JUST DO IT
HÖHERE ABTEILUNG FÜR
INFORMATIONSTECHNOLOGIE

tgm
Die Schule der Technik

Laborprotokoll

RMI

Systemtechnik Labor
4BHIT 2015/16, GruppeX

Thomas Fellner

Version 1
Note:
Betreuer: M. Borko
Begonnen am 22. April 2016
Beendet am 29. April 2016

Inhaltsverzeichnis

1 Einführung

Verteilte Objekte haben bestimmte Grunderfordernisse, die mittels implementierten Middlewares leicht verwendet werden können. Das Verständnis hinter diesen Mechanismen ist aber notwendig, um funktionale Anforderungen entsprechend sicher und stabil implementieren zu können.

1.1 Ziele

Diese Übung gibt eine einfache Einführung in die Verwendung von verteilten Objekten mittels Java RMI. Es wird speziell Augenmerk auf die Referenzverwaltung sowie Serialisierung von Objekten gelegt. Es soll dabei eine einfache verteilte Applikation in Java implementiert werden.

1.2 Voraussetzungen

- Grundlagen Java und Software-Tests
- Grundlagen zu verteilten Systemen und Netzwerkverbindungen
- Grundlegendes Verständnis von nebenläufigen Prozessen

1.3 Aufgabenstellung

Folgen Sie dem offiziellen Java-RMI Tutorial [1], um eine einfache Implementierung des PI-Calculators zu realisieren. Beachten Sie dabei die notwendigen Schritte der Sicherheitseinstellungen (Security-Manager) sowie die Verwendung des RemoteInterfaces und der RemoteException.

Implementieren Sie ein Command-Pattern [2] mittels RMI und übertragen Sie die Aufgaben/Berechnungen an den Server. Sie können am Client entscheiden, welche Aufgaben der Server übernehmen soll. Die Erweiterung dieser Aufgabe wäre ein Callback-Interface auf der Client-Seite, die nach Beendigung der Aufgabe eine entsprechende Rückmeldung an den Client zurück senden soll. Somit hat der Client auch ein RemoteObject, welches aber nicht in der Registry eingetragen wird sondern beim Aufruf mittels Referenz an den Server übergeben wird.

 Thomas Fellner

2 Ergebnisse

2.1 Tutorial

Zu finden ist das Tutorial entweder auf meinem Github `https://github.com/tfellner-tgm/SYT-rmi` [3] oder unter den Tutorials von Michael Borko [4]

2.1.1 Ant

Ant ist ein Buildtool, welches durch die `build.xml` definiert wird.
Dieses File kann von eclipse oder Intellij generiert werden und besteht dann aus vielen Properties und Targets. Die Properties bestehen als Voraussetzung für den build (z.B. Java Version, Binary Pfad ...) Die Targets können über die Commandline ausgeführt werden mittels `ant [target]`. Der `build` target kompelliert die Sources zu Binaries, welche dann von anderen Targets verwendet werden können. Hier können auch Commandline-Arguments definiert werden

2.1.2 RMI

Konkret in dem Beispiel gibt es die Targets `engine` und `compute`, wobei Engine quasi der Server und Compute der Client ist.

Hier der wichtigste Code der Engine:

```
1 Compute engine = new ComputeEngine();
  Compute stub = (Compute) UnicastRemoteObject.exportObject(engine, 0);
3 Registry registry = LocateRegistry.createRegistry(1099);
  registry.rebind("Compute", stub);
```

Listing 1: Engine

Zuerst wird hier das Objekt engine erstellt. Dieses Objekt dient Compute dazu eine Task auszuführen. Danach wird das engine Objekt exportiert und im stub gespeichert.
`createRegistry(port)` erstellt eine Registry, in welcher diese stubs dann mit dem bind oder rebind (bind würde noch überprüfen ob der Name schon vergeben ist) mittels eines Namen freigegeben werden können.

Die Klasse ComputePi verwendet `getRegistry(port)` um sich in die Registry einzutragen und holt sich dann das Compute Objekt mittels `registry.lookup(name)`

```
  Registry registry = LocateRegistry.getRegistry(1099);
2 Compute comp = (Compute) registry.lookup("Compute");
```

Listing 2: Compute

Wenn nun die Engine ausgeführt wird, bekommt man die Exception

Java.security.AccessControlException: access denied

2.1.3 Java Policy

RMI braucht noch bestimmte Permissions. Um dem Programm diese zu geben, muss die Datei `java.policy` unter dem Verzeichnis der JDK `/usr/lib/jvm/{java-version}/jre/lib/security/java.pol` ins Homeverzeichnis (`/home/{name}/.java.policy`) kopiert werden, wobei bei diesem Block

```
grant codeBase "file:${{java.ext.dirs}}/*" {
        permission java.security.AllPermission;
};
```

Listing 3: java.policy

der Ausdruck `${{java.ext.dirs}}/*` dann noch zu `/home/{name}/-` geändert werden muss. Dies sagt aus, dass alle Dateien (-) unter dem Homeverzeichnis diese permission haben. In diesem Fall ist alles erlaubt, wegen des Ausdrucks `java.security.AllPermission`.

 Thomas Fellner

2.2 Command Pattern mit Callback

2.2.1 Command Pattern

Das Command Pattern ist realisiert durch 2 Interfaces und deren dazugehörige Klassen.

Das Command Interface gibt vor, dass ein Command die Methode `execute` besitzen muss.

```
1 public interface Command extends Serializable {
      public void execute();
3 }
```

Listing 4: Command Interface

Das CommandExecuter Interface ist, wie der Name schon sagt, dafür zuständig ist, die gegebenen Commands auszuführen

```
1 public interface CommandExecutor extends Remote {
      public void executeCommand(Command c) throws RemoteException;
3 }
```

Listing 5: CommandExecuter Interface

In der Konkreten Implementierung einer Klasse (`ServerService`) wird nur `c.execute()` aufgerufen.

2.2.2 Callback

Für die Callback Funktion wurde ein Callback Interface erstellt. Wichtig beim Interface ist, dass alle Methoden eine RemoteException werfen und die konkrete Klasse Serializable ist.

```
1 public interface Callback<T> extends Remote, Serializable {
      public void set(T argument) throws RemoteException;
3     public void print() throws RemoteException;
      public T receive() throws RemoteException;
5 }
```

Listing 6: Callback Interface

`set(T argument)` dient hier als setzen eines Attributes, welches von `print()` und `receive()` verwendet wird.
`print()` gibt den Wert im Terminal aus.
`T receive()` gibt den Wert zurück;

Weiters habe ich eine konkrete Klasse (`CallbackBigDecimal`) erstellt, welche diese Interface mit `BigDecimal` realisiert.

Um dann den Callback zu verwenden muss das Callback Objekt exportiert werden. Es wird nicht in die Registry gespeichert werden, da nur der Server antworten soll und nicht irgendein anderer Client.

```
1 Callback cb = new CallbackBigDecimal();
  Callback cbStub = (Callback) UnicastRemoteObject.exportObject(cb, 0);
```

Listing 7: Callback

2.2.3 Calculation

Das Interface Calculation wird verwendet von `PICalc` und `EulerCalc`. Es gibt vor, dass in der Methode `calculate()` die Berechnungen gemacht werden und in `getResult()` das Resultat zurück gegeben wird.

```
1 public interface Calculation {
2     public void calculate();
3     public BigDecimal getResult();
4 }
```

Listing 8: Callback Interface

`PICalc` ist vom Tutorial übernommen und umgeschrieben, um mit dem Interface zu funktionieren. `EulerCalc` berechnet die Eulersche Zahl e mit einer bestimmten Genauigkeit. Die Formel für die Eulersche Zahl ist $\sum_{k=0}^{n} e = \frac{1}{k!}$ wobei n die Genauigkeit ist.

2.2.4 CalculationCommand

Im `CalculationCommand` werden der Callback und die Calculation zusammengeführt.

```
1  public class CalculationCommand implements Command, Serializable {
2
3      private static final long serialVersionUID = 12L;
4      private Calculation calc;
5      private Callback cb;
6
7      public CalculationCommand(Callback cb, Calculation calc){
8          this.cb = cb;
9          this.calc = calc;
10     }
11
12     @Override
13     public void execute() {
14         System.out.println("CalculationCommand called!");
15         calc.calculate();
16         try {
17             cb.set(calc.getResult());
18             cb.print();
19         } catch (RemoteException e) {
20             System.out.println("Calculation Command error " + e);
21         }
22     }
23 }
```

Listing 9: CalculationCommand Klasse

Der Konstruktor bekommt 2 Objekte der Typen Callback und Calculation und wenn `execute()` aufgerufen wird, werden die Methoden des Interfaces aufgerufen. Zuerst wird berechnet, dann die Nummer des Callbacks zum Resultat gesetzt und dann dieses ausgegeben.

Um das Command zu erstellen muss der vorhin erstellte callback Stub mitgegeben werden und eine neue Calculation erstellt werden
Hier beide Möglichkeiten des Programms:

```
1 Command calcPi = new CalculationCommand(cbStub, new PICalc(Integer.parseInt(args[0])));
2 Command calcEuler = new CalculationCommand(cbStub, new EulerCalc(Integer.parseInt(args[0])));
```

Listing 10: Command Erstellung

2.2.5 Programmstruktur

Hier die Programmstruktur nochmal als ganzes in einem UML Diagramm dargestellt:
Die Verbindung von `Client` zu `ServerService` soll darstellen, dass das Objekt über die Registry geholt wird.

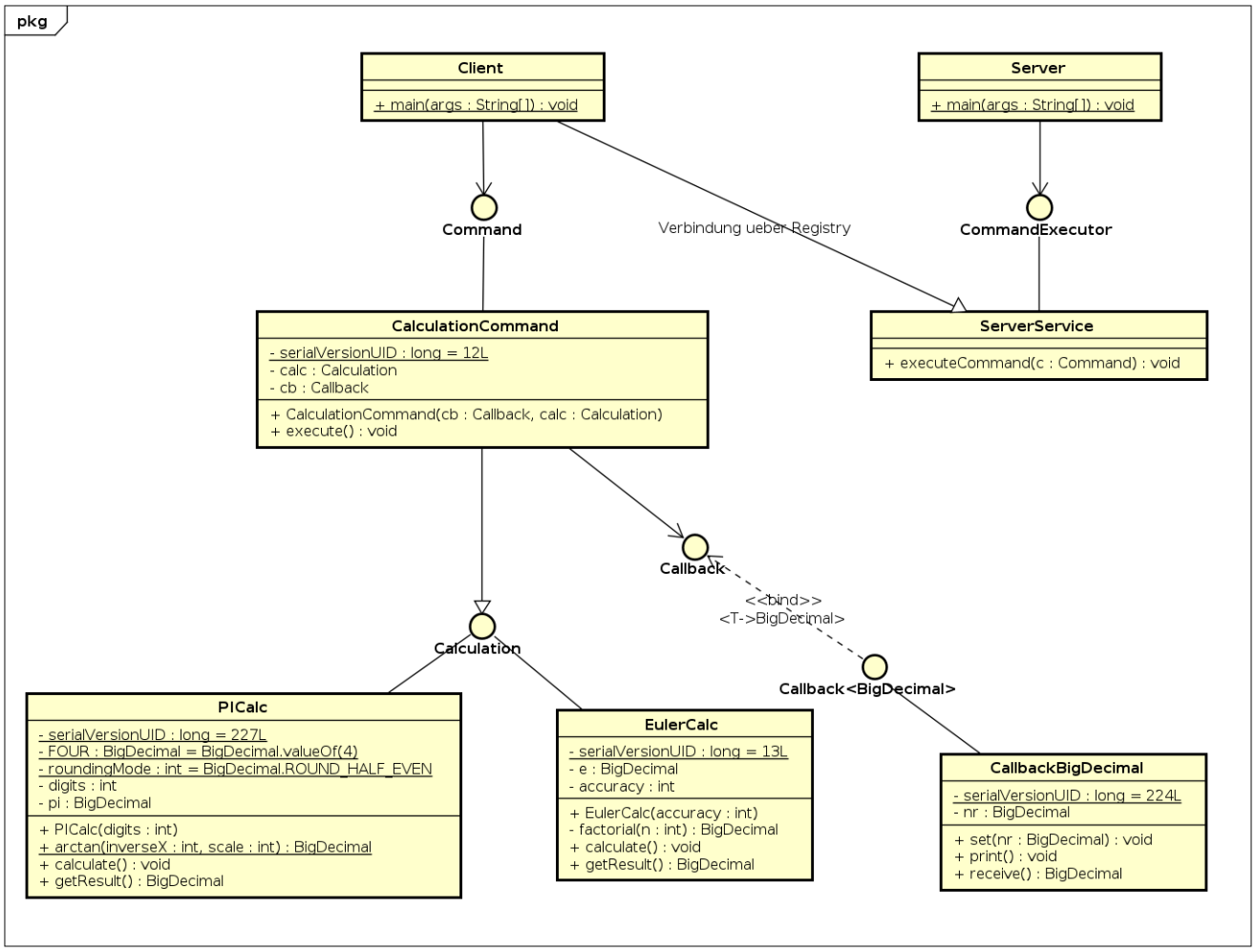

Abbildung 1: UML Diagramm der Aufgabe

3 Zeitaufzeichnung

Datum	Erwartete Dauer	Reale Dauer	Beschreibung
22.4.2016	2 Stunden	1 1/2 Stunden	Tutorial fertigstellen
22.4.2016	/	3 1/2 Stunden	Command Pattern und Callback Überlegung
24.4.2016	1/2 Stunde	1 Stunde	Protokoll beginnen
28.4.2016	2 Stunden	3 Stunden	Aufgabe fertigstllen
28.4.2016	2 Stunden	3 Stunden	Protokoll ferigstellen

Tabelle 1: Zeitaufzeichnung

 Thomas Fellner

Literatur

[1] The java tutorials - trail rmi. `http://docs.oracle.com/javase/tutorial/rmi/`.

[2] Vince Huston. Command pattern. `http://vincehuston.org/dp/command.html`.

[3] Thomas Fellner. Github repoisitory mit der aufgabe. `https://github.com/tfellner-tgm/SYT-rmi`.

[4] Michael Borko. Beispiel konstruktor für command pattern mit java rmi. `https://github.com/mborko/code-examples/tree/master/java/rmiCommandPattern`.

Tabellenverzeichnis

Listings

Abbildungsverzeichnis