UNIVERSITÄT SIEGEN

Fakultät IV
Betriebssysteme und verteilte Systeme
Prof. Dr. rer. nat. Roland Wismüller

Excercise Sheet 4

(To be processed until 22.05.2020)

Lecture Distributed Systems Summer Term 2020

Exercise 1: Programming: Java-RMI - Follow-up example

Create a distributed application that includes a remote object with a method that multiplies two integers. Proceed as follows: Create

a) first the interface `Numbers`, which defines the method `multiplyNumbers()`,

b) then a class `NumbersImpl` implementing `Numbers`,

c) a server application in the class `Server`. Replace the manual start of `rmiregistry` with a direct start of the RMI registry in the program code (Note: Use the two classes `java.rmi.registry.LocateRegistry` and `java.rmi.registry.Registry`),

d) a client application that calls the remote method in the server and returns the result of the multiplication.

Exercise 2: Programming: Generic proxy objects

Since JDK 1.5 the use of the RMI compiler `rmic` to create the client stub classes is no longer necessary when using Java RMI. Instead, the necessary stub classes are created at runtime using the class `java.lang.reflect.Proxy` from the Java Reflection framework. To learn more about this mechanism, you should create a generic proxy for the Java class `CalculatorImpl`, which you can find in the archive `u04eFiles.zip`[1] on the lecture's web page, which delegates all method calls to the actual object and logs them additionally.

First inform yourself about the following methods:

- `java.lang.reflect.Proxy.newProxyInstance()` – Dynamically creates a proxy object
- `java.lang.reflect.InvocationHandler.invoke()` – All method calls on the proxy object are delegated to this method
- `java.lang.Object.getClass()` – Returns the class of an object
- `java.lang.reflect.Class.getClassLoader()` – Returns the class loader for a class
- `java.lang.reflect.Class.getInterface()` – Returns all interfaces the class implements

Then add the `CalculatorProxy` class that implements the `InvocationHandler` interface. The method `invoke()` should output the called method and its arguments, delegate the call to the "real" object and then output the return value. In the `CalculatorExample` class, replace the "real" `Calculator` object (reference in the `calc` variable) with the proxy object and test your program.

[1] `http://www.bs.informatik.uni-siegen.de/web/wismueller/vl/vs/u04eFiles.zip`

Exercise 3: Generic dispatcher with Java Reflection

A component of a server skeleton is the *Dispatcher*, which passes the method call specified in the query message to the respective object. The server skeleton has been generic since Java 1.2, so it does not have to be built anew for every remote interface. The implementation is based on the mechanisms of Java Reflection.

Get a bit more familiar with Java Reflection, especially the two classes `java.lang.Class` and `java.lang.reflect.Method` and think about how

a) a concrete method call (incl. arguments) can be represented in a (request) message (i.e.: which information must be contained in the message),

b) the code of a generic dispatcher based on Java Reflection might look like, which receives the request message as argument and calls the method on the respective server object.

Exercise 4: Parameter passing

Consider the procedure `incr` that uses two integer parameters. This procedure adds a 1 to each parameter.

Suppose it is called twice with the same variable, for example `incr(i, i)`. If `i` is initially 0, what is its final value, if *call-by-reference* is used? What value would `i` have if *call-by-value* is used? What about *call-by-copy/result*?

Exercise 5: Transparency of Java RMI

An inexperienced programmer has been given the task of offloading the sorting of a list that was previously carried out locally in a client to a server. Since the interface had to remain absolutely unchanged (parts of the client are not available in source code), the programmer implemented a wrapper class that searches for the server object using the name service and handles the exceptions that occur. A (simplified) version of its code can be found in the archive `u04eFiles.zip`[2] on the lecture's web page. `LocalSorter` (in `Client.java`) is the original class, `RemoteSorter` is the new wrapper class. The server code consists of the remote interface `SortServer` and the implementation class `Server`. To test the compatibility between the classes `RemoteSorter` and `LocalSorter`, the method `testSorter()` in `Client.java` is called with one instance of each class. As the output of the client shows, sorting using the server does not seem to work.

"Repair" the remote implementation of sorting! Note that the `RemoteSorter` interface must not be changed. What is the problem? How can it be solved?

[2] `http://www.bs.informatik.uni-siegen.de/web/wismueller/vl/vs/u04eFiles.zip`