

**Laborprotokoll  
RMI –   
Remote Method Invocation**

**Systemtechnik Labor**

**4BHIT 2015/16, Gruppe X**

**Ramin Bahadoorifar**

|  |  |
| --- | --- |
|  | Version 1.0 |
| Note: | Begonnen am 22. April 2016 |
| Betreuer: Prof. Michael Borko | Beendet am 29. April 2016 |

Inhaltsverzeichnis

[1 Einführung 4](#_Toc449886650)

[1.1 Ziele 4](#_Toc449886651)

[1.2 Voraussetzungen 4](#_Toc449886652)

[1.3 Aufgabenstellung 4](#_Toc449886653)

[2 Ergebnisse 5](#_Toc449886654)

[2.1 Java RMI-Tutorial 5](#_Toc449886655)

[2.1.1 Java Policy 5](#_Toc449886656)

[2.1.2 Ausführung des Tutorials 5](#_Toc449886657)

[2.2 Command-Pattern inkl. Client-Callback 6](#_Toc449886658)

[2.2.1 Calculation 6](#_Toc449886659)

[2.2.2 Command-Pattern 7](#_Toc449886660)

[2.2.3 UML Diagram des Programmes 7](#_Toc449886661)

[3 Feedback 8](#_Toc449886662)

[3.1 Zeitaufzeichnung 8](#_Toc449886663)

# 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.

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.

Voraussetzungen

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

Aufgabenstellung

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

Implementieren Sie ein Command-Pattern[[2]](#footnote-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.[[3]](#footnote-3)

# Ergebnisse

RMI ist eine Java API, welche ermöglicht auf entfernte (in anderen JVMs) Objekte zugreifen zu können. Dies funktioniert auch in netzwerkentfernten JVMs. Solche Objekte müssen von einem Interface implementieren, welche dem Client und Server verfügbar sein muss. Dadurch kann das Objekt dann angesprochen werden.

Java RMI-Tutorial

Das Java-RMI Tutorial ist auf der offiziellen Seite[[4]](#footnote-4) oder in meinem Github-Repository[[5]](#footnote-5) zu finden. Beim Verwenden des offiziellen Tutorials ist zu beachten, dass Änderungen vorgenommen wurden. Empfohlen wird daher die Verwendung meines Github-Repositorys.

### Java Policy

Um das Programm starten zu können muss vorher noch die Java Policy angepasst werden. Diese befindet sich in Windows unter *C:\Program Files\Java\[jdk\_version]\jre\lib\security\java.policy.*

*Folgende Zeile muss hinzugefügt werden*

grant codeBase "file:/D:/Ramin/TGM-DATEN/4YHIT/Systemtechnik/-" {

permission java.security.AllPermission;

};

Die Java Policy definiert die Rechte eines Programmes, welches, wie in diesem Fall, im Ordner Systemtechnik gestartet wurde. Leider wurde die Java Policy in der IDE IntelliJ nicht verwendet, deswegen startete ich die kompilierten .class-Files in der Konsole. Die Konsole wurde im src-Ordner geöffnet und mit dem Befehl *java <package>.<classname without .class>* gestartet.

### Ausführung des Tutorials

Zum Ausführen des Tutorials muss die Klasse ComputeEngine ohne Argumente gestartet werden. Diese Klasse ist sozusagen der Server dieses Programmes. ComputePi hingegen ist der Client des Programmes, diese wird mit folgenden Argumenten gestartet: <ip-adresse> <genauigkeit von pi>

Bsp.: java client.ComputePi localhost 200

Command-Pattern inkl. Client-Callback

### Calculation

Die Klasse EulerCalculation implementiert vom Interface Calculation, welches nur eine execute()-Methode beinhaltet. Diese berechnet die Eulerzahl und ruft daraufhin die execute()-Methode von der CalculationCallback-Klasse mit der Eulerzahl als Parameter auf, welche die Zahl in der Konsole ausgibt.

**public** **class** **EulerCalculation** **implements** Calculation, Serializable {

**private** int digits;

**private** Callback callback;

**public** EulerCalculation(int digits, Callback callback) {

**this**.digits = digits;

**this**.callback = callback;

}

@Override

**public** void calculate() {

BigDecimal euler = BigDecimal.ONE;

BigDecimal fact = BigDecimal.ONE;

**for**(int i=1;i<digits;i++) {

fact = fact.multiply(**new** BigDecimal(i));

euler = euler.add(BigDecimal.ONE.divide(fact,

**new** MathContext(digits, RoundingMode.HALF\_UP)));

}

**try** {

callback.execute(euler);

} **catch**(RemoteException e) {

e.printStackTrace();

}

}

}

EulerCalculation-Klasse

### Command-Pattern

Das Command Pattern wurde realisiert durch ein Command-Interface, welches eine execute()-Methode beinhaltet. Von diesem Interface wurde eine Klasse CalculationCommand erstellt, welche dann die execute()-Methode definiert.

**package** server.commands;

**import** **java.io.Serializable**;

**public** **interface** **Command** **extends** Serializable {

**public** void execute();

}

Command-Interface

**package** server.commands;

**import** **java.io.Serializable**;

**import** **calculation.Calculation**;

**public** **class** **CalculationCommand** **implements** Command, Serializable {

**private** **static** **final** long serialVersionUID = 3202369269194172790L;

**private** Calculation calculation;

**public** CalculationCommand (Calculation calculation) {

**this**.calculation = calculation;

}

@Override

**public** void execute() {

System.out.println("CalculationCommand called!");

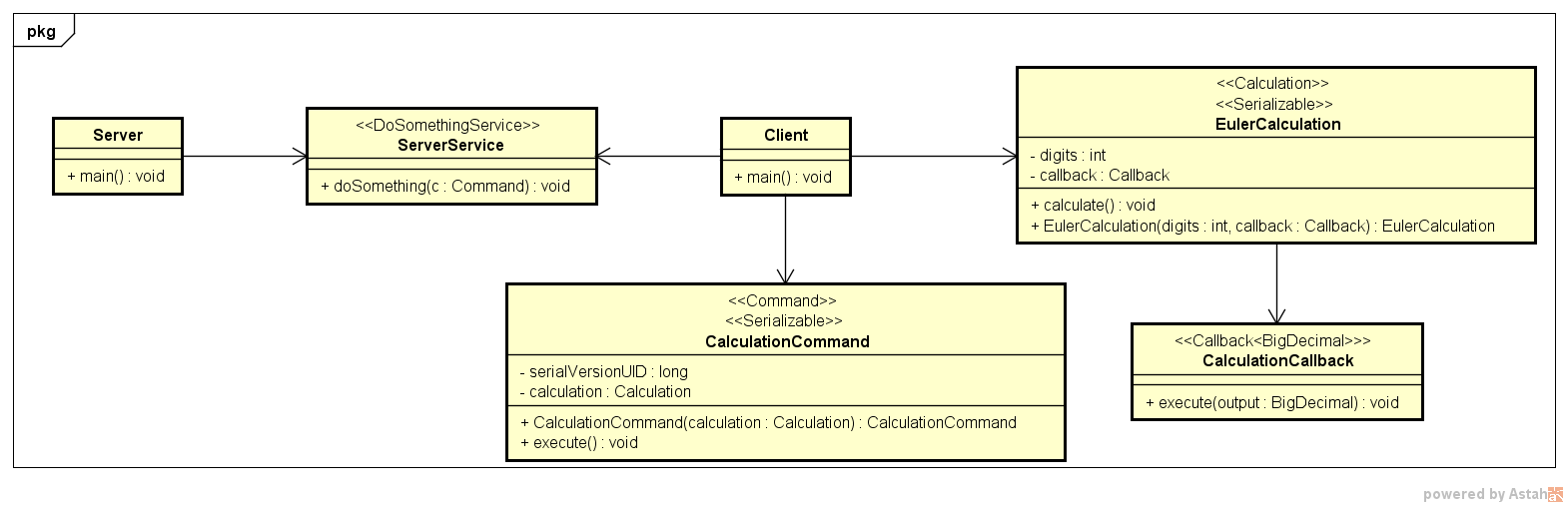
calculation.calculate();

}

}

CalculationCommand-Klasse

### UML Diagram des Programmes



# Feedback

Probleme gab es, wie erwähnt, beim Ausführen des Java Tutorials in der IntelliJ-IDE. Leider hab ich es bis jetzt nicht geschafft, dass IntelliJ die java.policy von der verwendeten JRE benutzt. Meine „Lösung“ war das Programm in der Konsole zu starten. (Siehe 2.1.1 Java Policy)

Zeitaufzeichnung

|  |  |
| --- | --- |
| Systemtechnik Labor | 4 Stunden |
| Privat | 6 Stunden |

1. "The Java Tutorials - Trail RMI"; online: http://docs.oracle.com/javase/tutorial/rmi/ [↑](#footnote-ref-1)
2. "Command Pattern"; Vince Huston; online: http://vincehuston.org/dp/command.html [↑](#footnote-ref-2)
3. "Beispiel Konstrukt für Command Pattern mit Java RMI"; Michael Borko; online:

   https://github.com/mborko/code-examples/tree/master/java/rmiCommandPattern [↑](#footnote-ref-3)
4. http://docs.oracle.com/javase/tutorial/rmi/ [↑](#footnote-ref-4)
5. https://github.com/rbahadoorifar-tgm/DezSys-RMI [↑](#footnote-ref-5)