JUST DO IT
HÖHERE ABTEILUNG FÜR INFORMATIONSTECHNOLOGIE

tgm
Die Schule der Technik

Laborprotokoll

Dezentrale Systeme [DezSys]

Verteilte Systeme über Java RMI

Git Repo: https://github.com/pkogler-tgm/DezSys-Distributed-Systems-rmi

---

Systemtechnik Labor

4CHITT 2015/16, Gruppe B

KOGLER Philipp

Version 0.6

Note:

Betreuer: BORKO Michael

Begonnen am 11. März 2016

Beendet am 18. März 2016

1 CONTENTS

2 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.

2.1 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..

2.2 VORAUSSETZUNGEN

Grundlegendes Verständins von Java und Verteilten Systemen.

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

2.3 AUFGABENSELLUNG

1. 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.

1. 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.

3 RMI – ALLGEMEIN (OVERVIEW)

Literaturverzeichnis [1], [2]

RMI (Remote Method Invocation) beschreibt den Aufruf einer Methode eines entfernten Java – Objektes. Dabei kann sich das Objekt in einer vollkommen anderen JVM befinden.

Meistens wird RMI als Client / Server Prinzip umgesetzt, das heißt wir haben einen Client, welcher eine bestimmte (meist komplexe) Tätigkeit, erledigen will. Dafür stellen wir ihm einen Server bereit, welcher die Ressourcen und die Mittel dazu hat diese Tätigkeit zu übernehmen.

Ein typisches server Programm erstellt mehrere remote Objects, referenziert auf diese bzw. übermittelt sein Skeleton, und wartet bis diese Clients seine Methoden aufrufen.

Ein typisches client Programm erhält eine reference eines remote Servers zu einem oder mehreren remote Objekten und ruft anschließend mittels Sockets über das Skeleton des Servers die gewünschten Methoden auf.

RMI bietet (in Java) den Grundmechanismus bei dem der Server und der Client, untereinander Informationen austauschen können (meist unter Verwendung von Sockets). So eine Art von Programm wird oft als distributed Object application bezeichnet.

3.1 WAS BIETET UNS RMI

- **LOCATE REMOTE OBJECTS**
  Es gibt zahlreiche Algorithmen um an Referenzen von remote bzw. Externen Objekten zu gelangen. Beispielsweise kann ein Programm seine remote Objects “registern” mithilfe der **REGISTRY.**
  Ebenfalls können Applikationen remote Objects als Teil eines Methoden Aufrufes referenzieren.

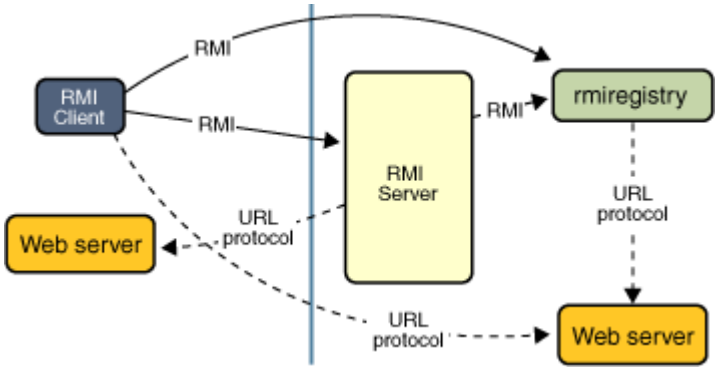
- **COMMUNICATE WITH REMOTE OBJECTS**
  Prinzipiell wird die ganze Kommunikation zwischen remote Objects von RMI umgesetzt. Für uns sieht die Kommunikation ähnlich wie ein simpler Methoden – Aufruf aus.

- **LOAD CLASS DEFINITIONS FOR OBJECTS THAT ARE PASSED AROUND**
  Da es möglich sein muss Objekte hin und her zu schieben. Es bietet die Möglichkeit Objekte zu laden, Klassen Definitionen sowie übermitteln der Daten eines Objektes.

3.2 AUFBAU RMI MIT REGISTRY

Folgende Abbildung beschreibt eine Applikation welche den Mechanismus von RMI implementiert.

Es wird eine RMI Registry verwendet um die Refernzen eines Remote Objects zu erlangen. Der Server ruft die Registry auf bzw binded sie. Somit wird gleichzeitig ein Name für das Remote Object gebindet.



Der Client sucht nach dem gebindeten Remote Object in der Registry des Servers und invoked anschließend die gewünschte Methode/Methoden davon.

Ebenfalls wird gezeigt, dass das RMI system einen existierenden Web Server benützt um Klassen Definitionen zu laden. Von dem Server auf den Client und von dem Client auf dem Server wenn gebraucht.

3.3 REMOTE INTERFACES, OBJECTS AND METHODS

Wie jede andere Java Applikation sollte auch ein RMI basierendes Programm korrekt mittels Interfaces und dazugehörigen Klassen aufgebaut sein. Die Klassen implementieren die Methoden, welche explizit in den Interfaces deklariert sind. Generell werden Objekte, welche Methoden haben die invoked werden können, remote Objects genannt.

Ein Objekt wird als remote gekennzeichnet, wenn dieses rein remote Interface implementiert. Dieses Interface muss folgende Kriterien erfüllen:

- A remote interface extends the interface `java.rmi.Remote`.
- Each method of the interface declares `java.rmi.RemoteException` in its `throws` clause, in addition to any application-specific exceptions.

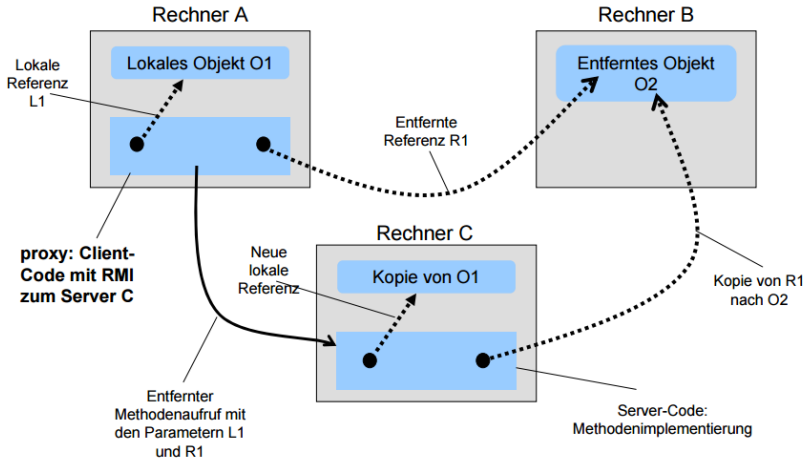
RMI behandelt remote Objekte anders als Objekte in der eigenen JVM. Es wird keine komplette Kopie des Objektes erstellt, sondern jedeglich wird ein remote stub für ein Remote Object übergeben. Dieser Stub verhält sich ähnlich wie ein Proxy. Für den Cleint ist es im Prinzip die remote Reference zu diesem Objekt.

Deises Skeleton ist verantwortlich die Methoden aufrufe an das gewünschte Objekt weiter zu liefern.
Das Skeleton implementiert die gleichen remote Interfaces wie das remote Object, das erlaubt uns diesen Stub auf jedes Interface, welches remote implementiert, zu casten.

3.4 REFERENZVERWALTUNG

Wie bereits beschrieben warden remote Objects nicht direkt kopiert, sondern jedeglich die Referenz zu dem Skeleton bzw. zu dem Objekt wird übermittelt.

Lokale Objekte werden als Kopie übergeben. Remote Objekte werden als Referenz übergeben.



3.5 SERILALISIERUNG / MARSHALLING

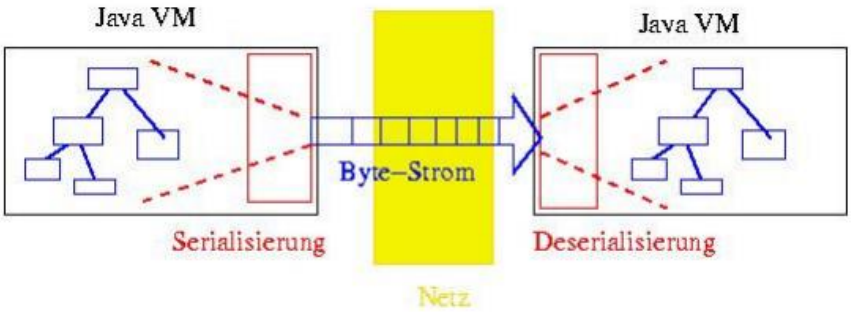
Literaturverzeichnis [5]

RMI war einer der Hauptgründe warum die Serilisierung damals in Java implementiert wurde.

Methoden Parameter könne auf zwei Arten verschickt werden:

1. Das entfernte Objekt wird als Remote Reference transportiert (das Objekt selbst "verlässt" dabei die JVM nicht, java.rmi.Remote)

2. Das entfernte Objekt wird als Kopie übertragen (entspricht einem "call by value", java.lang.Serializable)



Um nun das Call by Value Prinzip zu nutzen und die Objekt – Kopienzu anderen JVMs zu übertragen müssen die Datenstrukturen beim Sender in einen Datenstrom umgewandelt werden und beim Sender aus dem transportierten Datenstrom die Datenstrukturen rekonstruiert werden. Um dies umzusetzen bietet Java eine Serialisierungs-API. `java.io.Serializable`

Diese behinhalte folgendes:

- Alle Grundtypen sowie die meisten Klassen des JDK implementieren das Interface Seriliazable.
- Instanzen von nicht – serialisierbaren Klassen in einem zu serialisierenden Objekt – Graph führen zu einer NotSerializableException.

Bei RMI müssen Objekte welche Call by Value übertragen werden auf jeden Fall mit Serializabel gemarked sein!

3.5.1 TRANSFORAMTION EINES OBJEKTES IN EINEN BYTE-STREAM

„Das Objekt besteht aus einem Zustand (Inhalt seiner Objekt- und KlassenVariablen) und einer Implementierung (= class-Datei). Die class-Datei muss bekannt sein, also entweder über CLASSPATH lokal oder über das Netz. Der Zustand ist dynamisch und nur zur Laufzeit bekannt. Es wird immer ein "Schnappschuss" dieses Zustands übertragen.“

3.5.2 UNTERSCHIEDE ZWISCHEN REMOTE UND SERIALIZABLE OBKELTEN

Wenn sie als Argumente oder Ergebnis-Parameter von RMI-Aufrufen vorkommen, werden

• Serializable -Objekte "by value" übertragen
• Remote -Objekte "by reference" übertragen. Alle Felder von Remote-Objekten sind transient.

Werden Remote-Objekte z.B. auf ObjectOutputStreams geschrieben, gehen die Werte der Felder verloren!

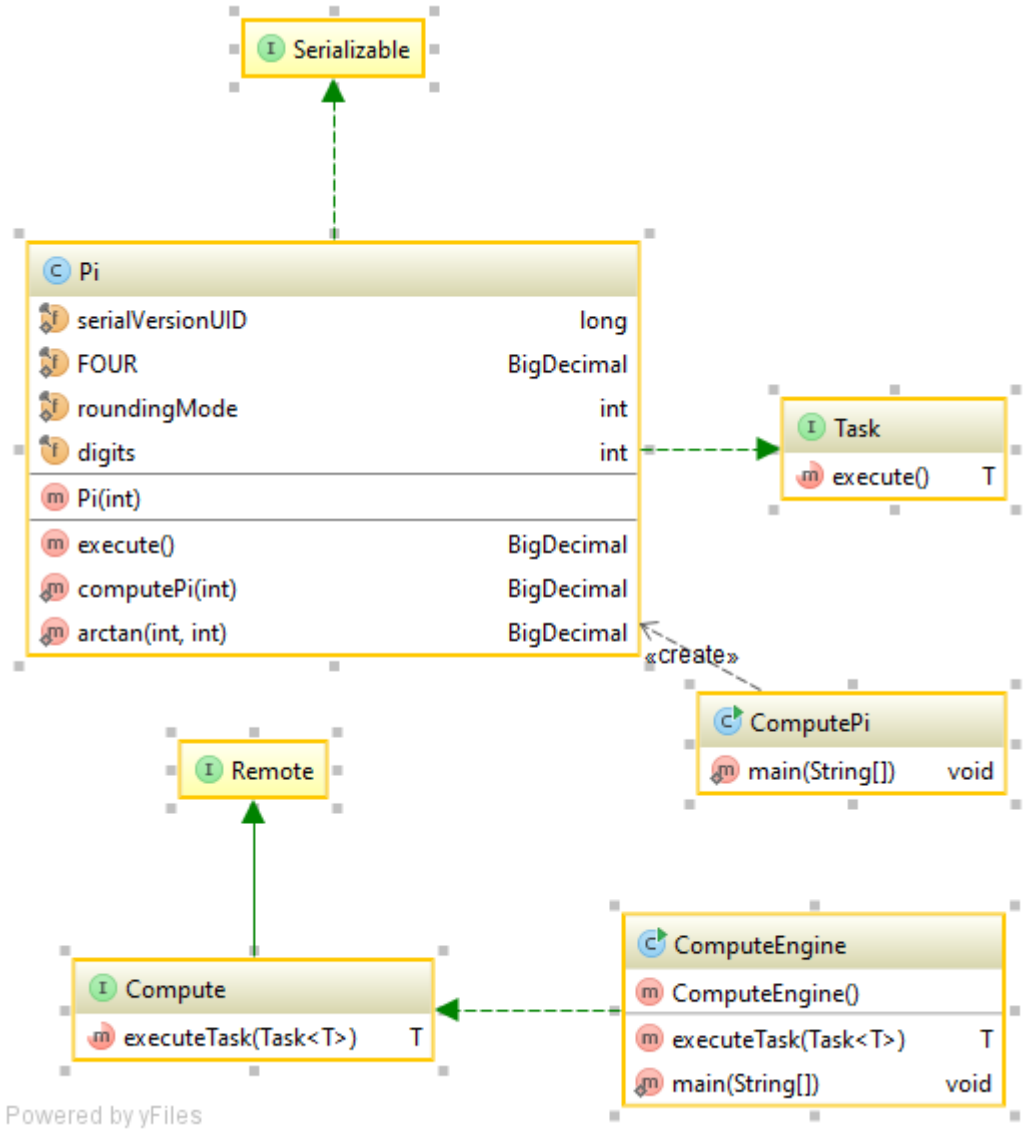
4 ERGEBNISSE

4.1 AUFGABENSTELLUNG 1 [RMI TUTORIAL IMPLEMENTIEREN]

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.

4.1.1 BEREITGESTELLTES GIT REPO CLONEN

Git: https://github.com/mborko/code-examples/tree/master/java/rmiTutorial



Folgende Klassen werden in dem Tutorial erstellt:

4.1.2 ERKLÄRUNG DER KLASSEN / INTERFACES

- **Client.ComputePi.class** **Client**

Lokalisiert die Registry

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

Verwendet registry um den gewünschten Eintrag zu finden

```
Compute comp = (Compute) registry.lookup(name);
```

Startet den task

```
Pi task = new Pi(Integer.parseInt(args[1]));
BigDecimal pi = comp.executeTask(task);
```

- **Client.PI.class** **Task**

Task um Pi zu berechnen. Dieser wird als remote Object Call by Value an den Server versandt. Deshalb muss dieser auch auf jeden Fall Serialiazable implementieren

- **Compute.Compute** **Interface**

Muss Remote extenden gibt execute Methode vor.

- **Compute.Task** **Interface for the Task**

Generic execute Aufruf

- **Engine.ComputeEngine.class** **Server**

Implementiert Compute und startet den Task.

Erstellt den Server Stub (Skelleton)

```
Compute stub = (Compute) UnicastRemoteObject.exportObject(engine, 0);
```

Erstellt und bindet die Registry auf einem spezifiziertem Namen

```
Registry registry = LocateRegistry.createRegistry(1099);
registry.rebind(name, stub);
```

4.1.3 SETZTEN / KONFIGURIEREN DER JAVA POLICY

Die Java Policy muss leicht abgeändert werden um den Security Manager zu verwenden. Sowohl Server als auch Client haben einen Security Manager installiert.

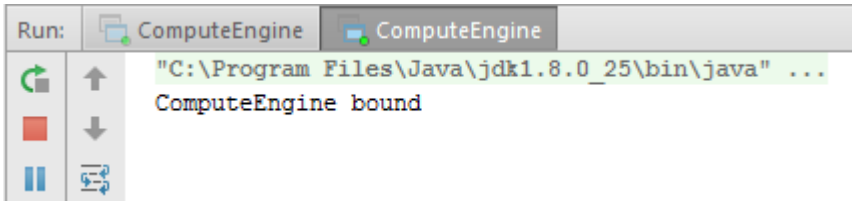
Um dies zu umgehen muss ein Java Policy File angegeben werden. Das Java Policy File schaut wie folgt aus:

```
grant codeBase "file:/home/pkogler/src/" {
    permission java.security.AllPermission;
};
```

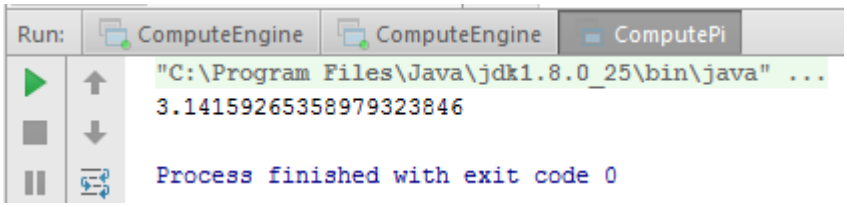
Wir granten für Server und Client AllPermissions, da den lokalen Pfaden des programmes vertraut werden kann. Beispielsweise die Komunikation der Sockets innerhalb des Security Managers nicht funktionieren würde.

4.1.4 STARTEN DES SERVERS / CLIENTS

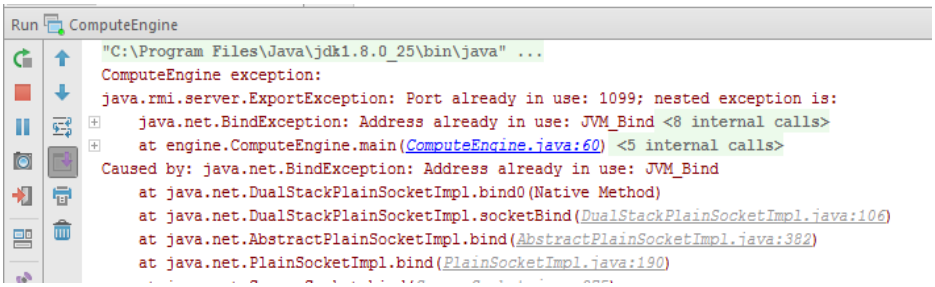
Zuerst muss der Server gestartet werden:



Anschließend kann der Client mit den gewünschten Argumenten gestartet werden:
Localhost anzahl der Pi Stellen



Wenn bereits eine Registry läuft kommt folgende Exception:



Wenn der Client keine Registry findet wird folgende Exception geworfen:

4.2 AUFGABENSTELLUNG 1 PROBLEME

Bei der Implementierung der Aufgabenstellung ist ein gröberer Fehler aufgetreten, welcher trotz simpler Ursache einen extremen zeitverzug verursachte.

Problemstellung:

Es wurde, trotz korrektem Source Code und Kompilierung aller Klassen, eine ClassNotFoundException geworfen. Es wurden keine imports bzw keine Klassen aus anderes Packages gefunden.

Nach „stundenlanger“ Suche nach dem vermeidlichen Fehler, wurde das Problem mithilfe des Betreuers Michael Borko gefunden und konnte beseitigt werden.

Ursache:

Da die Registry manuell gestartet wurde, wurde diese mit einer anderen JVM ausgeführt als der Programmcode. Fehlerhafterweise hat java eine ClassNotFoundException geworfen, welche den Fehler nicht beschreiben konnte.

Lösung:

Die Registry muss „per Hand“ im Programmcode gestartet werden. Mit dem Befehl:

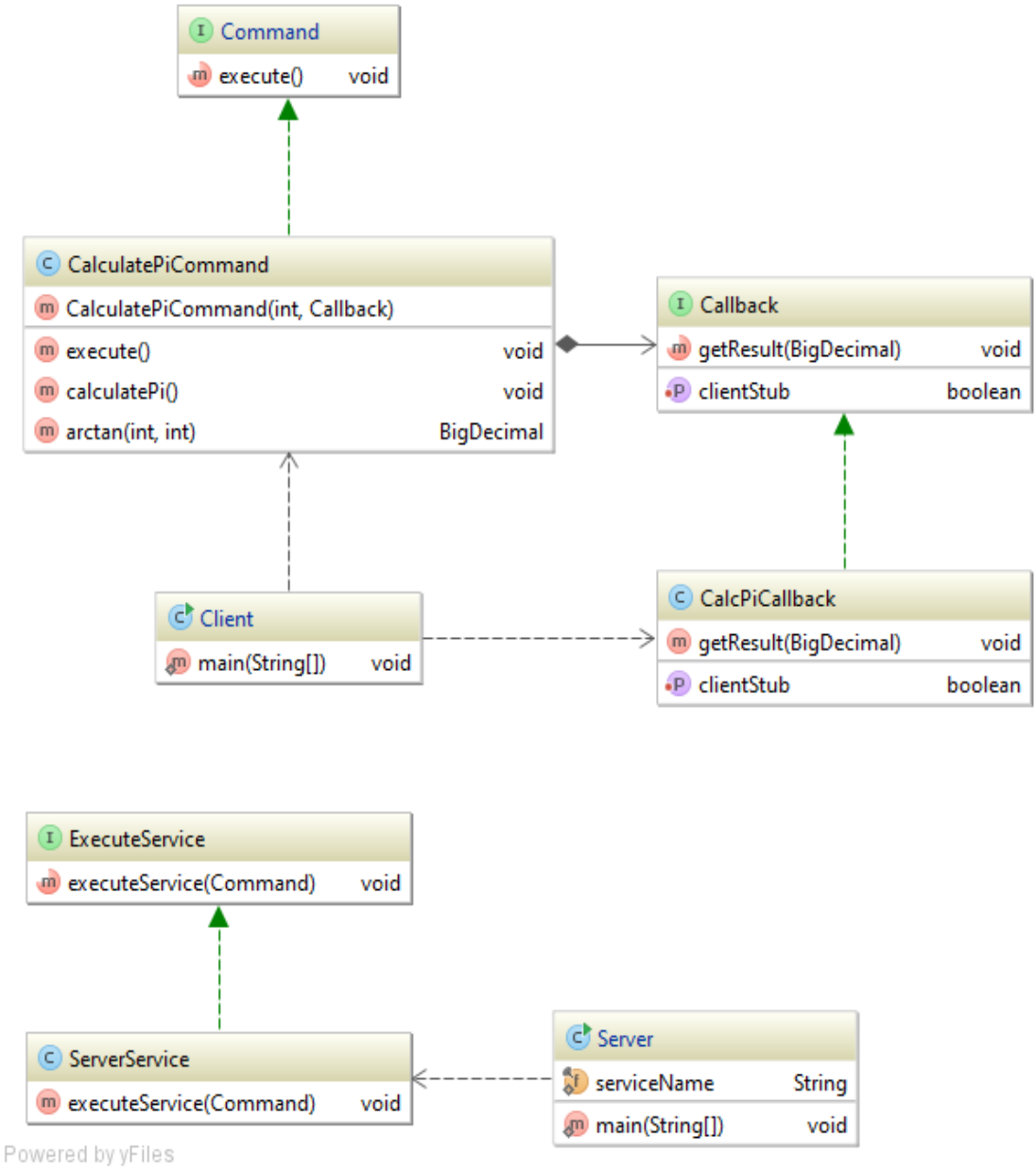
```
Registry registry = LocateRegistry.createRegistry(1099);
```

Aufgbrachte Zeit: ca 2 Stunden.

4.3 AUFGABENSTELLUNG 2 IMPLEMENTIERUNG MIT DEM COMMAND PATTERN

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.

4.3.1 AUFTEILUNG DER KLASSEN (UML – KLASSEN DIAGRAM)



4.3.2 ERKLÄRUNG DER KLASSEN / INTERFACES

- **Client.Client.class** **Client Class**

Lokalisiert die Registry

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

Verwendet registry um den gewünschten Eintrag zu finden

```
Compute comp = (Compute) registry.lookup(name);
```

Implementiert ein Callback Interface

Executet den Command

```
Pi task = new Pi(Integer.parseInt(args[1]));
BigDecimal pi = comp.executeTask(task);
```

- **clientCallback.Callback | CalcPiCallback** **Interface Remote | Implementation**

Extended Remote und stellt 2 Methoden zur Verfügung

- **getResult()**
  printet das Ergebnis und unexported den Client Stub, damit der Prozess vollständig geschlossen werden kann.
- **setClientStub()**
  Setter Method für das boolean Attribut clientStub

- **Command.Command** **Interface**

Muss Serilizable extenden gibt execute Methode vor.

- **Command.CalculatePiCommand** **Implementation for Command**

Berechnet Pi und sendet, wenn Aufgabe ausgeführt, das Ergebnis an den Callback Stub des Clients.

- **Server.Server** **Server Class**

Implementiert Compute und startet den Task.

Erstellt den Server Stub (Skelleton)

```
Compute stub = (Compute) UnicastRemoteObject.exportObject(engine, 0);
```

Erstellt und bindet die Registry auf einem spezifiziertem Namen

```
Registry registry = LocateRegistry.createRegistry(1099);
registry.rebind(name, stub);
```

4.3.3 IMPLEMENTIERUNG DES COMMAND – PATTERN

4.3.3.1 COMMAND – PATTERN ALLGEMEIN

Definieren einer Klasse welche einen Pointer auf ein Objekt, einen Pointer auf eine Funktion und all die dazu benötigten Argumente

Also ein Objekt wird verwendet um alle benötigten Informationen zu enkapseln um eine Aktion auszuführen oder ein Event zu triggern.

Diese Information beinhaltet:

- Den Methoden Namen
- Das Objekt welche die Methode implementiert
- Und die Werte, welche für die Methoden Paramter gebraucht werden.

Das Command Pattern basiert auf ein Command einen Receiver, invoker sowie einen Client. Ein Command Objekt weiß uber der Receiver bescheid und invoked eine Methode des Receivers. Werte für die Methode werden im Command gespeichert. Ab hier verarbeitet der Receiver den weiteren Vorgang des Geschehens.

4.3.3.2 IMPLEMENTIERUNG MITTELS RMI

Im Client wird ein Command aufgerufen → CalculatePiCommand. Dieser Command übernimmt als Parameter in unserem Fall die Stellen von Pi und das Callback Interface.

Über das ServerService wird der Command dann ausgeführt → executeCommand. Nun wird die Methode execute() von CalculatePiCommand aufgerufen. Diese berechnet Pi und gibt anschließend das Ergebnis an das Callback Interface zurück

4.3.4 IMPLEMENTIERUNG DES CLIENT – CALLBACK

4.3.4.1 WARUM BRAUCHEN WIR ÜBERHAUPT EIN CALLBACK

Der Callback in unserem Fall beschreibt einfach einen Stub am Client zu dem der Server bzw. das Skelleton also der Stub des Servers kommunizieren kann wenn dieser mit der Berechnung bzw. mit der Method invokation fertig ist.

Dies bringt uns einen großen Voreteil, da wir so Asynchron am Client weiter arbeiten können. Wenn der Server sich erst meldet, wenn die Ausführung der Methode abgeschlossen ist, muss der Client nicht die ganze Zeit auf das Result warten und kann somit andere Operationen erledigen.

4.3.4.2 IMPLEMENTIERUNG DES CLEIENT – CALLBACK

Callback stub am Client einrichten:

```
Callback callbackStub = (Callback) UnicastRemoteObject.exportObject(callback,0);
```

Callback stub muss dem Command als Paramter mitgegeben werden, damit das Result wieder zurückversandt werden kann. Unexporting des Stubs wenn Ergebnis ausgegeben mit `UnicastObject.unexportObject(this);`

5 ZEITSCHÄTZUNG

Fertigst.	Beschreibung	Geschätzte Zeit	Benötigte Zeit
4.12.2015	RMI Tutorial lauffähig machen	180 min	240 min
4.12.2015	Command Pattern umsetzten	60 – 90 min	120 min
4.12.2015	Client Callback implementieren	90 min	30 min
4.12.2015	Dokumentation Programmcode	30 min	30 min
6.12.2015	Protokoll / Dokumentation	100 min	200 min
	Ingesamt	**490 min**	**620 min**

6 LITERATURVERZEICHNIS

[1] "The Java Tutorials - Trail RMI"; online:
http://docs.oracle.com/javase/tutorial/rmi/

[2] Dezsy04_distributedObjects_RMI.pdf
https://elearning.tgm.ac.at/mod/resource/view.php?id=48172

[3] "Command Pattern"; Vince Huston; online:
http://vincehuston.org/dp/command.html

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

[5] TI Uni Tuebingen
http://www.ti.uni-tuebingen.de/fileadmin/assets/csp_ws0809/aufgabe2/aufgabe2a.pdf

[6] Git Repository pkogler
https://github.com/pkogler-tgm/DezSys-Distributed-Systems-rmi