UNIVERSITEIT TWENTE.

instructors only

Exercise series 3

Banking example:
Interoperability and Databases

Date	:	18-06-2012	
Course	:	Java Middleware Techonology	
Course code	:	192652110	
Group	:	1	
Students	:	Dennis Pallett	0167304
		Mark Wienk	1238531

Table of contents

1. Introduction

In the third exercise series we will be implementing the InterBank connectivity. This means that the banking systems of multiple groups will be working together. It will be able to transfer money to and from other bank's accounts to our bank. The authentication system will also be distributed, meaning everyone can login to our system using their own bank's credentials.

2. Software Architecture

2.1 Database layer

In this iteration of the development of our banking system, the data will come from a database. The database connection is a MySQL connection implemented using the JDBC library and the MySQL/J driver. Our application's architecture hasn't changed, since the design in the previous step already took care of a *DataAccess* layer. The implementation of the database connections therefore consisted of a change in the *DataAccessImpl* class.

Every method now builds a query, a *commitTransaction()* method commits the query to the database, while keeping data integrity. In our implementation we have set the database connection's *autoCommit* value to false, this means the *beginTransaction()* method in the *DataAccess* interface didn't need to be implemented.

2.2 Interbanking system

For the InterBank system, the Interbank interface was added to the project. This interface holds the actions that can be performed on the remote Interbank object (the interbank registry). In order to implement the InterBank functionality, we had to change the classes *TransactionProcessingImpl* (TP) and *AuthenticationImpl*. (Auth) These two classes now first scan the input string. The TP class scans the accountId, which it checks against our bankId, a static field added in the *BankService* class. If the accountId's first 3 digits are equal to our bankId, the *Transaction* class will be called, which handles the transaction request the same way it did in series 2. Figure 1 shows an updated class diagram of our banking layer project.

For the authentication of external clients, we had to implement some way to identify a bank of the client. It should still be able to have the same username on different accounts of different banks. So we decided to identify the bank with a prefix. If an external client wants to authenticate, the username should be prefixed with it's bankId and a dot, like 002.johndoe. The *AuthenticationImpl* class now scans the input string, if it finds a dot the request is forwarded to the identified bank.

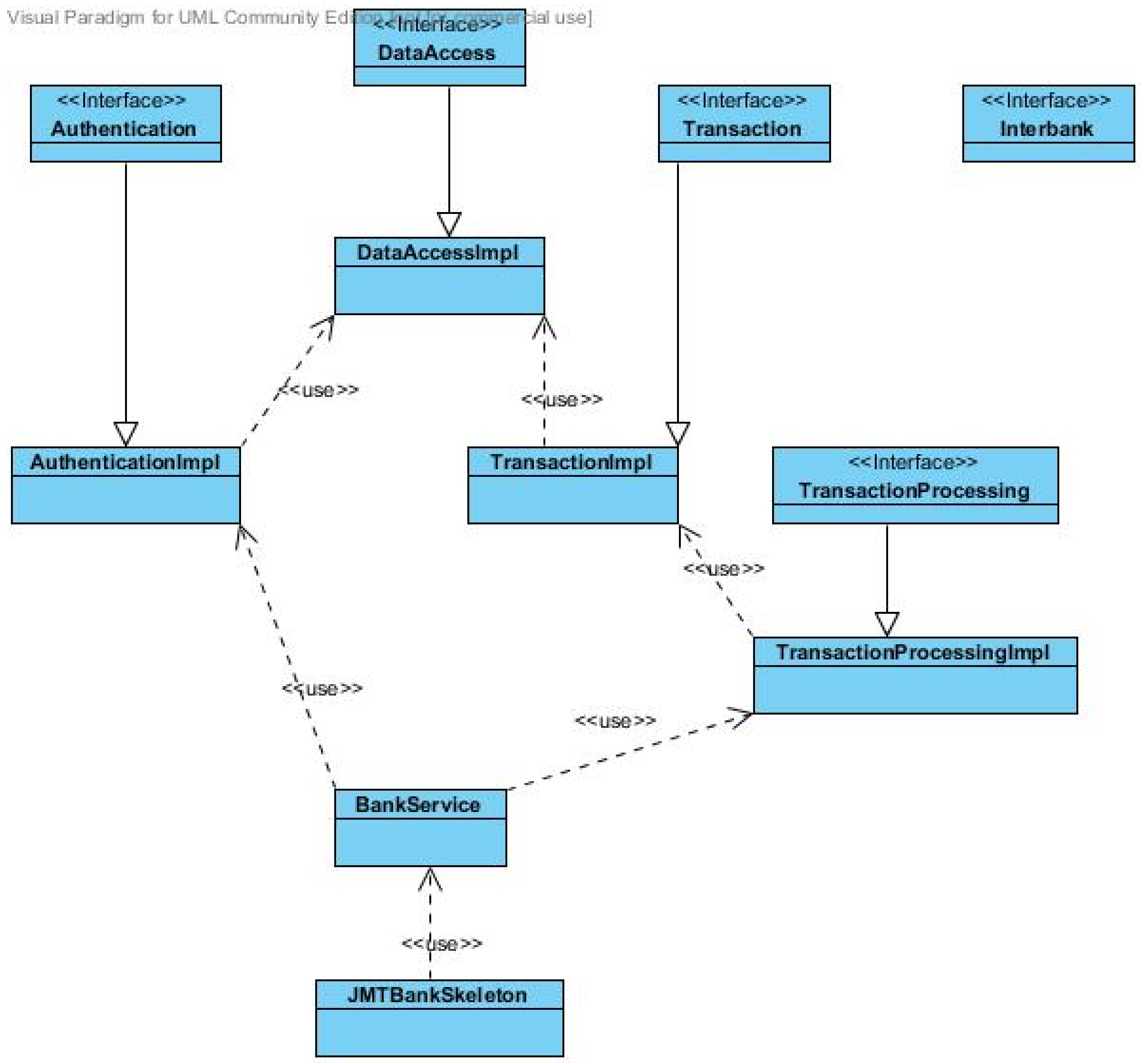
Visual Paradigm for UML Community Edition [not for commercial use]
<<Interface>> DataAccess
<<Interface>> Authentication
<<Interface>> Transaction
<<Interface>> Interbank
DataAccessImpl
<<use>>
<<use>>
AuthenticationImpl
TransactionImpl
<<Interface>> TransactionProcessing
<<use>>
TransactionProcessingImpl
<<use>>
<<use>>
BankService
<<use>>
JMTBankSkeleton

Figure 1: A class diagram of our banking layer

The registries communicate with each other via RMI mechanisms, a schematic representation can be found in Figure 2. In this figure, the InterBank only holds a link to another bank's registry (actually 2 links, one for Transaction and one for Authentication). The registry of the bank itself contains a link to the (remote) object, so it can be used by other banks. In the image the rectangles represent registries and the ovals represent objects.

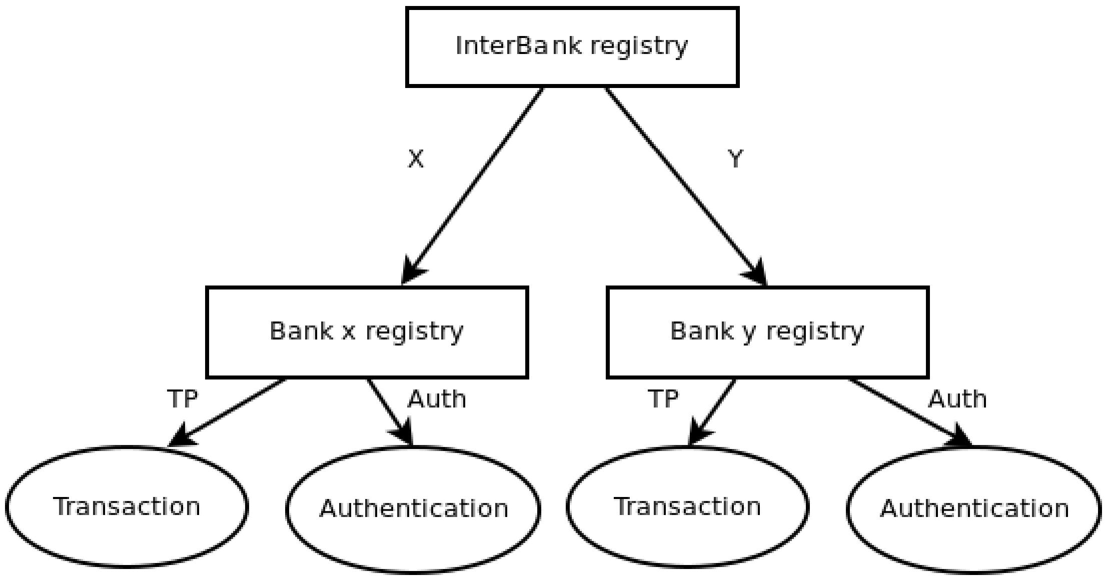

Figure 2: Flow of registries

3. Exception handling

When a data transaction method (*deposit()*, *withdraw()* or *transfer()*) is invoked, all checks are done prior to committing the statement. In case of a money transfer this means:

1. Check if the debit account has enough balance
2. Check if the credit account exists

After step 1 the *debitAccount()* method is already invoked, which prepares a query on the database. This query however is not run completely yet. If step 2 fails, an exception is thrown and caught. In that catch statement, the transaction will be undone by invoking a *rollback()* on the database. If no exception happens, a *commitTransaction()* is issued. This way, only complete transactions can occur (where the amount is credited and debited).

If however the database isn't connected, the *rollback()* statement can throw an exception and that might be unrecoverable. Since this is a separate statement, a 'fix' could be to send the initial values in an e-mail to a bank employee. Another idea is to make a queue for this type of exceptions. When the database is available again, the queue is first emptied before other transactions can be executed.

4. Database management system

We have implemented a database in the MySQL relational database management system. With help of the MySQL workbench we have created a database for our clients, accounts and bankcard (see Figure 3). The client and bankcard tables both have foreign keys to the account. This way, an account can hold multiple clients and multiple bankcards. This data is of course the very minimum of the data that can be stored by the system, but it suffices for this assignment.

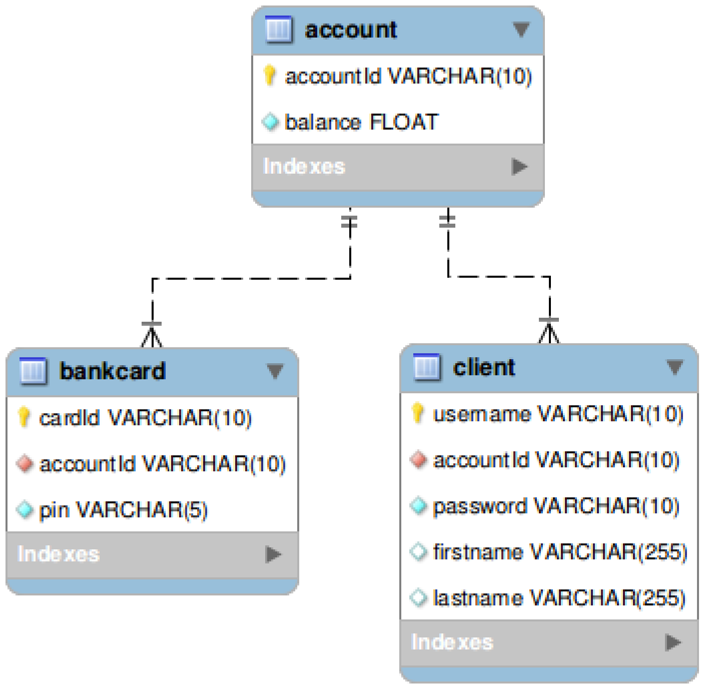

Figure 3: Database model

We connect to this database with the JDBC MySQL/J connector, all connections are handled in the *DataAccess* class, discussed in 2.1.

5. Testing strategy

To test our application the full JUnit test suite created during the previous assignments can be re-used. By running these tests again we can be sure that the application still functions correctly.

The results of running this test suite are the same as in the previous assignments, meaning all tests passed and the application functions as expected. This also implies we have correctly implemented the database connectivity since the *DataAccess* implementation, which is indirectly tested with the unit tests, now uses the database as a storage engine instead of in-memory.

Testing the InterBanking system is much more difficult since it requires multiple banks on different computers. The only way we can think of to test this is to manually run multiple banks on different computers and run through a list of test cases, similar to the test cases from the previous assignments. We do not see any practical automatic way of testing the InterBank functionality.

The test cases that we have done to test the InterBank system are listed in table 1 together with the results.

Nr.	Use-case	Expected outcome	Actual outcome
1	Enter an invalid username from an external bank in the homebanking login	- The user is not logged in - An error message is shown to the user	As expected
2	Enter a valid username and an invalid password from an external bank in the homebanking login	- The user is not logged in - An error message is shown to the user	As expected
3	Enter a valid username and a valid password from an external bank in the homebanking login	- The user is logged in - The user is redirected to the homebanking application index	As expected
4	Click the “Check account balance” link	- The current balance of the user is shown (via AJAX)	As expected
5	Money is transferred to an account on another bank	- The balance of the user decreases by the entered amount - The balance of the other account on the other bank increases by the entered amount - A message is shown indicating the success of the transfer	As expected
6	Money is received from an account on another bank	- The balance of the user increases by the right amount	As expected
7	Invalid values are entered into the transfer money form (i.e. incorrect account number)	- The balance of the user does not decrease - An error message is shown to the user	As expected

Table 1. The use-cases for black-box testing and the results of the InterBank system

During testing of the InterBank functionality many problems and errors were related to the security manager. Eventually we had to resort to practically disabling the security manager (by using broad wild-card permissions) to be able to get the InterBank system working. It’s possible to limit the security manager by selectively applying more restrictive rules until the system stops working again, however we have not done so in our system. The operating system's firewall also gave some problems, after disabling it, these problems were solved, but the computer does have a security risk. Here again should restrictive rules be applied selectively.