****

Autor

Titel

**Ausarbeitung/Abschlussarbeit**

Fach/Abschlussarbeit eingereicht im Rahmen Studium/Abschlussarbeit

im Studiengang Studiengang  
am Department Informatik  
der Fakultät Technik und Informatik  
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Erstprüfer  
Zweitgutachter : Zweitprüfer

Abgegeben am Datum

**Autor**

Titel

**Autor**

**Thema der Arbeit/Ausarbeitung**

…

**Stichworte**

…

**Kurzzusammenfassung**

…

**Autor**

**Title of the paper**

…

**Keywords**

…

**Abstract**

…

Inhaltsverzeichnis

[1 Ebene 1 5](#_Toc250905654)

[1.1 Ebene 2 5](#_Toc250905655)

[1.1.1 Ebene 3 5](#_Toc250905656)

1. Einleitung

Ziel dieser Arbeit ist die Konzeption einer einfachen objektorientieren Middleware, mit deren Hilfe Methodenaufrufe eines entfernten Objektes möglich sind.

Die Middleware soll aus nachfolgenden Teilen bestehen:

* **Namensdienst**
* Bibliothek ***mware\_lib***
* **Compiler**, der aus Definitionen in einer Schnittstellenbeschreibungssprache (im Folgenden IDL genannt) Quellcode (Java) generiert, den die Applikation zur Nutzung der Middleware als Bindeglied benötigt

Eine Applikation bindet zur Benutzung Ihrer Middleware *mware\_lib* und den mit Ihrem IDLCompiler generierten Code ein.

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

Der Namensdienst soll Namen auf Objektreferenzen abbilden. Er muss auf einem gesonderten Rechner im Netz laufen können. Sein Port muss zur Laufzeit mittels Startparameter einstellbar sein.

Die Bibliothek *mware\_lib* enthält alle Klassen und Interfaces, die die Middleware generell für den Betrieb benötigt und soll in einem Package zusammengefasst werden. Die zeitgleiche Nutzung ein und derselben Objektreferenz soll in der Middleware nicht zu Deadlocks führen.   
Diese ist in Java zu implementieren.

Das Projekt in Java hat nachfolgende Struktur:

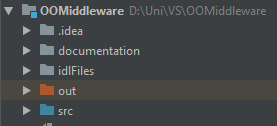


Abbildung 1: Projektstruktur in Java

* *documentation* enthält alle schriftlichen Ausarbeitung und Bilder
* *idlFiles* enthält die zu übersetzenden *.idl*-Dateien
* *out* enthält die Java-Binaries, welche über die Kommandozeile ausgeführt werden können
* *src* enthält den Sourcecode

1. Gesamtsystem

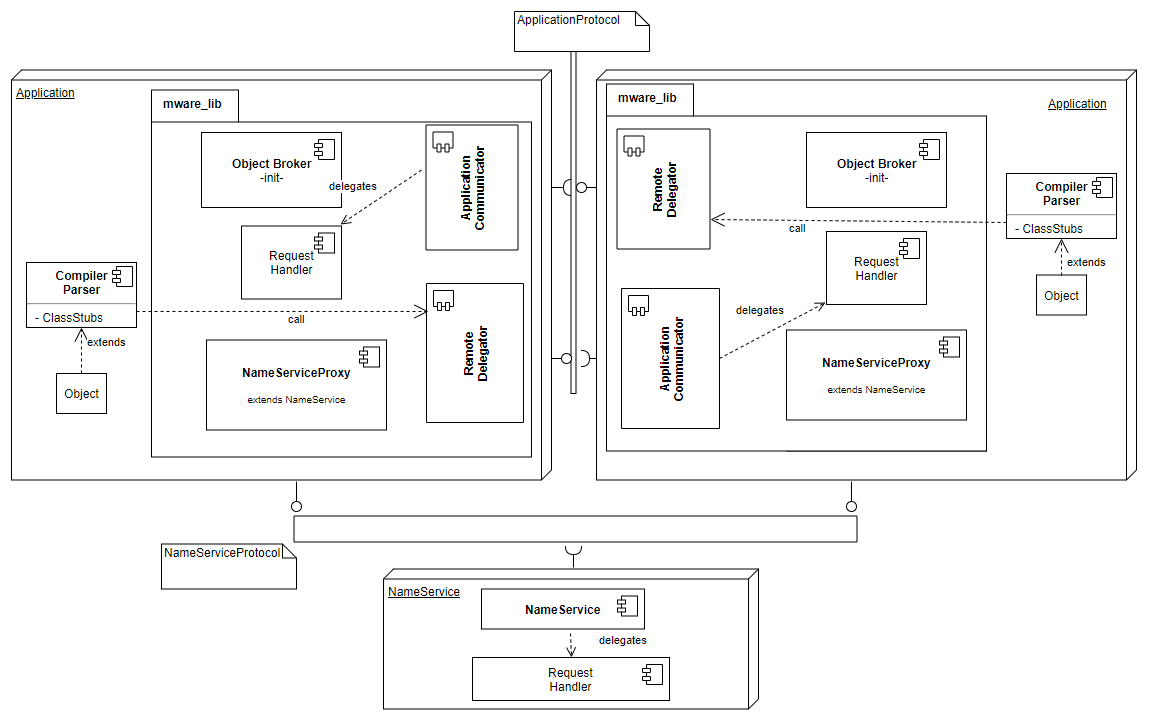


Abbildung 2: Überblick des Gesamtsystems

Obige Grafik verschafft einen ersten Eindruck über das Gesamtsystem. In den nachfolgenden Abschnitten wird auf jede Komponente näher eingegangen. Sämtliche Kommunikation zwischen Applikationen sowie zwischen den Applikationen und dem Namensdienst erfolgt über TCP/IP. Die Kommunikation erfolgt mittels *requests/reply*, deren Schema über Protokolle definiert ist.

1. Codegenerierung

Aus den Schnittstellenbeschreibung des Anwenders in IDL (*.idl*-Dateien) sollen entsprechenden benötigten Basisklassen generieren, welches als Bindeglied zwischen *mware\_lib* und dem Anwendungscode darstellen. Folgende Typbeschreibungen in *.idl* müssen erkannt werden:

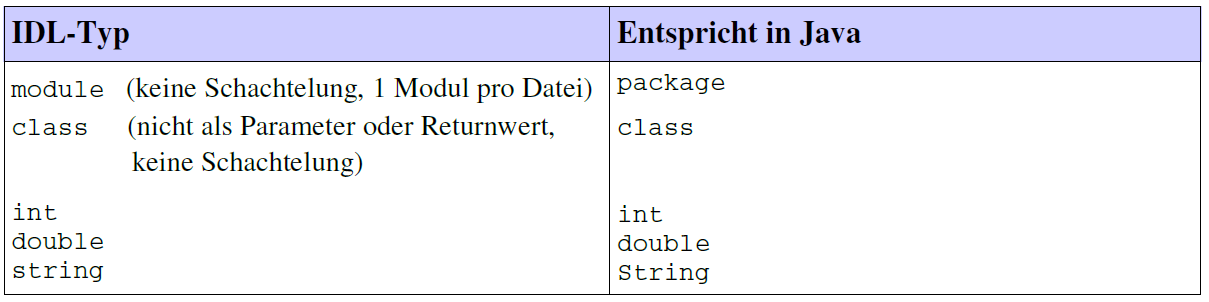


Abbildung 3: zu unterstützende IDL-Typen

Zur Einbindung der Anwenderklassen werden Schnittstellenklassen bereitgestellt, 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() (Namensdienst) gelieferten generischen Objektreferenz eine klassenspezifische Referenz liefert. Alle Klassen eines Moduls sollen in einem gleichnamigen Package zusammengefasst werden.

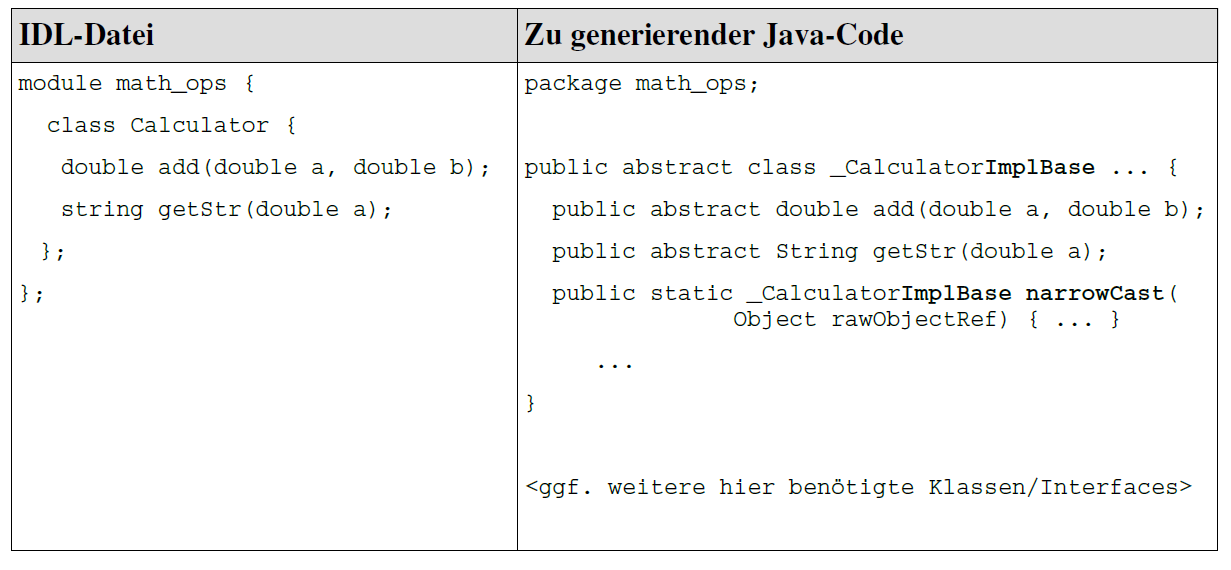


Abbildung 4: Java-Code Vorgabe

Die Struktur zur Kompilierung der benötigten *.java*-Dateien ergibt sich aus dem von H. Schulz vorgegebenen Beispielcode. Aus diesem Grund wird nur ein kurzer Überblick gegeben und dann auf Ergänzungen eingegangen. Das Paket *idl\_compiler* enthält alle benötigten Klassen:

* IDLmodule
* IDLclass
* IDLCompiler
* Parser

Wobei *IDLmodule, IDLclass* und *IDLCompiler* dazu dienen, die synaktische Korrektheit zu garantieren, indem sie festlegen, wie die IDL-Typen in Java zu übersetzen sind. Zugriff auf diese erfolgt über Methoden.

Der Parser nutzt oben genannte Klassen, um eine eingelesen *.idl*-Datei in eine *.java*-Datei zu übersetzen. Beim Start erhält er zwei Parameter:

* Pfad zu der zu übersetzenden *.idl*-Datei
* Pfad, wo das erzeugte package mit den dazugehörigen Klassen abgelegt werden soll

Beispielaufruf:



Der Parser wurde um die Methode writeToFile erweitert. Diese generiert ein package mit den dazugehörigen Javaklassen.

Der zu erzeugende Code hat die Struktur, welche in Abbildung 4 ersichtlich ist.

* 1. narrowCast

Es gab keine Vorgabe, wie genau die narrowCast(rawObjectRef) zu implementieren ist. Sie muss die Anforderung erfüllen, dass das von ihr zurückgelieferte Stellvertreterobjekt nach außen das gleiche Verhalten aufzeigt, wie die tatsächliche Implementation der Klasse, die möglicherweise auf einem anderen Rechner zu finden ist.

Das zurückgelieferte Stellvertreterobjekt speichert somit die ihm beim Aufruf übergebenen Objektreferenz, auf deren Struktur näher Im Kapitel *Nameservice* eingegangen wird.

Die Entwurfsentscheidungen zu narrowCast werden am folgenden Beispiel erläutert.

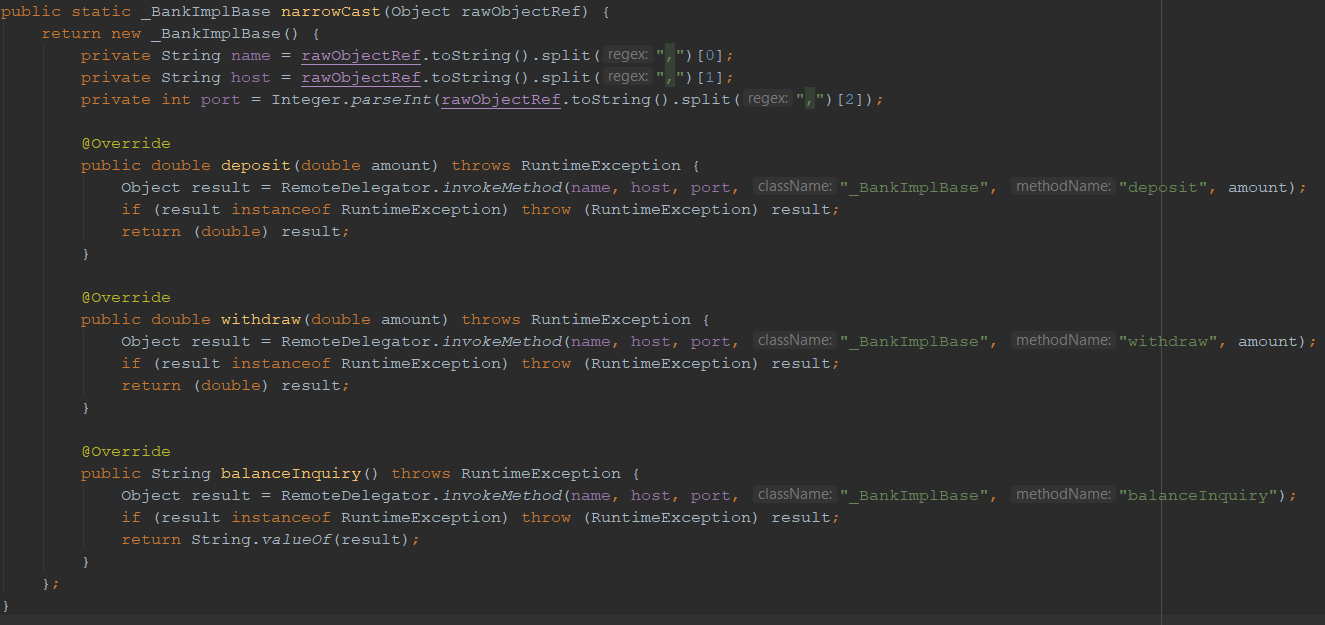


Abbildung 5: Beispiel narrowCast

Die Konstruktion des Objekts erfolgt direkt in der abstrakten Klasse. Auch möglich wäre die Erstellung einer eigenen Stellvertreterklasse, welche einem generischen Namensschema folgt und an die die Methodenaufrufe delegiert werden. Jedoch scheint dies eine unnötige zusätzliche Abstraktionsebene zu sein, welche zu geringerer Übersicht führt.

Da die tatsächliche Realisierung der Methoden nicht bekannt ist, kann deren Verhalten nicht vorhergesagt werden. Dementsprechende wird das Ergebnis eines Methodenaufrufs zwischengespeichert und überprüft. Sollte dies einer RuntimeException entsprechen, so wird diese weitergereicht. Somit kann nach außen nicht zwischen Stellvertreterobjekt und tatsächlicher Implementation unterschieden werden.

Der Rückgabewert des entfernten Objektaufrufs entspricht aus diesem Grund *Object* und wird erst im Nachhinein in den geforderten Rückgabewert gecastet. Für das Typecasting wurde der Compiler um die Methode getSupportedJavaDataTypeNameForReturnValue(SupportedDataTypes returnType) ergänzt. Um die Namen der Parameter aus der *.idl*-Datei zu erhalten, wurde der Parser um die Methode parseParamNames(int lineNo, String paramList) erweitert.

Der tatsächliche Methodenaufruf erfolgt durch den Aufruf der invokeMethod-Methode des RemoteDelegators, auf die im Kapitel *mware\_lib* genauer eingegangen wird. Alle dafür benötigten Parameter sind bereits zur Compilezeit bekannt, wobei *name, host* und *port* natürlich erst zur Laufzeit bei Übergabe der Objektreferenz feststehen.

1. Nameservice

Aufgabe des Namensdiensts ist, Namen auf Objektreferenzen abzubilden. Hierfür muss dieser über eine Datenstruktur verfügen, welche dies ermöglicht. Konkret wurde eine *ConcurrentHashMap* gewählt. Eine Map erfüllt genau die an den Namensdienst gestellten Anforderungen. Diese wurde in einen kritischen Bereich gelegt (*Concurrent*). Somit können keine unerwarteten *read/write*-Fehler auftreten. Diese sind möglich, da der Nameservice die erhaltenen Namensauflösungen delegiert, um eingehende Anfragen möglichst schnell beantworten zu können. Um zu gewährleisten, dass zur Laufzeit nur eine Instanz eines Namensdiensts zur Verfügung steht, folgt dieser dem Singleton-Pattern.

Der Aufruf des Namensdiensts folgt dem gleichen Muster, welches bereits beschrieben wurde. Ihm wird beim Start ein Parameter übergeben. Hierbei handelt sich um den Port, auf dem er auf eingehende Anfragen wartet.

* 1. RequestHandler

Sobald der Namensdienst eine Anfrage erhält, erstellt er einen neuen *RequestHandler*-Thread, welcher diese vorerst auf Protokollkonformität prüft. Sofern diese eingehalten wurde, führt er die Anfrage aus.

* rebind: legt Objektreferenz unter übergebenem Namen ab
* resolve: löst übergebenen Namen in Objektreferenz auf und übermittelt diese an den Anfrager

Anschließend terminiert sich der RequestHandler. Pro Anfrage wird folglich ein eigener Thread erstellt.

* 1. NameServiceProtocol

Die Kommunikation mit dem Namensdienst ist über ein Protokoll geregelt.

Das Protokoll ist als statische Klasse implementiert, welche Methoden liefert, um Anfragen zu erstellen und Antworten auszulesen. Eine Anfrage wird als Text zusammengefasst.

* + 1. Reply/Request types

Folgende Typen werden unterstüzt:

* REBIND: bindet Objektreferenz
* RESOLVE: lost Objektreferenz auf
* UNKNOWN: übergebener Typ nicht Protokollkonform
* SUCCESS: Namensauflösung erfolgreich
* FAILURE: Namensauflösung fehlgeschlagen
  + 1. Methoden

Die Erzeugung einer Anfrage erfolgt über

createRequest(String type, String objectName, String hostname, int hostport),

Die Erzeugung einer Antwort erfolgt über

createRequest(String type, String objectName, String hostname, int hostport)

Das Auslesen der Information erfolgt über die Methoden

* getType(String message)
* getObjectName(String message)
* getHost(String message)
* getPort(String message)

Wenn erwünscht, kann dem Kommunikationspartner mitgeteilt werden, dass seine Anfrage nicht verstanden wurde. In der Praxis ist es üblicher, gar nicht auf diese zu antworten (um CPU-Zeit zu sparen/keine Informationen preiszugeben). Diese Mitteilung erfolgt über

createUnknownProtocol

1. mware\_lib

*mware\_lib* ist ein package, dass alle Klassen und Interfaces enthält, die die Middleware generell für den Betrieb benötigt, unabhängig vom Aussehen der aktuellen Anwendungsschnittstellen.

In diesem Kapitel werden zuerst die einzelnen Bestandteile von *mware\_lib* aufgelistet. Diese Auflistung geht kurz auf die jeweilige Aufgabe sowie alle enthaltenen Methoden ein. Anschließend wird das Zusammenspiel der Module (in Java:Klassen) anhand eines Beispiels verdeutlicht.

* 1. Bestandteile
     1. ObjectBroker

Der *ObjectBroker* ist der zentrale Einstiegspunkt der Middleware aus Applikationssicht. Um zu gewährleisten, dass zur Laufzeit nur eine Instanz des *ObjectBrokers* zur Verfügung steht, folgt dieser dem Singleton-Pattern.

**Methoden**

* init(String serviceHost, int listenPort, boolean debug): liefert ein *ObjectBroker*-Objekt zurück, über das die Applikation mit der Middleware kommuniziert. Startet den *ApplicationCommunicater*
* getNameService(): liefert den Namensdienst zurück
* shutdown(): beendet die Benutzung der Middleware
  + 1. NameServiceProxy

Da der tatsächliche Namensdienst nicht Bestandteile der Middleware ist, benötigt diese ein Stellvertreterobjekt, welche das Verhalten des tatsächlichen Namensdiensts imitiert.

Dies wird gewährleistet, indem sich die Klasse *NameServiceProxy* von der abstrakten Klasse *NameService* ableitet. Um zu gewährleisten, dass zur Laufzeit nur eine Instanz des Namensdienst-Stellvertreters zur Verfügung steht, folgt dieser dem Singleton-Pattern.  
Der *NameServiceProxy* verfügt über ebenfalls über die gleichen Datenstrukturen wie der globale Namensdienst, mit dem Unterschied, dass bei dessen dictionary (in Java: Map) die konkreten Objekte abgelegt werden.

Da auch hier mehrere Anfragen gleichzeitig eintreffen können, ist die Map ebenfalls als *ConcurrentHashMap* implementiert.

**Methoden**

* rebind(Object servant, String name): legt ein Objektreferenz beim Namensdienst ab. Beim Aufruf von rebind wird eine Verbindung zum globalen (dessen Adresse beim Start des *ObjectBrokers* festgelegt wurde) Namensdienst hergestellt, und eine rebind-Anfrage gemäß Protokoll gesendet. Zusätzlich wird das tatsächliche Objekt in der Map gespeichert
* resolve(String name): löst übergebenen Namen in Objektreferenz auf. Beim Aufruf von resolve wird eine Verbindung zum globalen Namensdienst hergestellt, und eine resolve-Anfrage gemäß Protokoll gesendet
* resolveLocally(String name): löst übergebenen Namen in Objekt auf und gibt dieses zurück
  + 1. RemoteDelegator

Der *RemoteDelegator* ist der Ansprecher für die aus den *.idl*-Dateien generierten Stellvertreterklassen.

**Methoden**

Über invokeMethod(String objectName, String locationHost, int locationPort, String className, String methodName, Object... params)

reicht eine Stellvertreterklasse ihren Methodenaufruf an den *RemoteDelegator* weiter. Beim Aufruf von invokeMethod kontaktiert der *RemoteDelegator* die Adresse, welche über die Parameter locationHost und locationPort mitgegeben wurde. Dessen Kommunikation mit anderen Applikation ist über das *ApplicationProtocol* (siehe Abschnitt Protokolle) geregelt.

* + 1. ApplicationCommunicater

Der *ApplicationCommunicater* wird beim Start der Middleware durch den Objektbroker gestartet. Beim Start werden ihm der Port für eingehende Anfragen und die Adresse des Namensdiensts übermittelt.

Er lauscht auf dem beim Start übergebenen Port. Eingehende Anfragen delegiert er an den *RequestHandler,* indem er pro Anfrage einen neuen Thread startet.

**Methoden**

* run(): beginnt auf übergebenem Port auf Anfragen zu lauschen
* shutDown(): beendet den *ApplicationCommunicater*
  + 1. RequestHandler

Der *RequestHandler* bearbeitet eingehende Anfragen des *ApplicationCommunicaters*. Der RequestHandler wird als Thread ausgeführt und beantwortet genau eine Anfrage.  
Bei der Erzeugung werden ihm die Adresse des globalen Namensdiensts sowie das aktuelle Socket, von dem die Anfrage kam, übergeben.

**Methoden**

**Versicherung über Selbstständigkeit**

*Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.*

*Hamburg, den \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*