Team: TEAM 01, Falco Winkler (FW), Daniel Schruhl (DS)

Aufgabenteilung:

- IDL Compiler
- Namensdienst
- mware_lib Library

Quellenangaben:

- Aufgabe 4, 11.06.2017, C. Klauck & H. Schulz:
  http://users.informatik.haw-hamburg.de/ schulz/pub/Verteilte-Systeme/AI5-VSP/Aufgabe4/

Bearbeitungszeitraum:

- 11.06.2017 4 Stunden (DS)
- 12.06.2017 2 Stunden (DS)
- 14.06.2017 4 Stunden (DS)
- 15.06.2017 5 Stunden (DS)

Aktueller Stand:

- IDL Compiler fertig und getestet
- Namensdienst
- mware_lib Library

Änderung des Entwurfs:

- Keine Änderungen

1 Einführung und Ziele

Es soll eine einfache objektorientierte Middleware entworfen werden, die Methodenaufrufe eines entfernten Objektes ermöglicht.
Zur Orientierung gilt hierbei die CORBA Architektur. Genauer soll hier ein ORB zur Verfügung gestellt werden, der es ermöglicht Methoden von entfernten Objekten aufzurufen.
Zur Abstraktion und Beschreibung der Schnittstellen der Objekte soll eine IDL verwendet werden. Diese IDL wird dann zur Erzeugung von Klassen- und Methodenrümpfen verwendet.
Außerdem beinhaltet der ORB einen Namensdienst, der Objektreferenzen in einem Netz mit Namen finden kann.
Die Middleware an sich soll durch eine Library abstrahiert und verwendbar sein.

1.1 Randbedingungen

Der Namensdienst soll auf einem entfernten Rechner unabhängig von der Middleware Library lauffähig sein. Der Port muss zur Laufzeit einstellbar sein.
Der IDL-Compiler soll in einem Package oder einer `.jar` Datei zur Verfügung gestellt werden. Der Compiler soll folgende IDL Typen unterstützen:

- module (keine Schachtelung, 1 Modul pro Datei)
- class (nicht als Parameter oder Returnwert, keine Schachtelung)
- int
- double
- string

Ein Beispiel:

```
module math_ops {
  class Calculator {
   double add(double a, double b);
   string getStr(double a);
 };
};
```

Die Middleware Library soll in einem Package `mware_lib` zusammengefasst werden.
Wenn eine Serverapplikation während eines entfernten Methodenaufrufes eine RuntimeException wirft, soll diese an den Aufrufer weitergeleitet werden.
Es soll möglich sein, dass zwei oder mehrere Klienten die selbe Objektreferenz zeitgleich nutzen wollen. Das soll innerhalb der Middleware nicht zu Deadlocks führen.

1.2 Kontextbegrenzung

Die Implementierung soll in Java vorliegen.
Die Behebung von Deadlocks in den Anwendungen ist nicht Aufgabe der Middleware.

2 Gesamtsystem

Das Gesamtsystem besteht aus drei Subsystemen, die unabhängig voneinander lauffähig sind. Dabei sind das NameService System und das IDL-Compiler System Applikationen, die über die Kommandozeile gestartet werden können.
Die Middleware Library (mware_lib) ist eine Bibliothek, die in der jeweiligen Applikation, die die Middleware verwenden soll eingebunden werden muss. Die Middleware Library ist also nicht direkt über die Kommandozeile startbar.

2.1 Bausteinsicht

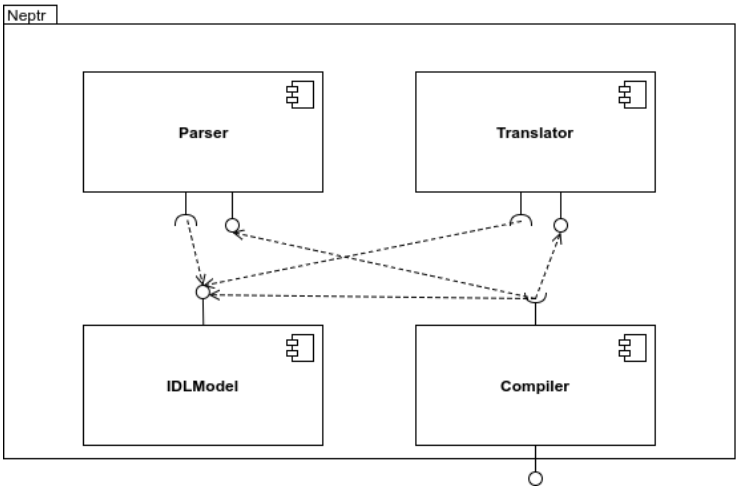

Abbildung 1: Komponentendiagramm des IDL-Compilers (Neptr)

2.2 Laufzeitsicht

3 Subsysteme und Komponenten

3.1 NameService

3.1.1 Aufgabe und Verantwortung

Der NameService bildet Namen auf Objektreferenzen ab. Er wird verwendet, um Objekte anhand ihres Namens zu finden und anzusprechen und um Objekte anzumelden.

3.1.2 Schnittstelle

```
Die Schnittstellen sind ansprechbar mittels u.g.
Nachrichtenformat per TCP an gegebenem Port.
public void rebind(Object servant, String name);
Bindet ein Objekt an einen Platzhalter im Namensdienst.
public Object resolve(String name);
Loest eine Namensreferenz auf ein Java – Objekt auf.
Das Ergebnis wird als serialisiertes Java – Objekt per TCP an den Aufrufer gesendet.
```

3.1.3 Entwurfsentscheidungen

Der Port, an dem der NameService läuft ist zur Laufzeit einstellbar. Das geschieht über den Startparameter.
Der Nameservice Empfängt Nachrichten im folgenden Format:

- Byte 0: Art des Befehls. (0 = Rebind, 1 = Resolve, 2 = Shutdown)
- Byte 1 - 11: Alias für rebind / resolve
- Byte 12 - n: Serialisiertes Objekt

3.1.4 Konfigurationsparameter

- Port des NameServices

3.2 IDL Compiler (Neptr)

3.2.1 Aufgabe und Verantwortung

Der IDL Compiler hat die Aufgabe, eine gegebene Modulbeschreibung aus einer `.idl-Datei` im IDL Format einzulesen und daraus Java-Code zu generieren.

Es wird eine Abstrakte Java Klasse generiert (ImplBase bzw. Stub). Diese wird auf serverseitig eingebunden und zu einer konkreten Klasse abgeleitet, die die Methodenrümpfe implementiert.

Darüber hinaus soll der Compiler eine Proxy - Klasse für die Clientseite generieren, welche die Methodenaufrufe auf dem entfernten Server Objekt weiterleitet und

ausführt. Das Proxy Objekt wird mit der Abstrakten Klasse (Stub) im Client per narrowCast Methode erstellt.

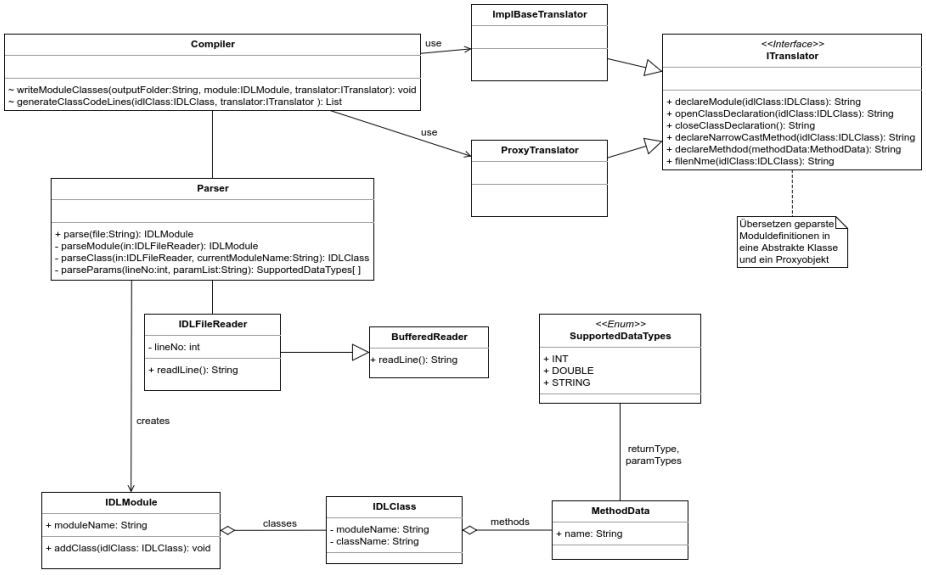

Abbildung 2: Fachliches Daten Modell des IDL-Compilers

3.2.2 Schnittstelle

```
public class Compiler {
    /**
     * Ueber die main – Methode der Klasse Compiler kann der Compiler ausgefuehrt werden.
     */
    public static void main(String[] args);
}
```

Zu übergebene Argumente sind:

- args[0]: Pfad zur IDL - Datei
- args[1]: Pfad zum Ordner für die Ausgabedateien

Die erzeugten Binaries können wie folgt ausgeführt werden:

```
./bin/neptr <IDL-Dateipfad> <Ausgabe-Ordner>
```

3.2.3 Entwurfsentscheidungen

Der Compiler besteht aus vier Komponenten (siehe Abbildung 1). In der Hauptkomponente (Compiler) ist der eigentliche Compiler mit seiner Mainmethode. Diese wird nach außen frei gegeben.

Der Compiler benutzt den Parser um eine IDL-Datei zu parsen und daraus ein IDLModule zu erstellen (siehe Abbildung 2). Diese IDLModule beinhaltet alle Klassen. Jede Klasse hat eine Ansammlung von Methoden. Der Compiler schreibt nachdem die IDL-Datei geparsed wurde alle erkannten Klassen im Modul der IDL-Datei mit Hilfe der Translator an den angegeben Ausgabepfad.

Die Translator haben ein fest definiertes Interface. Dadurch können leichter neue Translator erstellt werden und Translator ausgetauscht werden. Außerdem ist dadurch der Übersetzungsprozess logisch von der Syntax der zu übersetzenden Klasse getrennt. Es gibt insgesamt zwei Translators. Einen um die Stubs (ImplBase) zu erstellen und einen um die Proxy Klassen zu erstellen. Die Translators sind also dafür zuständig die IDLModules zu kompilieren in die gewünschten Java Klassen.

Die Parser Komponente und die IDLModel Komponente wurden mit Hilfe der vorgegeben Klassen aus der Aufgabe erstellt und erweitert.

3.3 mware lib

3.3.1 Aufgabe und Verantwortung

Die Bibiothek / das Package mware lib stellt die Kernfunktionalität bereit. Die Schnittstelle nach außen ist die Klasse ObjectBroker, die eine Referenz auf eine Implementierung des NameService Protokolls hält.

3.3.2 Schnittstelle

public NameService getNameService();
Gibt die Referenz auf das Proxyobjekt fuer den NameService an den Aufrufer.
public void shutDown();
Beendet den Object Request Broker und seine Dependencies.

3.3.3 Entwurfsentscheidungen

Der Object Request Broker wird gemäß Vorgabe in einer statischen Methode initialisiert. Hierbei wird ein Nameservice Proxyobjekt erzeugt, und zur weiteren Verwendung referenziert. Das Nameservice Proxyobjekt leitet alle Methodenaufrufe des NameService an eine entfernte NameService Instanz per TCP weiter.

3.3.4 Konfigurationsparameter

- host: Der Hostname des entfernten NameService
- Port: Der Port des entfernten NameService
- Debug: Konfiguriert, ob debug ausgaben geloggt werden sollen.