Übungsaufgabe #0: Grundlagen

Diese Übungsaufgabe soll die notwendigen Vorkenntnisse für die Belegung der Veranstaltung „Verteilte Systeme“ nahelegen. Im Gegensatz zu den weiteren Aufgaben im Semester ist sie nicht als Gruppenaufgabe konzipiert, sondern sollte von allen Teilnehmenden selbstständig bearbeitet werden. Die Bearbeitung ist freiwillig, wird aber stark empfohlen.

In der Aufgabe geht es darum, folgende Bereiche zu üben bzw. zu wiederholen:

- Grundlagen der Implementierung in Java
- Verwendung von Java auf der Konsole: Kompilieren und Ausführen über die Kommandozeile
- Notwendige Kenntnisse in Linux: SSH, einfache Bash-Skripte
- Synchronisation

Alle diese Themen und mehr werden auch ihm Rahmen der Grundlagenfolien und -videos auf der Webseite behandelt, die als Vorbereitung zu „Verteilte Systeme“ empfohlen werden. Zur Bearbeitung der Aufgabe ist zudem ein Account im CIP-Pool der Informatik notwendig. Falls noch nicht vorhanden, kann dieser mithilfe von *Single Sign-On* über `https://account.cip.cs.fau.de/` erstellt werden.

Hinweis: Da die Aufgabe als Vorbereitung und Selbsteinschätzung konzipiert ist, liegt sowohl ihr Umfang als auch der Schwierigkeitsgrad weit unter den eigentlichen Übungsaufgaben im Semester!

0.1 Sirexa Chatbot

Ziel des ersten Teils der Aufgabe ist die Erstellung und Verwendung eines einfachen Chatbots[1]. Die Server-Seite ist dabei schon vorgegeben, die Client-Seite dagegen muss im Rahmen der Aufgabe noch implementiert werden.

0.1.1 Server-Seite

Die Server-Seite des Programms ist bereits vollständig implementiert und in Form eines JAR-Archivs im Dateisystem der CIP-Pool-Rechner unter `/proj/i4vs/pub/aufgabe0/sirexa.jar` verfügbar. Das Programm erwartet als einziges Kommandozeilenargument einen Port, über den es nach dem Start eigenständig seinen Dienst anbietet.

Achtung: Der Server ist bewusst so gestaltet, dass er sich nur auf einem CIP-Rechner ordnungsgemäß ausführen lässt und auf anderen Rechnern direkt beim Start mit einer Fehlermeldung abbricht. Entsprechend empfiehlt es sich, zunächst per SSH eine Verbindung in einen CIP-Pool aufzubauen [Folie 0.2 : 3] und den Chat-Server anschließend über die Kommandozeile zu starten. Da genau diese Form der verteilten Ausführung bei dieser Teilaufgabe im Vordergrund steht, sollte dieses Vorgehen auch dann zum Einsatz kommen, falls vor Ort im CIP-Pool gearbeitet wird.

Verwendung des Server-JAR-Archivs Als erstes soll die Benutzung des Server-JAR-Archivs über die Kommandozeile erfolgen. Ein Java-Programm wird mittels `java` ausgeführt, wobei Archive per `-cp` zum Classpath hinzugefügt werden müssen [Folie 0.2 : 1]. Die zu startende Hauptklasse ist im Fall des Chat-Programms `vsue.chat.VSSirexa`. Wie bereits erwähnt ist zudem der Port, auf dem der Server laufen soll, als Parameter mitzugeben. Eine erfolgreiche Inbetriebnahme des Servers ist dadurch zu erkennen, dass nach dem Start zunächst folgende Kommandozeilenausgabe erscheint:

```
Sirexa ready to chat! Host: <hostname> Port: <port>
Waiting on client...
```

Bash-Skripte Um nicht bei jedem Start den gesamten Java-Befehl eintippen zu müssen, lässt sich der Kommandozeilenaufruf in einem Skript kapseln. Hierfür liegt im Verzeichnis `/proj/i4vs/pub/aufgabe0/` bereits ein halbfertiges Bash-Skript, dass nun vervollständigt werden soll. Zuerst ist der Java-Befehl an der entsprechend gekennzeichneten Stelle einzufügen. Der Port soll dabei nicht fest im Skript definiert sein, sondern auch hier wieder über die Kommandozeile übergeben werden. Eine Variable `PORT` ist dafür bereits im Skript angelegt; auf den Inhalt dieser Variablen lässt sich mit `$PORT` zugreifen (z. B. um sie im Java-Aufruf zu verwenden). Zugriff auf übergebene Kommandozeilenparameter erhält man in Bash-Skripten mittels `$1`, `$2`, `$3` usw., wobei `$n` den Parameter an Position `n` referenziert.

Aufgaben:

→ Starten des Server-JAR-Archivs über die Kommandozeile
→ Vervollständigen des vorgegebenen Bash-Skripts zum leichteren Starten des Servers

Hinweise:

- Das Server-JAR-Archiv ist kein *executable* JAR und kann daher nicht mithilfe von `-jar` ausgeführt werden.
- Das Bash-Skript soll mit `./startServer.sh <port>` ausgeführt werden [Folie 0.2 : 5].
- Um ein Skript ausführbar zu machen, verwendet man `chmod +x <Dateiname>`.

[1] Basierend auf der ELIZA-Implementierung von Charles Hayden

0.1.2 Client-Seite

Nachdem die Server-Seite funktioniert, soll jetzt die Client-Seite des Chatbots implementiert werden. Der Client funktioniert dabei wie folgt: Nachdem eine Verbindung zum Server aufgebaut wurde, kann der Nutzer eine Nachricht an den Server schreiben. Diese wird über die Kommandozeile eingelesen und mit Enter bestätigt. Der Chat-Client schickt die Nachricht dann an den Server und wartet auf eine Antwort. Diese wird anschließend mittels Standardausgabe dargestellt und der Nutzer kann eine neue Nachricht schreiben. Ein möglicher Ablauf sieht also wie folgt aus:

```
Server: Hello <User>, I am Sirexa. How can I help you today?
Client: What day is today?
Server: Today is Monday. Such a beautiful day!
Client: Tell me a joke
Server: [...]
```

Kommunikation mit dem Server Damit der Client auch die Adresse (Hostname und Port) des Servers kennt, soll er diese beim Start als Parameter in `args` mitbekommen. Unter Angabe der Adresse kann er dann mithilfe der Klasse `java.net.Socket` eine TCP-Verbindung zum Server öffnen. Um Daten über den Socket zu senden oder zu empfangen, soll im Rahmen dieser Aufgabe ein `Data{Out,In}putStream` über den `Output-` bzw. `InputStream` des Socket gelegt werden. Zum Austausch von Nachrichten (in dieser Aufgabe repräsentiert durch einfache `String`-Objekte) bieten diese Datenströme die Methoden `writeUTF(String)` bzw. `readUTF()` an. Die Methoden kümmern sich dabei selbstständig darum, dass jeweils die gesamte Zeichenkette vollständig versendet bzw. empfangen wird.

Anmeldung beim Server Im Anschluss an den Verbindungsaufbau muss sich der Client beim Server registrieren indem er ihm als erstes einen Nutzernamen sowie ein Passwort in Form jeweils separater Nachrichten schickt. Der Nutzername kann frei gewählt werden, das Passwort soll dagegen vom Client aus der Datei `credentials.txt` (siehe `/proj/i4vs/pub/aufgabe0/`) ausgelesen werden. Als Reaktion auf die Anmeldung antwortet der Server mit einer Willkommensnachricht an den Client und begrüßt diesen.

Eingabe und Austausch von Nachrichten Nachdem die Anmeldung beim Server erfolgreich war, soll der Client auf eine mit Enter bestätigte Nutzereingabe warten. Zum Einlesen der Standardeingabe bietet sich die Klasse `java.util.Scanner` an, deren Methode `nextLine()` die nächste Zeile als Zeichenkette zur Verfügung stellt.

```
Scanner scanner = new Scanner(System.in);
String line = scanner.nextLine();
```

Die eingelesene Zeile sendet der Client an den Server, pausiert anschließend bis zu dessen Antwort, und gibt diese dann per Standardausgabe auf dem Bildschirm aus. Analog verfährt der Client mit allen weiteren Nutzereingaben.

Starten und Testen der Implementierung Auch für den Start des Clients soll wieder ein Bash-Skript zur einfacheren Handhabung erstellt werden. Hierfür bietet sich das Server-Skript aus Teilaufgabe 0.1.1 als Vorbild an.

Nach der Fertigstellung des Clients soll nun abschließend das Zusammenspiel mit dem Server getestet werden. Hierzu sind Server und Client auf zwei unterschiedlichen Rechnern zu starten und ein paar Nachrichten zu schicken. Was passiert zum Beispiel, wenn der *Sirexa*-Server nach dem Wetter gefragt wird oder einen Witz erzählen soll?

Aufgaben:

→ Implementierung der Client-Seite
→ Schreiben eines Bash-Skripts zum leichteren Starten des Clients
→ Testen der eigenen Client-Implementierung mit dem vorgegebenen Server

Hinweise:

- Zu Übungszwecken ist es ratsam, die Client-Implementierung nicht von der eigenen IDE kompilieren zu lassen, sondern den Code (zumindest einmal) selbst händisch über die Konsole zu kompilieren.
- Beim Betrieb des Clients auf Privatrechnern kann es unter Umständen zu Firewall-Problemen kommen. Sollten derartige Probleme auftreten und sich nicht lösen lassen, ist es ratsam, den Client im CIP-Pool auszuführen.
- Das Passwort für den *StudOn-Kurs* kann mit `"What is the password?"` erfragt werden.
- Eine Beispielimplementierung von Aufgabe 0 kann ebenfalls von Sirexa erfragt werden (z.B. mit `"Show me a solution"`).

0.2 Bad Bank

Der Fokus der zweiten Teilaufgabe liegt auf dem Thema Synchronisation. Als Grundlage hierfür dient die (fehlerhafte) Implementierung einer Bank-Anwendung, die im Verzeichnis `/proj/i4vs/pub/aufgabe0/` bereit liegt und in einem Subpackage `vsue.badbank` realisiert ist. In der Anwendung sind zwei Synchronisationsfehler versteckt, die es im Rahmen dieser Aufgabe zu finden und zu beheben gilt.

Details zur Implementierung Die Bankanwendung besteht aus den beiden Klassen `VSBankAccount` und `VSBroker`.

`VSBankAccount` repräsentiert dabei ein einzelnes Konto, das sich in Form des Besitzers (`owner`) eindeutig identifizieren lässt und den aktuellen Kontostand (`balance`) kapselt. Zur Modifikation des Kontostands dienen drei Methoden: Mit `withdraw()` und `deposit()` kann Geld direkt abgehoben bzw. eingezahlt werden, wogegen `transfer()` für die direkte Überweisung von Beträgen zwischen zwei Konten zum Einsatz kommt. Ein Konto soll unter keinen Umständen überzogen werden können, der Kontostand darf also niemals unter 0 fallen.

`VSBroker` verwaltet die Liste aller Konten und hat die Aufgabe, in periodischen Abständen (z. B. alle 100 ms) einen aktuellen Übersichtsbericht über die Konten zu erstellen. Eine solche Zusammenfassung soll dabei einen *stabilen* Zustand der Konten enthalten, das heißt während der Erstellung des Berichts darf sich kein Geld *in transit* befinden. Solange nur Geld zwischen den vorhandenen Konten mit `transfer()` transferiert und nicht anderweitig abgehoben oder eingezahlt wird, muss die Gesamtsumme aller Kontostände im Bericht also über die Zeit gleich bleiben.

Testfälle Zum Testen der Anwendung steht die Klasse `VSBankTest` zur Verfügung. Diese ermöglicht per Parameter entweder die Ausführung eines einfachen (`simple`) oder eines komplexeren (`fancy`) Testfalls. Für den Einstieg und das Finden des ersten Synchronisationsfehlers bietet es sich an, mit dem einfachen Test zu beginnen. `simple` erstellt 10 Konten, jeweils mit einem Anfangsbetrag von 100, und startet anschließend mehrere Threads, die parallel Geld zwischen den Konten hin und her transferieren. `fancy` funktioniert ähnlich, hier kommt aber zusätzlich ein `VSBroker` hinzu, der regelmäßig den aktuellen Stand aller Konten in Form des Übersichtsberichts ausgibt. Beide Tests sollen überprüfen, ob die vorher definierten Eigenschaften von `VSBankAccount` und `VSBroker` (z. B. keine ungewollten Änderungen der Kontostände, keine Überziehung eines Kontos, Stabilität des im Bericht dokumentierten Zustands) erfüllt werden.

Synchronisationsfehler Ohne weitere Maßnahmen liefert keiner der beiden Testfälle ein korrektes Ergebnis, da zwei Synchronisationsfehler in der Bank-Implementierung existieren. Einer davon ist recht einfach zu beheben und benötigt keine größeren Änderungen, der andere ist dagegen etwas komplexer. Ziel dieser Teilaufgabe ist es, beide Fehler durch geeignete Anpassungen der vorliegenden Implementierung in möglichst effizienter Art und Weise zu beseitigen. Erlaubt sind hierfür beliebige Änderungen und Erweiterungen an den Klassen `VSBankAccount` und `VSBroker`, solange sich die von `VSBankTest` verwendeten Schnittstellen dabei nicht ändern. Die Klasse `VSBankTest` selbst muss im Gegensatz zu den beiden anderen Klassen zwingend in ihrem vorgegebenen Zustand verbleiben.

Aufgabe:

→ Anpassung der Klassen `VSBankAccount` und `VSBroker`, so dass die definierten Eigenschaften erfüllt werden

Hinweis:

- Es ist weder im Sinne der Aufgabe noch zielführend, alle Methoden als `synchronized` zu markieren.