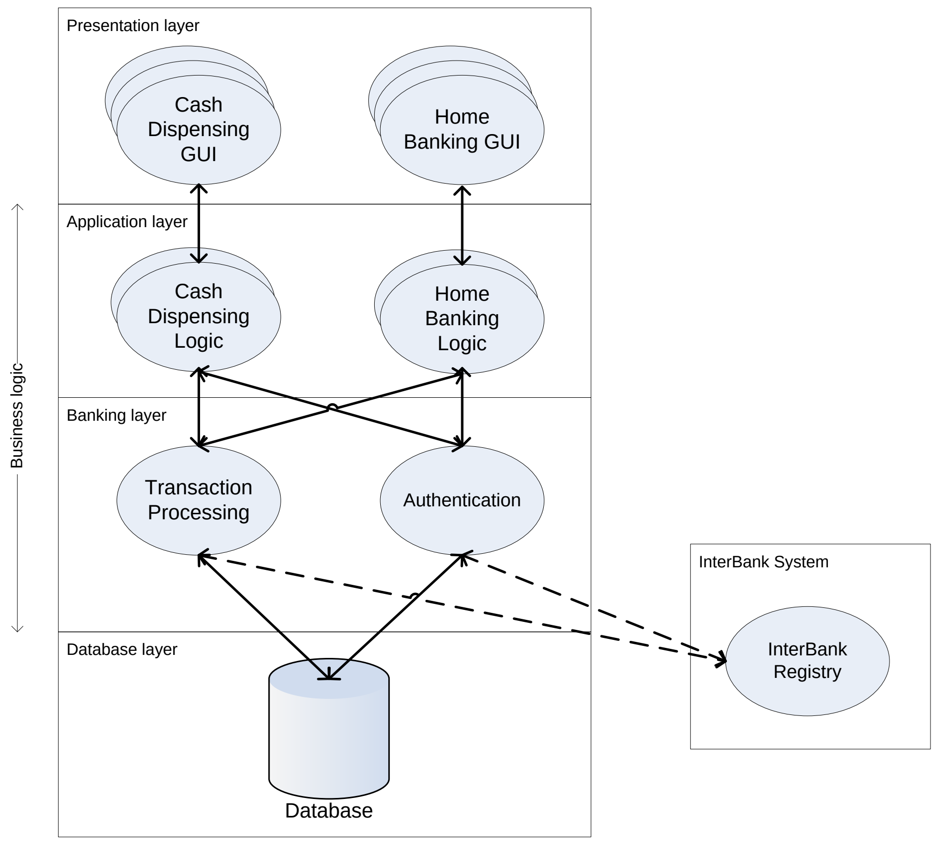


Figure 1. Banking example architecture.

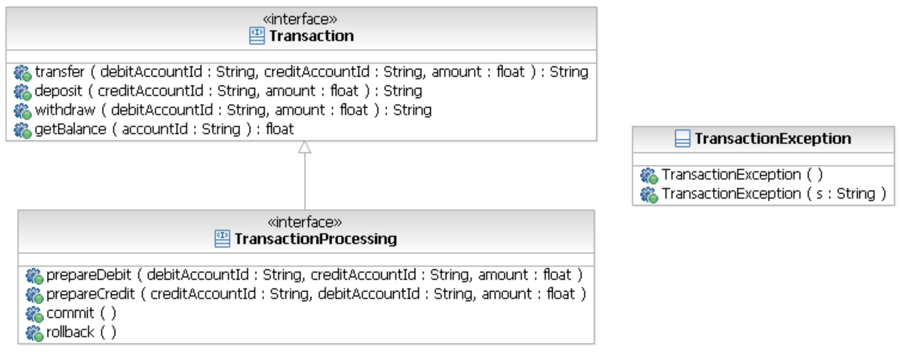
2 Transaction processing

The `TransactionProcessing` subsystem is responsible for transferring money between bank accounts and for retrieving account balance information. This subsystem provides the following interfaces:

– interface `Transaction`, which is provided to the application layer; and

– interface `TransactionProcessing`, which is provided to the transaction processing subsystems of other bank systems. The operations of this interface do not have to be implemented in this series. These operations will be implemented in series 3.

Figure 2 shows both interfaces and their relationship.


```
«interface»
Transaction
transfer ( debitAccountId : String, creditAccountId : String, amount : float ) : String
deposit ( creditAccountId : String, amount : float ) : String
withdraw ( debitAccountId : String, amount : float ) : String
getBalance ( accountId : String ) : float

«interface»
TransactionProcessing
prepareDebit ( debitAccountId : String, creditAccountId : String, amount : float )
prepareCredit ( creditAccountId : String, debitAccountId : String, amount : float )
commit ( )
rollback ( )

TransactionException
TransactionException ( )
TransactionException ( s : String )
```

Figure 2. Transaction processing interfaces

The banking layer supports the two usage scenarios discussed below, namely the delivery of balance information and the transfer of money to another account. The money deposit and withdraw scenarios can be seen as simpler variants of money transfer, and are not discussed here.

Operation `getBalance(accountId: String)` gives the balance of account `accountId`. In this series we only consider accounts of a single bank, but we have prepared the data structures so that interactions between different banks can be supported in the exercise series 3. The structure of the account identifier is discussed in Section 4.

Operation `transfer(d: String, c: String, m : float)` tries to transfer some amount of money `m` from debit account `d` to the credit account `c`. This transfer is performed only if `old.getBalance(d)` ≥ `m`, where `old.getBalance(d)` denotes the balance of account `d` before `transfer()` is executed. Furthermore, the transfer should be performed atomically, such that either the balances of `d` and `c` get values `old.getBalance(d)` - `m` and `old.getBalance(c)` + `m` in case of a successful transaction, respectively, or the balances of `d` and `c` remain unchanged in case of an unsuccessful transaction.

Operation `transfer` delivers `null` in case it is executed successfully, and a String value containing additional information in case the transfer is not successful. Operation `transfer` throws a `TransactionException` in case a failure occurs in the execution of the transfer transaction.

Figure 3 shows the sequence of interactions between subsystems `TransactionProcessing` and `DataAccess` that guarantees that operation `transfer()` has the atomic properties defined above.

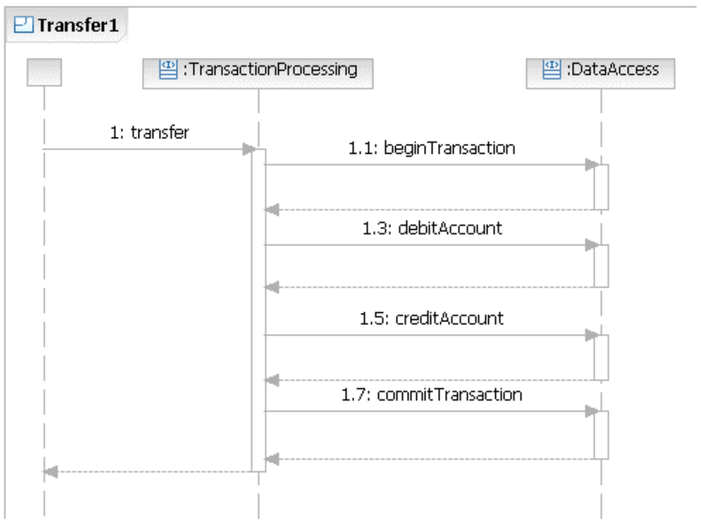


Figure 3. Successful money transfer.

Figure 4 shows the use of a rollback mechanism to undo the modification of the debit account, after the modification of the credit account fails.

The `DataAccess` interface is presented in Section 4.

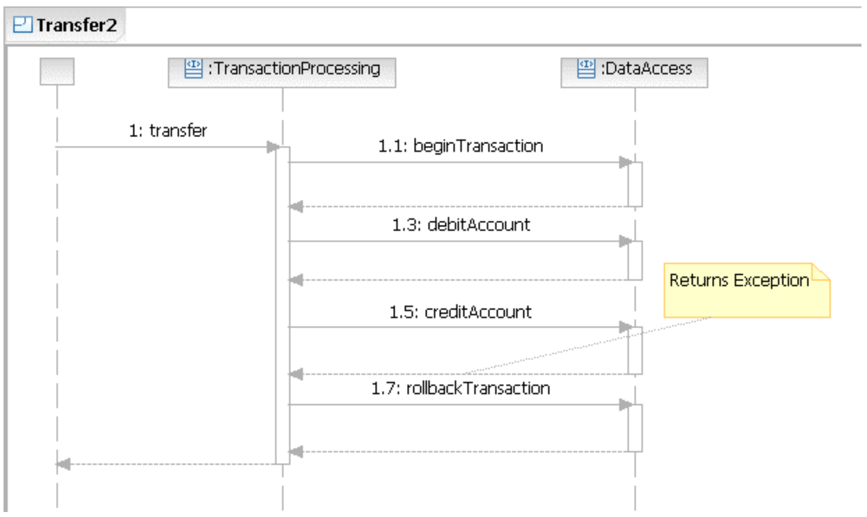


Figure 4. Money transfer failure.

3 Authentication

The `Authentication` subsystem is responsible for checking the authentication of bank application users. Figure 5 shows interface `Authenticator`, which is supported by the `Authentication` subsystem.

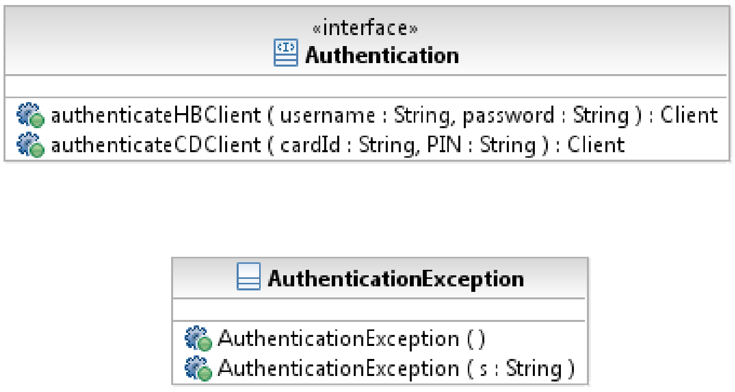

«interface»
Authentication
authenticateHBClient (username : String, password : String) : Client
authenticateCDClient (cardId : String, PIN : String) : Client
AuthenticationException
AuthenticationException ( )
AuthenticationException (s : String)

Figure 5. `Authentication` interface.

Operation `authenticateCard()` authenticates a cash dispensing client through its card, based on a bank card identifier and a PIN code. In case the authentication is successful this operation returns a `Client` object (see Section 4), which contains information about the authenticated client. Otherwise, this operation returns `null`. A PIN code consists of 5 ASCII characters.

Operation `authenticateClient()` authenticates a home banking client based on the client's username and password. In case the authentication is successful this operation returns a `Client` object (see Section 4), which contains information about the authenticated client. Otherwise, this operation returns `null`. A password consists of minimal 6 and maximal 10 ASCII characters.

4 Data access layer

The `DataAccess` subsystem shields the access to the bank's database, hiding the particular way in which data is stored in the bank's database. Figure 6 shows the `DataAccess` interface that is provided by this subsystem to the banking and application layers.

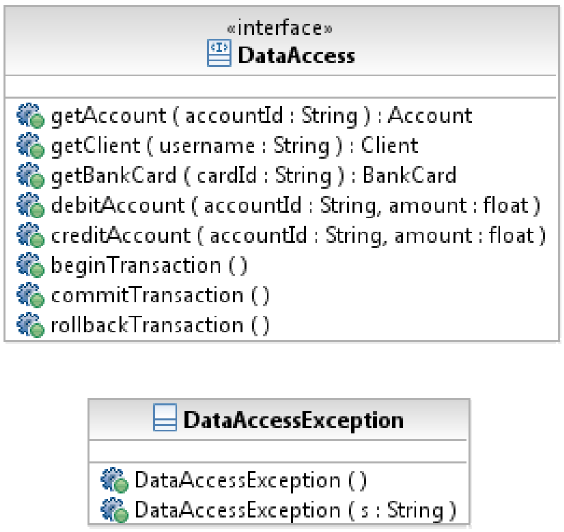

«interface»
DataAccess
getAccount (accountId : String) : Account
getClient (username : String) : Client
getBankCard (cardId : String) : BankCard
debitAccount (accountId : String, amount : float)
creditAccount (accountId : String, amount : float)
beginTransaction ( )
commitTransaction ( )
rollbackTransaction ( )
DataAccessException
DataAccessException ( )
DataAccessException (s : String)

Figure 6. `DataAccess` interface.

The `DataAccess` interface depends on the classes `Client`, `Account` and `BankCard`, which are defined in the class diagram shown in Figure 7. Since the Client class is also used in the Transaction processing and Authentication subsystems, it is absolutely necessary to support this class definition in order to guarantee interoperability between the implementations built by the different groups.

The bank that manages some account can be recognized by the first part of the account identifier. An account identifier consists of two parts: a *bank code*, which uniquely represents the bank that manages the account, and an *account code*, which uniquely identifies the account within this bank. An account identifier is encoded as an ASCII string of length 10, such that the first 3 characters are used for the bank code and the remaining 7 characters are used for the account code.

Each bank implementation built by a group will be represented as the `String` `00X`, where `X` denotes the group number. For example, the implementation of group 1 is identified as `001`, group2 as `002`, etc. An example of an account id is `0030000012`, which is account `0000012` of bank `003` (bank implemented by group 3).

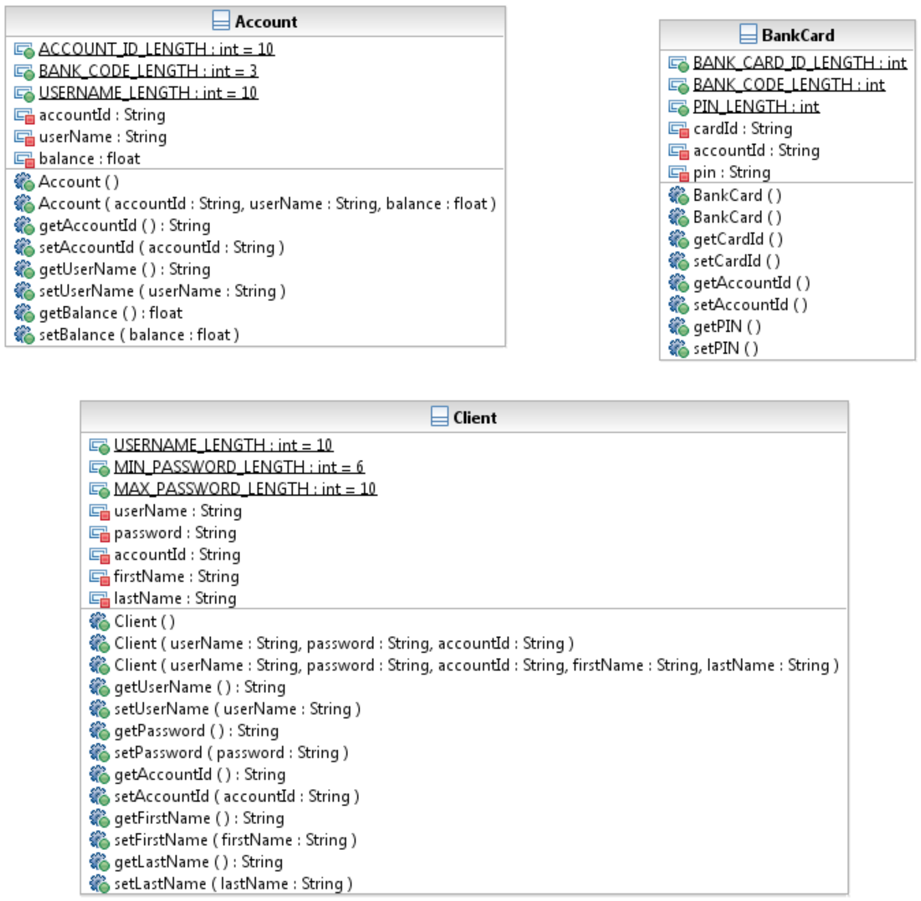

Account
ACCOUNT_ID_LENGTH : int = 10
BANK_CODE_LENGTH : int = 3
USERNAME_LENGTH : int = 10
accountId : String
userName : String
balance : float
Account ( )
Account (accountId : String, userName : String, balance : float)
getAccountId ( ) : String
setAccountId (accountId : String)
getUserName ( ) : String
setUserName (userName : String)
getBalance ( ) : float
setBalance (balance : float)
BankCard
BANK_CARD_ID_LENGTH : int
BANK_CODE_LENGTH : int
PIN_LENGTH : int
cardId : String
accountId : String
pin : String
BankCard ( )
BankCard ( )
getCardId ( )
setCardId ( )
getAccountId ( )
setAccountId ( )
getPIN ( )
setPIN ( )
Client
USERNAME_LENGTH : int = 10
MIN_PASSWORD_LENGTH : int = 6
MAX_PASSWORD_LENGTH : int = 10
userName : String
password : String
accountId : String
firstName : String
lastName : String
Client ( )
Client (userName : String, password : String, accountId : String)
Client (userName : String, password : String, accountId : String, firstName : String, lastName : String)
getUserName ( ) : String
setUserName (userName : String)
getPassword ( ) : String
setPassword (password : String)
getAccountId ( ) : String
setAccountId (accountId : String)
getFirstName ( ) : String
setFirstName (firstName : String)
getLastName ( ) : String
setLastName (lastName : String)

Figure 7. Class diagram for the banking business objects.

Important notice

Google App Engine (GAE) does not support Java RMI, so the students who have deployed their implementation of series 1 in the GAE will not be able to extend it directly to incorporate the banking layer. There are at least to possible solutions to this problem:

1. Port the code of series 1 to an on-premise Tomcat installation (away from the cloud) before starting with series 2.
2. Build the banking layer components with two interfaces, one with Java RMI for the interaction with the other banks, and one with web services (SOAP) for the interaction with the GAE-based front-end.

Students who are comfortable with web services are suggested to try the second solution, while the others should simply apply the first solution.

Question 1
Draw UML class diagrams showing the classes you implemented. Explain the relations among the classes. Explain the sequence of interaction between your software components and where these interactions take place. Add diagrams to indicate how these software components can be assigned to concrete computers (end-user computers and servers). Explain how your code works and how it has been integrated with your code of series 2, especially discussing if modifications were necessary and what you have done to integrate this code. Particularly the implementation of the `deposit()`, `withdraw()` and `transfer()` operations should be properly discussed.

Question 2
Describe the steps you have taken to implement, register and discover RMI remote objects in your system, as well as your system's configuration. How are RMI remote objects registered and discovered in your system? Try to make your system as flexible as possible with respect to the actual computer system in which the subsystems execute and explain what you have done to accomplish this flexibility.

Question 3
Describe the approach you took in order to test your application. How did you test your software components? How did you test the application as a whole? How can you be sure the system works? Describe both the testing architecture (implementation-under test and testing components) and the testing strategy (test cases) that you applied, and the results produced in these tests. Be sure you discuss how you tested the transaction processing scenarios of Section 2 and authentication functionality of Section 3, and how you can be sure your system properly supports these functions.