1. Vorbemerkungen

In dieser Aufgabe soll eine einfache objektorientierte Middleware konzipiert und realisiert werden, mit deren Hilfe Methodenaufrufe eines entfernten Objektes möglich sind.

2. Grundsätzlicher Aufbau der Middleware und Schnittstellen

Die Middleware soll aus nachfolgenden Teilen bestehen

- **Namensdienst**
- Bibliothek ***mware_lib*** (in Java ein Package)
- **Compiler**, der aus Definitionen in einer Schnittstellenbeschreibungssprache (im Folgenden IDL genannt) Quellcode (Java) generiert, den die Applikation zur Nutzung Ihrer Middleware als Bindeglied benötigt

Eine Applikation bindet zur Benutzung Ihrer Middleware *mware_lib* und den mit Ihrem IDL-Compiler generierten Code ein. (Vgl. Beispiel unten)

Wirft eine Serverapplikation beim Remoteaufruf eine `RuntimeException`, soll diese an den Aufrufer weitergeleitet werden, d.h. gleicher Exceptiontyp und gleicher Meldungstext.

3. Namensdienst

Dieser soll Namen auf Objektreferenzen abbilden. Er muss auf einem gesonderten Rechner im Netz laufen können. Sein Port muss zur Laufzeit einstellbar sein (Startparameter).

4. Bibliothek *mware_lib*

Sie soll alle Klassen und Interfaces enthalten, die die Middleware generell für den Betrieb benötigt, unabhängig vom Aussehen der aktuellen Anwendungsschnittstellen.
Die Bibliothek soll in einem Package (Name *mware_lib*) zusammengefasst werden.

Schnittstellen nach aussen:

```
public class ObjectBroker {  //- Front-End der Middleware -
       public static ObjectBroker init(String serviceHost,
                                       int listenPort, boolean debug) { ... }
          // Das hier zurückgelieferte Objekt soll der zentrale Einstiegspunkt
          // der Middleware aus Applikationssicht sein.
          // Parameter: Host und Port, bei dem die Dienste (hier: Namensdienst)
          //            kontaktiert werden sollen. Mit debug sollen Test-
          //            ausgaben der Middleware ein- oder ausgeschaltet werden
          //            können.

       public NameService getNameService() {...}
          // Liefert den Namensdienst (Stellvetreterobjekt).

       public void shutDown() {...}
          // Beendet die Benutzung der Middleware in dieser Anwendung.
}

public abstract class NameService {     //- Schnittstelle zum Namensdienst -
       public abstract void rebind(Object servant, String name);
          // Meldet ein Objekt (servant) beim Namensdienst an.
```

```
        public abstract Object resolve(String name);
           // Liefert eine generische Objektreferenz zu einem Namen. (vgl. unten)
}
```

(Anwendungsbeispiele hierzu finden Sie unten)

5. IDL-Compiler

Dieser soll aus den Schnittstellenbeschreibungen des Anwenders in IDL die entsprechenden benötigten (Basis-)Klassen der Middleware in (Java-)Quellcode generieren. Sie stellen das Bindeglied zwischen *mware_lib* und dem Anwendercode dar.

Der Compiler soll in *einem* Package oder JAR-Archiv vorliegen. Der IDL-Code soll aus einer Datei gelesen werden (Startparameter).

Um das Parsing zu vereinfachen, soll der Funktionsumfang auf nachfolgende Typen beschränkt werden:

IDL-Typ	Entspricht in Java
`module` (keine Schachtelung, 1 Modul pro Datei)	`package`
`class` (nicht als Parameter oder Returnwert, keine Schachtelung)	`class`
`int`	`int`
`double`	`double`
`string`	`String`

Zur Einbindung der Anwenderklassen soll der Compiler Schnittstellenklassen bereitstellen, von denen der Anwender seine Klassen ableitet. Namenskonvention ist dabei "**`_<`***name*`**>ImplBase`**". Weiter sollen diese Klassen eine `narrowCast()`-Methode implementieren, die zu einer von `resolve()` gelieferten generischen Objektreferenz eine klassenspezifische Referenz liefert. Alle Klassen eines Moduls sollen in einem gleichnamigen Package zusammengefasst werden.

Beispiel für eine IDL-Datei und dem daraus vom Compiler generierten Java-Quellcodeschema:

IDL-Datei	Zu generierender Java-Code

`module math_ops {`
`  class Calculator {`
`   double add(double a, double b);`
`   string getStr(double a);`
`  };`
`};`
	`package math_ops;`
	
	`public abstract class _Calculator`**`ImplBase`**` ... {`
	`  public abstract double add(double a, double b);`
	`  public abstract String getStr(double a);`
	`  public static _Calculator`**`ImplBase narrowCast`**`(`
	`  Object rawObjectRef) { ... }`
	`  ...`
	`}`
	
	`<ggf. weitere hier benötigte Klassen/Interfaces>`

Um das Parsing einfach zu halten, soll jede IDL-Datei dem obigen Format entsprechen (Position der Klammern, 1 Methode = 1 Zeile, 1 Modul pro Datei). Ein Beispiel für einen einfachen Parser finden Sie im Download,

6. Beispiel: Integration in den Code einer Server-Anwendung

```
import math_ops.*;

...

public class Calculator extends _CalculatorImplBase {
   public double add(double a, double b) {  return a + b;  }
      ...
}
...

   ObjectBroker objBroker = ObjectBroker.init(host, port, false);
   NameService nameSvc = objBroker.getNameService();
   nameSvc.rebind((Object) new Calculator(), "zumsel");

...

   objBroker.shutDown();

...
```

7. Beispiel: Integration in den Code einer Client-Anwendung

```
import math_ops.*;

...

ObjectBroker objBroker = ObjectBroker.init(host, port, false);
NameService nameSvc = objBroker.getNameService();
Object rawObjRef = nameSvc.resolve("zumsel"); // generische Objektreferenz

_CalculatorImplBase remoteObj = _CalculatorImplBase.narrowCast(rawObjRef);
// liefert klassenspezifisches Stellvertreterobjekt

...

try { // Entfernter Methodenaufruf
  double s = remoteObj.add(1, 567);
catch (RuntimeException e) {  ... }

...

objBroker.shutDown();

...
```

8. Hinweise und Tipps

Überlegen Sie sich, welche Informationen für einen Aufruf ausgetauscht werden müssen und wie. Entwerfen Sie ein geeignetes Request/ReplyProtokoll.

Sockets sollten nur in möglichst wenigen Klassen verwendet werden.

Es kann vorkommen, dass zwei oder mehr Klienten ein und dieselbe Objektreferenz zeitgleich

nutzen wollen. Die Erfüllung solcher Anforderungen soll innerhalb der Middleware nicht zu Deadlocks führen. (Die Behebung von Deadlocks in den Anwendungen ist hingegen nicht Aufgabe der Middleware.)

An einigen Stellen ist ein Übergang zwischen der (statischen) Bibliothek *mware_lib* und den vom IDL-Compiler generierten Code notwendig. Hier helfen die Vererbungsmechanismen weiter. Der Einsatz der Java-Reflection ist nur mit entsprechender Erfahrung zu empfehlen.

Den Namen einer Klasse können Sie in Java zur Laufzeit mit dem Stacktrace von Throwable ermitteln: `new Throwable().getStackTrace()[0].getClassName()`

Vorbereitung: Bis zum **Donnerstagabend 20:00 Uhr** vor dem Praktikumstermin ist ein Konzeptpapier als **PDF**-Dokument (mit ausgefülltem Dokumentationskopf) per E-Mail über den Abgabeverteiler abzugeben. Darin ist der geplante Aufbau der Middleware sowie alle wesentlichen Interaktionen und Abläufe mit geeigneten Diagrammen zu dokumentieren. Die wichtigsten Klassen und Methoden müssen hier bereits erkennbar sein.

Die **Vorführung** beginnt um ca. **08:20**. Die Middleware wird dabei mit fremden Applikationen getestet.

Abgabe als ZIP-Archiv **am Abend vor dem Praktikum bis 20:00 Uhr** von allen Teams an den o.g. Abgabeverteiler mit CC an den Praktikumspartner.

Die Abgabe muss folgendes enthalten:

- README-Datei, die beschreibt, wie der Namensdienst und wie der IDL-Compiler von der Kommandozeile zu starten ist,
- Binär- und Quellcode der Middleware-Pakete sowie der Testapplikationen,
- *mware_lib* binär (1 Package bzw. Verzeichnis),
- Vom IDL-Compiler für die Vorführungsläufe generierter Code,
- Ausgaben der Applikationen aus den Vorführungsläufen,
- aktualisierter Dokumentationskopf.

Abgaben, die diese Form nicht erfüllen oder unvollständig sind, werden nicht akzeptiert. Im Übrigen gelten die Regelungen aus den vorangegangenen Aufgaben