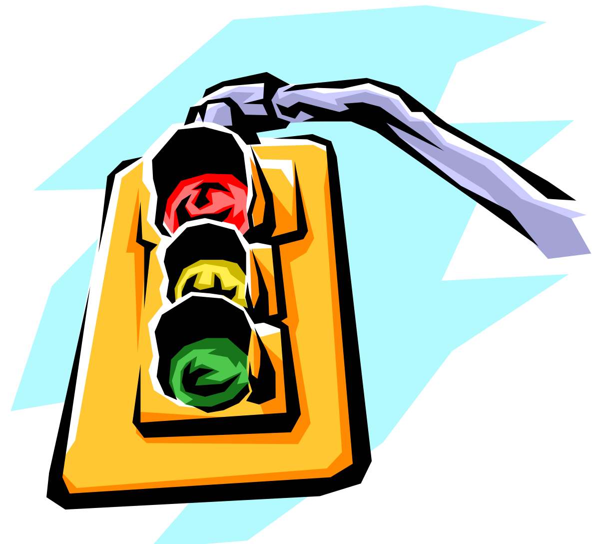
Sommersemester 2003

Implementierung IV: Schnittstelle zur Umgebung

und

Integration und Test



Aufgabe 6

Implementierung IV: Schnittstelle zur Umgebung und Integration und Test

Umfang: **2 Wochen**

Punkte: **100 P.**

Jedes eingebettete System ist mit seiner Umgebung (Environment, Außenwelt) über Sensoren (Messfühler) und Aktuatoren (z.B. Motoren, Relais, Signalleuchten) gekoppelt.

Da man eingebettete Systeme während der Entwicklung meistens nicht direkt mit der endgültigen Umgebung validieren kann – die Umgebung existiert zum Teil noch gar nicht oder Experimente in einer frühen Entwurfsphase wären zu gefährlich – verwenden wir einen Simulator, der das Verhalten der Umgebung simuliert.

Um diesen Simulator gegen eventuell andere Simulatoren oder am Ende gegen die physikalische Umgebung austauschen zu können, definieren wir eine einheitliche Schnittstelle. Dazu wählen **Java RMI** (**Remote Method Invocation**) als technische Basis. Durch Java RMI wird die Implementierung einer verteilten Applikation ermöglicht (d.h. Simulator und Lichtsignalsteuerung können auf

unterschiedlichen Rechnern ablaufen). In einer solchen verteilten Applikation müssen „entfernte" Objekte lokalisiert („gefunden"), die Kommunikation zwischen den verteilten Objekten realisiert und bei Bedarf Java Code ausgetauscht werden.

1 RMI-Grundlagen

Mit RMI wird in Java ein Mechanismus zur Verfügung gestellt, der die einfache netzwerkweite Bereitstellung von Objektdiensten erlaubt. Damit können spezielle Aufgaben einer Applikation (z.B. Datenbankzugriffe oder Kommunikation mit externen Systemen) an geeignete Server delegiert und so die Applikationslast gleichmäßiger verteilt und die Skalierbarkeit des Systems erhöht werden. Die prinzipielle Vorgehensweise beim Arbeiten mit RMI lässt sich wie folgt skizzieren:

- Definition von Methoden, die als aufrufbare Dienste anderen zur Verfügung gestellt werden sollen, in einem **Remote-Interface**.
- Implementierung dieses Interfaces durch eine Serverklasse und Erzeugung einer oder mehrerer Instanzen, die als Dienstleister im Netzwerk in Erscheinung treten sollen (**Remote-Objekte**).
- Registrierung der Remote-Objekte bei einem **Namens-Service** (***RMI-Registry***), der von potentiellen Clients zur Lokalisierung der Remote-Objekte abgefragt werden kann.
- Clients erhalten so genannte **Remote-Referenzen** auf die benötigten Remote-Objekte mittels der RMI-Registry. Beim Aufruf der gewünschten Methode werden Aufrufparameter an das Remote-Objekt übertragen, die passende Methode des Serverobjekts ausgeführt und der Rückgabewert an den Client zurück übertragen. Die Übertragung von Aufrufparametern und Rückgabewerten unterliegt gewissen Beschränkungen (siehe Java RMI Spezifikation [RMI99]).

Die Kommunikation zwischen Aufrufer und Remote-Objekt geschieht dabei transparent über so genannte **Stubs** und **Skeletons**. Ein Stub-Objekt dient als lokaler Vertreter eines Remote-Objekts, daher implementiert die Stub-Klasse dasselbe Remote-Interface wie die eigentliche Serverklasse. Der Stub kommuniziert über das Netzwerk mit dem als Skeleton bezeichneten Gegenstück auf der Serverseite. Das Skeleton kennt das tatsächliche Applikationsobjekt, leitet

Aufruf und Parameter an dieses weiter und transferiert den Rückgabewert an den Stub. Stub und Skeleton werden mittels eines JDK-Tools aus der Serverklasse generiert und verbergen die komplizierten Details der Netzwerkkommunikation zwischen Server und Client.

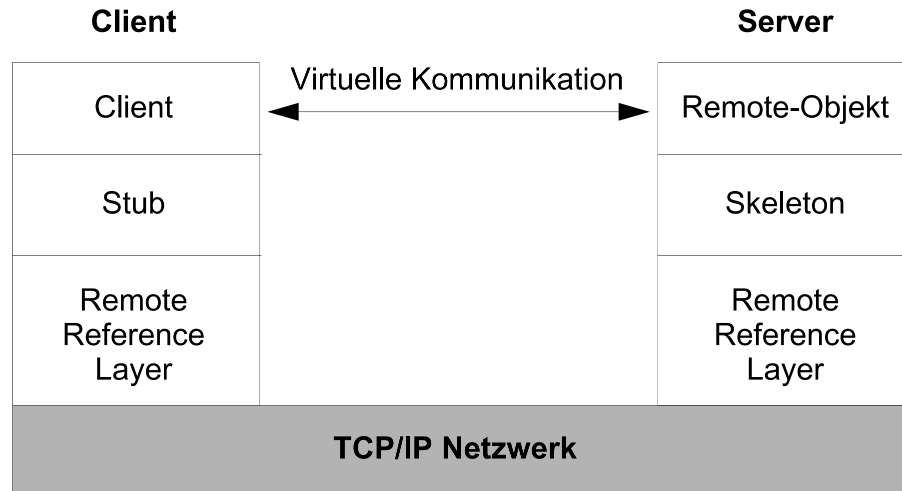


Abbildung 24 Kommunikation zwischen Client und Remote-Object über Stub und Skeleton

Eine weitere interessante Eigenschaft von RMI ist die Fähigkeit, fehlenden Code dynamisch nachzuladen. Benötigt beispielsweise ein Server zur Ausführung eines Auftrags eine ihm bis dato unbekannte Klasse vom Client, so lädt er diese nach. Dazu muss diese Clientklasse allerdings ein der Serverklasse zur Compile-Zeit bekanntes Interface implementieren.

Eine detaillierte Beschreibung der Konzepte von Java RMI bietet die Java RMI Spezifikation [RMI99]. Desweiteren findet man in [RMITut] Programmierbeispiele.

2 Der Aufbau der Schnittstelle

2.1 Objektstruktur

Die Schnittstelle zur Umgebung (in diesem Fall zum Simulator) wird durch die drei in Abbildung 25 gezeigten Klassen realisiert, wobei die Interfaces der Klassen `Signalgeber` und `Detektor` Ihnen seit dem Aufgabenteil 'Implementierung II' bekannt sind.

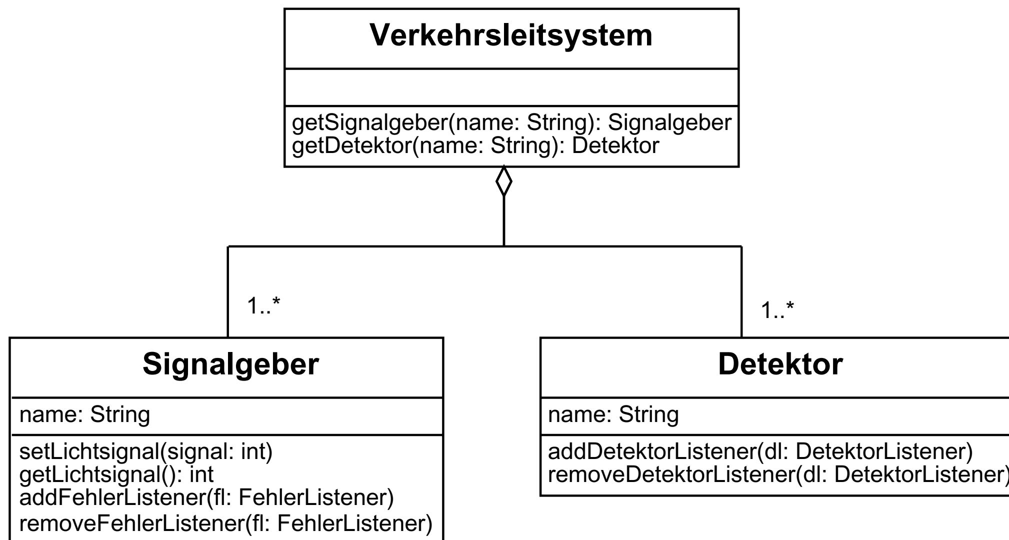


Abbildung 25 Klassen der RMI-Schnittstelle

Dabei tritt `Verkehrsleitsystem` als sog. „Bootstrapping“-Klasse auf, d.h. sobald man eine Instanz dieser Klasse über den RMI-Naming-Service erhalten hat, können Remote-Referenzen auf die eigentlichen `Signalgeber`- und `Detektor`-Objekte von diesem angefordert werden (Methoden `getSignalgeber()` und `getDetektor()`). Die benötigten Remote-Interfaces, deren mögliche Exceptions und weitere verwendete Klassen sind in [SimDoc] dokumentiert.

Wir wollen die Realisierung dieser Schnittstelle im folgenden Abschnitt grob am Beispiel eines einfachen Steuerungssystems vorstellen.

2.2 Ein einfaches Steuerungssystem

Um die Anwendung des RMI-Mechanismus zu verdeutlichen, soll hier für ein simples Steuerungssystem die wichtigsten Stellen im Code erklärt werden. Mit Hilfe dieses Schemas sollten Sie dann die Anbindung Ihrers Steuerungssystems an den Simulator implementieren können.

```
public class ControlSystem {

    /* ... */
```

```
public static void main(String[] args) {

// Install security manager
System.setProperty("java.security.policy",
 "./java.policy");
if(System.getSecurityManager() == null) {
   System.setSecurityManager(new SecurityManager());
}
```

Zunächst wird eine Instanz der Java `SecurityManager` Klasse installiert. Jede RMI-Applikation benötigt solch einen **Security Manager**, da sie potenziell nicht vertrauenswürdigen Kode herunterladen kann und somit die Sicherheit des Systems beinträchtigen könnte. Mit JDK Version 1.2 hat sich das **Sicherheitskonzept** von Java deutlich verändert. Besonders das Konzept der **Policies** ist für uns von Interesse, denn der Security-Manager ermittelt die Berechtigungen für den nachgeladenen Code aus **Policy-Files**. Wir setzen im Praktikum die Vertrauenswürdigkeit unseres eigenen Codes voraus und erzeugen eine „`java.policy`“ Datei mit folgendem Inhalt, welche sämtliche Aktivitäten erlaubt:

```
grant {
    permission java.security.AllPermission;
};
```

Für ein endgültiges Softwareprodukt sollten die Berechtigungen allerdings auf die minimal benötigten Permissions beschränkt sein. Es gibt verschiedene Möglichkeiten, dem Security-Manager die Permissions mitzuteilen, z.B. über benutzerspezifische Policy-Files („`java.policy`“ im User-Heimverzeichnis wie oben) oder frei über das `java.security.policy`-Property (bei Aufruf oder im Programm vor Benutzung einer RMI-Klasse). Weitere Details zu dem Sicherheitskonzept von Java findet man in [JSec1–3].

```
// Get classServerPort
int portClassServer = /* ... */;

// Install class file server
try {
   String classpath = System.getProperty("user.dir");
   ClassFileServer cfs =
   new ClassFileServer(portClassServer, classpath);
   } catch (Exception exc) {
      System.exit(0);
}
```

Nun installieren wir einen sog. **Class-File-Server** (ein spezieller Web-Server), damit der Simulator den Code für die Implementierung von *DetektorListener* und *FehlerListener* herunterladen kann. Java

RMI benutzt nämlich vorhandene URL-Protokolle (z.B. HTTP, FTP) um Bytecode zwischen Java Virtual Machines auszutauschen. Wir könnten natürlich auch einen gewöhnlichen Webserver für die dynamische Weitergabe von Code einsetzen. Dazu müsste ein solcher Web-Server aber immer installiert sein, was wir nicht voraussetzen wollen.

```
// remote classes must be annotated with
// the location (the URL) of class binaries.
//
String localhost = /* ... */;
// IMPORTANT: The trailing / in the following argument
// MUST be used!
System.setProperty("java.rmi.server.codebase", "http://" +
        localhost + ":" + portClassServer + "/");
```

Nachdem der Class-File-Server initialisiert wurde, müssen wir unserer Applikation noch mitteilen, mit welcher Adresse (URL) sie die RMI-Objekte **annotieren** soll, so dass der Code von einer anderen Virtual Machine geladen werden kann. Geschieht dies nicht, werden die Class-Binaries von der anderen Virtual Machine nur in derem lokalen CLASSPATH gesucht!

```
// Get RMI-Registry Host and Port
String hostRegistry = /* ... */;
int portRegistry = /* ... */;

// Get object references and start 'simple' control system
try {
   Verkehrsleitsystem vl = (Verkehrsleitsystem)
      Naming.lookup("rmi://"+hostRegistry+":"+portRegistry+
      "/Verkehrsleitsystem");

      Signalgeber sg7_1;
      sg7_1 = vl.getSignalgeber("lsa7_sg1");

      Detektor d8_1;
      d8_1 = vl.getDetektor("lsa8_d1");

      // create DetektorListener
      DetektorListenerImpl dl = new DetektorListenerImpl();
      d8_1.addDetektorListener(dl);

      sg7_1.setZustand(Signalgeber.GRUEN);

} catch(Exception exc) {
}

}
```

Wir besorgen uns das `Verkehrsleitsystem`-Objekt des Simulators über dessen Registry (die Portnummer wird beim Start des Simulators angegeben). Dieses Objekt liefert uns die einzelnen

Signalgeber-Objekte (hier stellvertretend das für lsa7_sg1) und analog die Detektor-Objekte (hier beispielhaft lsa8_d1).

Danach wird eine Implementierung von `DetektorListener` als Listener beim Detektor-Objekt registriert. Das ist auch der Grund, warum wir einen Class-Server auf Steuerungssystem-Seite benötigen. Der Ampelsimulator muss ja den Bytecode der DetektorListener (oder analog dazu den von FehlerListener) bekommen. Details dazu findet man wieder in [RMI99].

2.3 Der AmpelSimulator

Das Simulatorpaket wird auf dem Ausbildungscluster installiert und zusätzlich auf den Praktikums-Webseiten zum Download bereit gestellt.

Die RMI-Registry und die Class-File-Server kommunizieren über IP-Ports, die auf einem Rechner eindeutig vergeben und dem Kommunikationspartner bekannt sein müssen. Damit sich verschiedene Praktikumsgruppen bei den Tests nicht gegenseitig stören, erhält jede Gruppe einen Bereich von **100 Ports** zugewiesen, die sie exklusiv nutzen dürfen. Die **erste Portnummer** eines Gruppenbereichs berechnet sich wie folgt:

$$\text{erste Portnummer} = 20000 + (\text{Gruppennummer} \cdot 100)$$

Also z.B. der Bereich von 21400-21499 für Gruppe 14.

3 Aufgaben

3.1 Ergänzung der initialen Implementierung (50 Punkte)

In der letzten Aufgabe haben Sie bereits die Steuerung der einzelnen Lichtsignalanlagen implementiert. Erweitern Sie Ihr Steuerungssystem nun dahingehend, dass die Detektoren und Signalgeber des AmpelSimulators verwendet werden.

3.2 Integration und Test (50 Punkte)

3.2.1 Ampelsimulator

Testen Sie Ihre Ampelsteuerung mit Hilfe des AmpelSimulators. Überprüfen Sie insbesondere Ihre Konzepte zur Verkehrskontrolle („Grüne Welle“, etc.), indem Sie das Verhalten des Systems über einen längeren Zeitraum beobachten. Hier sollten Sie nun auch Ihre bislang noch unbekannten Parameter (Abstände der Kreuzungen, etc.) bestimmen. Gegebenenfalls müssen Sie auch Ihre Strategien modifizieren.

3.2.2 Dokumentation der Testläufe

In der Aufgabe „Implementierung III“ sollten Sie mit Hilfe eigener Testroutinen die Richtigkeit Ihrer Implementierung in Bezug auf die bei der Analyse entstandenen Sequenzdiagramme überprüfen.

Dokumentieren Sie für den Korrektor nachvollziehbar die Übereinstimmung Ihrer (evtl. modifizierten) Sequenzdiagramme und Implementierungen an Hand von Testprotokollen.

3.3 Abgabe

Abzugeben sind:

- die neuen und eventuell korrigierten Java-Sourcen, inkl. JavaDoc
- (evtl. modifizierte) Sequenzdiagramme
- zugehörige Testprotokolle (inklusive sinnvoller Dokumentation)

4 Literatur

[RMI99] *Java Remote Method Invokation Specification*. Mountain View, Ca: Sun Microsystems. 1999
—Link ist auf Praktikums-Site vorhanden—

[RMITut] *Fundamentals of RMI – Short Course*
—Link ist auf Praktikums-Site vorhanden—

[SimDoc] Dokumentation zum AmpelSimulator
—Link ist auf Praktikums-Site vorhanden—

[JSec1] Java SecurityManager, API Documentation
—Link ist auf Praktikums-Site vorhanden—

[JSec2] Permissions in the Java 2 SDK
—Link ist auf Praktikums-Site vorhanden—

[JSec3] Java Security
—Link ist auf Praktikums-Site vorhanden—