Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Ausarbeitung

Philipp Goemann

Konzeption einer Objektorientieren Middleware

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Philipp Goemann

Konzeption einer Objektorientieren Middleware

Facharbeit eingereicht im Rahmen des Studiums

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

Abgegeben am 12.06.2018

Inhaltsverzeichnis

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:

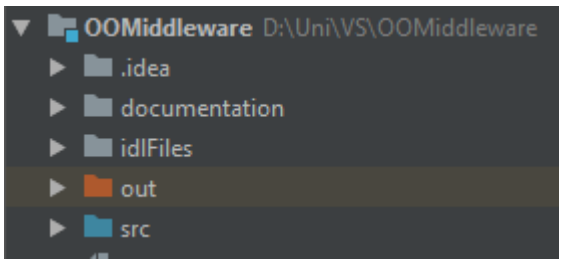

Abbildung 1: Projektstruktur in Java

- *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

2 Gesamtsystem

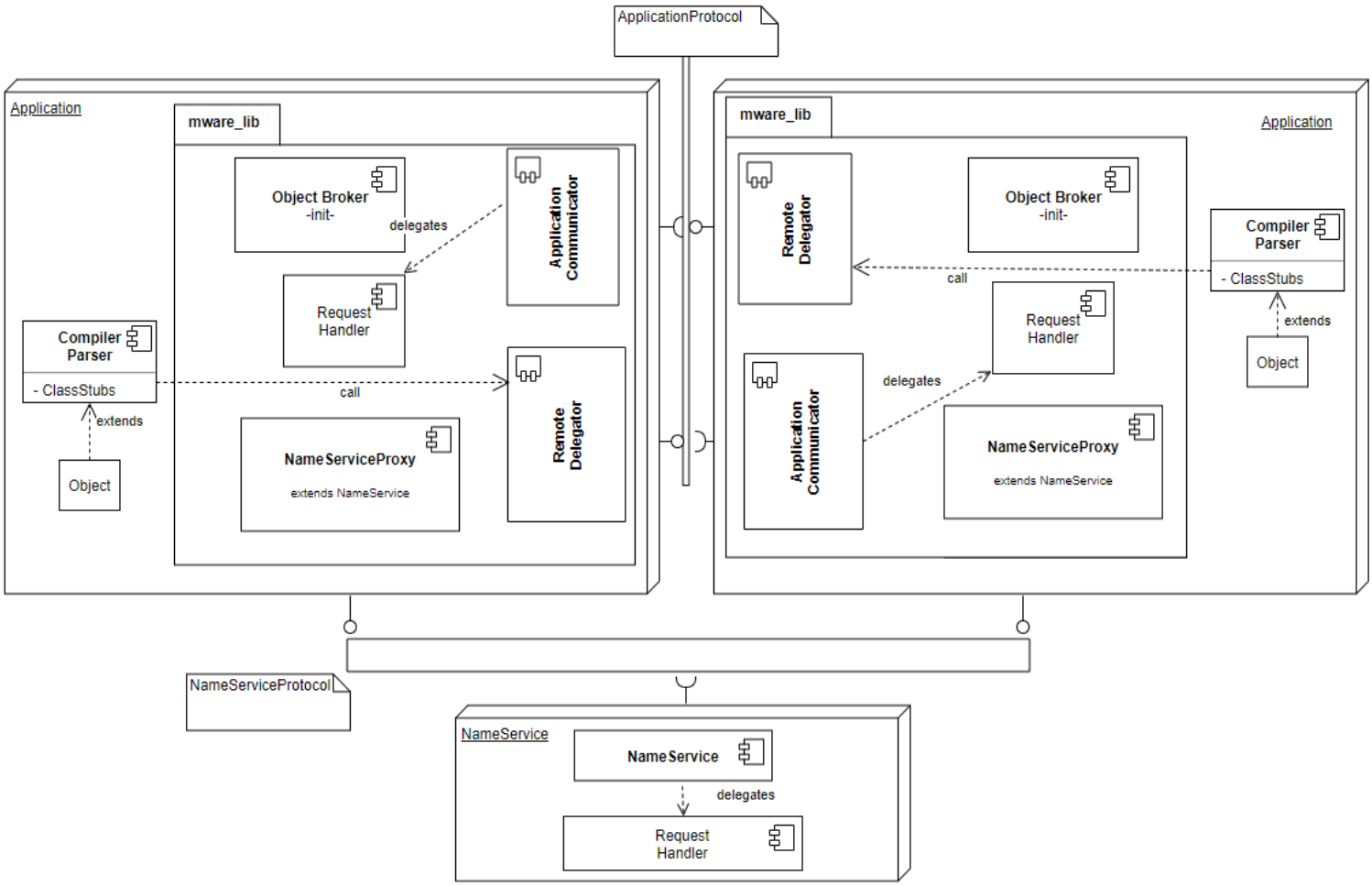

Abbildung 2: Überblick des Gesamtsystems

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.

3 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:

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`

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.

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 _CalculatorImplBase ... {
	  public abstract double add(double a, double b);
	  public abstract String getStr(double a);
	  public static _CalculatorImplBase narrowCast(
	  Object rawObjectRef) { ... }
	  ...
	}
	
	<ggf. weitere hier benötigte Klassen/Interfaces>

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 syntaktische 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:

```
java idl_compiler/Parser D:\\Uni\\VS\\OOMiddleware\\idlFiles\\bank.idl D:\\Uni\\VS\\OOMiddleware\\src
```

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.

3.1 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.

```
public static _BankImplBase narrowCast(Object rawObjectRef) {
    return new _BankImplBase() {
        private String name = rawObjectRef.toString().split( regex: ",")[0];
        private String host = rawObjectRef.toString().split( regex: ",")[1];
        private int port = Integer.parseInt(rawObjectRef.toString().split( regex: ",")[2]);

        @Override
        public double deposit(double amount) throws RuntimeException {
            Object result = RemoteDelegator.invokeMethod(name, host, port, className: "_BankImplBase", methodName: "deposit", amount);
            if (result instanceof RuntimeException) throw (RuntimeException) result;
            return (double) result;
        }

        @Override
        public double withdraw(double amount) throws RuntimeException {
            Object result = RemoteDelegator.invokeMethod(name, host, port, className: "_BankImplBase", methodName: "withdraw", amount);
            if (result instanceof RuntimeException) throw (RuntimeException) result;
            return (double) result;
        }

        @Override
        public String balanceInquiry() throws RuntimeException {
            Object result = RemoteDelegator.invokeMethod(name, host, port, className: "_BankImplBase", methodName: "balanceInquiry");
            if (result instanceof RuntimeException) throw (RuntimeException) result;
            return String.valueOf(result);
        }
    };
}
```

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.

4 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. Die Objektreferenz wird konkret als ein zweistelliges Array der Form [LocationHost, LocationPort] gespeichert. 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.

4.1 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.

4.2 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.

4.2.1 Reply/Request types

Folgende Typen werden unterstützt:

- REBIND: bindet Objektreferenz
- RESOLVE: lost Objektreferenz auf
- UNKNOWN: übergebener Typ nicht Protokollkonform
- SUCCESS: Namensauflösung erfolgreich
- FAILURE: Namensauflösung fehlgeschlagen

4.2.2 Methoden

Die Erzeugung einer Anfrage erfolgt über

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

Die Erzeugung einer Antwort erfolgt über

- createResolveResponse(String objectName, String locationHost, String locationPort)

Das Auslesen der Information erfolgt über die Methoden

- getType(String message)
- getObjectName(String message)
- getHost(String message)
- getPort(String message)
- extractObjectHost(String[] response)
- extractObjectPort(String[] response)

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

5 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. Im darauf folgenden Kapitel wird das Zusammenspiel der Module (in Java:Klassen) anhand eines Beispiels verdeutlicht.

5.1 Bestandteile

5.1.1 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

5.1.2 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

5.1.3 RemoteDelegator

Der *RemoteDelegator* ist der Ansprechpartner 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.

5.1.4 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 (genauer: pro Verbindungsaufbau) einen neuen Thread startet.

Methoden

- run(): beginnt auf übergebenem Port auf Anfragen zu lauschen
- shutDown(): beendet den *ApplicationCommunicater*

5.1.5 RequestHandler

Der *RequestHandler* bearbeitet die eingehende Anfrage, die auf seinem Socket eintrifft. 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

- run(): startet den *RequestHandler*, wartet auf eingehende Anfrage
- invokeMethod(String objectName, String className, String methodName, String[] params): führt den gewünschten Methodenaufruf aus. Dazu holt der *RequestHandler* sich das Objekt vom lokalen Namensdienst, indem er dessen resolveLocally()-Methode mit objectName aufruft. Anschließend wird mittels eigener „Reflection“ die Methode ausgeführt. Die Antwort wird über das *ApplicationProtocol* generiert und über das Socket weitergereicht.

5.1.6 ReflectionUtil

Das *ReflectionUtil* beinhaltet Methoden, um Methodenaufrufe auf konkreten Objekten zu ermöglichen.

Methoden

- getParameterTypes(String[] params): gibt eine Liste vom Typ Class mit den jeweiligen Klassen der Parameter zurück. Iteriert dazu über params und ruft für jedes Objekt getParameterType(String param) auf
- getParameterValues(String[] params): gibt eine Liste vom Typ Object mit den tatsächlichen Werten zurück. Iteriert dazu über params und ruft für jedes Objekt getParameterValue(String param) auf
- getParameterType(String param): gibt den tatsächlichen Typen zurück
- getParameterValue(String param): gibt den tatsächlichen Wert zurück
- getException(String param): liest aus dem übergebenen String Exception-Typ und Exception-Nachricht aus und erstellt eine entsprechende Exception.

5.2 Protokolle

5.2.1 NameServiceProtocol

Das verwendete Protokoll, damit der *NameServiceProxy* den globalen Namensdienst ansprechen kann, ist dasselbe.

5.2.2 ApplicationProtocol

Das *ApplicationProtocol* regelt die Kommunikation der Applikationen untereinander.
Die Erzeugung einer Anfrage erfolgt über

- requestMethodExecution(String objectName, String className, String methodName, Object... params)

Die Erzeugung einer Antwort erfolgt über

- createReply(Object result, Throwable throwable)

Die Antwort entspricht dabei der Form [<Result>,<Throwable>]. Einer der beiden Werte ist immer null (welcher ist abhängig davon, ob die Methoden erfolgreich aufgerufen werden konnte oder nicht) und der andere entspricht immer dem Rückgabetyp der Funktion.

Das Auslesen der Information erfolgt über die Methoden

- getObjectName(String message)
- getClassName(String message)
- getEssentialClassName(String message)
- getMethodName(String message)
- getParams(String message)

Da hier der Name von getEssentialClassName nicht zwingend selbsterklärend ist, wird diese kurz erläutert: vom übergebenen Klassennamen des Stellvertreterobjekts wird der führende Unterstrich sowie der Zusatz „Impl“ entfernt, sodass der zurückgegebene Name dem tatsächlichen Klassennamen entspricht. Da dies jedoch auf der Vereinbarung beruht, dass tatsächliche Implementationen von Klassen dieser Namensgebungkonvention folgen, was nicht gewährleistet werden kann, wurde auf den Einsatz der Methode verzichtet.

6 Ablauf

In diesem Kapitel wird beispielhaft ein erfolgreiches Binden eines Objekts einer Applikation gezeigt (hier: Server) und der anschließende erfolgreiche Aufruf einer Methode einer Applikation (hier: Client) auf einem Stellvertreterobjekt aufgezeigt.

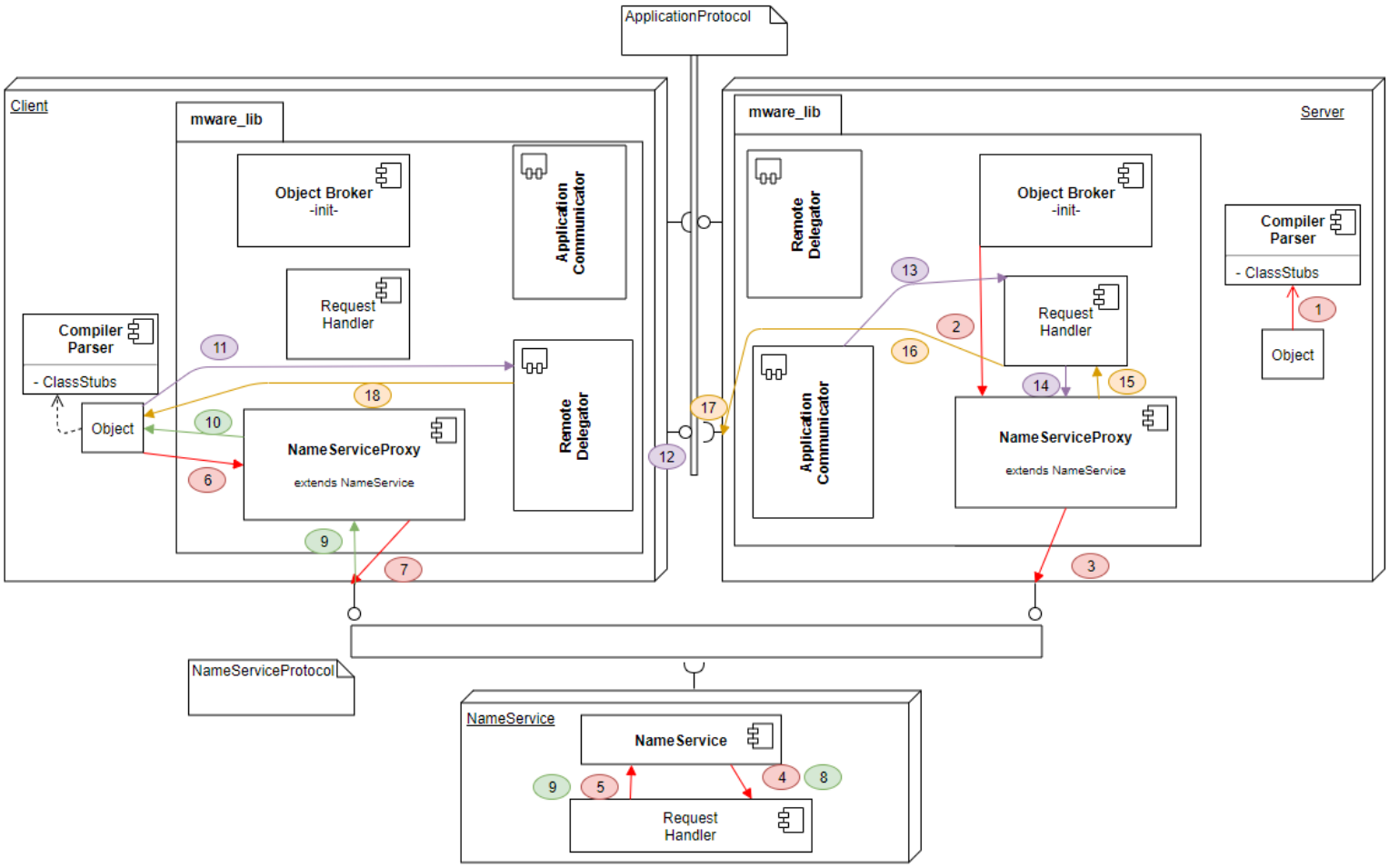

Abbildung 6: Programmablauf

1. Ein Objekt des Servers ist eine Subklasse einer der generierten Stellvertreterklassen.
2. Über den ObjektBroker wird der lokale Namensdienst ermittelt und diesem wird ein *rebind*-Request gestellt. Dieser speichert das konkrete Objekt unter übergebenem Namen ab.
3. Der lokale Namensdienst erstellt eine Verbindung zum globalen Namensdienst, und übermittelt einen *rebind*-Request, welcher den Name des Objekts sowie die Adresse des Rechners, auf dem die Applikation läuft, enthält.
4. Der globale Namensdienst delegiert diese Anfrage an den RequestHandler.
5. Der RequestHandler hinterlegt die Objektreferenz unter dem Namen in der Datenstruktur des globalen Namensdiensts. Das *Rebind* ist hiermit abgeschlossen.
6. Die Clientapplikation erzeugt ein Objekt der gleichen generierten Stellvertreterklasse mittels *resolve.*
7. Der lokale Namensdienst holt sich die dafür benötigte Referenz, wo das tatsächliche Objekt hinterlegt ist, beim globalen Namensdienst. Hierfür baut er Verbindung zu diesem auf und sendet einen *resolve*-Request.
8. Der globale Namensdienst delegiert diese Anfrage an den RequestHandler.
9. Der RequestHandler löst die Objektreferenz unter dem Namen in der Datenstruktur des globalen Namensdiensts auf und sendet diese an lokalen Namensdienst.
10. Die Erzeugung des Stellvertreterobjekts ist abschlossen. Diesen enthält die Adresse, an der die tatsächliche Implementierung vorliegt. Das *Resolve* ist hiermit abgeschlossen.
11. Es wird eine Methode des Stellvertreters aufgerufen. Diese leitet den gewünschten Methodenaufruf an den *RemoteDelegator* weiter.
12. Der *RemoteDelegator* stellt eine Verbindung mit dem der ihn übergebenen Adresse dar und stellt einen *requestMethodExecution*-Request.
13. Der *ApplicationCommunicater* des Servers erhält diese Anfrage und delegiert sie an den *RequestHandler*.
14. Der *RequestHandler* stellt eine *resolveLocally*-Anfrage an den lokalen Namensdienst.
15. Der lokale Namensdienst übermittelt dem RequestHandler das Objekt, welcher unter übergebenem Namen von diesem gespeichert wurde (siehe Schritt 2).
16. Auf dem übergebenen Objekt wird mit Hilfe des *ReflectionUtils* die gewünschte Methode mit (eventuell) übergebenen Parametern aufgerufen und das Ergebnis protokollkonform übergeben.
17. Der *RemoteDelegator* des Clients erhält die Antwort auf seine Anfrage und extrahiert das Ergebnis des Methodenaufrufs aus dieser.
18. Das Objekt, welches in Schritt 11 die Methode aufgerufen hat, erhält das Ergebnis dieser, übersetzt es in den entsprechenden Rückgabetyp und gibt es zurück.

7 Vorgehen

Dieses Kapitel setzt sich damit auseinander, wie diese Aufgabe bearbeitet wurde.
Es wurde viel Zeit in eine gute Struktur investiert. Diese beinhaltet

- Entwurf/Erwerbung fehlender Kenntnisse
- Logging
- Testumgebung

7.1 Entwurf

Der Entwurf wurde hauptsächlich auf Papier erstellt. Beispiele dessen sind unter anderem in Abbildung 2 und Abbildung 6 zu sehen.
Es wurde ein Top-Down Ansatz verfolgt. Als erstes wurden die benötigten Komponenten entworfen und dann um deren Klassen ergänzt. Anschließend wurden diese mit Methoden befüllt. Die Methodenrümpfe blieben vorerst leer. Somit ließ sich gut die Sinnhaftigkeit des Systems überprüfen, ohne Kaschierungen durch Implementationen zu ermöglichen. Nachdem alle Methoden mit deren Parametern + Rückgabewerten feststanden, wurde erneut auf Papier alle möglichen Szenarien durchgespielt um den Entwurf auf Vollständigkeit zu prüfen. Der nächste Schritt war die Identifikation von Wissenslücken. Mangelnde Kenntnis bestand in den Bereichen der Kompilierung, der Konzeption von Protokollen und Reflection.

7.2 Testumgebung

Als nächstes wurden Tests geschrieben, um die vom Entwurf zu erwartenden Funktionalität zu überprüfen. Diese wurde vor der Implementation geschrieben, um unvoreingenommen zu sein. Im Verlauf wurden der Implementation wurde diese bei Bedarf erweitert. Es wurden keine Tests für die Umwandlung von *.idl*-Dateien in *.java*-Dateien geschrieben, da die Tests nur Laufzeitfehler abdecken. Syntaktische Fehler können durch bloßes Hinsehen + IDL-Highlighting erkannt werden. Die Logik der generierten Stubs wurde durch die anderen Tests mit abgedeckt. Nachfolgende Grafik zeigt, dass unter Anbetracht der begrenzten Zeit eine zufriedenstellende Testabdeckung erfolgt ist.

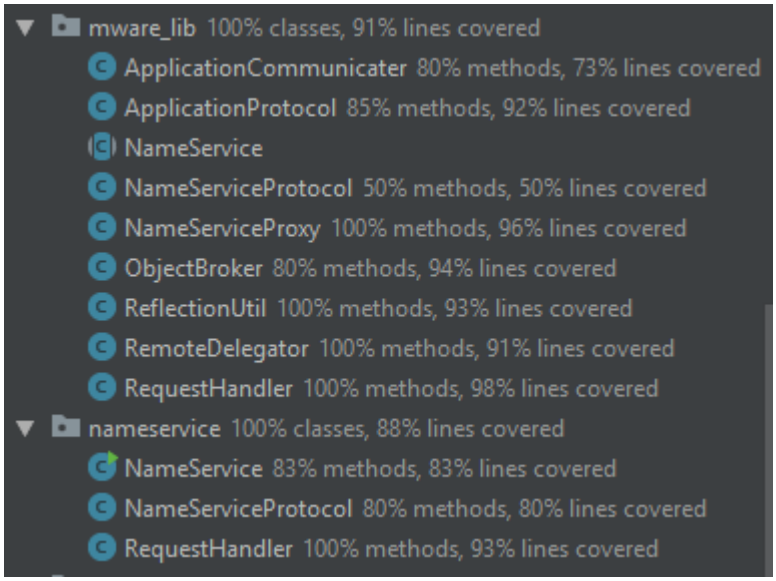

Abbildung 7: Testabdeckung

Während der Implementation wurde darauf geachtet, die Tests, welche aufeinander aufbauen, nacheinander zu erfüllen. Bei jedem neuen Feature wurden die Tests erneut durchlaufen um zu gewährleisten, dass keine unerwarteten Nebeneffekte auftreten.

7.3 Logging

Alle Klassen wurden mit einem Logger versehen, der an relevanten Stellen Ausgaben erzeugt. Somit kann bei unerwartetem Verhalten der Fehler schnell aufgefunden werden. Da unter IntelliJ IDEA entwickelt wurde und dies nützliche Werkzeuge zum Bearbeiten von Logs enthält, wird der Log nicht in eine Datei geschrieben. Eine Beispielausgabe aller durchlaufenen Tests ist dieser Arbeit jedoch beigelegt, welche auch die relevanten Loggingausgaben enthält. Ist das Schreiben in eine Datei jedoch erwünscht, so kann dies durch das Anhängen eines FileHandles (siehe Quellen).

8 Reflektion

Die strukturierte Vorgehensweise ermöglichte mir eine Aufgabe zu lösen, die für meinen aktuellen Kenntnisstand einen sehr hohen Anspruch hatte. Leider wurde bezüglich der Konzeption der Protokolle beim Entwurf eine schlechte Entscheidung getroffen, die mangels verbleibender Zeit nicht mehr korrigiert werden konnte. Hierbei handelt es sich um die Entscheidungen, dass die Nachrichten als Textrepräsentation versandt werden. Wobei die übergebenen Parameter mittels Komma getrennt sind. Beim Lesen dieser wird also das Komma als Separator genutzt. Sollte jedoch als Parameter ein String übergeben werden, welcher ein Komma enthält, so schlägt das Extrahieren der gewünschten Information fehl, da dieser String fälschlicherweise als mehrere Parameter angesehen wird. Bei der Erkennung dieses Fehlers wurden Mechanismen eingebaut, diesen teilweise zu umgehen. Jedoch ist dies eine grundlegende Entwurfsentscheidung, die nicht ohne Weiteres komplett behoben werden kann. Vor allem sind alle Ansätze, diesen Fehler zu beheben, nur Kaschierungen und damit lässt sich über deren Sinnhaftigkeit streiten.
Im Nachhinein hätten die Daten als Bytes übergeben werden, welches die Problematik der Separatoren behebt. Im Protokoll kann festgelegt werden, wie die Parametern voneinander zu trennen sind, ohne dass es zu erwünschten Nebeneffekten kommen kann.

9 Fazit

Dafür das diese Art von Aufgabe neu für mich war und ich viele Dinge zum ersten Mal gemacht habe, bin insgesamt mit dem Ergebnis zufrieden. Wobei sich diese Zufriedenheit weniger an der tatsächlichen Implementation misst, sondern vielmehr um den Ansatz, wie eine solches Projekt angegangen wird. Hier habe ich den vergangenen Wochen viel dazugelernt, und vor allem bin ich froh darüber, dass ich meine Erkenntnisse aus dem gescheiterten Versuch von Aufgabe 3 direkt umsetzen konnte und sehe, wie sehr mir diese die Art zu entwickeln erleichtert haben. Somit hat die Bearbeitung dieser Aufgabe sogar sehr viel Spaß gemacht. Dem Zeitdruck geschuldet bin ich mit der Umsetzung dennoch nicht zufrieden. Allerdings ist es auch schön zu wissen, dass man direkt Verbesserungsmöglichkeiten sieht, sodass man weiß, dass man dazugelernt hat.

10 Quellenangaben

Es wurde der Von Hartmut Schulz zur Verfügung gestellte Code zum parsen/kompilieren genutzt.

Weitere Quellen:

Zum Zeitpunkt der Abgabe (12.06.2018) sind alle Quellen erreichbar und unverändert.

https://docs.oracle.com/javase/tutorial/essential/io/pathOps.html - Informationen zum Dateisystem unter Java

https://docs.oracle.com/javase/tutorial/networking/sockets/index.html - Informationen über die Netzwerkkommunikation in Java

https://docs.oracle.com/javase/tutorial/reflect/member/methodInvocation.html - Informationen zu Javas Reflection-Mechanismus

https://docs.oracle.com/javase/7/docs/api/java/util/logging/FileHandler.html
Informationen dazu, wie der im Projekt genutzte Logger in eine (oder mehrere) Dateien schreibt

11 Abbildungsverzeichnis

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 12.06.2018