Laborübung zu Verteilte Systeme : Aufgabe zu Java-RMI (1)

Aufgabenstellung :

Es ist ein über RMI aktivierbarer **generischer Problemlösungs-Server** in Java zu entwickeln.
Kern des Servers ist ein mittels RMI ansprechbares entferntes Objekt, das prinzipiell beliebige Aufgaben (= zu lösende Probleme) von Clients entgegennimmt, diese bearbeitet und das erzeugte Ergebnis zurückliefert.
Die jeweils zu bearbeitende Aufgabe wird durch ein Objekt beschrieben (Aufgabenobjekt) , das der Problemlösungsfunktion des Server-Objekts vom Client als Parameter übergeben werden muß.
Die Problemlösungsfunktion des Server-Objekts ruft ihrerseits die im Aufgabenobjekt implementierte eigentliche Bearbeitungsfunktion auf. Da bei RMI bei Bedarf auch der Code von (Parameter- und Rückgabe-) Objekten übertragen wird, muß die eigentliche Bearbeitungsfunktion dem Server a priori nicht bekannt sein.
Der Server ist somit in der Lage beliebige Aufgaben zu bearbeiten, die zum Zeitpunkt seines Starts (und erst recht zum Zeitpunkt seiner Erstellung) noch nicht einmal definiert sein müssen. Einzige Voraussetzung ist, daß die Aufgabenobjekte ein Interface implementieren, über das ihre eigentliche Bearbeitungsfunktion aufgerufen werden kann.
Clients können somit die Bearbeitung von Aufgaben, die ihre eigenen Fähigkeiten übersteigen (Zeitbedarf, Resourcen), auf den Problemlösungs-Server auslagern.
Der Problemlösungs-Server soll beispielhaft zur Ermittlung der Zahl PI mit einer angebbaren Genauigkeit eingesetzt werden. Hierfür ist ein geeigneter Client zu entwickeln.

Hinweise zur Lösung (Interfaces und Server-Seite) :

- Die Schnittstelle zum Aufruf der Server-Funktionalität wird durch das Remote-Interface **ProblemSolver** beschrieben. Es deklariert– als einzige Komponente – die vom Client aufrufbare Problemlösungsfunktion

  Object solveProblem(ProblemTask t);

  Da in Java alle Klassen von der Klasse Object – direkt oder indirekt – abgeleitet sind, kann eine konkrete Implementierung dieser Funktion ein beliebiges Ergebnis durch ein Objekt beliebigen Typs zurückgeben.

- Der Problemlösungsfunktion solveProblem() wird vom Client das Aufgabenobjekt als Parameter vom Typ **ProblemTask** übergeben.ProblemTask ist ein – non-remote – Interface, das – als einzige Komponente – die Bearbeitungsfunktion

  Object executeTask();

  deklariert.
- Das tatsächliche Aufgabenobjekt muß Instanz einer Klasse sein, die dieses Interface ProblemTask implementiert. Jede derartige Klasse repräsentiert eine zu bearbeitende konkrete Aufgabe. Die Bearbeitung der Aufgabe wird von der von der Klasse definierten Funktion executeTask() realisiert.

- Da das Aufgabenobjekt – als Kopie – vom Client zum Server übertragen wird, muß das Interface ProblemTask vom Interface Serializable abgeleitet sein.

- Beide Interfaces – ProblemSolver und ProblemTask – werden sowohl auf der Client- als auch auf der Server-Seite benötigt.

- Auf der Server-Seite wird das Interface ProblemSolver durch die Klasse **ProblemSolverImpl** implementiert. Diese Klasse definiert die Problemlösungsfunktion solveProblem(), die – als einzige wesentliche Aktivität – die Funktion executeTask() für das ihr als Parameter übergebene Aufgabenobjekt aufruft. Sie gibt deren Rückgabewert (Objekt beliebigen Typs) als eigenen Rückgabewert zurück

- Die Klasse ProblemSolverImpl definiert auch – in ihrer statischen main()-Methode – die Start-Funktionalität des Servers. Diese umfaßt : Installation eines RMISecurityManagers, falls noch kein SecurityManager in der JVM installiert ist.
  Der **muss** installiert werden, wenn **vom** Client/Server aus **auf** eine externe Url zugegriffen werden soll.

  Anmerkung: wird später der Client/Server gestartet kann in der Kommandozeile mittels: -Djava.security.policy=file:///<Laufwerksbuchstabe>:\<verzeichnis>/ angegeben werden wo sich die Datei .java.policy befindet, wenn sie nicht standardgemäß im Heimatverzeichnis des Benutzers Linux: /home/<Benutzer> Windows: C:\Dokumente und Einstellungen\<Benutzer> abgelegt ist.
  Beispiel Inhalt der Textdatei .java.policy unter Windows:

```
grant
{
  permission java.net.SocketPermission "*:1024-65535", "connect,accept";
  permission java.net.SocketPermission "*:80", "connect";
  permission java.io.FilePermission "C:\\-", "read";
  permission java.io.FilePermission "${user.home}\\-", "read";
};
```

  1.Zeile: * bedeutet nach überall hin ist erlaubt ;-))
  3.Zeile: Wenn ausprobiert wird bei Codebaseangabe ohne http:// sondern die Codebase dann über das Filesystem ansprechbar sein soll: Das '-' heißt alle Subdirectories sind auf C:\ zugänglich.

- Erzeugung eines Objekts der – eigenen – Klasse ProblemSolverImpl.
- Registrierung dieses Objekts bei der RMI-Registry unter dem Namen "ProblemSolver".
- Ausgabe des Textes "ProblemSolver bound in registry" an der Konsole (= Standardausgabe).

Hinweise zur Lösung (Client) :

- Die zu bearbeitende Aufgabe – Ermittlung der Zahl PI mit einer angebbaren Genauigkeit – wird durch die Klasse **PiTask** definiert. Diese Klasse implementiert das Interface ProblemTask. Sie besitzt die folgenden Komponenten:
- Eine **Datenkomponente** vom Typ **double**, die die gewünschte Genauigkeit aufnimmt.
- Einen **Konstruktor** mit einem Parameter vom Typ double. Der Konstruktor initialisiert die Genauigkeits-Datenkomponente mit diesem Parameter.
- Die die Bearbeitungsfunktionalität realisierende Memberfunktion **Object executeTask();** Ein – auf der Potenzreihenentwicklung für den arctan beruhender – Algorithmus zur iterativen Ermittlung der Zahl PI mit der Genauigkeit genauigkeit kann wie folgt beschrieben werden :

```
pi_viertel = 1.0;
inc = 1.0;
inv_inc = 1/inc;
fak = 1;
while (4*inc > genauigkeit)
{  inv_inc += 2;
   inc = 1/inv_inc;
   fak = -fak;
   pi_viertel += fak*inc;
}
pi = 4*pi_viertel;
```

- Der Rückgabewert von executeTask() muß ein Objekt (und kein Wert eines elementaren Datentyps) sein. Der ermittelte Wert für PI (double-Wert pi) muß daher in einem Objekt der Wrapper-Klasse Double gekapselt werden. Dies kann erfolgen mit : new Double(pi);
- Der Client wird durch die Klasse CompPi implementiert. Diese Klasse besitzt nur eine statische main()-Methode, in der die gesamte Client-Funktionalität zusammengefaßt ist. Diese Funktionalität besteht aus :
  - Installation eines RMISecurityManagers, falls noch kein SecurityManager in der JVM installiert ist (Siehe Server).

- Überprüfung auf die richtige Anzahl von Kommandozeilenparametern.. Das Client-Programm muß mit zwei Kommandozeilenparametern aufgerufen werden : Server (Name oder IP-Adresse) und gewünschte Genauigkeit.

Bei einer falschen Anzahl von Kommandozeilenparametern ist ein Hinweis auf das richtige Aufrufformat an der Konsole (=Standardausgabe) auszugeben und das Programm zu beenden.Bei einer richtigen Anzahl von Kommandozeilenparametern sind die folgenden weiteren Aktivitäten auszuführen :

- Erfragen einer lokalen Referenz auf das entfernte RMI-Server-Objekt bei der RMI-Registry
- Ermittlung der internen Darstellung des Zahlenwerts der Genauigkeit (Typ double) aus dem entsprechenden Kommandozeilenparameter:

```
double prec = Double.parseDouble(args[1]);
```

- Erzeugung eines Objekts der Klasse PiTask (Aufgabenobjekt).
- Aufruf der Methode solveProblem() für die erhaltene Referenz auf das RMI-Server-Objekt unter Übergabe des erzeugten Aufgabenobjekts als Parameter. Der Rückgabewert dieses Methodenaufrufs ist ein Objekt der Wrapper-Klasse Double, das den ermittelten Wert für PI (Typ double) kapselt.
- Entkapselung des ermittelten Werts res für PI aus dem Rückgabeobjekt resObj (der Klasse Double):

```
double res = resObj.doubleValue();
```

- Ausgabe des Ergebnisses an der Konsole (= Standardausgabe), begrenzt auf die genaue Stellenanzahl digits.
- Diese läßt sich aus der Genauigkeit prec wie folgt ermitteln :

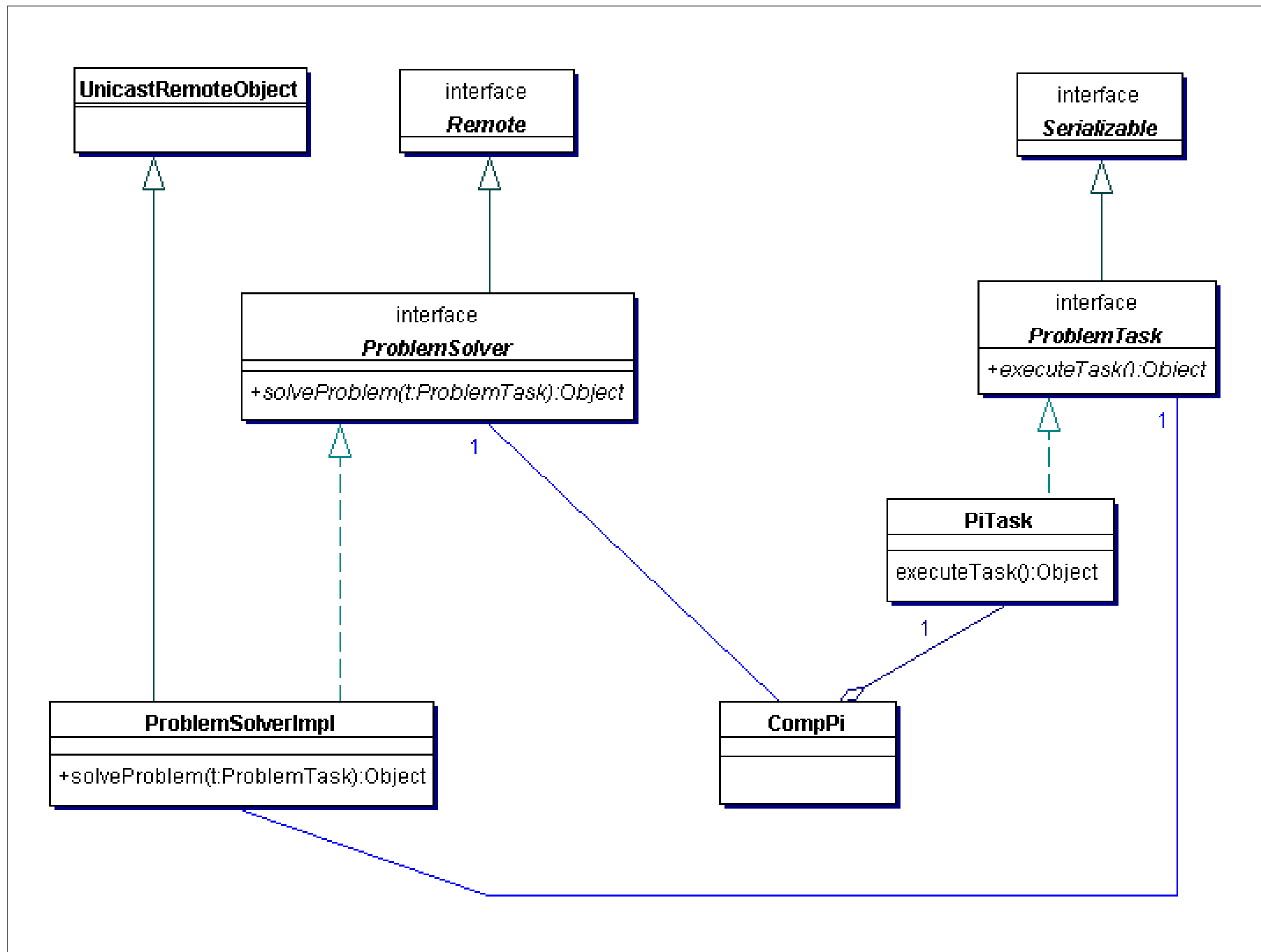
```
int digits = -(int)Math.round(Math.log(prec)/Math.log(10.0))-1;
```

- Die Ergebnisausgabe kann dann erfolgen mittels :

```
System.out.println((new
BigDecimal(res)).setScale(digits,BigDecimal.ROUND_HALF_UP));
```

Klassendiagramm zur Lösung :

Die statischen Beziehungen der für die Lösung benötigten Klassen (Server- und Client-Seite) können durch das folgende Klassendiagramm beschrieben werden :



Anmerkungen zur Entwicklungsumgebung in Labor:

- Die Entwicklung findet unter **Windows 2000/XP** statt.
- Als Entwicklungswerkzeuge stehen nur der Editor **MPEDIT** und die Tools des Java Development Kits (**JDK**) 5.x/6.x zur Verfügung.
- Der Editor Sicite ist als Windows-Applikation zu starten (Icon auf Desktop)
- Die Tools des JDK müssen über Kommandozeilenaufrufe aus einem Konsolen-Fenster (CMD-Box) gestartet werden.

- Machen Sie ein CMD-Fenster auf (Start->Ausführen:cmd) **und wechseln Sie nach L:\Projects\.**
- Kontrollieren Sie im CMD Fenster , ob
  - die Environment-Variable **PATH** u.a. den Pfad zum Aufruf der JDK-Tools enthält (**D:\jdk1.x\bin**)
  - die Environment-Variable **CLASSPATH** auf das aktuelle Arbeitsdirectory (**.**) gesetzt ist

- Legen Sie im – vorhandenen – Directory **L:\Projects** die Directories **Server** und **Client** an.
- Das Directory **L:\Projects\Server** ist das **Arbeits-Directory** für die Entwicklung und Start des **Servers**.
- Das Directory **L:\Projects\Client** ist das **Arbeits-Directory** für die Entwicklung und Start des **Clients**.
- Legen Sie in beiden Arbeits-Directories jeweils die Directory-Struktur **vs\rmiprobsolv** an.

- Alle vom Editor und den JDK-Tools erzeugten Dateien für den Server bzw den Client sind in dem jeweiligen Directory **rmiprobsolv** abzulegen.
- Das Directory **L:\public_html** ist das **Directory** für die Bereitstellung der .class Dateien die der Client von Server benötigt un umgekehrt. Auch hier erwartet der Cleient/Server die Directory-Struktur **vs\rmiprobsolv . Legen Sie die Directory-Struktur deshalb an.**
- **Vergessen Sie nicht in jeder ihrer Klassendateien .java das package Statement anzugeben:**
  ```
  package vs.rmiprobsolv;
  ```
  Ansonsten werden die Dateien bei der Ausführung nicht gefunden

Anmerkungen zur Realisierung :

Allgemeines :

- **Jedes** Interface und jede Klasse muß in einer **eigenen Datei** definiert werden. Jede Quell-Datei muß den jeweiligen Klassennamen als Hauptnamen und die Extension **.java** tragen.
- Alle Interfaces und Klassen sollen zum Package **vs.rmiprobsolv** gehören. → jede Java-Quelldatei muß die folgende Package-Deklaration enthalten :

  package vs.rmiprobsolv;
- Jede Java-Quellcode-Datei ist getrennt mit dem Java-Compiler **javac** zu übersetzen
  → Erzeugung von Java-Bytecode-Dateien, Extension **.class**.
  Ausgehend vom jeweiligen Arbeitsdirectory (L:\Projects\Server bzw L:\Projects\Client) ist die zu übersetzende Quelldatei durch eine relative Pfadangabe zu referieren, z.B.

  javac vs\rmiprobsolv\ProblemSolverImpl.java

Erzeugung der Interfaces

- Die beiden Interfaces **ProblemTask** und **ProblemSolver** werden sowohl vom Client als auch vom Server benötigt.
- Erzeugen Sie die Quellcode-Dateien für die Interfaces und legen Sie sie im Server-Dateien-Directory
  L:\Projects\Server\vs\rmiprobsolv ab.
- Nach erfolgreicher Übersetzung sind die erzeugten Java-Bytecode-Dateien .class in das Client-Dateien-Directory **L:\Projects\Client\vs\rmiprobsolv** und in das Codebasedirectory **L:\public_html\vs\rmiprobsolv** zu kopieren..

Erzeugung und Start des Servers

- Erzeugen Sie die Quellcode-Datei für die Server-Klasse **ProblemSolverImpl** und legen Sie sie im Server Dateien-Verzeichnis L:\Projects\Server\vs\rmiprobsolv ab.
- Bleiben Sie im Konsolenfenster in dem Arbeits-Directory des Servers **L:\Projects\Server.** Übersetzen Sie die Quellcode-Datei (s. oben, Allgemeines)
- Erzeugen Sie Stub und Skeleton-Klassen für die Server-Klasse mit dem RMI-Compiler **rmic** ausgehend vom Arbeitsdirectory L:\Projects\Server\. Die – als Java-Bytecode-Datei vorliegende – Server-Klasse muß mit ihrem vollqualifizierten Package-Namen angegeben werden.

  rmic vs.rmiprobsolv.ProblemSolverImpl
- Kopieren Sie die erzeugte Datei **ProblemSolverImpl_stub.class** zusammen mit den Java-Bytecode-Dateien für die Interfaces **ProblemSolver.class** und **ProblemTask.class** in das Directory

  L:\public_html\vs\rmiprobsolv.
- Damit liegen sie tatsächlich auf dem Server **ti1** und können von dort (durch den Client) heruntergeladen werden.
- Starten Sie die RMI-Registry auf dem Standardport (1099). in einem Konsolen-Fenster, in dem die Environment-Variable **CLASSPATH nicht** gesetzt ist :

set CLASSPATH=
rmiregistry

- Starten Sie in einem anderen Konsolenfenster (CLASSPATH muß wie oben gesetzt sein: set CLASSPATH=) aus dem Server-Arbeits-Directory **L:\Projects\Server** den Server. Geben sie beim Aufruf der JVM die von der RMI-Registry zu verwendende Codebase (= Server und Directory),die die Stub-Klasse enthält, an :
  java –Djava.rmi.server.codebase=http://ti1/~<vsxx>/ vs.rmiprobsolv.ProblemSolverImpl

!!vsxx ist Ihre Praktikumskennung!!

Erzeugung und Start des Clients :

- Erzeugen Sie die Quellcode-Dateien für die Client-Klassen **PiTask** und **CompPi** und legen Sie sie im Client-Dateien-Verzeichnis **L:\Projects\Client\vs\rmiprobsolv** ab.

- Öffnen Sie ein neues Konsolen-Fenster und wechseln Sie in das Arbeits-Directory des Clients **L:\Projects\Client.** Übersetzen Sie die Quellcode-Dateien (s. oben, Allgemeines)
- Kopieren Sie die Java-Byte-Code-Datei für die Klasse des Aufgabenobjekts **PiTask.class** in das Directory **L:\public_html\vs\rmiprobsolv**
  Damit liegt die Datei tatsächlich auf dem Server **ti1** und kann von dort (durch das Serverprogramm) heruntergeladen werden.
- Öffnen Sie ein Konsolen-Fenster und gehen Sie in das Arbeits-Directory des Clients **L:\Projects\Client**
- Überprüfen Sie, daß die Environment-Variablen PATH und CLASSPATH wie auf dem Entwicklungsrechner gesetzt sind.

- Starten Sie in dem Konsolen-Fenster den Client. Geben Sie beim Aufruf der JVM die Codebase, in der sich die Bytecode-Datei **PiTask.class** befindet, an :

java –Djava.rmi.server.codebase=http://ti1/~<vsxx>/ vs.rmiprobsolv.CompPi localhost <genauigkeit>

Anmerkung: Die für den Client-Aufruf benötigte **IP-Adresse des Servers** (Kommandozeilenparameter **<server_host>** könnte auch durch Aufruf des Kommandos **ipconfig** in einem Konsolen-Fenster des Servers ermittelt werden.

- Geben Sie anstelle localhost **ti1** an, dann benutzen Sie die Implementierung des Servers des Labors. D.h. Ihr Client arbeitet mit dem Laborserver zusammen.

Erzeugung und Start des Clients und Servers auf einem Rechner zu Hause:

- Ersetzte **L:\Projects** durch ein beliebiges eigenes Verzeichnis: z.B. **C:\VsLab**
- Anstelle des **L:\public_html\vs\rmiprobsolv** Ordner: z.B. **C:\VsCode\vs\rmiprobsolv**
- Übersetzen mittels javac und rmic wie oben beschrieben, aber im Verzeichnis C:\VsLab\Server und C:\VsLab\Client
- Aufruf rmiregistry wie gehabt (siehe oben)
- Aufruf Server:

java –Djava.rmi.server.codebase=file:///C:\VsCode/ vs.rmiprobsolv. ProblemSolverImpl

- Aufruf Client:

java –Djava.rmi.server.codebase=file:///C:\VsCode/ vs.rmiprobsolv.CompPi localhost <genauigkeit>

- **Vergessen Sie nicht die Datei .java.policy zu erstellen (siehe oben) und in ihrem Heimatverzeichnis abzulegen (Linux /home/<Benutzer>, Windows C:\Dokumente und Einstellungen\<Benutzer>)**