Übung 1: Socket Implementierung

# Grundidee

In der ersten Übung geht es darum, für die lokale Implementierung der Bank ein Socket-Frontend zu schreiben. In meiner Implementierung möchte ich Wert auf folgende Punkte legen:

1. Das Interface soll komplett objektbasiert gestaltet werden
2. Die Übertragung der Befehle soll mit einem Serializer und Deserializer erfolgen. Es soll also kein zeichen- oder tokenbasiertes Protokoll erbastelt werden, da dieses schlecht wartbar, fehleranfällig und zu oft zu unsicher und nicht robust genug implementiert wird

Verknüpft man die beiden Anforderungen kommt man zu folgender Grundidee:

* Es gibt eine Klasse Task, mit welcher jede Aktion als konkreter Task implementiert wird (z.B. einen Account erstellen)
* Diese Klasse wird vom Client serialisiert und an den Server gesendet
* Der Server deserialisiert die Klasse und führt den konkreten Task aus. Dabei speichert er das Resultat und allfällige Exceptions im Task
* Der Task wird serialisiert und wieder an den Client zurückgesendet
* Der Client deserialisiert den empfangenen Task, wirft allfällige Exceptions und verwendet das Resultat

# Implementierung

Die abstrakte Klasse Task sieht wie folgt aus:

|  |
| --- |
| public abstract class Task implements Serializable {  protected Object result;  protected Exception exception;  public abstract void executeTask(BankDriver driver) throws Exception;  public void executeHandledTask(BankDriver driver) {  try {  executeTask(driver);  } catch (Exception exception) {  this.exception = exception;  }  }  public Object getResult() {  return result;  }  public void throwPossibleIoException() throws IOException {  if (exception != null && exception instanceof IOException) {  throw new IOException(exception.getMessage(), exception);  }  }  public void throwPossibleIllegalArgumentException() throws IllegalArgumentException {  if (exception != null && exception instanceof IllegalArgumentException) {  throw new IllegalArgumentException(exception.getMessage(), exception);  }  }  public void throwPossibleInactiveEception() throws InactiveException {  if (exception != null && exception instanceof InactiveException) {  throw new InactiveException(exception.getMessage(), exception);  }  }  public void throwPossibleOverdrawException() throws OverdrawException {  if (exception != null && exception instanceof OverdrawException) {  throw new OverdrawException(exception.getMessage(), exception);  }  }  } |

Eine konkrete Implementierung eines Tasks sieht dann wie folgt aus:

|  |
| --- |
| public static class CreateAccountTask extends Task {  private final String owner;  public CreateAccountTask(String owner) {  this.owner = owner;  }  @Override  public void executeTask(BankDriver driver) throws Exception {  this.result = driver.getBank().createAccount(owner);  }  } |

Der CreateAccountTask wird wie eingangs erwähnt durch den Client erstellt, serialisiert und an den Server gesendet. Die Methode executeTask wird dementsprechend auf dem Server mit der lokalen Bankentreiberimplementierung ausgeführt (und nicht im Kontext des Clients, wie man fälschlicherweise annehmen könnte). Durch das Injizieren des Bankentreibers kann der Task problemlos serialisiert werden.

Im Client wird dann der spezifische Task wie folgt aufgerufen:

|  |
| --- |
| @Override  public String createAccount(String owner) throws IOException {  Task task = Sender.send(new Task.CreateAccountTask(owner));  task.throwPossibleIoException();  return (String) task.getResult();  } |

Der Task wird an die Sender Hilfsklasse gesendet, wo das Objekt serialisiert und an den Server gesendet wird. Nachdem der Server den blockierenden Aufruf wieder zurückgibt, werden allfällige Exceptions geworfen und das Resultat wird gecastet und verwendet.

Die Hilfsklasse sieht wie folgt aus (Der statische Hostname und Port werden von aussen gesetzt):

|  |
| --- |
| public class Sender {  public static String hostname = "127.0.0.1";  public static int port = 8080;  public static Task send(Task task) {  try {  Socket socket = new Socket(hostname, port);  ObjectOutput outputstream = new ObjectOutputStream(socket.getOutputStream());  outputstream.writeObject(task);  ObjectInputStream inputstream = new ObjectInputStream(socket.getInputStream());  task = (Task) inputstream.readObject();  inputstream.close();  outputstream.close();  socket.close();  return task;  } catch (Exception exception) {  return new Task.InvalidTask();  }  }  public static Account fromLocalAccountToSocketAccount(Account localaccount) throws IOException {  if (localaccount == null) {  return null;  }  return new SocketAccount(localaccount.getNumber(), localaccount.getOwner());  }  public static Account fromSocketAccountToLocalAccount(Account socketaccount) throws IOException, InactiveException {  if (socketaccount == null) {  return null;  }  LocalAccount localaccount = new LocalAccount(socketaccount.getOwner(), socketaccount.getNumber());  if (socketaccount.isActive()) {  localaccount.deposit(socketaccount.getBalance());  } else {  localaccount.close();  }  return localaccount;  }  } |

Dabei ist es wichtig zu erwähnen, dass zwischen lokalen und entfernten Accounts unterschieden und hin und her konvertiert werden muss. Geschieht dies nicht, können durch die Serialisierung sogenannte Round Trips entstehen und die Applikation schaukelt sich mit rekursiven Request hoch und stürzt irgendwann ab.

Der Server ist mittels einem einfachen dynamischen Threadpool realisiert und verwendet die lokale Implementierung des Bankentreibers:

|  |
| --- |
| public class RemoteDriver {  private final ServerSocket serversocket;  private final BankDriver bankdriver;  public RemoteDriver(int port) throws IOException {  serversocket = new ServerSocket(port);  bankdriver = new LocalDriver();  }  public void startServer() throws IOException {  ExecutorService service = Executors.newCachedThreadPool();  while (true) {  Socket socket = serversocket.accept();  service.execute(new RequestHandler(bankdriver, socket));  }  }  public static void main(String[] args) throws IOException {  new RemoteDriver(1234).startServer();  }  } |

Der Request Handler serialisiert wie erwähnt die Tasks, führt Sie aus und sendet den veränderten Task zurück (Es wird hoffentlich endgültig ersichtlich, warum zwischen lokalen und entfernten Accounts unterschieden werden muss):

|  |
| --- |
| public class RequestHandler implements Runnable {  private final BankDriver bankdriver;  private final Socket socket;  public RequestHandler(BankDriver bankdriver, Socket socket) {  this.bankdriver = bankdriver;  this.socket = socket;  }  @Override  public void run() {  try {  ObjectInputStream inputstream = new ObjectInputStream(socket.getInputStream());  Task task = (Task) inputstream.readObject();  task.executeHandledTask(bankdriver);  ObjectOutputStream outputstream = new ObjectOutputStream(socket.getOutputStream());  outputstream.writeObject(task);  inputstream.close();  outputstream.close();  socket.close();  } catch (Exception exception) {  exception.printStackTrace();  }  }  } |

# Verbesserungsmöglichkeiten

Die momentane Version ist designbedingt schon sehr sauber realisiert. Problematisch könnte jedoch die Performance werden, da das mehrmalige Serialisieren und Deserialisieren schon sehr zeitaufwändig ist. Eventuell müsste man früher oder später auf einen eigenen Serialisierer/Deserialisierer umsteigen, welcher performanter implementiert ist.

Hinzu kommt, dass die aktuelle Implementierung über keinen Cache verfügt. Würde man pushbare Sockets oder Protokolle wie HTTP2 verwenden, könnte man Änderungen vom Server auf die Clients pushen und so mit einem Cache arbeiten, welcher die Zahl notwendiger Requests minimiert.

# Source Code

Der Source Code ist hier zu finden: <https://github.com/swaechter/fhnw/tree/master/Module/vesys/bank>