JUST DO IT
HÖHERE ABTEILUNG FÜR INFORMATIONSTECHNOLOGIE

tgm
Die Schule der Technik

Java RMI

Systemtechnik Labor
4CHIT 2015/16, Gruppe C

Martin Weber

Version 1.0

Note:

Begonnen am 1. April 2016

Beendet am 7. April 2016

Betreuer: Borko Michael

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, 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] 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.

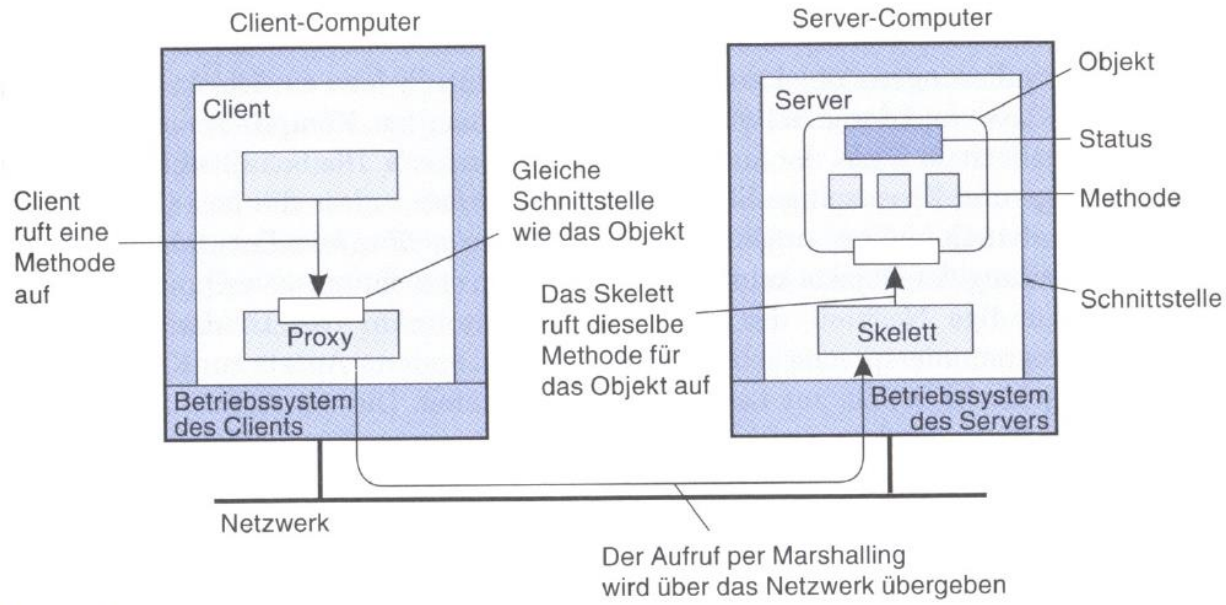
1.4. Quellen

[1] "The Java Tutorials - Trail RMI"; online: http://docs.oracle.com/javase/tutorial/rmi/
[2] "Command Pattern"; Vince Huston; online: http://vincehuston.org/dp/command.html
[3] "Beispiel Konstrukt für Command Pattern mit Java RMI"; Michael Borko; online: https://github.com/mborko/code-examples/tree/master/java/rmiCommandPattern

Bewertung: 16 Punkte
- Java RMI-Tutorial lauffähig (5 Punkte)
- Implementierung des Command-Patterns mittels RMI (6 Punkte)
- Implementierung des Client-Callbacks (2 Punkte)
- Protokoll entsprechend der Richtlinien mit entsprechendem theoretischen Background (3 Punkte)

2. Ergebnisse

2.1. RMI



Quelle: A.Tanenbaum „Verteilte Objekte“

Der Client sendet an den Server welche Methode dieser aufrufen soll und welche Parameter benutzt werden sollen. Über ein gemeinsames Interface kennt der Server diese Methode kann kann sie ausführen. Das Ergebnis schickt er dann an den Client zurück. Um Objekte übertragen zu können müssen sie das Interface Serializable implementieren um verschickt werden zu können. Das Remote Objekt über welches dann der Aufruf getätigt wird muss von java.rmi.remote erben. Dadurch kann man dann eine Remote Referenz auf das Objekt erhalten mit dem dieses angesprochen werden kann.

2.2. Java policy file

Mithilfe des Java policy files können Rechte an Programme vergeben werden

Das Java policy file ist unter folgendem Pfad zu finden:

C:\Program Files\Java\<verwendete Java Version>\lib\security\ java.policy
In diesem mussten dann folgende Zeilen eingefügt werden.

```
grant codeBase "file:/C:/Users/Martin/-" {
        permission java.security.AllPermission;
};
```

In diesem Verzeichnis und allen untergeordneten Verzeichnis wurden nun alle

Rechte vergeben. (mithilfe des – werden alle jar und class Files im Ordner so wie in allen Unterordnern ausgewählt, wäre nur ein / wären alle class Files im Verzeichnis Martin ausgewählt, mit /* wären jar und class Files im Verzeichnis Martin ausgewählt)

2.3. Java RMI Tutorial

Beim Java RMI Tutorial wurde das von Prof. Borko gegebene Beispiel aus seinem GitHub Repository übernommen.
Die Aufteilung erfolgt hier in 3 packages.

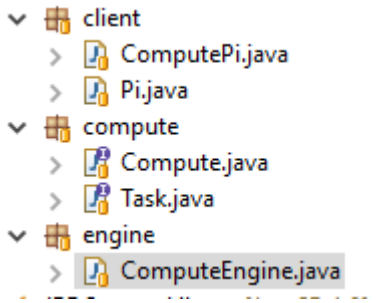


Abbildung 1 vorhandene Files

Client ist der Client PC der Tasks an den Server schicken will welcher sich in engine befindet. In Compute befinden sich die benötigten Interfaces.

2.3.1. Server

Der Server bietet die Möglichkeit, dass auf ihm Remote Methoden ausgeführt werden. Die Klasse ComputeEngine hat dafür eine Methode executeTast welche einen generischen Task als Parameter nimmt und einen generischen Datentypen zurückgibt.

```
public <T> T executeTask(Task<T> t) {
    return t.execute();
}
```

Mithilfe dieser Methode kann ein an den Server übergebener Task ausgeführt werden (t.execute()) und das Ergebnis an den Client zurückgeliefert werden.

In der Main Methode des Servers wird überprüft ob ein Security Manager vorhanden ist. Falls nicht wird ein neuer erstellt. Dieser überprüft die geltenden Policies mithilfe des java.policy Files. Um diese Remote Zugriffe zu erlauben musste das policy file wie am Anfang angegeben abgeändert werden.

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

```
            registry.rebind(name, stub);
```

Nun wird ein ComputeEngine() Objekt erstellt. Dieses wird dann Mithilfe von UnicastRemoteObject.exportObject(engine,0) auf Port 0 exportiert um auf Aufrufe reagieren zu können. Es wird eine Registry erstellt die auf Port 1099 Requests akzeptiert. Nun wird das exportierte ComputeEngine Objekt mit dem Service Namen Compute auf die Registry gebindet. Würde statt rebind die Methode bind genutzt werden würde es, falls es schon einen Dienst mit demselben Namen gibt eine AlreadyBoundException geworfen werden.

2.3.2. Compute

In Compute sind 2 Interfaces definiert. Task und Compute.
Task ist ein generisches Interface welches eine generische Methode T execute bietet um verschiedenste Tasks durchzuführen.

```
public interface Task<T> {
    T execute();
}
```

Das Interface Compute bietet nun eine Methode welche einen generischen Taks erhält einen generischen Rückgabewert hat.

```
public interface Compute extends Remote {
    <T> T executeTask(Task<T> t) throws RemoteException;
}
```

Compute erbt außerdem von Remote und die Methode wirft eine RemoteException um einen Remoteaufruf zu ermöglichen.

2.3.3. Client

Hier gibt es die Klasse Pi. Sie implementiert die beiden Interfaces Task und Serializable. Task wird mit dem Datentyp BigDecimal Implementiert. Pi bietet eine Methode execute() welche einen BigDecimal Wert zurückgibt der ein auf digits Stellen genau berechnetes Pi ist.

```
public BigDecimal execute() {
        return computePi(digits);
    }
```

Außerdem gibt es noch ein paar weitere Methoden die die Berechnung von Pi ermöglichen aber diese sind für das Protokoll und zum Verständnis von RMI unwichtig. Dadurch, dass Pi Sowohl Task<BigInteger> als auch Serializable implementiert kann es serialisiert werden und an den Server gesendet werden.

ComputePi enthält die Main Methode vom Client. Hier wird als erstes wieder mithilfe des SecurityManagers das Policy File überprüft.

```
if (System.getSecurityManager() == null) {
            System.setSecurityManager(new SecurityManager());
        }
```

Als nächster Schritt wird nun die Registry geladen. Diese bekommt einen Parameter über die Argumente welche beim Programmaufruf übergeben werden. Die Registry wird am Standardport 1099 gesucht, welchen wir auch beim Server angegeben haben.

```
Registry registry = LocateRegistry.getRegistry(args[0]);
```

Nun erhält man von der Registry eine remote Referenz auf das gebindete Compute Objekt des Servers.
Als nächster Schritt wird ein neuer Task erstellt um Pi zu berechnen. Die Genauigkeit der Stellen wird dabei über die Command Line Arguments übergeben.

```
            Pi task = new Pi(Integer.parseInt(args[1]));
```

Nun wird die eigentliche Berechnung durchgeführt indem wird dem Compute Objekt auf dem Server einen Task übergeben und diesen ausführen lassen.

```
            BigDecimal pi = comp.executeTask(task);
```

Der Aufruf der executeTask() Methode und der execute() Methode erfolgt nun am Server und es wird der für Pi berechnete Wert zurückgegeben.

2.4. Command Pattern

2.4.1. Beschreibung

Für die Umsetzung des Command Patterns wurde auch der Code von Prof. Borko übernommen und es wurden noch Anpassungen vorgenommen um ein Client callback und die Berechnung von Pi zu ermöglichen.

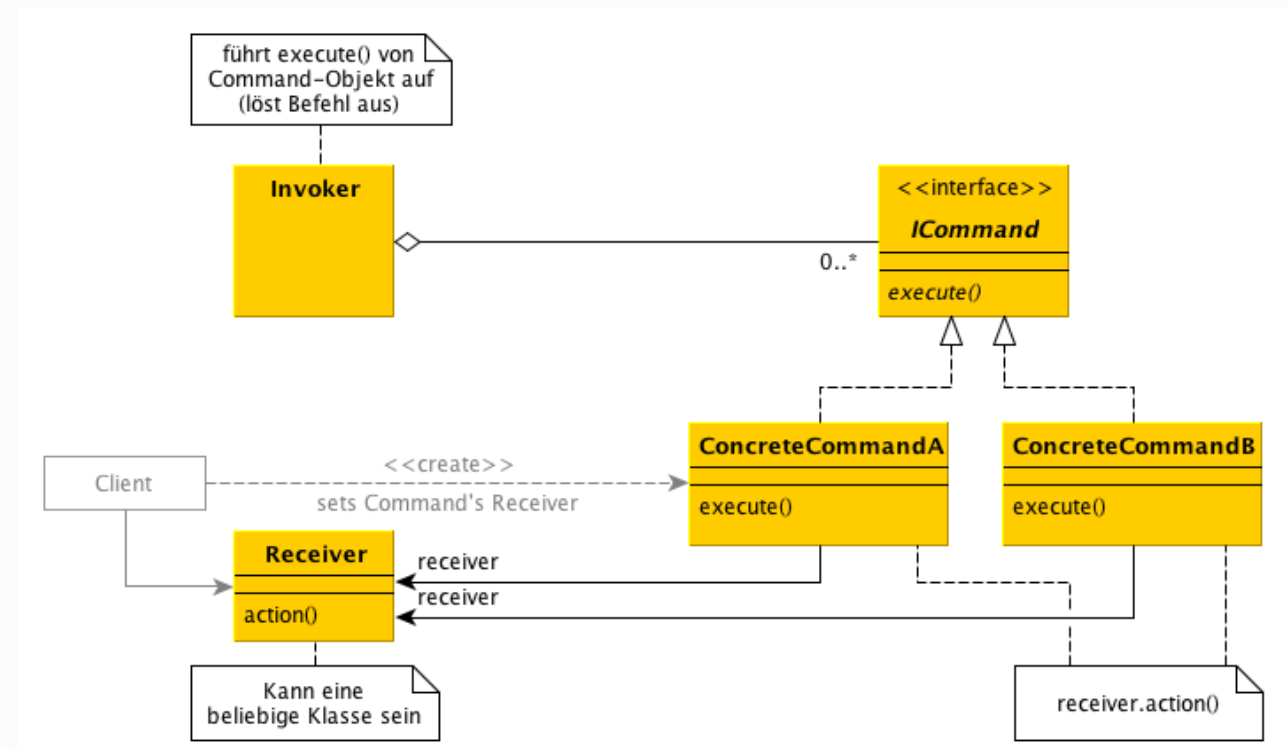


Abbildung 2 Command Pattern [PHI09]

Beim Command Pattern gibt es ein vordefiniertes Command Interface. Dieses besitzt eine Methode execute. Das Command ruft dann die Methode action() auf um bestimmte Aktionen durchzuführen. Das execute Command wurde vom Invoker aufgerufen. Dadurch sind Invoker und Reciever entkoppelt. [PHI09]

Im folgenden Beispiel wird nun das Berechnen von Pi mithilfe des Command Interfaces durchgeführt.

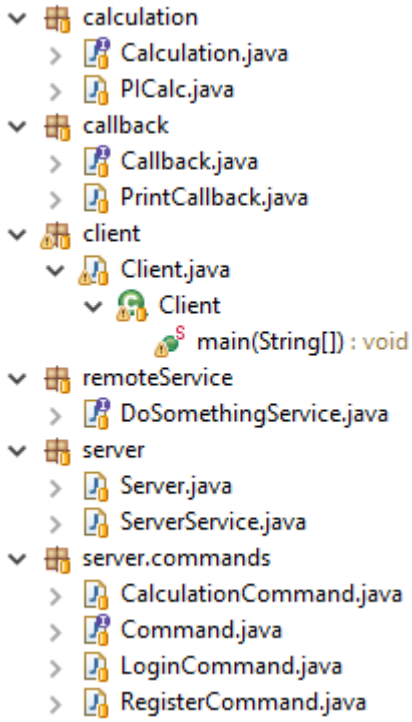


Abbildung 3 packages Command Beispiel

Hier gibt es das client package welches den Client enthält, ein calculation package welches ein Interface für Berechnungen und eine Umsetzung für die Berechnung von Pi enthält. Weiters ein callback package mit den benötigten Klassen für ein Callback sowie ein remoteService Package für den Server. Ein Server Package mit der Server Funktionalität sowie ein Package mit den verschiedensten Commands die dann auf dem Server ausgeführt werden. Auf das calculation package werde ich nicht näher eingehen da die Berechnung von Pi der vom 1. Beispiel so gut wie ident ist.

2.4.2. Callback

Hier wird die Möglichkeit geboten dem Client das Ergebnis zurückzuliefern. Er muss daher nicht darauf warten, bis die Berechnung durchgeführt wird sondern bekommt diese dann vom Server zurückgegeben.

Dafür gibt es ein generisches Interface Callback welches von Remote erbt um den Remote Zugriff zu ermöglichen. Außerdem gibt es eine Methode mit einem generische Parameter die dann vom Server aufgerufen wird um auf dem Client die gewünschte Aktion durchzuführen.

```
public void callback(T value) throws RemoteException;
```

Außerdem gibt es noch die generische Klasse PrintCallback welche es ermöglicht den erhaltenen T value auszugeben.

```
public void callback(T value) throws RemoteException {
            System.out.println(value);
      }
```

2.4.3. Server

Am Server gibt es die Klasse ServerService welche die Methode doSomething(Command c) enthält.

```
public void doSomething(Command c) throws RemoteException {
            c.execute();
      }
```

Diese Methode wird vom DoSomethingService Interface vorgegeben. Dieses erbt von Remote um die Remotezugriffe zu ermöglichen.
Diese Methode ruft dann die execute() Methode vom übergebenen Command auf.

Die Serverklasse selbst ist die main Methode.
Hier werden wieder die Java Policies geprüft. Dann wird ein neues ServerService Objekt erstellt und dieses dann mit UnicastRemoteObject.exportObject exportiert um auf remote Aufrufe reagieren zu können. Die vorgehensweise ist hier die selbe wie beim vorherigen Beispiel. Außerdem wird eine Registry auf dem Port 1234 erstellt und das Remote Object an diese gebunden.

```
      ServerService uRemoteObject = new ServerService();
                  DoSomethingService stub = (DoSomethingService)
UnicastRemoteObject.exportObject(uRemoteObject, 0);
                  Registry registry = LocateRegistry.createRegistry(1234);
                  registry.rebind("Service", stub);
```

Wird in der Kommandozeile Enter gedrückt wird das Objekt wieder entbunden und das Programm endet. Durch den Parameter true wird das unexporten geforced. D.h. auch wenn gerade ein call durchgeführt wird wird das Objekt entfernt.

```
UnicastRemoteObject.unexportObject(uRemoteObject, true);
System.out.println("Service unbound, System goes down ...");
```

2.4.4. Commands

In diesem Package sind die verschiedensten Commands die alle das Interface Command implementieren, damit der Server dann mithilfe seiner doSomething Methode das Command ausführen kann.

Die beiden Commands Login und Register sind uninteressant weil diese nur am Server ausgeben, dass sie aufgerufen worden sind.
Das CalculationCommand gibt hingegen mithilfe von Callbacks einen Wert an den Client zurück.

```
public CalculationCommand(Calculation calc,Callback<BigDecimal> cb) {
            this.calc = calc;
            this.cb=cb;
      }
```

Das Calculation Command benötigt hierfür auch eine Calculation calc die in unserem Fall dann PiCalc sein wird und ein Callback Objekt vom Typen BigDecimal.

```
@Override
	public void execute() {
		calc.calculate();
		try{
			cb.callback(calc.getResult());
			System.out.println("Calculation Command called!");
		}
		catch(RemoteException re){
			System.err.println("Exception while recieving result");
		}
	}
```

In der execute methode selbst wird dann zuerst der Wert berechnet und dann wird versucht diesen mithilfe der callback Methode an das callback Objekt zurückzuschicken. Hier kann natürlich eine RemoteException geworfen werden.

2.5. Client

Am Client wird nun auch wieder die Java Policy geprüft sowie eine Verbindung mit der Registry hergestellt und ein DoSomethingService Objekt vom Server wird mithilfe der lookup Methode erhalten.
Dann werden die beiden Beispielcommands ausgeführt welche wie schon erwähnt nur am Server eine Ausgabe tätigen.

```
Registry registry = LocateRegistry.getRegistry(1234);//Tries to locate the
registry on port 1234
		DoSomethingService uRemoteObject = (DoSomethingService)
registry.lookup("Service");//looks up service in the registry
		System.out.println("Service found");

		//two example commands
		Command rc = new RegisterCommand();
		Command lc = new LoginCommand();
```

Als nächstes wird ein Calculation Objekt erstellt welches wieder von den Command Line Arguments die Genauigkeit für Pi bekommt. Außerdem wird ein Callback Objekt pcb erstellt welches dann exportiert wird um einen Remote Call vom Server aus zum Client zu ermöglichen.

```
Calculation pi = new PICalc(Integer.parseInt(args[0]));
PrintCallback<BigDecimal> pcb=new PrintCallback<BigDecimal>();//Callback object
Callback<BigDecimal> cbstub = (Callback<BigDecimal>)
UnicastRemoteObject.exportObject(pcb, 0);//exported stub callback object
Command pc = new CalculationCommand(pi,cbstub);//Command for calculation Pi
```

Nun werden die einzelnen Commands ausgeführt.

```
uRemoteObject.doSomething(rc);
uRemoteObject.doSomething(lc);
uRemoteObject.doSomething(pc);
```

Um das Programm beenden zu können wird das Callback Objekt wieder unexported:

```
UnicastRemoteObject.unexportObject(pcb, true);//removes callback so it can no longer recieve incoming RMI calls
```

3. Zeitaufzeichnung

Schätzung:

Aufgabenstellung	Dauer
Java RMI	3h
Command Pattern	3h
Client-Callback	2h
Protokoll	2,5h
Gesamt	10,5h

Wirklich benötigt:

Aufgabenstellung	Dauer
Java RMI	3h
Command Pattern	1,5h
Client-Callback	1h
Protokoll	3
Gesamt	8,5h

4. GitHub Link

https://github.com/mweber-tgm/JavaRMI

5. Für das Protokoll benutzte Quellen

[BOR16] Michael Borko 2016 "Beispiel Konstrukt für Command Pattern mit Java RMI"; Michael Borko; online: https://github.com/mborko/code-examples/tree/master/java/rmiCommandPattern [abgerufen am 06.04.2016]

[BOR16] Michael Borko 2016 "Beispiel Konstrukt für Java RMI"; Michael Borko; online https://github.com/mborko/code-examples/tree/master/java/rmiTutorial [abgerufen am 06.04.2016]

[PHI09] Philipp Hauer (2009-2010) Das Command Design Pattern [Online] Available at: http://www.philipphauer.de/study/se/design-pattern/command.php [abgerufen am 07.04.2016]