

Aufgabenblatt 4

(Zu bearbeiten bis 14.11.)

Vorlesung Verteilte Systeme Wintersemester 2024/25

Aufgabe 1: Programmierung: Java-RMI - Folgebeispiel (Pflichtaufgabe für 6 LP, Abgabe über moodle!)

Erstellen Sie eine verteilte Anwendung, die ein entferntes Objekt mit einer Methode beinhaltet, welche zwei Ganzzahlen multipliziert. Gehen Sie wie folgt vor: Erstellen Sie

- zunächst die Schnittstelle `Numbers`, die die Methode `multiply()` definiert,
- dann eine Klasse `NumbersImpl`, die `Numbers` implementiert,
- eine Server-Anwendung in der Klasse `Server`,
- eine Client-Anwendung, die die entfernte Methode im `Server` aufruft und das Ergebnis der Multiplikation ausgibt.

Versuchen Sie, die RMI-Registry direkt im Programmcode des Servers zu starten, so dass auf der Kommandozeile kein Aufruf von `rmiregistry` notwendig ist. Verwenden Sie die Klassen `java.rmi.registry.LocateRegistry` und `java.rmi.registry.Registry` (siehe [Java API Dokumentation](#)).

Aufgabe 2: Programmierung: Generische Proxy-Objekte

Seit JDK 1.5 kann bei der Nutzung von Java RMI auf die Verwendung des RMI-Compilers `rmic` zur Erzeugung der Client-Stub-Klassen verzichtet werden. Stattdessen werden die notwendigen Stub-Klassen zur Laufzeit mit Hilfe von Java Reflection und der Klasse `java.lang.reflect.Proxy` gearbeitet. Um diesen Mechanismus näher kennenzulernen, sollen Sie für die Java-Klasse `CalculatorImpl`, die Sie im Archiv [u04Files.zip](#)¹ auf der Vorlesungswebseite finden, einen generischen Proxy erzeugen, der alle Methodenaufrufe an das eigentliche Objekt delegiert und sie zusätzlich protokolliert.

Informieren Sie sich zunächst über folgende Methoden:

- `java.lang.reflect.Proxy.newProxyInstance()` – Erzeugt dynamisch ein Proxy-Objekt
- `java.lang.reflect.InvocationHandler.invoke()` – An diese Methode werden alle Methodenaufrufe auf dem Proxy-Objekt delegiert
- `java.lang.Object.getClass()` – Liefert die Klasse eines Objekts
- `java.lang.reflect.Class.getClassLoader()` – Liefert den Klassenlader zu einer Klasse
- `java.lang.reflect.Class.getInterface()` – Liefert alle Schnittstellen, die die Klasse implementiert

Ergänzen Sie dann die Klasse `CalculatorProxy`, die das Interface `InvocationHandler` implementiert. Die Methode `invoke()` soll die aufgerufene Methode und deren Argumente ausgeben, den Aufruf dann an das „echte“ Objekt delegieren und anschließend den Rückgabewert ausgeben. Ersetzen Sie in dann der Klasse `CalculatorExample` das „echte“ `Calculator`-Objekt (Referenz in der Variable `calc`) durch das Proxy-Objekt und testen Sie Ihr Programm.

¹<http://www.bs.informatik.uni-siegen.de/web/wismueller/vl/vs/u04Files.zip>

Aufgabe 3: Generischer Dispatcher mit Java Reflection

Eine Komponente eines Server-Skeletons ist der *Dispatcher*, der den in der Anfragenachricht spezifizierten Methodenauf-
ruf an das jeweilige Objekt weitergibt. Das Server-Skeleton ist seit Java 1.2 generisch, muß also nicht für jede Remote-
Schnittstelle neu erzeugt werden. Die Implementierung stützt sich dabei auf die Mechanismen von Java Reflection.

Machen Sie sich etwas tiefer mit Java Reflection vertraut, insbesondere den beiden Klassen `java.lang.Class` und
`java.lang.reflect.Method` und überlegen Sie sich, wie

- a) ein konkreter Methodenaufruf (inkl. Argumente) in einer (Anfrage-)Nachricht dargestellt werden kann (also: welche
Informationen in der Nachricht enthalten sein müssen),
- b) der Code eines generischen Dispatchers auf Basis von Java Reflection aussehen könnte, der die Anfragenachricht
als Argument erhält und die Methode auf dem jeweiligen Server-Objekt aufruft.

Aufgabe 4: Parameterübergabe

Betrachten Sie die Prozedur `incr`, die zwei ganzzahlige Parameter verwendet. Diese Prozedur addiert zu jedem Parameter
eine 1.

Angenommen, sie wird mit der selben Variable zweimal aufgerufen, beispielsweise `incr(i, i)`. Wenn `i` anfänglich 0
ist, welchen Wert hat es, anschließend, wenn *call-by-reference* verwendet wird? Welchen Wert hätte `i`, wenn *call-by-value*
genutzt wird? Wie sieht es bei *call-by-copy/result* aus?

Aufgabe 5: Transparenz von Java RMI (Pflichtaufgabe für 6 LP, Abgabe über moodle!)

Ein unerfahrener Programmierer hat die Aufgabe bekommen, das Sortieren einer Liste, das bisher lokal in einem Cli-
ent vorgenommen wurde, ein einen Server auszulagern. Da dabei die Schnittstelle absolut unverändert bleiben mus-
ste (Teile des Clients liegen nicht im Quelltext vor), hat der Programmierer eine Wrapper-Klasse implementiert, die
das Server-Objekt beim Namensdienst aufsucht und die auftretenden Exceptions behandelt. Eine (vereinfachte) Versi-
on seines Codes finden Sie im Archiv [u04Files.zip](http://www.bs.informatik.uni-siegen.de/web/wismueller/v1/vs/u04Files.zip)² auf der Vorlesungswebseite. Dabei ist `LocalSorter` (in
`Client.java`) die Originalklasse, `RemoteSorter` die neue Wrapper-Klasse. Der Servercode besteht aus der Remote-
Schnittstelle `SortServer` und der Implementierungsklasse `Server`. Um die Kompatibilität zwischen den Klassen
`RemoteSorter` und `LocalSorter` zu testen, wird die Methode `testSorter()` in `Client.java` jeweils mit ei-
ner Instanz der beiden Klassen aufgerufen. Wie die Ausgabe des Clients zeigt, scheint das Sortieren mit Hilfe des Servers
jedoch nicht zu funktionieren.

„Reparieren“ Sie die Remote-Implementierung des Sortierens! Beachten Sie, daß die Schnittstelle von `RemoteSorter`
nicht verändert werden darf. Wo liegt das Problem? Wie kann es gelöst werden?

²<http://www.bs.informatik.uni-siegen.de/web/wismueller/v1/vs/u04Files.zip>