Übung 1: Socket Bank

Die Technologien, mit welchen verteilte Systeme gebaut werden können, werden im Kontext einer kleinen Bank-Applikation illustriert, welche an Vorgänge in einer Bank angelehnt ist. Kunden können neue Konten bei einer Bank-Instanz eröffnen. Jedes Konto hat eine Kontonummer und einen Kontoinhaber. Auf Konten können Geldbeträge einbezahlt und abgehoben werden, das Konto darf jedoch nicht überzogen werden. Zusätzlich können Geldbeträge von einem Konto auf ein anderes transferiert werden. Ein Konto darf nur gelöscht werden, wenn es saldiert worden ist, d.h. wenn der Saldo 0.00 ist. Wird ein Konto gelöscht, so wird es als passiv markiert (ein Klient könnte ja noch eine Referenz darauf besitzen). Auf passiven Konten dürfen keine Transaktionen mehr ausgeführt werden. Nummer, Name und Saldo kann jedoch auch auf einem passiven Konto abgefragt werden (der Saldo ist dabei immer 0.00).

Damit Sie sich voll auf den Aspekt Verteilung konzentrieren können, haben wir Ihnen ein Benutzerinterface bereitgestellt. Dieses Programm verwendet folgende Interfaces um auf die Funktionalitäten der Bank zuzugreifen (alle Methoden deklarieren zusätzlich die Ausnahme `java.io.IOException`, diese wird geworfen, wenn bei der Kommunikation Probleme auftreten).

```
package bank;

public interface Bank {
    String createAccount(String owner);          // creates a new account and returns the account number
    boolean closeAccount(String number);         // closes the specified account and returns success of closing
                                                 //          and passivates the account
    Set<String> getAccountNumbers();             // returns a set of the account numbers of all currently active accounts
    Account getAccount(String number);           // returns the account with the given number

    void transfer(Account a, Account b, double amount)
                    throws IllegalArgumentException , InactiveException, OverdrawException;
}

public interface Account {
    String getNumber();                          // returns the account number
    String getOwner();                           // returns the account owner
    boolean isActive();                          // returns the state of the account.

    void deposit (double amount) throws IllegalArgumentException, InactiveException;
    void withdraw(double amount) throws IllegalArgumentException , InactiveException, OverdrawException;
    double getBalance();
}

public class OverdrawException extends Exception {
    public OverdrawException (String reason) { super(reason); }
}

public class InactiveException extends Exception {
    public InactiveException (String reason) { super(reason); }
}
```

Um auf die Implementierung der Bank zugreifen zu können wird das Interface `BankDriver` definiert. Das Klientenprogramm verwendet dieses Interface, um die Verbindung zum Server auf- bzw. abzubauen.

```
package bank;

public interface BankDriver {
    void connect(String[] args);                 // establish connection to the (remote) bank
    void disconnect();                           // close connection to the (remote) bank
    Bank getBank();
}
```

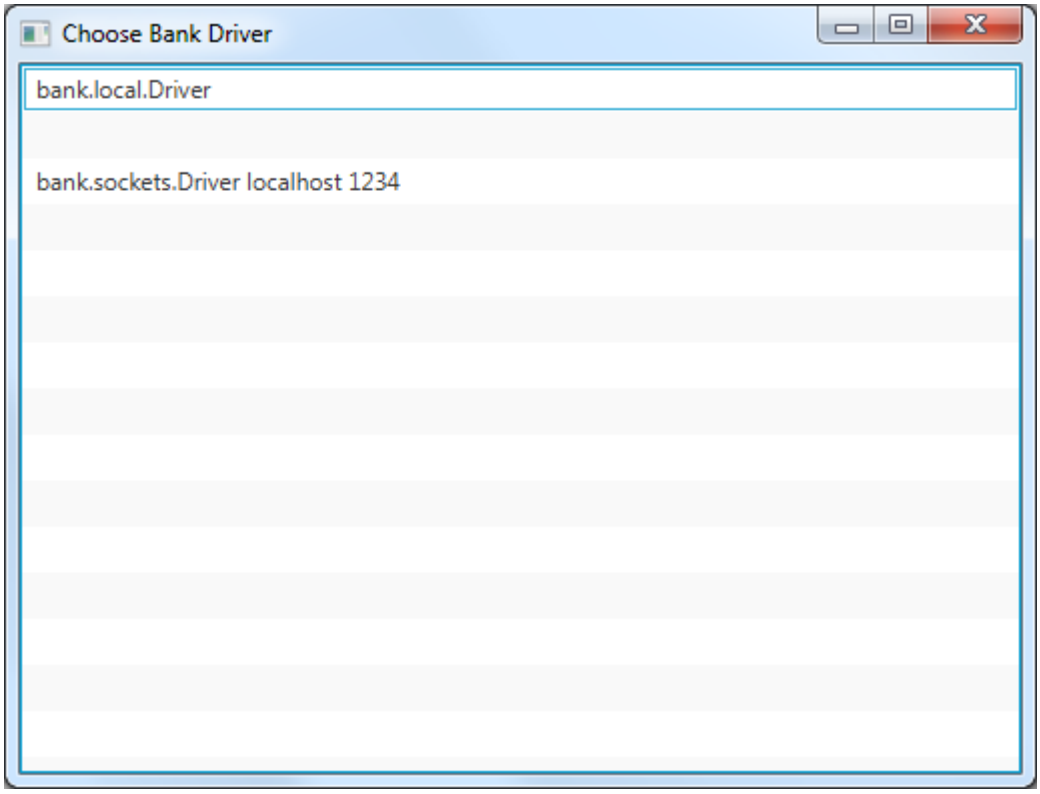
Beim Start des Klientenprogramms muss als erster Parameter eine Klasse angegeben werden, welche das Interface `BankDriver` implementiert. Das Klientenprogramm lädt dann diese Klasse und ruft über dieses Objekt die Methode `connect` auf und übergibt als Parameter die restlichen Kommandozeilenargumente als String-Array. Argumente wie Rechner- oder Portnummern lassen sich so der Implementierung des Server-Proxies übergeben.

Im Beispiel

```
java bank.gui.Client bank.sockets.Driver localhost 1234
```

lädt das Klientenprogramm die Klasse `bank.sockets.Driver` und ruft in dieser Klasse die Methode `connect` auf und übergibt als Parameter ein String-Array der Länge 2 mit den beiden Elementen "localhost" und "1234".

In den folgenden Übungen wird die Kommunikation über Rechnergrenzen mit jeweils anderen Technologien realisiert werden, d.h. Sie werden in jeder Übung eine neue Klasse implementieren, welche das Interface `BankDriver` implementiert. Da es etwas umständlich ist, die Laufzeitargumente die beim Start übergeben werden jeweils zu ändern, haben wir einen „Konfiguration-Choser“ bereitgestellt. Die Laufzeitargumente können in der Text-Datei `src/main/resources/Servers.txt` bereitgestellt werden. Wenn das Klientenprogramm `bank.gui.Client` *ohne* Argumente gestartet wird, dann kann die Konfiguration aus den in der Datei Servers.txt bereitgestellten Konfigurationen gewählt werden.



Per Doppelklick kann dann das Bank-GUI mit der entsprechenden Konfiguration gestartet werden.

Dieses Programm kann auch aus der Konsole mit dem Befehl

```
gradlew run
```

gestartet werden, oder mit

```
gradlew run --args="bank.sockets.Driver localhost 1234"
```

falls man einen speziellen Bank-Klienten starten möchte.

Aufgabe a)

Bevor wir eine *verteilte* Bank implementieren können müssen wir die Funktionalität der Bank *lokal* implementieren. Eine lokale Implementierung kann mit einer einfachen BankDriver - Factory getestet werden. Die Methode `getBank` dieser Factory gibt eine Instanz Ihrer Implementierung der Bank zurück, welche in der Methode `connect` erzeugt worden ist und welche in der Methode `disconnect` wieder auf null zurückgesetzt wird. Dies Klasse `bank.local.Driver` ist gegeben.

```
package bank.local;
public class Driver implements bank.BankDriver {
	private bank.Bank bank = null;

	public void connect(String[] args) {
		bank = new Bank();
		System.out.println("connected...");
	}
	public void disconnect() {
		bank = null;
		System.out.println("disconnected...");
	}
	public bank.Bank getBank() {
		return bank;
	}

	public static class Bank implements bank.Bank {
		…
	}
}
```

Für die Verwaltung der Konto-Objekte in der Klasse Bank empfehlen wir eine `Map<String, Account>` zu verwenden, in welcher die Konten mit ihrer Kontonummer assoziiert werden. Die Klassen `Driver.Bank` und `Driver.Account` enthalten `TODO` Annotationen die implementiert werden müssen. Diese Implementierung der Bankfunktionalität kann in *allen* folgenden Bank-Übungen verwendet werden (referenziert werden, nicht kopiert werden).

Im Klientenprogramm steht ein Menu *"Test->Functionality Test"* zur Verfügung, mit welchem die Funktionalität der Bank getestet werden kann. Dieser Test ist aktiv, sobald ein Konto erzeugt worden ist. Konten können z.B. mit dem Befehl „*File->Create Accounts*“ erzeugt werden. Mit dem Befehl „File->CreateAccounts“ werden fünf Test-Konten erzeugt.

Die Dateien zur Bank stehen auf dem AD in der Datei `Uebungen\Uebung01\bank-assignment.zip` zur Verfügung.

Aufgabe b)

In dieser Übung soll die lokale Bank aus Aufgabe a) zu einer verteilten Lösung erweitert werden. Die Kontodaten soll dabei auf einem Host verwaltet werden, und die Klienten sollen die Anforderungen über eine Socket-Stream-Verbindung an den Server übertragen und die Resultate über dieselbe Verbindung lesen.

Gegenüber der Lösung von Aufgabe a) muss in dieser Aufgabe neu eine unabhängige Server-Applikation entwickelt werden. Diese soll, sobald sie auf dem Server gestartet wird, einen Internet-Socket auf einem bestimmten Port (z.B. 6789) eröffnen und soll warten, bis ein Klient eine Verbindung zu diesem Server aufbaut. Sobald eine Verbindung aufgebaut ist, soll der Server einen separaten Thread starten, der alle Anfragen von diesem Klienten liest und verarbeitet. Auf den Bankserver kann somit gleichzeitig von mehreren Klienten her zugegriffen werden. Während der Entwicklung kann der Server auch auf derselben Maschine gestartet werden, auf der auch der Klient läuft.

Da das GUI nur über Schnittstellen mit Ihrer Implementierung kommuniziert, müssen auch auf der Klientenseite Implementierungen für die Interfaces `Bank` und `Account` bereitgestellt werden! Diese Klassen schicken dann jeweils die Argumente bei Methodenaufrufen über die Socketverbindung an den Server. Die Verbindung zum Server wird aufgebaut, sobald vom Klienten-GUI her die Methode `Driver.connect` aufgerufen wird. Kommandozeilenargumente, die beim Start des Klienten angegeben werden, werden (bis auf das erste) an die Methode `connect` weitergereicht. Startet man das GUI mit dem Kommando

```
java bank.gui.Client bank.sockets.Driver localhost 6789
```

so wird die Klasse `bank.sockets.Driver` geladen und über eine Instanz dieser Klasse die Methode `connect` aufgerufen und als Parameter der String-Array `{"localhost", "6789"}` übergeben. So kann beim Start des Klienten festgelegt werden, auf welcher Maschine der Bankserver läuft. Die Interpretation der Kommandozeilenargumente geschieht in Ihrer Implementierung der Schnittstelle `bank.BankDriver`.

Die Kommunikation mit dem Server kann auf verschiedene Arten gelöst werden. Entweder Sie verschicken die Funktionsart (z.B. als Integer) und die Argumente einzeln über die Socketverbindung, oder Sie verpacken die Argumente in ein (Funktions)-Objekt und schicken dieses Objekt über einen Object-Stream an den Server.

Beachten Sie, dass die Ausnahmen, die der Server erzeugen kann, auch irgendwie über die Socketverbindung an den Klienten übertragen werden müssen.

Test:

Testen Sie Ihr Programm, indem Sie *zwei* Klienten starten, die sich mit demselben Server verbinden und prüfen Sie, ob die Änderungen, die Sie über den einen Klienten vornehmen, auch über den zweiten Klienten (nach einem Refresh) sichtbar sind. Prüfen Sie was passiert, wenn Sie ein Konto über einen Klienten löschen und danach über den anderen Klienten (ohne ein Refresh gemacht zu haben) eine Einzahlung auf diesem Konto vornehmen.

Geben Sie neben dem Code eine kurze Beschreibung Ihrer Lösung ab. Daraus soll hervorgehen, wie Sie die Kommunikation mit dem Server realisiert haben, und soll mir den Einstieg in Ihre Lösung erleichtern.

Wer die besondere Herausforderung sucht der kann

a) den Server in einer anderen Programmiersprache programmieren. Es bietet sich dazu C/C++, C#, Kotlin oder Go an.
b) anstelle der verbindungsorientierten Sockets auch Datagram-Sockets verwenden. Die Schwierigkeit dabei ist, verlorene Pakete zu erkennen und richtig zu behandeln. Pakete können dabei sowohl auf dem Weg vom Klienten zum Server, als auch umgekehrt vom Server an den Klienten verloren gehen. Trotzdem sollte sichergestellt sein, dass z.B. Beträge nicht mehrfach abgehoben oder gutgeschrieben werden.

Abgabe: 04. März 2020