



# TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät für Informatik  
Institut für Systemarchitektur  
Professur für Rechnernetze  
Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill

## DIPLOMARBEIT

zum Thema

### **Dynamisches Deployment XMPP-basierter Dienste für Mobile Social Apps**

vorgelegt am 31.01.2012 von:

**Danny Kiefner**

geboren am: 01.01.1986 in Lutherstadt Wittenberg

s0274122@mail.zih.tu-dresden.de

Matrikelnummer: 3340893

Betreuer:

Dr.-Ing. Daniel Schuster

Daniel.Schuster@tu-dresden.de



## AUFGABENSTELLUNG FÜR DIE DIPLOMARBEIT

Name, Vorname:

Kiefner, Danny

Studiengang:

Informatik

Matr.-Nr.: 3340893

Thema:

**Dynamisches Deployment XMPP-basierter Dienste für Mobile Social Apps**

### ZIELSTELLUNG

Mobile Social Apps (z.B. kollaborative Spiele, Buddy-Finder, Social Networks) werden meist als Client-Server-Anwendungen mit komplexen, auf den Anwendungsfall zugeschnittenen Netzwerkprotokollen realisiert. Die Entwicklung und das Deployment der Server-Komponenten gestalten sich für den Entwickler schwierig und zeitaufwendig. Clients auf verschiedenen Endgeräten müssen jeweils neu entwickelt und das Protokoll aufwendig auf unterschiedlichen mobilen Plattformen realisiert werden.

In der Arbeit soll deshalb ein Service-orientierter Ansatz zur Entwicklung solcher mobiler sozialer Anwendungen realisiert werden, der den Entwickler bei der Entwicklung und dem Deployment des Netzwerkprotokolls und des zugehörigen Server-Dienstes unterstützt. Es soll eine XML-basierte Sprache entwickelt werden, die die Beschreibung der Dienstschnittstellen und Protokollnachrichten von XMPP-basierten Diensten erlaubt und mittels Code-Generierung auf verschiedenen Plattformen als Basis für die Client- und Server-Entwicklung dient. Des Weiteren soll ein dynamisches Binden der Clients an den Dienst zur Laufzeit unterstützt werden. Das Konzept soll im Rahmen der Mobilis-Plattform prototypisch implementiert und auf zwei ausgewählten mobilen Plattformen getestet werden.

### SCHWERPUNKTE

- Grundlagen: Service-orientierte Architekturen, XMPP-Erweiterungen, mobile Plattformen
- Entwurf der Dienstbeschreibungssprache
- Konzeption für Codegenerierung, dynamisches Deployment, Discovery/Binden von Diensten
- Implementierung und Test innerhalb der Mobilis-Plattform

Betreuer:

Dr.-Ing. Daniel Schuster

Betreuernder

Hochschullehrer:

Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill

Beginn am:

01.08.2011

Einzureichen am:

31.01.2012

i.A. Braun

Unterschrift des betreuenden Hochschullehrers



# Danksagung

Ich möchte mich an erster Stelle bei Daniel Schuster, dem Betreuer dieser Diplomarbeit, für die Geduld und die wertvollen Ratschläge bedanken, die ich während dieser Zeit von ihm erhalten habe.

Ein großes Dankeschön möchte ich auch den Mitarbeitern des Mobilis-Projektes und vor allem Robert Lübke zukommen lassen. Sie hatten immer ein offenes Ohr und viele Ideen, die mich über die Zeit im Mobilis-Projekt begleitet haben.

Meiner Familie und meinen Freunden möchte ich für die großartige Unterstützung während meiner gesamten Studienzeit danken. In ihnen fand ich stets den Rückhalt, der in schwierigen Zeiten unbezahlbar war. Meinen Nachbarn möchte ich für die kleinen aber feinen Aufmunterungen und wohltuenden Worte während der Diplomphase danken.

Abschließend danke ich von ganzem Herzen meiner Freundin Christin dafür, dass sie stets verständnisvoll und mit viel Fürsorge für mich da war – Tag für Tag. Du hast mir nicht nur die Studienzeit zu einem unvergesslichen und einzigartigen Erlebnis gemacht. Ich danke dir.



# Abstract

Die Entwicklung von Social Software im mobilen Bereich verlangt in der heutigen Zeit ein hohes Maß an Dynamik. Neben einfachen kollaborativen Anwendungsaspekten, wie zum Beispiel Chatten, gewinnen *Value Added Services* immer mehr an Bedeutung. Insbesondere die Integration sozialer Netzwerke, wie zum Beispiel Twitter oder Facebook, sind eine oft gesehene Erweiterungen. Unterwegs die Freunde aus Facebook in der Nähe finden und mit diesen ad hoc ein ortsbasiertes Spiel starten, stellt nur eines dieser Szenarien dar.

Mit der Entwicklung einer Mobile Social App muss der Entwickler in den meisten Fällen die kollaborativen oder sozialen Komponenten und Schnittstellen selber entwickeln und in das System integrieren. Im ungünstigsten Fall muss er das für jede Plattform erneut machen und kann sich somit nicht auf die wesentliche Logik der Applikation konzentrieren. Die meisten dieser Anwendungen werden als Client-Server-Architektur umgesetzt, sodass diese *Value Added Services* vorwiegend auf dem Server integriert werden.

In dieser Arbeit soll der Entwicklungsprozess zur Erstellung von Mobile Social Apps optimiert werden. Ziel ist die Umsetzung einer serviceorientierten Architektur, sodass die Kommunikation plattformunabhängig beschrieben werden kann. Mithilfe der Codegenerierung soll aus dieser Beschreibung der Quellcode für die jeweiligen Plattformen erzeugt werden können. Der Entwickler soll für die Realisierung des Servers die bereits entwickelte Mobilis-Plattform verwenden, auf der diese Dienste betrieben werden. Damit die Flexibilität und Erweiterbarkeit der Dienste gewährleistet wird, sollen diese dynamisch zur Laufzeit integriert werden können.



# Inhaltsverzeichnis

<b>Aufgabenstellung</b>	I
<b>Abstract</b>	V
<b>Inhaltsverzeichnis</b>	VII
<b>1 Einführung</b>	1
1.1 Motivation . . . . .	1
1.2 Aufbau der Arbeit . . . . .	1
<b>2 Grundlagen</b>	3
2.1 Serviceorientierte Architekturen . . . . .	3
2.1.1 Merkmale . . . . .	3
2.1.2 Komponenten . . . . .	4
2.2 Dynamisches Deployment . . . . .	5
2.2.1 Prinzip . . . . .	6
2.2.2 Dynamisches Deployment in Java . . . . .	6
2.2.3 Die OSGi Plattform . . . . .	6
2.3 Das Extensible Messaging and Presence Protocol . . . . .	9
2.3.1 Architektur . . . . .	10
2.3.2 XMPP Extension Protocols . . . . .	11
2.4 Die Mobilis-Plattform . . . . .	15
2.5 Related Work . . . . .	17
2.5.1 Komponentenbasierte Entwicklung . . . . .	17
2.5.2 Serviceorientierte Entwicklung . . . . .	19
2.5.3 Serviceorientierte Komponentenmodelle . . . . .	21
2.5.4 Abgrenzung . . . . .	25
2.6 Zusammenfassung . . . . .	27
<b>3 Anforderungsanalyse</b>	29
3.1 Entwicklung in der Mobilis-Plattform . . . . .	29
3.1.1 Definition und Erstellung des Kommunikationsprotokolls . . . . .	29
3.1.2 Entwicklung eines Mobilis-Service . . . . .	30
3.1.3 Entwicklung einer Mobilis-Client-Anwendung . . . . .	31
3.1.4 Grenzen der Mobilis-Plattform . . . . .	32
3.2 Use Case Analyse . . . . .	32
3.2.1 Dienstnutzer . . . . .	33
3.2.2 Dienstanbieter . . . . .	33
3.2.3 Entwickler . . . . .	34
3.3 Anforderungen . . . . .	35
3.3.1 Funktionale Anforderungen . . . . .	35

3.3.2 Nichtfunktionale Anforderungen . . . . .	38
3.4 Abgrenzungskriterien . . . . .	39
3.5 Zusammenfassung . . . . .	40
<b>4 Konzeption</b>	<b>41</b>
4.1 Architekturkonzept . . . . .	42
4.1.1 Allgemeines Schema . . . . .	42
4.1.2 Ansatz 1: Externe Mobilis-Service-Komponenten . . . . .	43
4.1.3 Ansatz 2: Mobilis-Service Module . . . . .	46
4.1.4 Diskussion . . . . .	48
4.2 Dynamische Service-Plattform . . . . .	48
4.2.1 Architektur . . . . .	49
4.2.2 Mobilis-Service-Container . . . . .	51
4.2.3 Mobilis-Service-Lifecycle . . . . .	53
4.3 Beschreibung XMPP-basierter Dienste . . . . .	57
4.3.1 Elemente einer Dienstbeschreibung für Mobilis-Services . . . . .	57
4.3.2 Ein Vergleich zu WSDL . . . . .	59
4.3.3 Mobilis Service Description Language . . . . .	60
4.4 XMPP-Mapping . . . . .	65
4.5 Codegenerierung anhand der MSDL . . . . .	67
4.5.1 Besonderheiten bei der Codegenerierung von XMPP . . . . .	67
4.5.2 Codegenerierung mit XSLT . . . . .	68
4.5.3 Grenzen der Codegenerierung . . . . .	70
4.6 Zusammenfassung . . . . .	71
<b>5 Implementierung</b>	<b>73</b>
5.1 Übersicht . . . . .	73
5.2 Protokoll- und Dienstbeschreibung . . . . .	73
5.3 Code Generierung . . . . .	75
5.4 Deployment eines neuen Mobilis-Service . . . . .	79
5.4.1 Implementierung . . . . .	79
5.4.2 Packen . . . . .	80
5.4.3 Life cycle Management . . . . .	82
5.5 Weiterführende Aufgaben . . . . .	86
5.6 Zusammenfassung . . . . .	87
<b>6 Evaluation</b>	<b>89</b>
6.1 Testumgebung . . . . .	89
6.2 Erstellung der MSDL . . . . .	90
6.3 Implementierung des MobilisXHunt-Service . . . . .	90
6.4 Aktualisierung des MobilisXHunt Service . . . . .	92
6.5 Zusammenfassung . . . . .	93
<b>7 Zusammenfassung und Ausblick</b>	<b>95</b>
<b>A System Services Paket-Beispiele</b>	<b>97</b>

<b>B MSDL</b>	<b>103</b>
B.1 Transformation WSDL zu MSDL . . . . .	103
B.2 Schema . . . . .	103
B.3 MSDL TreasureHunt . . . . .	106
B.4 MSDL Mobilis XHunt . . . . .	108
<b>Abkürzungsverzeichnis</b>	<b>117</b>
<b>Abbildungsverzeichnis</b>	<b>119</b>
<b>Tabellenverzeichnis</b>	<b>121</b>
<b>Literaturverzeichnis</b>	<b>126</b>
<b>Selbstständigkeitserklärung</b>	<b>127</b>



# 1 Einführung

## 1.1 Motivation

Soziale Elemente, wie zum Beispiel das Finden von Freunden in der näheren Umgebung oder das Spielen von kollaborativen Spielen, erlangen in der heutigen Zeit einen immer höheren Stellenwert, vor allem im Bereich der mobilen Anwendungen. Diese sogenannten *Mobile Social Apps* unterliegen einer ständigen Veränderung und Erweiterung. Viele besitzen Schnittstellen zu sozialen Netzwerken, aber eher weniger zu anderen kollaborativen Diensten, wie zum Beispiel *Content-Sharing* oder Methoden für die Erstellung und Verwaltung von ortsbasierten Gruppen.

Die meisten dieser *Mobile Social Apps* bestehen aus einer Client-Server-Architektur, da die Endgeräte oft mit geringen Ressourcen, wie zum Beispiel einer geringen Rechenleistung oder einer begrenzten Speicherkapazität, auskommen müssen. Aus diesem Grund dient die mobile Applikation in erster Linie zur Repräsentation von Inhalten und zur Interaktion mit dem dahinter liegendem Server, der die eigentlichen sozialen und kollaborativen Elemente realisiert. Diese Architektur setzt eine zuverlässige und fehlertolerante Kommunikation zwischen Client und Server voraus. Aus diesem Grund stellt sich zu Beginn einer solchen Entwicklung zuerst die Frage, welches Kommunikationsprotokoll eingesetzt werden soll. Jeder Entwicklung liegt dabei ein gewisser Entwicklungsprozess zu Grunde. In einer *Serviceorientierte Architektur (SOA)* besteht der erste Schritt in der Erstellung des Kommunikationsprotokolls mithilfe einer Dienstbeschreibung. Im zweiten Schritt wird diese Dienstbeschreibung auf dem Client und Service umgesetzt und an die Anwendungslogik angebunden.

Das Ziel dieser Arbeit ist es, die bereits bestehende Mobilis-Plattform dahingehend weiterzuentwickeln, dass diese alle Komponenten einer *SOA* unterstützt. Damit wird die Erstellung des Kommunikationsprotokolls des Clients und Service für den Entwickler vereinfacht. Die Mobilis-Plattform bietet neben der Integration von sozialen Netzwerken bereits vorgefertigte kollaborative Dienste zur Verwendung an, die der Entwickler nutzen kann. Der Prozess für die Erstellung von *Mobile Social Apps* soll, mithilfe dieser Plattform, optimiert werden.

## 1.2 Aufbau der Arbeit

Als Einsteig dieser Arbeit soll in Kapitel 2 eine Einführung in die notwendigen Architekturen und Techniken gegeben werden. Neben diesen Grundlagen wird die Mobilis-Plattform vorgestellt. Diese soll als Basis für die Umsetzung dieser Arbeit genutzt werden. Darauf aufbauend werden in Kapitel 3 diejenigen Bereiche in der Mobilis-Plattform aufgezeigt, in denen diese von einer herkömmlichen Dienstplattform abweicht und eine gewisse Inflexibilität aufweist. Mit diesen Erkenntnissen und einer

## *1 Einführung*

Anwendungsfallanalyse sollen im Anschluss die Anforderungen an die Optimierung dieser Plattform definiert werden. Die Konzepte für die Architektur, die Umsetzung des dynamischen Deployments und die Beschreibungssprache werden in Kapitel 4 besprochen. Hier wird zudem ein Ansatz für die Codegenerierung mit vorgestellt und diskutiert. Die Umsetzungen, der zuvor erstellten Konzepte, sollen in Kapitel 5 genauer vorgestellt und in Kapitel 6 anhand eines ausgewählten Beispiel-Projekts evaluiert werden. Zum Ende der Arbeit wird in Kapitel 7 eine Zusammenfassung, sowie ein Ausblick auf die Weiterentwicklung der entwickelten Lösung gegeben.

## 2 Grundlagen

In diesem Kapitel soll ein Überblick über die Grundlagen gegeben werden, auf die diese Arbeit aufbaut. Dafür wird in Abschnitt 2.1 eine kurze Einführung in das Thema **SOA** in Bezug auf Merkmale und Komponenten gegeben. Die abstrakte Sichtweise einer dynamischen **SOA**-Plattform wird anschließend durch eine konkrete Sichtweise in Abschnitt 2.2 ergänzt. Hier wird auf das Prinzip und die Verwendung des dynamischen Deployment in Java eingegangen und eine Umsetzung in der OSGi-Plattform beispielhaft erläutert. Um ein Verständnis für das kommunikationsspezifische Verhalten der in dieser Arbeit weiterzuentwickelnden Dienstplattform Mobilis zu schaffen, wird in Abschnitt 2.3 das Kommunikationsprotokoll XMPP vorgestellt. Anschließend wird die Mobilis-Plattform in Abschnitt 2.4 im Detail besprochen. Zum Schluss werden in Abschnitt 2.5 verwandte Arbeiten betrachtet, die sich mit den Bereichen *Komponentenbasierte Entwicklung* und *Serviceorientierte Entwicklung* befassen. Die in diesen Arbeiten entwickelten Projekte sollen mit der Mobilis-Plattform verglichen werden, um daraus die Schwerpunkte einer dynamischen Dienstplattform abzuleiten.

### 2.1 Serviceorientierte Architekturen

**Serviceorientierte Architekturen** ermöglichen die Konstruktion von Client-Server Architekturen mit einer hohen Flexibilität und Verfügbarkeit. Die Verknüpfung von Anwendungen, die auf unterschiedlichen Plattformen und in verschiedenen Sprachen implementiert sind, können so mit dieser Architektur einfacher miteinander kommunizieren und interagieren.

Nach [31] ist **SOA**:

”... eine Systemarchitektur, die vielfältige, verschiedene und eventuell inkompatible Methoden oder Applikationen als wiederverwendbare und offen zugreifbare Dienste repräsentiert und dadurch eine plattform- und sprachunabhängige Nutzung und Wiederverwendung ermöglicht.“

Im Folgenden soll ein kurzer Überblick über **SOA**, anhand der Merkmale und Komponenten, gegeben werden.

#### 2.1.1 Merkmale

Eine **SOA** definiert lediglich eine abstrakte Architektur und keine konkrete Umsetzung. Nach [31] kann man diese Architektur anhand der folgenden Merkmale beschreiben:

**Lose Kopplung** Die Dienste sind lose miteinander gekoppelt, wenn Bindungen zwischen Diensten nicht in der Entwurfsphase fest implementiert werden. Dienste können somit während der Laufzeit andere Dienste finden, selber von anderen Diensten gefunden und letztendlich gebunden werden.

## 2 Grundlagen

**Dynamisches Binden** Während der Laufzeit sollen nicht nur beliebige Dienste eingebunden werden, sondern es soll gezielt nach bestimmten Diensten gesucht werden können. Dafür sollen komplexere Suchmethoden zur Verfügung stehen, sodass der bestmögliche Dienst gefunden und gebunden werden kann.

**Verzeichnisdienst** Das Finden der Dienste wird durch einen Verzeichnisdienst unterstützt. Dienste können sich bei einem Verzeichnisdienst anmelden und bestimmte Metadaten hinterlegen. Anhand dieser Metadaten können diejenigen Dienste gefunden werden, die die komplexen Suchkriterien am besten erfüllen.

**Verwendung von Standards** Durch Plattform- und Sprachunabhängigkeit der Dienste muss eine einheitliche Kommunikationsebene geschaffen werden, auf der sich Dienste unterschiedlicher Implementierungen miteinander verständigen können. Offene Standards fördern dabei die Verwendung globaler Schnittstellen zur Kommunikation.

**Einfachheit** Die Trennung eines Dienstes in eine Schnittstellenbeschreibung und eine konkrete Implementierung vereinfacht dabei den Umgang. Intern können somit Veränderungen am Dienst vorgenommen werden, während die Schnittstelle nach außen hin unverändert bleibt.

**Sicherheit** Sicherheit spielt gerade in Geschäftsbereichen eine sehr wichtige Rolle. Vertraulichkeit und Integrität zwischen dem Client und Service müssen hierbei sichergestellt werden, indem zum Beispiel ein sicherer Kommunikationskanal verwendet wird.

### 2.1.2 Komponenten

In einer [SOA](#)-Umgebung stellt ein Dienst eine Softwarekomponente dar, die Funktionalitäten für andere Dienste zur Verfügung stellt. Der Ablauf und die verwendeten Rollen sind in Abbildung 2.1 dargestellt und beinhalten folgende Komponenten:

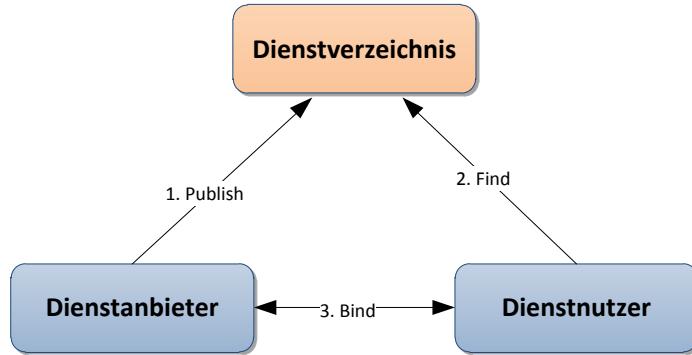
**Dienstanbieter** Der Dienstanbieter stellt die durch die Dienstbeschreibung definierten Funktionalitäten für andere Dienste zur Verfügung. Der Dienstanbieter ist für Authentifizierung und Authentisierung zuständig, sowie für [Quality of Service \(QoS\)](#). Ein Dienstanbieter kann für die Bearbeitung von Anfragen wiederum andere Dienste nutzen und Aufgaben delegieren.

**Dienstnutzer** Der Dienstnutzer kann über ein Verzeichnisdienst Dienstanbieter suchen und gegebenenfalls das Suchergebnis nach bestimmten Attributen filtern. Wurde ein passender Dienstanbieter gefunden, kann eine direkte Verbindung mit diesem hergestellt werden, indem ein auf beiden Seiten bekanntes Kommunikationsprotokoll, wie zum Beispiel SOAP, verwendet wird.

**Verzeichnisdienst** Ein Verzeichnisdienst ermöglicht das Auffinden von Dienstanbietern durch Dienstnutzer. Angebotene Dienste werden in Kategorien für eine effizientere Suchanfrage eingeteilt.

**Dienstbeschreibung** Eine Dienstbeschreibung ist eine öffentliche und vollständige Beschreibung eines Dienstes. Diese Beschreibung sollte unabhängig von der konkreten Implementierung des Dienstes und für andere Dienste lesbar sein. Neben

den rein funktionalen Beschreibungen ist es auch möglich, spezielle Attribute wie die maximale Antwortzeit oder Verfügbarkeit mit anzugeben. Eine sehr weit verbreitete Dienstbeschreibung ist die auf dem Webservice-Standard basierende [Web Service Description Language \(WSDL\)](#) [14].



**Abbildung 2.1:** Übersicht der SOA Komponenten

Wie zuvor erwähnt ist die Betrachtung der einzelnen Dienste losgelöst von der eigentlichen Implementierung und der Plattform auf der diese Dienste laufen. Abstrakt betrachtet werden Dienste dynamisch durch die De- und Registrierung an einem Dienstverzeichnis dynamisch der Dienstumgebung hinzugefügt. In einer konkreten Umsetzung kann es durchaus möglich sein, dass mehrere dieser Dienste auf ein und derselben Dienstplattform laufen. Aus diesem Grund müssen Dienste nicht nur auf der Dienstebene, sondern auch auf der lokalen Ebene dynamisch in die Plattform integriert werden können. Durch die Einführung einer komponentenbasierten Entwicklung wird jeder Dienst als eine oder mehrere Komponenten betrachtet und kann durch die Technik des dynamischen Deployments während der Laufzeit in die Dienstplattform integriert und auf der Dienstebene genutzt werden.

## 2.2 Dynamisches Deployment

Unter dem Begriff **Dynamisches Deployment** ist die Integration neuer Softwarekomponenten in ein bereits laufendes System zu verstehen. In einem herkömmlichen Entwicklungsprozess werden die Konzepte in Quellcode verfasst und der in einer Programmiersprache geschriebene Code in eine maschinenlesbare Form kompiliert. Erst an dieser Stelle kann das System gestartet und genutzt werden. Die gesamte Programmstruktur muss vor der Kompilierung bekannt sein und ist danach nicht mehr veränderbar. Software unterzieht sich jedoch einer stetigen Modifikation. Dies kann ein kleines Bug-fixing oder die Aktualisierung einer Anwendung mit einer komplett neuen Programmstruktur sein. Das Einspielen neuer Programmteile erfordert jedoch das Herunterfahren und wieder Hochfahren des gesamten Systems. Dieser Umstand soll mit dem Dynamischen Deployment überbrückt werden.

Im Folgenden soll kurz das Prinzip des Dynamischen Deployments erläutert und gezeigt werden, warum es in der Java-Umgebung mit geringem Aufwand zu realisieren

## 2 Grundlagen

ist. Anschließend soll ein detaillierter Einblick in die dynamische OSGi-Plattform gegeben werden.

### 2.2.1 Prinzip

Beim Dynamischen Deployment gibt es eine Komponente im laufenden System, die es ermöglicht weiteren Programmcode nachzuladen und auszuführen. Der Programmcode muss dabei häufig in einer kompilierten Form vorliegen. Im besten Fall ist dieser mit den vom Programm genutzten Ressourcen in einem Archiv zusammengepackt. Um bereits vorhandene Programmteile zu ersetzen ist es notwendig, dass das gesamte System in einer modularisierten Form vorliegt. Jedes Modul besitzt dabei eine Schnittstelle, in der definiert ist, wie ein Modul angesprochen werden kann.

Ein typisches Beispiel stellt der bereits erwähnte Webservice dar. In einer [SOA](#)-Umgebung können einzelne Dienste auf anderen Plattformen gestartet werden und durch einen Eintrag im Verzeichnisdienst dynamisch für andere Dienste zur Verfügung gestellt werden.

### 2.2.2 Dynamisches Deployment in Java

Java unterstützt das Prinzip der Reflection, genauso wie C# oder Python . *Reflection* ist eine Technik, die es ermöglicht, dass ein Programm seine eigene Struktur kennt und modifizieren kann. Das trifft auch für Programmteile zu, die während der Erstellung des Programms noch nicht bekannt waren und somit dem System erst später hinzugefügt wurden. In Abbildung 2.2 ist zu sehen, wie die Klasse `String` mittels Reflection ermittelt und dafür ein `Constructor` erstellt wird. Mit diesem `Constructor` kann eine neue Instanz der Klasse `String`, mit dem Inhalt "*Hello Refl String*", erstellt werden. Dieser wird danach auf der Konsole ausgegeben.

Das Austauschen von Klassen in Java während der Laufzeit wird als *Hot Deployment* bezeichnet. In Java kommen dafür die sogenannten `ClassLoader` zum Einsatz, die im Dateisystem auf eine neue Klassendatei warten und gegebenenfalls eine alte Klasse durch eine neue ersetzen. In Java können bereits implementierte Klassenlader wie der `URLClassLoader` verwendet werden, der Klassendateien von einer [Uniform Resource Locator \(URL\)](#) laden kann und diese in das Laufzeitsystem integriert. Es ist aber auch möglich eigene Klassenlader zu realisieren, die zum Beispiel komprimierte oder verschlüsselte Klassendateien laden können.

### 2.2.3 Die OSGi Plattform

**OSGi** ist eine auf Java basierende, dynamische Service Plattform, die es ermöglicht Softwaresysteme zu modularisieren, dynamisch neue Komponenten in das System zu integrieren und zu verwalten. Das OSGi Alliance Konsortium [1], welches 1999 gegründet wurde, ist für die Erstellung und Pflege der offenen Spezifikationen verantwortlich.

Nach [46] besteht die OSGi-Plattform aus beiden Bereichen *OSGi Framework* und *OSGi Standard Services*.

```

Constructor constructor;
try {
    clazz = Class.forName( "java.lang.String" );
    constructor = clazz.getConstructor( String.class );
    String refString = (String)constructor.newInstance( "Hello Refl String" )
    System.out.println( refString );
} catch ( Exception e ) {
    e.printStackTrace();
}

```

**Abbildung 2.2:** Beispiel Reflection in Java

### Das OSGi Framework

Das OSGi Framework bildet ein Rahmenwerk für die folgenden Anwendungskomponenten:

**Bundle** Ein Bundle stellt eine Komposition von Klassen und Ressourcen dar und ist eine Teilkomponente der Gesamtanwendung. Das Bundle kann in der Gesamtanwendung installiert, deinstalliert, gestartet und gestoppt werden, ohne dass die Anwendung selber in ihrem Lebenszyklus beeinträchtigt wird. Bundles können mithilfe von Sichtbarkeiten wiederum andere Bundles verwenden, wofür jedoch die explizite Deklaration in der Konfiguration der Bundles erforderlich ist.

**Service** Services können von Bundles im gesamten System verwendet werden. Um einen Service nutzen zu können, muss dieser jedoch erst in der Service Registry registriert werden, sodass er von anderen Bundles gefunden werden kann.

**Management Agent** Die Verwaltung der Bundles übernimmt der Management Agent. Dieser Agent kann selber wieder aus Bundles bestehen und ermöglicht das starten, stoppen, installieren und deinstallieren von Bundles.

Das OSGi Framework wird nach [4] in folgende logische Schichten unterteilt, die sich in die OSGi-Plattform einordnen (siehe auch Abbildung 2.3):

**Service** Die Schicht der Services bietet primär den Management Agents Unterstützung an, wie zum Beispiel der *Package Admin Service*, welcher Informationen über Abhängigkeiten von Bundles vorhält oder dem *Start Level Service*, der die Startreihenfolge von Bundles festlegt, sodass diese in der richtigen Reihenfolge zur Verfügung stehen.

**Life cycle** In der Schicht *Life cycle* werden die Zustände eines Bundles verwaltet, die es während der Laufzeit annehmen kann. Abbildung 2.4 zeigt, welche Zustände ein Bundle annehmen kann. Wie bereits erwähnt, sind diese Zustände über den Management Agent modifizierbar.

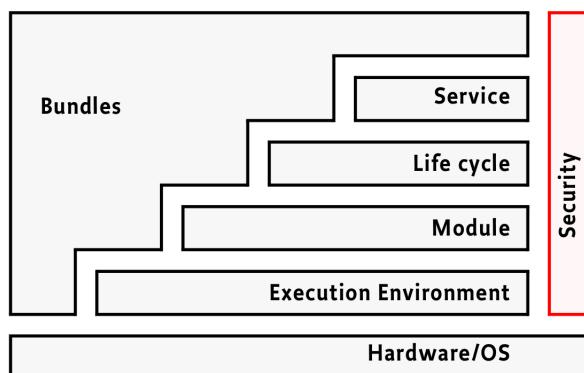
**Module** Die *Module*-Schicht ist für die Modularisierung der Anwendung zuständig, deren Module die Bundles bilden. Da ein Bundle ein Archiv ist, ist für die Konfiguration und Verwendung der Bundles eine Konfigurationsdatei vonnöten, die

## 2 Grundlagen

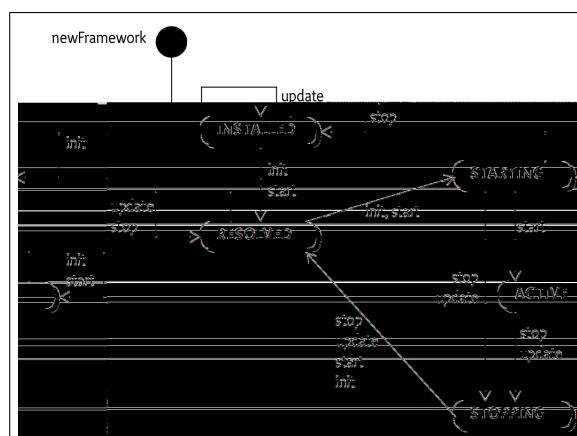
**MANIFEST.MF**. In dieser Datei werden zum Beispiel die Version und der Name des Bundles hinterlegt. Des Weiteren wird in dieser Datei festgelegt, welche Komponenten von anderen Bundles benötigt werden und welche es selber für andere zur Verfügung stellt.

**Execution Environment** Für die Ausführung eines Bundles auf unterschiedlichen Java-Versionen kümmert sich die *Execution Environment*. Diese unterschiedlichen Ausprägungen sind in den `java.*` Namespaces zu finden. Während Java SE 6 zu Java SE 5 rückwärts kompatibel ist, gilt dies nicht für die Umsetzung von Java SE 7 zu Java SE 5. Aus diesem Grund enthalten die Bundles keine Imports der Form `java.*` und sind somit nicht von der auszuführenden Java-Version abhängig, da dies von der Execution Environment übernommen wird.

**Security** In der Schicht Security können einzelne Rechte zur Ausführung von Bundles definiert werden. Dieses System basiert auf dem Java SE Security Model [3] und wurde um OSGi-spezifische Anforderungen an die Bundles erweitert .



**Abbildung 2.3:** OSGi Plattform Schichten (Quelle: [4])



**Abbildung 2.4:** Lebenszyklus eines OSGi Bundles (Quelle: [4])

### Die OSGi Standard Services

Die Standard Services stellen möglichst allgemein nutzbare Anwendungskomponenten dar, die in jeder Anwendungskomponente genutzt werden können, um bestimmte Problemstellungen zu lösen. In [5] werden diese Services in die Kategorien *Framework Services*, *System Services*, *Protocol Services* und *Miscellaneous Services* unterteilt. Die folgende Liste stellt einen kleinen Ausschnitt aus interessanten Services vor:

**Log Service** Um Protokollierungen aus der Anwendung heraus zu ermöglichen, stellt OSGi zwei *Log Services* bereit. Einer dient dem Schreiben und einer dem Lesen der Log-Nachrichten. Log-Nachrichten können zudem mit einem Log-Level, wie *Debug*, *Warnung* oder *Fehler*, versehen werden.

**Configuration Admin Service** Der *Configuration Admin Service* ermöglicht die Konfiguration einer Anwendungskomponente während der Laufzeit, ohne dass diese neu kompiliert werden muss.

**Metatype Service** Der *Metatype Service* stellt zusätzliche Informationen über eine Anwendungskomponente zu Verfügung, die während der Laufzeit abgefragt werden können. Dies ermöglicht Informationen flexibel für andere Anwendungskomponenten zur Verfügung zu stellen.

**Preference Service** Zur Verwaltung der system- und benutzerspezifischen Einstellungen wird der *Preference Service* verwendet.

**User Admin Service** Zur Reglementierung der Zugriffe kann der *User Admin Service* verwendet werden. Dieser definiert anhand von Benutzer- und Gruppeninformationen die Zugriffssteuerung auf andere Anwendungskomponenten.

**Http Service** Zur Verwendung externer Protokolle kann zum Beispiel der *Http Service* verwendet werden, der es ermöglicht, Bundles und deren Ressourcen über das Http-Protokoll anzusprechen.

Seit der OSGi Version 4.3 ist es möglich, die in den Core integrierten *Remote Services* zu verwenden, um OSGi Services in einer verteilten Umgebung zu nutzen. Die *Remote Services* bieten dabei Endpunkte unabhängig vom verwendeten Protokoll an, sodass Bundles über verschiedene virtuelle Java Maschinen miteinander kommunizieren können. In der Praxis wird dafür fast ausschließlich das *Hypertext Transfer Protocol (HTTP)* als Protokoll verwendet. Die Dienste können dadurch als Webservices angeboten und mit SOAP über *HTTP* angesprochen werden. Die Nutzung anderer Protokolle, wie dass in dieser Arbeit verwendete XMPP, ist möglich, aber praktisch bislang noch nicht umgesetzt worden.

## 2.3 Das Extensible Messaging and Presence Protocol

Das *Extensible Messaging and Presence Protocol (XMPP)* ist ein offenes Kommunikationsprotokoll, das primär für die Bereiche *Instant Messaging (IM)* entwickelt wurde und den Austausch von Presence-Informationen zwischen mehreren Teilnehmern ermöglicht [7]. Die grundlegenden Techniken für dieses Protokoll wurden 1999 noch

## 2 Grundlagen

unter der Jabber Community entwickelt. Seit 2002 ist die XMPP Foundation<sup>1</sup> für die Standardisierung und Weiterentwicklung des Protokolls zuständig. XMPP ist dabei nicht an eine konkrete Netzwerkarchitektur gebunden und kann durch sogenannte *XMPP Extension Protocol (XEP)* erweitert werden [8].

### 2.3.1 Architektur

Ein XMPP-Netzwerk besteht aus einer Client-Server-Architektur in der es möglich ist, dass verschiedene XMPP-Clients über einen XMPP-Server, sowie die XMPP-Server selbst miteinander kommunizieren können. Damit ein einzelner Teilnehmer im Netzwerk identifiziert werden kann, bekommt dieser eine feste *Jabber ID (JID)* zugewiesen. Die JID setzt sich aus den Teilen Benutzername(*user*) für die Identifizierung des Benutzerkontos, einer Domäne(*domain*) für die Identifizierung des genutzten XMPP-Servers und einer Ressource(*resource*) für die Verwendung eines Benutzerkontos auf verschiedenen Endgeräten zusammen (*user@domain/resource*).

Die Kommunikation zwischen Client und Server, sowie zwischen Server und Server ist asynchron und nutzt das *Transmission Control Protocol (TCP)* als Übertragungsprotokoll zum Austausch von Daten. Datenpakete werden als strukturierte *Extensible Markup Language (XML)* basierte Informationen übertragen. Ein Datenpaket wird in XMPP als XMPP-Stanza bezeichnet. Für die Übertragung dieser Pakete zwischen Client und Server werden XML-Streams als Container verwendet.

In XMPP existieren drei Arten von XMPP-Stanzas:

- message** Dient primär zum Austausch von reinen textbasierten Nachrichten und kann als eine Art *Push*-Funktion verwendet werden.
- presence** Stellt Presence-Informationen eines Clients zur Verfügung und unterstützt Subscription.
- iq** Bietet einen einfachen Request-Response-Mechanismus mit generischem Inhalt.

Jedes XMPP-Stanza nutzt für das Routing und die Identifizierung der Datenpakete festgelegte Attribute wie sie in Tabelle 2.1 definiert sind.

**Tabelle 2.1:** Übersicht der Attribute eines XMPP-Stanzas

Attribut	Funktion
to	Enthält die JID des Empfängers
from	Enthält die JID des Absenders
id	(optional) zur Identifikation des Pakets (z.B. für Request-Response-Zuordnung)
type	Ist Stanza-abhängig und beschreibt den Zweck des Pakets

---

<sup>1</sup> <http://xmpp.org/>

### 2.3.2 XM Extension protocols

Eines der Hauptziele von **XMPP** ist, neben der dezentralisierten Kommunikation, die Erweiterbarkeit. Auf der Basis der drei **XMPP**-Stanzas ist es möglich, bereits existierende oder auch neue Protokolle zu definieren und zu veröffentlichen. Für die Standardisierung dieser **XEPs** ist die **XMPP Standards Foundation (XSF)** verantwortlich. Diese kategorisiert dabei jedes **XEP** in eines der drei Entwicklungsstadien *experimental*, *draft* oder *final* [9].

Zur Zeit gibt es mehr als 300 veröffentlichte **XEPs**, von denen bereits einige in dem Referenzprojekt Mobilis Anwendung finden. Diese, sowie für das Konzept dieser Arbeit relevante und interessante **XEPs**, sollen hier kurz vorgestellt werden.

**Data Forms (XEP-0004)** Das XEP-0004 [17] hilft dabei, eine strukturierte Form der Daten, ähnlich den Forms aus XHTML 1.0 [36], zu ermöglichen. Die einzelnen Elemente werden dabei über Key-Value-Pairs verfügbar gemacht und bieten somit vielen anderen **XEPs** die Möglichkeit, die Daten generisch zu beschreiben.

**Service Discovery (XEP-0030)** Das XEP-0030 [22] ermöglicht es, Informationen über andere Entitäten in einer **XMPP**-Umgebung zu beschreiben. Zur Entität selber sollen Informationen wie die Identität und unterstützte Features ermittelt werden können, sowie mit der Entität verbundene und adressierbare Elemente. Diese Erweiterung stellt einen einfachen Verzeichnisdienst dar, wie man es von **SOA** kennt. In Abbildung 2.5 ist ein Beispiel einer Discovery-Anfrage dargestellt, die eine Antwort wie in Abbildung 2.6 zur Folge haben kann.

```
<iq to="DiscoService@openfire" from="client@openfire/java"
    id="packet1" type="get" >
    <query xmlns="http://jabber.org/protocol/disco#info" />
</iq>
```

Abbildung 2.5: XEP Beispiel einer Service Discovery Anfrage

```
<iq to="client@openfire/java" from="DiscoService@openfire"
    id="packet1" type="result" >
    <query xmlns="http://jabber.org/protocol/disco#info" >
        <identity category="client" type="pc" name="JavaClient" />
        <feature var="http://jabber.org/protocol/disco#info" />
        <feature var="http://jabber.org/protocol/disco#items" />
        <feature var="http://jabber.org/protocol/muc" />
        <feature var="jabber:iq:register" />
        <feature var="jabber:iq:search" />
        <feature var="jabber:iq:time" />
        <feature var="jabber:iq:version" />
    </query >
</iq>
```

Abbildung 2.6: XEP Beispiel einer Service Discovery Antwort

## 2 Grundlagen

**Multi-User Chat (XEP-0045)** Das XEP-0045 [41] bietet, im Gegensatz zu herkömmlichen Eins zu Eins Chats, ganze Chaträume an, in denen mehrere Teilnehmer miteinander kommunizieren können. Chaträume werden, wie jeder Teilnehmer im XMPP-Netzwerk, durch eine JID identifiziert, wobei die Ressource für den Nicknamen eines einzelnen Chat-Teilnehmers verwendet wird. Nachrichten in einem Multi-User Chat werden dabei wie ein Broadcast an alle Teilnehmer versendet. Ein Chatraum kann durch verschiedene Attribute konfiguriert werden. So kann zum Beispiel der Zugang durch ein Passwort beschränkt oder innerhalb eines Chatraumes können verschiedene Rechte durch Rollen vergeben werden.

**File Transfer (XEP-0052)** Mit dem XEP-0052 [33] ist es möglich, Dateien über das XMPP-Netzwerk anzubieten und zu übertragen. Dabei findet eine Übertragung immer zwischen zwei *JIDs* statt. Mit dieser Erweiterung werden zusätzliche Metainformationen für die zu übertragende Datei, wie zum Beispiel der Name, die Größe und der MIME-Type, bereitgestellt. Der Empfänger kann aufgrund dieser Metainformationen die Datei annehmen oder ablehnen. Eine Beispieldatenübertragung ist in Abbildung 2.7 zu sehen und in Abbildung 2.8 die Annahme der Datei durch den Empfänger.

```
<iq to="receiver@openfire/Service" from="sender@openfire/Client"
    id="packet1" type="set" >
  <si xmlns="http://jabber.org/protocol/si"
      id="jsi_6890111119277926802"
      profile="http://jabber.org/protocol/si/profile/file-transfer">
    <file xmlns="http://jabber.org/protocol/si/profile/file-transfer"
        name="DeploymentTestService.jar" size="19938"/>
    <feature xmlns="http://jabber.org/protocol/feature-neg">
      <x xmlns="jabber:x:data" type="form">
        <field var="stream-method" type="list-multi">
          <option>
            <value>http://jabber.org/protocol/bytestreams</value>
          </option>
          <option>
            <value>http://jabber.org/protocol/ibb</value>
          </option>
        </field>
      </x>
    </feature>
  </si>
</iq>
```

Abbildung 2.7: XEP Beispiel File Transfer anbieten

**Publish-Subscribe (XEP-0060)** Ein PubSub-Service, wie man es vor allem von News Feeds kennt, wird durch das XEP-0060 [32] bereitgestellt. Im Mittelpunkt liegt dabei ein Knoten in einem Teilbaum, den ein Teilnehmer in der XMPP-Umgebung abonnieren kann. Wird eine Information über diesen Knoten publiziert, so werden alle abonnierten Teilnehmer darüber informiert. Das Verhalten ist eventbasiert und gleicht einem Broadcast an alle Teilnehmer in einem Subnetz. Die publizierte Information kann dabei Daten enthalten oder auch leer sein.

```
<iq to="sender@openfire/Client" from="receiver@openfire/Service"
    id="packet1" type="result" >
  <si xmlns="http://jabber.org/protocol/si">
    <feature xmlns="http://jabber.org/protocol/feature-neg">
      <x xmlns="jabber:x:data" type="submit">
        <field var="stream-method">
          <value>http://jabber.org/protocol/bytestreams</value>
          <value>http://jabber.org/protocol/ibb</value>
        </field>
      </x>
    </feature>
  </si>
</iq>
```

Abbildung 2.8: XEP Beispiel File Tranfer akzeptieren

**SOAP Over XMPP (XEP-0072)** Das XEP-0072 [20] ermöglicht die Einbettung von *XMPP* in SOAP. Damit soll es ermöglicht werden, *XMPP*-basierte Dienste mit einer *WSDL* zu beschreiben, sodass anstatt dem synchronen *HTTP*, das asynchrone *XMPP* für die Kommunikation verwendet werden kann. Für die Umsetzung der Pakete können das IQ- und das Message-Stanza verwendet werden. Jedes Stanza muss zudem einen SOAP-Envelope im First-Level-Child definieren und einen SOAP-Header, sowie SOAP-Body anbieten. Eine Möglichkeit der Codegenerierung anhand der *WSDL*, sowie die Beschreibung von Abhängigkeiten von Diensten, wird nicht mit vorgestellt. Eine beispielhafte Umsetzungen ist in den Abbildungen 2.9 und 2.10 dargestellt. Aus Platzgründen wurden in diesen Abbildungen die SOAP-Header-Elemente weggelassen.

```
<iq to="service@openfire/soapServer"
    from="client@openfire/soapClient"
    id="packet1" type="get" >
  <env:Envelope xmlns:env="path{}" >
    <env:Header>
      <!-- Typical SOAP Header -->
    </env:Header>
    <env:Body>
      <p:carrequest xmlns:p="path{}" >
        <p:carproperties >
          <p:make >audi</p:make >
          <p:model >a3</p:model >
          <p:maxprice >25.000</p:maxprice >
        </p:carproperties >
      </p:carrequest>
    </env:Body>
  </env:Envelope>
</iq>
```

Abbildung 2.9: XEP Beispiel SOAP Anfrage

**Bidirectional-streams Over Synchronous HTTP (XEP-0124)** *XMPP* basiert auf langlebigen *TCP*-Verbindungen, was in Szenarien in denen keine stabile Verbindung garantiert werden kann, zu Problemen führt. Mit dem XEP-0124 [35] wird eine Möglichkeit geboten, langlebige *TCP*-Verbindungen zu emulieren, indem Request-Response-Abfolgen über *HTTP* ermöglicht werden. Um dies zu realisieren wird ein spezieller Proxy benötigt, der zwischen *HTTP* Request-Response und den von *XMPP* genutz-

```

<iq to="client@openfire/soapClient"
    from="service@openfire/soapServer"
    id="packet1" type="result" >
  <env:Envelope xmlns:env="path{}" >
    <env:Header>
      <!-- Typical SOAP Header -->
    </env:Header>
    <env:Body>
      <p:caroffer xmlns:p="path{}" >
        <p:cardealership >
          <p:name >audistar</p:name >
          <p:city >springfield</p:city >
          <p:price >15.000</p:price >
          <p:status >used</p:status >
        </p:cardealership >
        <p:cardealership >
          <p:name >rudis</p:name >
          <p:city >shelbyville</p:city >
          <p:price >22.586</p:price >
          <p:status >new</p:status >
        </p:cardealership >
      </p:caroffer>
    </env:Body>
  </env:Envelope>
</iq>
```

**Abbildung 2.10:** XEP Beispiel SOAP Antwort

ten [XML-Streams](#) vermittelt. Dieser Mechanismus ähnelt dem *Long Polling* in webbasierten Szenarien. Neben emulierten langlebigen Verbindungen bietet [Bidirectional-streams Over Synchronous HTTP \(BOSH\)](#) einen Weg, Webapplikationen an ein [XMPP](#)-Netzwerk anzubinden.

**Service Discovery Extensions (XEP-0128)** Das XEP-0128 [39] ist eine Erweiterung zum bereits beschriebenen [Service Discovery \(XEP-0030\)](#), da diese bereits im Stadium *final* ist und somit nicht mehr verändert werden kann. Das XEP-0128 ermöglicht es, die Informationen einer Entität, die durch das XEP-0030 angeboten werden, um beliebige Informationen zu erweitern. Diese generische Erweiterung wird mithilfe des [Data Forms \(XEP-0004\)](#) angeboten und anhand von Namespaces eingeschränkt.

**Service Administration (XEP-0133)** Zur Verwaltung von [XMPP](#)-basierten Komponenten kann das XEP-0133 [40] verwendet werden. Dazu zählt zum Beispiel das Hinzufügen und Löschen von Benutzern, das Festlegen und Ändern von Passwörtern und das Neustarten oder Herunterfahren von Diensten.

**Service Delegation (XEP-0291)** In XEP-0291 [25] wird eine Möglichkeit beschrieben, Anfragen an einen Dienst weiterzuleiten. Davon ausgehend, dass ein einziger [XMPP](#)-Server und somit auch [XMPP](#)-Client, nicht alle notwendigen Features selber anbieten kann, sollen Anfragen an Andere delegiert werden. Als Beispielszenario wird in dabei ein Cloud-basiertes System genannt, welches Features lediglich über einen [XMPP](#)-Account anbietet, aber jede Anfrage an einen anderen Account weiterleitet.

In der folgenden Tabelle 2.2 wurden alle [XEPs](#) zusammengefasst und gekennzeichnet, welche bereits in die Mobilis-Plattform integriert sind, beziehungsweise für das

Konzept dieser Arbeit interessant sind. Die bereits in die Mobilis-Plattform integrierten XEPs werden im folgenden Abschnitt 2.4 näher erläutert.

**Tabelle 2.2:** Übersicht der XEPs in der Mobilis-Plattform

XEP	In Mobilis integriert	Verwendung Konzept
XEP-0004	✓	–
XEP-0030	✓	✓
XEP-0045	✓	–
XEP-0052	✓	✓
XEP-0060	✓	–
XEP-0072	–	✓
XEP-0124	–	✓
XEP-0128	–	✓
XEP-0133	–	–
XEP-0291	–	✓

## 2.4 Die Mobilis-Plattform

Die **Mobilis-Plattform** ist ein Projekt der TU Dresden, das eine kollaborative Dienstplattform realisiert, die zur Kommunikation der Teilnehmer eine XMPP-Umgebung verwendet. Das Ziel des Mobilis-Projekts ist es, den Entwicklungsprozess von kollaborativen und sozialen Anwendungen zu optimieren. Dabei sollen dem Entwickler bereits vorgefertigte Dienste zur Verfügung gestellt werden, welche die Anbindung an soziale Dienste vereinfacht. Zu den wiederverwendbaren Diensten zählen zum Beispiel ein Gruppenmanagementsystem zur Verwaltung von ortsbasierten Gruppen, ein System zum gemeinsamen Bearbeiten von Dokumenten und die Anbindung an soziale Netzwerke wie Facebook und Foursquare. Neben diesen wiederverwendbaren Diensten werden zudem komplexere Dienste, wie zum Beispiel *MobilisBuddy*<sup>2</sup>, einer mobilen Anwendung zum Auffinden von Freunden in der näheren Umgebung, auf Basis von Näherungssensoren und Benachrichtigungen, angeboten. Die ortsbasierten Informationen von mehreren Teilnehmern spielen auch in dem *Location-based Game (LGB) Mobilis XHunt* [27] eine wichtige Rolle. Das, auf dem Brettspiel Scotland Yard<sup>3</sup> basierende, Hide and Seek Spiel wird dabei live von mehreren Spielern in der realen Welt, mithilfe von mobilen Endgeräten, gespielt.

Die Architektur der Mobilis-Plattform wurde in [30] aufgearbeitet und optimiert und ist in Abbildung 2.11 dargestellt. Die Basis der Plattform bilden die **Mobilis Services**, die den Mobilis-Clients unter anderem Funktionalitäten, wie zum Beispiel das Bilden von Gruppen, kollaboratives Editieren, Spielen von *LGB*, anbieten. Das Registrieren und Auffinden dieser Dienste wird durch ein Service-Discovery unterstützt. Der Mobilis-Server ermöglicht es zudem von einem Dienst mehrere Instanzen zu starten, die unabhängig voneinander laufen, aber dieselben Funktionen anbieten. Zur Unterscheidung dieser Instanzen dient hier wiederum das Service-Discovery nach der

<sup>2</sup> <http://mobilisplatform.sourceforge.net/mobilisbuddy.html>

<sup>3</sup> [http://de.wikipedia.org/wiki/Scotland\\_Yard\\_%28Spiel%29](http://de.wikipedia.org/wiki/Scotland_Yard_%28Spiel%29)

## 2 Grundlagen

Vorgabe der Erweiterung **Service Discovery (XEP-0030)**. Weitere **XEPs** die in der Mobilis-Plattform Anwendung finden sind der **Multi-User Chat (XEP-0045)** zur Realisierung von Sitzungen und normalen Chataktivitäten, **File Transfer (XEP-0052)** zur Übertragung von Spielementen in *Mobilis XHunt*, **Publish-Subscribe (XEP-0060)** zur Nutzung der User Context Informationen und **Bidirectional-streams Over Synchronous HTTP (XEP-0124)** zur Realisierung eines Web-Clients, wie zum Beispiel dem *XHunt Spectator* [24], einer Webanwendung, um als Beobachter an einem *MobilisXHunt*-Spiel teilzunehmen.

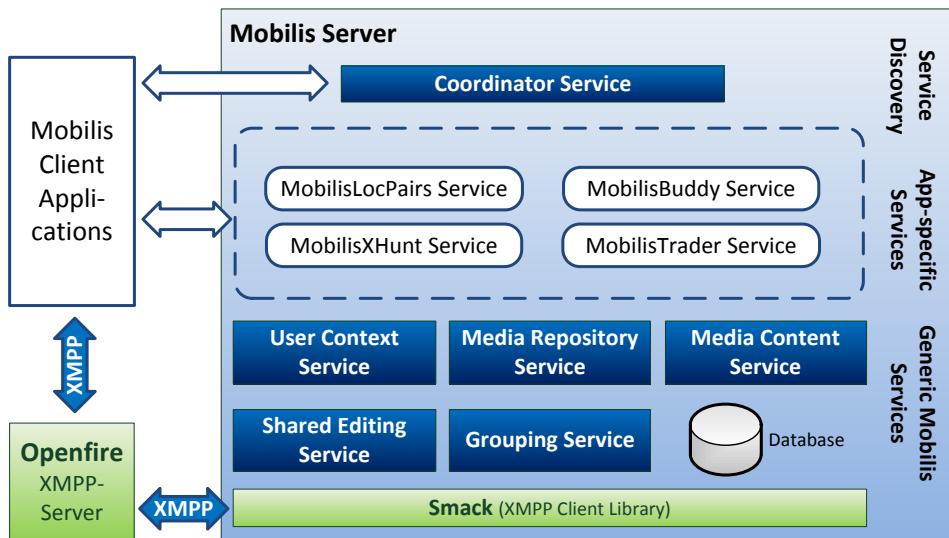


Abbildung 2.11: Übersicht Mobilis Service Plattform

In der Mobilis-Plattform werden die Mobilis-Services in *Generic Service* und *App-specific Service* kategorisiert. Ein **Generic Service** ist ein Dienst, der von verschiedenen Diensten genutzt werden kann und meist einen bestimmten Aspekt kapselt, wie zum Beispiel die ortsbasierte Gruppenbildung oder die Verwaltung und persistente Speicherung von Kontextinformationen der Benutzer. Ein **App-specific Service** bietet hingegen sehr spezielle Funktionen an, die meist nicht von anderen Diensten genutzt werden können. Zum Beispiel stellt ein *Mobilis XHunt* Service ein aktuell laufendes Spiel dar und kann somit nur schwer von anderen Diensten wiederverwendet werden.

Jeder Dienst ist in der Lage, sich mit einer eigenen **JID** anzumelden. Dafür können Einstellungen zum Agent, sowie zum Dienst selber getätigt werden. Diese werden dann beim Start des Mobilis-Server ausgewertet. Diese Service-Instanzen eines Dienstes verwenden alle den selben **XMPP**-Account und unterscheiden sich nur hinsichtlich der Ressource voneinander.

Die Kommunikation zwischen Client und Service erfolgt ausschließlich über das IQ-Stanza. Um die selbst definierten IQs in einer einheitlichen Form von Client und Service nutzen zu können, werden diese als sogenannte **Mobilis-Beans** implementiert. Diese Abstraktion des IQ-Stanzas, optimiert die Verwendung der eigentlichen Nutzdaten aus dem IQ und bildet diese auf Business-Objekte ab. Die Mobilis-Beans werden

vom Entwickler global verfügbar gemacht und können somit auf Client- und Serviceseite genutzt werden. Ein Mobilis-Bean bietet zudem die Möglichkeit, die Business-Objekte in ein IQ-Stanza zu De-/ Serialisieren.

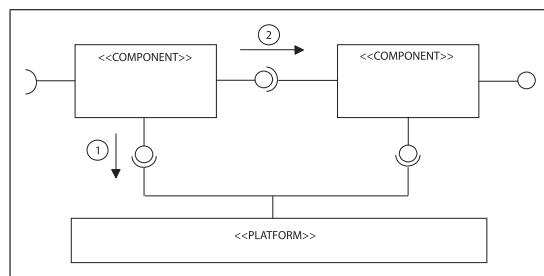
## 2.5 Related Work

Dieser Abschnitt dient dem Überblick über vorhandene Forschungsarbeiten, die thematisch mit dieser Arbeit verwandt sind. Für diese Betrachtungen wird als erstes die komponentenbasierte Entwicklung 2.5.1 erläutert, die den Grundstein für das dynamische Deployment und einer serviceorientierten Architektur bildet. Darauf aufbauend wird die daraus entstandene serviceorientierte Entwicklung in Abschnitt 2.5.2 noch einmal aufgegriffen und aus Sicht der Forschung auf die einzelnen Elemente und deren Verwendung eingegangen. Zum Schluss wird in 2.5.3 die Verbindung dieser beiden Entwicklungsmethoden erläutert und anhand von Referenzimplementierungen, wie zum Beispiel dem *Gravity* Projekt [13] und dem iPOJO Framework [18], deren Umsetzung gezeigt.

### 2.5.1 Komponentenbasierte Entwicklung

Die Idee, Software in einzelne Module beziehungsweise in Komponenten zu zerlegen, ist sehr alt. Schon 1972 beschrieb Parnas in [34] die Modularisierung von Software als einen Mechanismus zur Verbesserung der Flexibilität und Verständlichkeit eines Systems. Durch die Wiederverwendung und Austauschbarkeit einzelner Module konnte die benötigte Entwicklungszeit und Wartung eines Systems um ein Vielfaches verbessert werden.

Die ausschließliche Verwendung von Softwarekomponenten ist heute unter dem Begriff *Component-based Software Engineering (CBSE)* zusammengefasst. Nach [15] definiert eine Komponente Eigenschaften, durch die sie beschrieben wird und wie es ermöglicht wird, diese Komponente mit anderen Komponenten zu kombinieren. Ein komponentenbasiertes System besteht dabei, wie in Abbildung 2.12 gezeigt, aus einzelnen Komponenten, einer Plattform auf der diese Komponenten laufen und einem Mechanismus zum Binden, sodass Komponenten miteinander interagieren können.



**Abbildung 2.12:** Ein komponentenbasiertes System (Quelle: [15])

In [15] wird ein Framework vorgestellt, das verschiedene Komponentenmodelle klas-

## 2 Grundlagen

sifiziert und miteinander vergleicht. Um Klassifizierungen und Vergleiche zu erstellen, müssen jedoch folgende grundlegende Merkmale definiert werden:

**Life cycle** In einem *Component-based System (CBS)* existieren einmal ein Life cycle für das gesamte *CBS* und einer für jede einzelne Komponente. Der Life cycle einer Komponente ist etwas komplexer und wird in verschiedene Stadien unterteilt, wie in Abbildung 2.13 dargestellt ist.

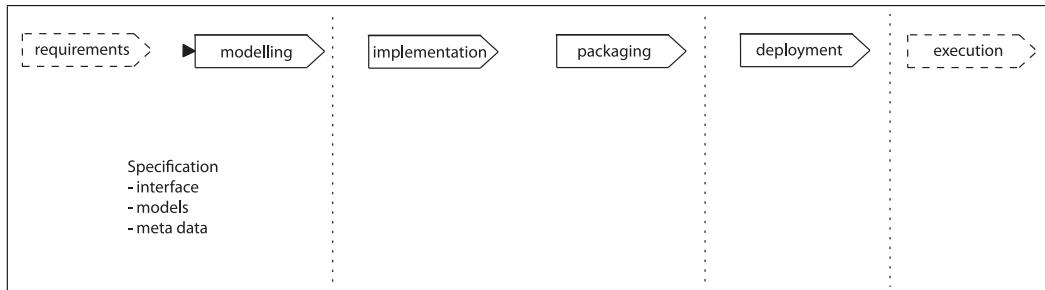
**Construction** Um zu beschreiben, wie ein System zusammengebaut wird und welche Funktionen in welcher Weise genutzt werden können, wird folgendes benötigt:

**Interface** Gibt an, welche Funktionen angeboten werden

**Binding** Beschreibt, wie Komponenten miteinander verbunden werden, zum Beispiel horizontal oder vertikal

**Interaction** Definiert, wie die verschiedenen Komponenten miteinander kommunizieren

**Extra-functional Properties** Für die Erweiterung der Komponenten um Funktionen im Management, in der Spezifikation oder in der Komposition, werden Extra-functional Properties definiert. Dadurch können unter anderem Parameter für *QoS*, Zuverlässigkeit und Antwortzeit definiert und überwacht werden.



**Abbildung 2.13:** Stadien eines Lifecycles einer Komponente (Quelle: [15])

Die Implementierung von Komponenten, deren Integration in eine Plattform und Interaktion mit anderen Komponenten, wird meist zur Entwurfszeit festgelegt. Ein System, das zur Laufzeit um Komponenten erweitert werden soll, muss zusätzliche Funktionen mitbringen, damit auch externe Komponenten während der Laufzeit in das System integriert werden können. Eine in der Java-Welt weit verbreitete Methode ist die Verwendung einer OSGi-Laufzeitumgebung, welche bereits Funktionen für dieses Szenario mitbringt. In [19] wird gezeigt, welche Schritte notwendig sind, um das OSGi-Konzept nach .NET zu portieren. Hierfür ist vor allem der abstrakte Ansatz von Interesse, wie es ermöglicht wird, eine dynamische Service Plattform aufzubauen. Das Ziel von OSGi ist es, die Ausführung komplexer Softwaresysteme in einer dynamisch erweiterbaren Umgebung zu verwalten.

Richard S. Hall [21] gibt einen Einblick in eine beispielhafte Implementierung von OSGi. Dabei werden Nachteile dieser Plattform, wie zum Beispiel eine sehr flache Registry für Dienste oder in bestimmten Situationen auftretende Probleme durch gleichzeitiges Laden von Klassen, aufgezeigt. Einige der Probleme sind in der aktuellen

Version von OSGi behoben wurden, doch gerade das mehrfache Laden von Klassen kann unter Umständen immer noch zu Deadlocks führen, da OSGi, anders als im reinen Java, Klassen nur hierarchisch lädt.

### 2.5.2 Serviceorientierte Entwicklung

Ein großer Nachteil von [CBS](#) ist die Beschränkung der Komponenten auf eine Plattform, wie zum Beispiel JavaBeans oder [Component Object Model \(COM\)](#). Die Interaktion von zwei unterschiedlichen Plattformen ist nur mit sehr viel Aufwand möglich. Eine Lösung dieser Problemstellung bietet die serviceorientierte Entwicklung, welche eine Interaktion unabhängig von der Plattform durch verschiedene Standards ermöglicht.

Michael Stal [44] beschreibt im Speziellen die Webservices als eine vielversprechende Technologie, mit der es möglich ist, solche heterogenen Systeme in homogene, komponentenbasierte Lösungen zu überführen. Webservices sind eine weit verbreitete Technik, um Interoperabilität, meist über das Internet, verschiedener Implementierungen zu gewährleisten. Für die Umsetzung einer [SOA](#) sind demnach folgende Schritte notwendig:

**Implementierung des Protokolls** Ein Protokoll sollte in erster Linie selbst beschreibend sein. Das heißt, es sollten keine notwendigen Informationen extern ausgelagert sein, um das Protokoll nutzen zu können. Des Weiteren muss ein Protokoll die verwendeten Nachrichten zur Interaktion syntaktisch und semantisch beschreiben, sowie deren Abfolge bei einem Aufruf.

**Implementierung der Schnittstellendefinition** Die Schnittstelle, welche die Funktionen eines Service nach außen definiert, sollte Informationen bezüglich der Struktur und des Deployments in einer Weise definieren, die von einer Plattform unabhängig ist. In [WSDL](#) werden dafür die verwendeten Datentypen und Operationen, unabhängig vom verwendeten Übertragungsprotokoll, beschrieben. Erst die Bindungen und Endpoints definieren die Verwendung für ein bestimmtes Protokoll und deren Adressierung.

**Implementierung eines Dienstverzeichnis** Damit Dienstanbieter die Beschreibung ihrer Schnittstelle der Öffentlichkeit zugänglich machen und Konsumenten diese nutzen können, um ein konkreten Dienst zu binden, wird ein Verzeichnis benötigt.

In [16] wird an einem Beispiel gezeigt, wie ein komponentenbasiertes System zu einem serviceorientierten System erweitert wird, um dessen Flexibilität und dynamische Erweiterbarkeit zu verbessern. Als Laufzeitumgebung wird dafür die OSGi-Plattform verwendet. Dafür werden die JavaBeans in OSGi-spezifische Bundles überführt und die native Unterstützung von Diensten genutzt.

Eine weitere [SOA](#) zeigt das eFlow Projekt [12]. Das eFlow Projekt legt einen abstrakten Prozessablauf über eine Komposition von mehreren Diensten. Das Ziel ist es dabei Dienste adaptiv zu gestalten, sodass diese sich selber an Veränderungen in der Dienstumgebung anpassen, mit geringem oder sogar ohne manuellem Eingriff. Um dies zu bewerkstelligen, werden eine Vielzahl von Konsistenz- und Verhaltensregeln

## 2 Grundlagen

an die Dienste gestellt und überwacht.

Neben den Vorteilen, die eine dynamische Dienstumgebung mit sich bringt, entstehen auch neue Anforderungen an jene Umgebung. In [43] wird darauf eingegangen, wie diese Dienstumgebungen modelliert werden können. Für die Modellierung wird der Ansatz des *Model-Driven-Architecture (MDA)* verfolgt. *MDA* basiert auf den drei Standards: *Unified Modeling Language (UML)*, MetaObject Facility und Common Warehouse Metamodel. Diese verfolgen das Ziel, die eigentliche Spezifikation von der Implementierung zu trennen und durch Transformation in ausführbaren Code zu überführen. Dafür existieren drei Transformationsstufen:

**Computation Independent Model (CIM)** Auch Domain Model genannt, welches die Umgebung und die Anforderungen eines Systems modelliert, ohne auf strukturelle oder prozessspezifische Details einzugehen.

**Platform Independent Model (PIM)** Beschreibt das System in einer plattformunabhängigen Form, dass sich nur auf die Funktionen des Systems fokussiert.

**Platform Specific Model (PSM)** Beschreibt die Kombination der *PIM* Beschreibung, unter Beachtung der darunter laufenden Plattform auf der das System laufen wird.

Der eigentliche Prozess ist in Abbildung 2.14 abgebildet. Der Ansatz der *MDA* findet gerade in der *SOA* einen weit verbreiteten Einsatz, um heterogene Systeme zu verbinden. Die Beschreibungssprache eines Dienstes obliegt dabei dem *CIM* und *PIM*. Durch Codegenerierung kann dann ein plattformspezifischer Code für verschiedene Plattformen erzeugt werden.



**Abbildung 2.14:** Verlauf einer Komposition nach MDA (Quelle: [43])

Eine letzte Betrachtung in diesem Abschnitt befasst sich mit der Fehlersemantik in serviceorientierten Umgebungen. In [23] spielen vor allem orchestrierte Dienste eine wichtige Rolle, da ein Fehler in der Kette von Aufrufen das ganze System in einen nicht definierten Zustand bringen kann. Als Lösung und allgemeingültiges Konzept für die Fehlerbehandlung in serviceorientierten Umgebungen, wird das Crash-Only Prinzip eingeführt. Dieses Prinzip besagt, dass jeder Dienst zu jeder Zeit damit rechnen muss, dass ein Fehler auftreten kann, auf den er reagieren muss. Wird ein Fehler festgestellt, soll der Dienst sich beenden und gegebenenfalls wieder neu starten. Diese Methode soll vor allem das Fehlermodell und das Testen vereinfachen. Damit das

Crash-Only Prinzip auch in Webservices angewendet werden kann, werden folgende Eigenschaften definiert und von einem Crash-Manager überwacht:

- Begrenzte externe Bindungen
- Alle Interaktionen zwischen Diensten haben einen Time-out
- Ressourcen sollen nur an Dienste und nicht an das System gebunden werden
- Anfragen an einen Dienst sollen vollkommen selbst beschreibend sein

### 2.5.3 Serviceorientierte Komponentenmodelle

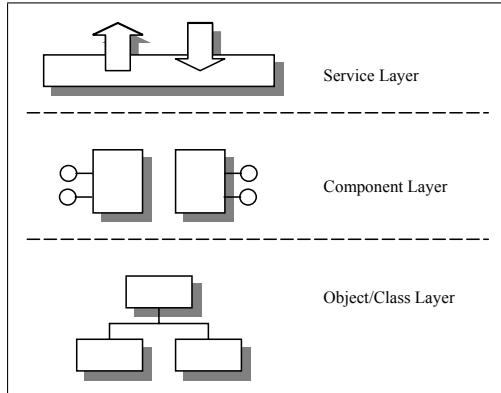
Komponentenbasierte und serviceorientierte Architekturen haben nach Petritsch [37] die selben Ziele, wie zum Beispiel eine hohe interoperable Architektur, lose Kopplung und fehlerfreie Entwicklung. Die serviceorientierte Entwicklung stellt nach Petritsch eine Erweiterung, beziehungsweise eine Verbesserung der komponentenbasierten Entwicklung dar. Der größte Unterschied liegt in der Art und Weise, wie Dienste miteinander verbunden sind und wie diese externen Dienstnutzern angeboten werden. Dabei muss aber auch Petritsch eingestehen, dass sich eine exakte Trennung der beiden Architekturen als schwierig erweist. Schnittstellen und Verzeichnisdienst werden zudem auch in komponentenbasierten Architekturen eingesetzt.

In [11] beschreiben die Autoren, wie mit Hilfe einer **SOA** und der komponentenbasierten Entwicklung, Webservice-Anwendungen erstellt werden können. Dabei liegt der Fokus auf verteilte Enterprise Lösungen. Anhand eines solchen Szenarios wird ein tiefer Einblick in die komponentenbasierte Entwicklung, und wie diese in eine **SOA** eingebettet werden kann, gegeben. Abbildung 2.15 stellt die einzelnen Schichten in einem serviceorientierten Komponentenmodell dar. Dabei spielt in der Service- und in der Komponentenschicht die schnittstellenorientierte Entwicklung eine wichtige Rolle. Die Dienstschnittstellen stellen Funktionen nach außen hin, für externe Nutzer bereit, wohingegen die darunter liegenden Komponentenschnittstellen Funktionen auf lokaler Ebene anbieten und nutzen. Dies wird auch als externe Sicht und interne Komposition beschrieben.

Einen detaillierteren Einblick in die Entwicklung und Architektur geben die Autoren in [13] und [38]. Im *Gravity* Projekt, welches in [13] vorgestellt wird, kommunizieren zum Beispiel alle Komponenten ausschließlich über Dienste. Das ermöglicht, dass Dienste auf ein und derselben Plattform laufen, aber auch ohne weitere Anpassungen ausgelagert werden können. Um Dienste und Komponenten zu kombinieren, wurden folgende Richtlinien definiert:

1. Ein Dienst muss Funktionalität in Form von Operationen anbieten und wieder verwendbar sein.
2. Ein Dienst muss durch einen Vertrag charakterisiert werden.
3. Komponenten implementieren den Dienstvertrag und folgen deren Beschränkungen, wobei weitere implementierungsspezifische Abhängigkeiten definiert werden können.

## 2 Grundlagen



**Abbildung 2.15:** Schichten eines serviceorientierten Komponentenmodells (Quelle: [11])

4. Dienstabhängigkeiten müssen auch während der Laufzeit aufgelöst werden.
5. Kompositionen müssen im Dienstvertrag beschrieben werden.
6. Komponenten dürfen nur durch alternative Komponenten ersetzt werden, wenn sie die gleichen Dienstverträge implementieren.

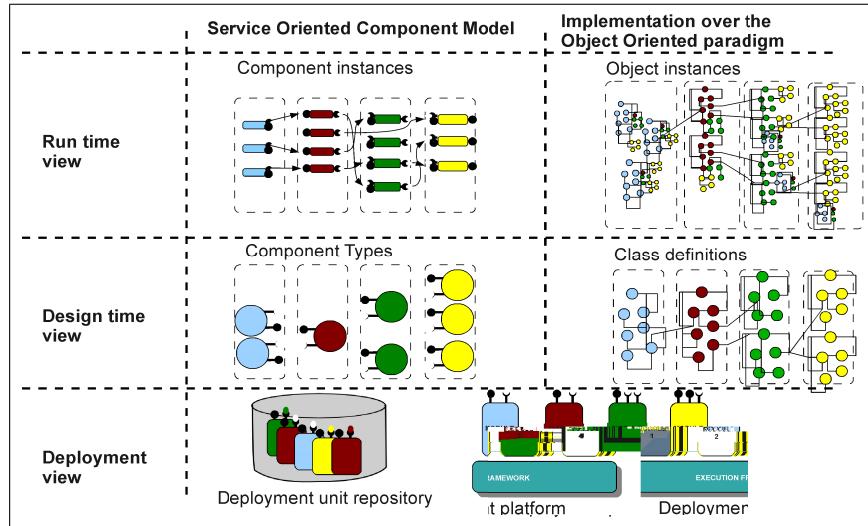
Im Gravity Projekt steht vor allem die Adaption dynamisch verfügbarer Dienste zur Laufzeit im Fokus. Dafür werden die Szenarios *Ankunft einer neuen Komponente*, *einer neuen Instanz einer Komponenten* oder *eines Dienstes*, sowie Entfernung der gleichen betrachtet. Der *Instance-level Manager* ist zum Beispiel für die Überwachung und Verwaltung der einzelnen Instanzen zuständig. Zur Verwaltung dieser einzelnen Instanzen wird eine Beschreibung für die Komponente benötigt, wie sie in Abbildung 2.16 zu sehen ist.

```
<component class="org.gravity.webbrowser.BrowserImpl"
           factory="yes">
    <provides
        service="org.gravity.services.Application"/>
    <provides service="org.gravity.services.WebBrowser"/>
    <property name="version" value="1.0" type="string"/>
    <requires
        service="org.gravity.services.BrowserPlugin"
        filter=""
        cardinality="0..n"
        policy="dynamic"
        bind-method="addPlugin"
        unbind-method="removePlugin"
    />
    <requires
        service="org.gravity.services.WindowManager"
        filter=""
        cardinality="1..1"
        policy="dynamic"
        bind-method="set WindowManager"
        unbind-method="unset WindowManager"
    />
</component>
```

**Abbildung 2.16:** Beschreibung einer Komponente im Gravity Projekt(Quelle: [13])

Einen Schritt weiter geht das in [38] entwickelte Framework. In einem serviceorientierten Komponentenmodell sind zum Beispiel *dynamische Rekonfigurierung* und *Auflösen von fehlenden Abhängigkeiten* von Diensten essentielle Aspekte, die verwaltet

werden müssen. In Abbildung 2.17 ist zu sehen, dass allein die Nutzung von Komponenten einen wesentlichen Vorteil gegenüber der Nutzung einer reinen objektorientierten Architektur bringt. In dieser Abbildung ist des Weiteren zu sehen, in welche drei Abstraktionsebenen eine Komponente unterteilt werden kann. Zur Laufzeit werden die Instanzen der Komponenten verwaltet, während der Entwurfsphase werden die Komponenten anhand einer Beschreibung spezifiziert und in der Deploymentphase wird der Life cycle der Komponenten verwaltet. Komponenten können zum Beispiel nur gestartet werden, wenn alle Abhängigkeiten aufgelöst wurden. Dabei wird in statische und dynamische Abhängigkeiten unterschieden. Werden statisch abhängige Dienste neu konfiguriert, ist ein Neustart aller abhängigen Dienste nötig. Dynamisch abhängige Dienste hingegen können neue Konfigurationen ohne einen Neustart übernehmen und somit ihren alten Zustand beibehalten.



**Abbildung 2.17:** Abstraktionen in einem serviceorientierten Komponentenmodell (Quelle: [38])

Das iPOJO Framework [18] verwendet die zuvor genannten Ideen und Umsetzungen aus [13] und [38]. iPOJO ist ein serviceorientiertes Komponentenmodell, das neben den typischen Eigenschaften, wie *loose coupling*, *late binding*, *resiliency to dynamics* und *location transparency*, noch einen Erweiterungsmechanismus für Komponenten anbietet. iPOJO ermöglicht es, *Plain-Old-Java-Object (POJO)* in das serviceorientierte Framework zu integrieren, indem es diese um nichtfunktionale Eigenschaften erweitert. Zu den nichtfunktionalen Eigenschaften zählen zum Beispiel Persistenz, Sicherheit und automatisches Management. Das Ziel ist es dabei, den Entwicklungsprozess dahingehend zu vereinfachen, dass reine *POJOs*, die in diesem Fall die Business-Objects repräsentieren, dynamisch über eine serviceorientierte Umgebung angeboten werden. So kann sich ein Entwickler auf die Realisierung der Businesslogik konzentrieren, während das iPOJO Framework den Rest übernimmt. Abbildung 2.18 zeigt, wie so ein *POJO* von einem Container verwaltet wird und mit der externen Welt interagieren kann. Für diese Interaktion können vom iPOJO Core Framework unabhängige Handler entwickelt werden, um die nichtfunktionalen Eigenschaften zu handhaben.

## 2 Grundlagen

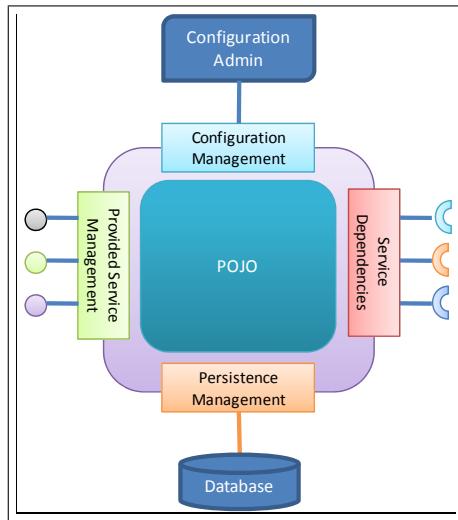


Abbildung 2.18: Erweiterung einer iPOJO Komponente (Quelle: [18])

In [10] bieten die Autoren mit dem *USENET Project* einen sehr aktuellen Einblick in die Thematik des dynamischen Deployments kollaborativer Komponenten in einer serviceorientierten Architektur und gehen somit noch einen Schritt weiter. Neben dieser Architektur wurde ein Profilmechanismus entwickelt, der es ermöglicht, nutzerspezifische Einstellungen und Eigenschaften des Endgerätes zu verwalten. Das bewirkt eine bessere, vom Nutzerkontext abhängige, Integration und Anpassung an eine kollaborative Umgebung. Zielstellung ist hier wieder die Vereinfachung des Entwicklungsprozesses solcher Komponenten, indem das Deployment, die Konfiguration und das Aktualisieren dynamisch und separat vom Entwickler verläuft. Um dies zu realisieren wird, die *Maschine-zu-Maschine (M2M)*-Kommunikation ausgereizt, deren Elemente in Abbildung 2.19 dargestellt sind. Im Zentrum dieser Architektur befindet sich die kollaborative *M2M*-Plattform, die sich um die Verwaltung und Interaktion der Nutzer und deren Endgeräte kümmert. Um Informationen zwischen den Teilnehmern auszutauschen, wird, wie im iPOJO Framework, auf die Verwendung von Business-Objects gesetzt. Die Umsetzung des dynamischen Deployments wird in dieser Plattform mit Hilfe des OSGi Frameworks realisiert. Abbildung 2.20 zeigt den Ablauf einer Interaktion in dieser Plattform.

Abschließend wird in [26] ein interessanter Ansatz vorgestellt, wie eine *SOA* über eine *Peer-to-Peer (P2P)* Netzwerkstruktur erstellt werden kann. Ein Dienst wird somit nicht mehr wie üblich über eine feste Netzwerkstruktur angeboten, sondern dynamisch mit der Teilnahme eines Peers in der *P2P* Umgebung. Die Anforderungen in einer heterogenen, mobilen *P2P* Umgebung sind weit höher, als in gängigen Client-Server-Architekturen. Zum Beispiel existiert ein weit höherer Datenverkehr und eine höhere Anforderung an die Vertraulichkeit einzelner Peers. Aus diesem Grund wurde eine Zweischichtenarchitektur erstellt, die sich zum Einen um die reinen *P2P* Mechanismen kümmert, wohingegen sich die zweite Schicht um die anwendungsspezifischen Dienste kümmert, wie in Abbildung 2.21 zu sehen ist.

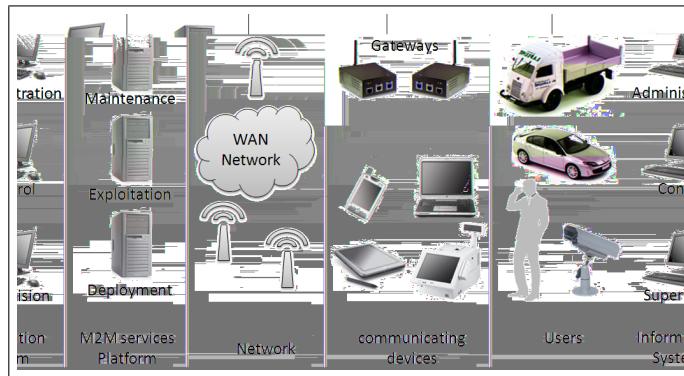


Abbildung 2.19: Elemente einer M2M Infrastruktur (Quelle: [10])

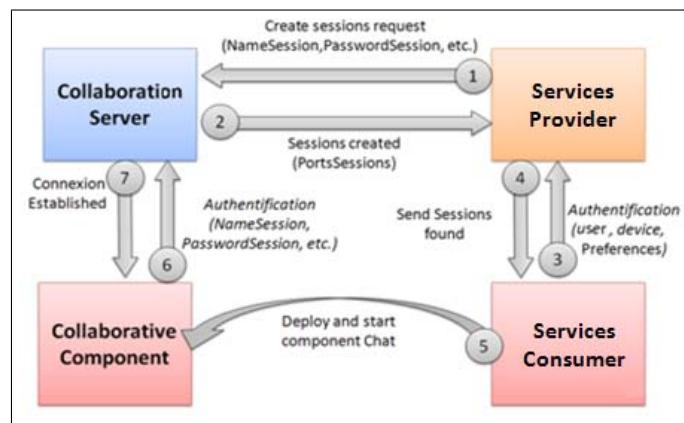


Abbildung 2.20: Kommunikationsablauf in der kollaborativen Plattform (Quelle: [10])

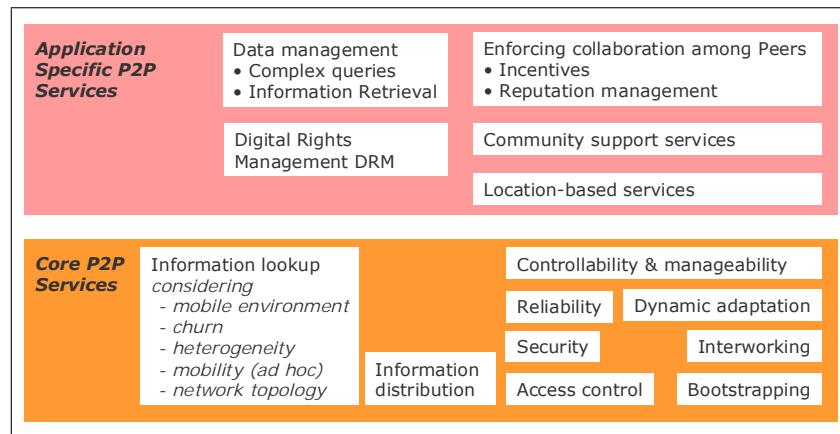


Abbildung 2.21: Die zwei Schichten der P2P-Plattform (Quelle: [26])

#### 2.5.4 Abgrenzung

In Tabelle 2.3 sind Eigenschaften der Dienstplattformen aufgelistet und die Unterstützung der Plattformen und Frameworks aus den Related Works darin vermerkt. Klar zu erkennen ist, dass alle Plattformen unterschiedliche Schwerpunkte aufweisen.

**Tabelle 2.3:** Gegenüberstellung der Plattformen

	OSGi	iPOJO	Gravity	[38]	USENET	P2P [26]	Mobilis	benötigte Features
Verteilte Plattform	(√)	(√)	–	(√)	–	✓	(√)	–
Multiple Plattformen	–	–	–	–	✓	✓	✓	–
Erweiterbare Dienste	✓	✓	✓	✓	✓	✓	✓	✓
Dienste austauschbar	✓	✓	?	✓	?	(√)	–	✓
Versionierung	✓	✓	?	✓	?	–	✓	✓
Lebenszyklus	✓	✓	✓	✓	✓	(–)	–	✓
Dienstbeschreibung	–	–	–	–	–	–	–	✓
Veröffentlichen	✓	✓	✓	✓	✓	✓	✓	✓
Discovery	✓	✓	✓	✓	✓	✓	✓	✓
Dynamisches Binden	✓	✓	✓	✓	✓	✓	–	✓
Codegenerierung	–	–	–	–	–	–	–	✓
<b>XMPP</b> -Unterstützung	–	–	–	✓	–	–	✓	✓
Kollaborative Plattform	–	–	–	–	✓	–	✓	✓
NF Eigenschaften	✓	✓	–	✓	✓	–	✓	–
Multiple Instanzen	(–)	–	✓	–	–	–	✓	✓
Autorisierung	(–)	–	–	–	✓	✓	–	✓
<b>P2P</b> -Kommunikation	–	–	–	–	–	✓	(√)	–
Dienst-Abhängigkeiten	✓	–	✓	✓	–	✓	–	✓

Abgesehen von der Mobilis-Plattform, unterstützen alle Plattformen das Austauschen von Diensten während der Laufzeit und definieren einen Life cycle für die Komponenten. Etwas mehr als die Hälfte definiert explizit bestehende Abhängigkeiten zwischen den Diensten, die in der Mobilis-Plattform nicht vermerkt sind.

Die Mobilis-Plattform ist jedoch eine der wenigen, die primär für kollaborative Dienste native eine Unterstützung mitbringt, das Kommunikationsprotokoll **XMPP** verwendet und es ermöglicht multiple Instanzen eines Dienstes zu erstellen.

Die **P2P**-Plattform aus [26] erzielt den höchsten Grad an Dynamik von allen Plattformen. Diese kommt ohne eine Server-Komponente für die Verwaltung aus, was aber einen sehr hohen Traffic-Overhead mit sich bringt, da für ein Discovery immer ein DNS-Broadcast verwendet wird.

Die meisten Plattformen, bis auf OSGi und Mobilis, sind lediglich Forschungsprojekte und bieten keine voll einsatzfähige Implementierung ihrer Plattform. Als interessant stellt sich die Plattform aus [38] heraus, da diese wahrscheinlich **XMPP** für die Umsetzung der Aktionen benutzt. Eine Implementierung ist jedoch nicht verfügbar, genauso

wenig wie ein detailliertes Konzept der Umsetzung von [XMPP](#) als Kommunikationsprotokoll in einer SOA.

In der letzten Spalte sind sie Anforderungen markiert, die die Mobilis-Plattform am Ende dieser Arbeit unterstützen soll. Eine detaillierte Definition dieser Anforderungen wird in Kapitel 3 vorgenommen.

## 2.6 Zusammenfassung

Am Anfang wurde ein kurzer Überblick über die Merkmale und Komponenten einer [SOA](#) gegeben. Im darauffolgenden Abschnitt wurde eine Einführung in das Dynamische Deployment gegeben und gezeigt, wie dieses unter Java funktioniert. Im Anschluss wurde anhand der OSGi Plattform ein detaillierter Einblick in eine Dynamische Service-Plattform gegeben. In Abschnitt 2.3 wurde das Kommunikationsprotokoll [XMPP](#) eingeführt und ein Überblick über die Architektur gegeben, sowie für diese Arbeit relevante [XEPs](#) betrachtet. Darauf aufbauend wurde die Mobilis-Plattform vorgestellt, die eine kollaborative serviceorientierte Plattform darstellt und zur Kommunikation [XMPP](#) als Grundlage verwendet. Im letzten Abschnitt 2.5 wurden verwandte Arbeiten betrachtet. Dabei wurde vor allem die Entwicklung, ausgehend von einem komponentenbasierten bis hin zu einem serviceorientierten Komponentenmodell, dargestellt. In der letzten Arbeit wurde ein wichtiger Ausblick eines serviceorientierten Komponentenmodells in Richtung [P2P](#) gegeben, zu der sich die Mobilis-Plattform auf langfristigem Weg hin entwickeln sollte.



# 3 Anforderungsanalyse

Das folgende Kapitel soll in die Aufgabenstellung und Problematik dieser Arbeit einführen. Dafür wird in diesem Kapitel der aktuelle Stand der Mobilis-Plattform betrachtet. Zu Beginn wird dafür auf die Entwicklung in der Mobilis-Plattform in Abschnitt 3.1 eingegangen, in welcher der aktuelle Entwicklungsprozess der Clients und Services vorgestellt wird und die damit verbundenen Problembereiche aufgezeigt werden. In Abschnitt 3.2 wird erörtert, welche Anwendungsfälle und Rollenverteilung für diese Arbeit gefordert werden. Die ausgearbeiteten Anwendungsfälle sollen im Anschluss, in Abschnitt 3.3, formalisiert und nach Priorität geordnet werden. Neben den Anforderungen an das System sollen in 3.4 Kriterien definiert werden, die bewusst nicht in dieser Arbeit umgesetzt werden.

## 3.1 Entwicklung in der Mobilis-Plattform

Die Entwicklung einer Client-Server-Anwendung in der Mobilis-Plattform gliedert sich in den meisten Szenarien in folgende drei Teilaufgaben:

1. Definition und Erstellung des Kommunikationsprotokolls
2. Entwicklung einer Server-Anwendung(Mobilis-Service) und Adaption des Kommunikationsprotokolls unter Verwendung des Mobilis-Server
3. Entwicklung einer Client-Anwendung(Mobilis Client) und Adaption des Kommunikationsprotokolls unter Verwendung eines clientseitigen XMPP-Frameworks

### 3.1.1 Definition und Erstellung des Kommunikationsprotokolls

Wie bereits beschrieben, wird für den Austausch von IQs in der Mobilis-Plattform die Verwendung der XMPP-Beans empfohlen (siehe Kapitel 2.4). Jedes Kommunikationspaket, dass zwischen Client und Service ausgetauscht wird, sollte von der Basiskomponente XMPP-Bean Beschreibung ableiten.

Die Erstellung eines XMPP-Beans ist in den meisten Fällen unkompliziert. Es ist nötig von der Klasse XMPP-Bean erben zu lassen und die jeweiligen Methoden mit Logik zu füllen. Wichtig ist, dass genau auf die Verwendung der Syntax für die Konvertierung in eine XML-Struktur geachtet wird, da sich hier schnell Fehler einschleichen, die erfahrungsgemäß auf der mobilen Plattform nur schwer gefunden werden. Die Konvertierung der XML-Struktur in ein XMPP-Bean, muss unter Verwendung eines XML-Parsers erfolgen.

Die Schnittstelle eines XMPP-Beans und deren Informationen, bildet die XMPP-Info.

### 3 Anforderungsanalyse

Damit wird einheitlich beschrieben, wie die De- und Serialisierung von [XMPP](#)-Nachrichten ermöglicht wird. Informationen eines XMPP-Beans können von XMPP-Info erben und somit beliebig tief verschachtelt werden. Für einen detaillierten Einblick wird an dieser Stelle auf die Arbeiten von Benjamin Söllner [45] und Robert Lübke [30] verwiesen, die eine konzeptionelle Definition und Verwendung zur Verfügung stellen. Aus letzterer Arbeit ist das abstrakte Schema in Abbildung 3.1 zu sehen.

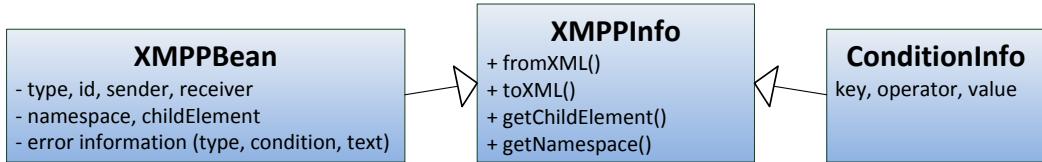


Abbildung 3.1: Schema XMPP-Bean (Quelle: [30])

Alle vom Entwickler erstellen XMPP-Beans müssen im Server, sowie im Client referenziert werden oder zumindest die gleiche Struktur aufweisen, sodass sie bei der Registrierung im [XMPP](#)-Netzwerk auch von beiden Seiten erkannt und verstanden werden.

#### 3.1.2 Entwicklung eines Mobilis-Service

In der Mobilis-Plattform ist zur Ausführung der Mobilis-Services der Mobilis-Server notwendig. Der Mobilis-Server verwaltet, startet und stoppt alle ihm bekannten Service-Instanzen. Eine Mobilis-Service-Instanz ist dabei ein instantiiertes Objekt einer Mobilis-Service-Klasse, die vom Entwickler erstellt wurde und vom Mobilis-Service abgeleitet wird.

In Abbildung 3.2 ist der Entwicklungsprozess dargestellt, der notwendig ist, um einen neuen Mobilis-Service in die Mobilis-Plattform zu integrieren. Bevor ein neuer Mobilis-Service integriert werden kann, muss der laufende Mobilis-Server gestoppt werden. Dies hat zur Folge, dass alle aktuell laufenden Service-Instanzen angehalten und für die Clients nicht mehr erreichbar sind. Erst dann kann der neu entwickelte Mobilis-Service implementiert werden, indem der Quellcode in das Serverprojekt integriert wird. Daraufhin kann die Konfiguration des neuen Mobilis-Service vorgenommen werden, um zum Beispiel dessen Verbindungsverhalten im [XMPP](#)-Netzwerk zu definieren. Danach kann der Mobilis-Server wieder gestartet werden. Der neue Mobilis-Service ist beim Start des Mobilis-Server bereits registriert und kann, falls er vom Typ App-specific-Service ist, mehrere Service-Instanzen starten.

Für die Verwendung der XMPP-Beans wird ein sogenannter *Glue Code* bereitgestellt, der von Benjamin Söllner [45] und Robert Lübke [30] entwickelt wurde, um ein [XMPP](#)-IQ-Stanza in ein XMPP-Bean zu konvertieren. Das Schema des *Glue Code* ist in Abbildung 3.3 dargestellt. Die wichtigste Klasse stellt die `BeanIQAdapter`-Klasse dar, die für die Konvertierung verantwortlich ist.

Eine Benachrichtigung über das Eintreffen eines XMPP-Beans wird auf der serverseite mit Hilfe eines Packet-Listener vorgenommen, der registriert wird und dann auf eingehende IQs reagiert. IQs, die dem Server nicht bekannt sind, werden mit einem

ERROR-IQ beantwortet. Eine detaillierte Beschreibung mit Beispielen ist im Tutorial von Robert Lübke [29] zu finden.

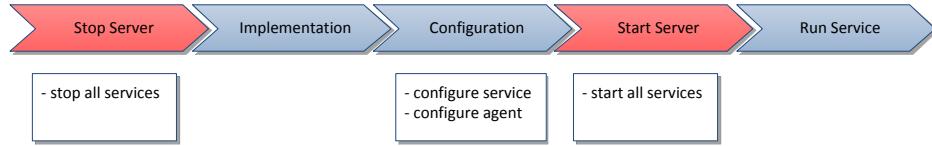


Abbildung 3.2: Entwicklungsprozess eines Mobilis-Service

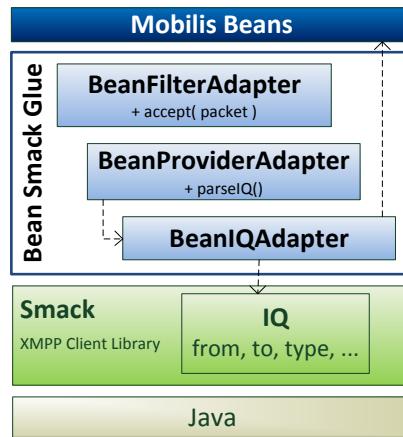


Abbildung 3.3: XMPP Glue Code im Mobilis Server (Quelle: [30])

### 3.1.3 Entwicklung einer Mobilis-Client-Anwendung

Die Registrierung aller benötigten XMPP-Beans muss, wie im Service, im verwendeten XMPP-Framework erfolgen, sodass die Erkennung und Weiterleitung vorgenommen werden kann. Zum aktuellen Zeitpunkt ist die Hauptplattform für Mobilis-Clients die Android-Plattform. Für diese wurde ein spezielles Framework namens [Mobilis XMPP for Android \(MXA\)](#) entwickelt, das eine Portierung von SMACK auf Android ermöglicht.

Für die Handhabung der XMPP-Beans wird die `Parceller`-Klasse empfohlen, welche bereits Funktionalitäten mitbringt, um ein XMPP-Bean zu registrieren und eingehende IQs in das entsprechende XMPP-Bean zu konvertieren. Von jedem zu nutzenden XMPP-Bean wird im Parceller ein Prototyp registriert, mit dem eingehende IQs verglichen werden.

Mithilfe einer Schnittstelle kann sich eine Clientapplikation für alle registrierten eingehenden IQs benachrichtigen lassen. Jedes erkannte IQ wird somit an die Schnittstelle weitergereicht und kann mithilfe des Parcellers in ein XMPP-Bean konvertiert werden. Diese Funktion wurde im Mobilis-Server durch den `BeanIQAdapter` vorgenommen. Eine detaillierte Beschreibung mit Beispielen ist im Tutorial von Robert Lübke [29] zu finden.

### 3 Anforderungsanalyse

#### 3.1.4 Grenzen der Mobilis-Plattform

An erster Stelle ist zu sehen, dass die Entwicklung des Netzwerkprotokolls eines Mobilis-Service sehr aufwendig und unübersichtlich ist. Die Reihenfolge der Pakete ist nicht explizit festgelegt und erschwert somit das Monitoring der einzelnen Mobilis-Services. Zudem ist ein Mobilis-Service nicht in der Lage sich selbst zu beschreiben, sodass ein Client-Entwickler Einsicht in den Quellcode des Mobilis-Server benötigt und diesen aufwendig analysieren muss, bis er diesen korrekt ansprechen und nutzen kann.

Um dem Mobilis-Server neue Dienste hinzuzufügen, muss der Mobilis-Server erst beendet werden. Dann müssen Änderungen im Quellcode vorgenommen und der Mobilis-Server neu gestartet werden. Hierbei meldet sich dieser selbst und seine Dienste wieder neu im XMPP-Netzwerk an, was unter Umständen einige Zeit in Anspruch nehmen kann. Des Weiteren können Dienste nicht während der Laufzeit aktualisiert werden, was bei zeitkritischen Updates nicht zu unterschätzen ist.

Die einzelnen Dienste können unter Umständen bei einem Fehler den gesamten Mobilis-Server zum Absturz bringen. Dies ist durch die geringe Fehlertoleranz des Mobilis-Server begründet. Sollte ein Dienst in einen Fehlerzustand übergehen, so wird vom Mobilis-Server kein Fehlerprotokoll ausgeführt, welches den Dienst beendet und Ressourcen wieder frei gibt oder versucht den Dienst neu zu starten.

Ein Mobilis-Service besitzt des Weiteren nur einen primitiven Life cycle, welcher von außen nur eingeschränkt beeinflussbar ist. Ein Mobilis-Service kann nur gestartet und gestoppt werden. Es ist nicht möglich einen Dienst zu Veröffentlichen, Installieren oder Deinstallieren. Von außen ist es nur möglich eine Service-Instanz zu Starten, wohingegen das Stoppen nur intern möglich ist.

Zuletzt ist der Aufwand einer Portierung hin zu neuen Plattformen sehr hoch. Für die Kommunikation werden die XMPP-Beans verwendet, welche Java-spezifisch implementiert sind. Soll nun ein neuer Client, zum Beispiel auf der Basis von Windows Mobile Phone(WMP), mit in das Netzprotokoll integriert werden, so müssen alle XMPP-Beans in C# neu geschrieben werden. Dies stellt ein erhebliches Defizit für ein System dar, das plattformunabhängig sein soll. Wenn davon ausgegangen wird, dass ein Entwickler einen Dienst erstellt, der Android-, iPhone-, WMP- und Browser-Clients unterstützen soll, so müssen die gesamten XMPP-Beans und deren Logik zur De- und Serialisierung für jede Technologie neu implementiert werden. Hier fehlt es an einer einheitlichen Vorlage.

## 3.2 Use Case Analyse

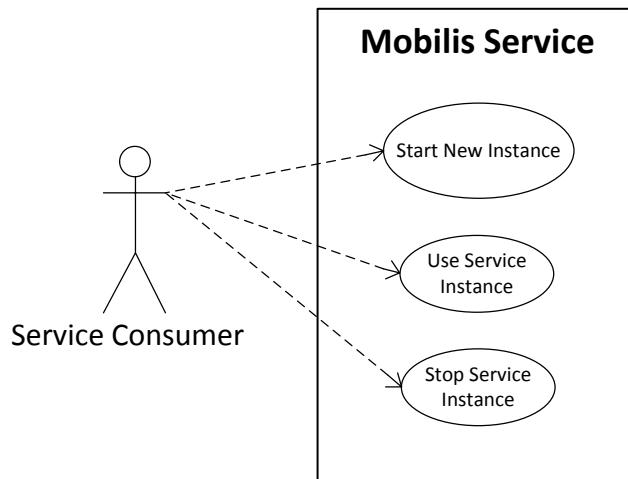
Ausgehend von der aktuellen Mobilis-Plattform und den damit aufgezeigten Problemen im vorherigen Abschnitt, sollen hier die existierenden Rollen, sowie die Anwendungsfälle ausgearbeitet werden. Damit soll eine erste Anforderungsanalyse entstehen, die dann im darauffolgenden Abschnitt [3.3](#) formalisiert wird.

In der Mobilis-Plattform existiert, neben den beiden Standardrollen Dienstnutzer und

Dienstanbieter, noch der Dienstentwickler. Dieser ist für die konkrete Umsetzung eines Mobilis-Service oder Clients zuständig und kann somit im Speziellen der Client- oder der Serviceentwickler sein. Typischerweise vertritt eine reale Person meist mehrere Rollen, so ist der Entwickler der Service-Komponente meist auch der Client-Entwickler und Dienstprovider.

### 3.2.1 Dienstnutzer

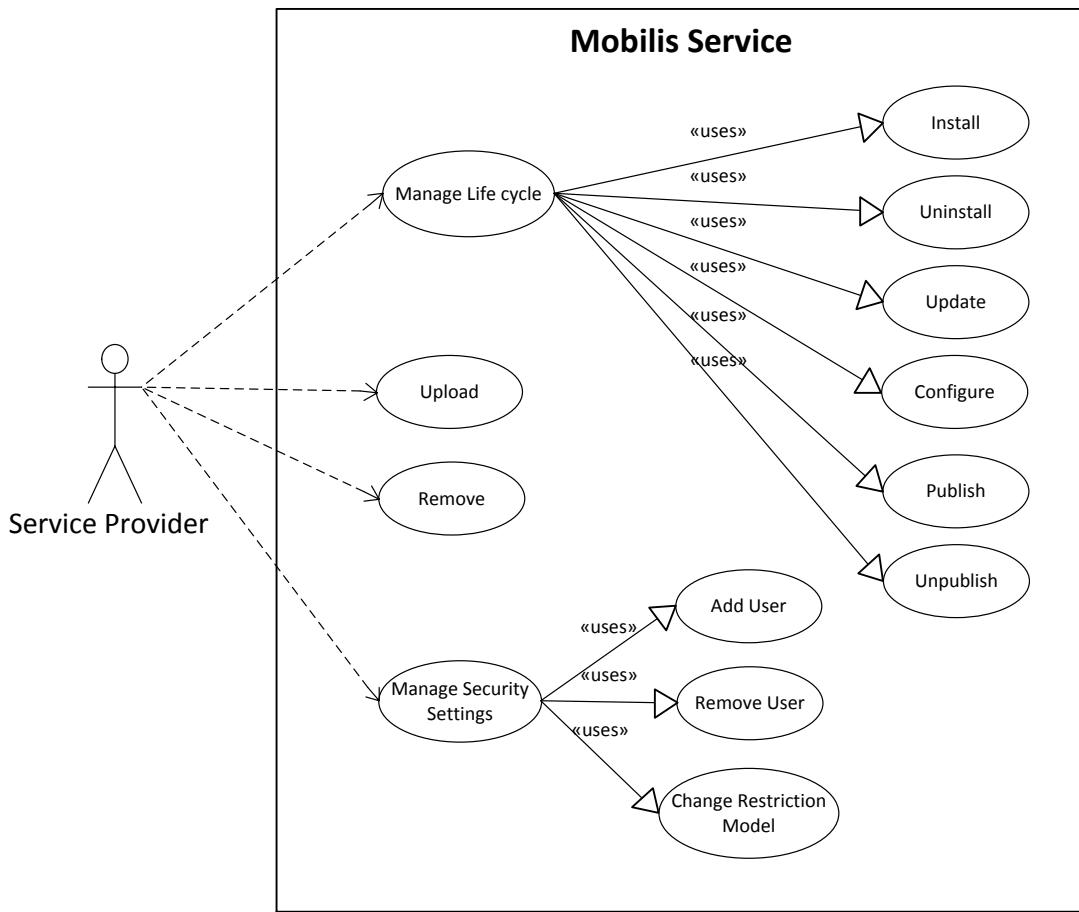
Der Dienstnutzer soll in der Mobilis-Plattform in der Lage sein, eine neue Service-Instanz zu starten, bereits laufende Service-Instanzen zu nutzen und zu stoppen. Dies entspricht im weitestgehenden Sinne der Rolle eines Dienstnutzer in einer normalen **SOA**-Umgebung. Der Unterschied ist jedoch, dass ein Mobilis-Service mehrfach instantiiert werden kann. In der Abbildung 3.4 sind alle genannten Anwendungsfälle in einem Use-Case Diagramm dargestellt.



**Abbildung 3.4:** Use-Case Mobilis-Service Consumer

### 3.2.2 Dienstanbieter

Der Dienstanbieter hat die Funktion eines Administrators. Die einzige Voraussetzung ist, dass der von ihm zu administrierende Mobilis-Service bereits in einer kompilierten und gepackten Form, zusammen mit der Dienstbeschreibung, vorhanden sein muss. Erst dann ist er in der Lage, den neuen Mobilis-Service zum Mobilis-Server hochzuladen oder diesen wieder zu entfernen. Die wichtigste Funktion des Dienstanbieters ist es, den Life cycle eines Mobilis-Service zu verwalten. Dazu gehört das Installieren, Deinstallieren, Aktualisieren, Konfigurieren, Veröffentlichen und die Veröffentlichung wieder rückgängig zu machen, wie in Abbildung 3.5 dargestellt ist. Zusätzlich soll dem Dienstanbieter ermöglicht werden, Einstellungen zur Authorisierung zu verwalten. Dazu zählt im einfachsten Fall das Zuweisen und wieder Entfernen von Benutzern, denen es erlaubt ist einen Mobilis-Service zu nutzen.



**Abbildung 3.5:** Use Case Mobilis-Service Provider

### 3.2.3 Entwickler

Ein Entwickler wird in der Mobilis-Plattform in einen Cliententwickler und einen Dienstentwickler unterschieden. Grundlegend ist ein Entwickler für die Logik einer Client- oder Serviceanwendung zuständig. Im ersten Schritt muss dafür das Kommunikationsverhalten in einer Dienstbeschreibung definiert werden. Die Dienstbeschreibung muss dafür so gestaltet sein, dass diese vom Cliententwickler, sowie vom Dienstentwickler gleichermaßen interpretiert und umgesetzt werden kann.

Aus dieser Dienstbeschreibung soll der Entwickler, unter Verwendung der Codegenerierung, einen Client- oder Service-Stub generieren. Dadurch sollen die Elemente der Dienstbeschreibung auf sprachspezifische Konstrukte portiert werden und in der Implementierung auf einfache Weise verwendet werden können, um über das [XMPP](#)-Netzwerk zu kommunizieren.

Der Dienstentwickler soll zusätzlich nach der Implementation des Mobilis-Service diesen in einer gepackten Form anbieten. Dabei sollen alle benötigten Referenzen, wie Bibliotheken und andere Ressourcen, in diesem Paket enthalten sein. Eine Gesamtübersicht der Anwendungsfälle eines Entwicklers ist in Abbildung 3.6 zu sehen.

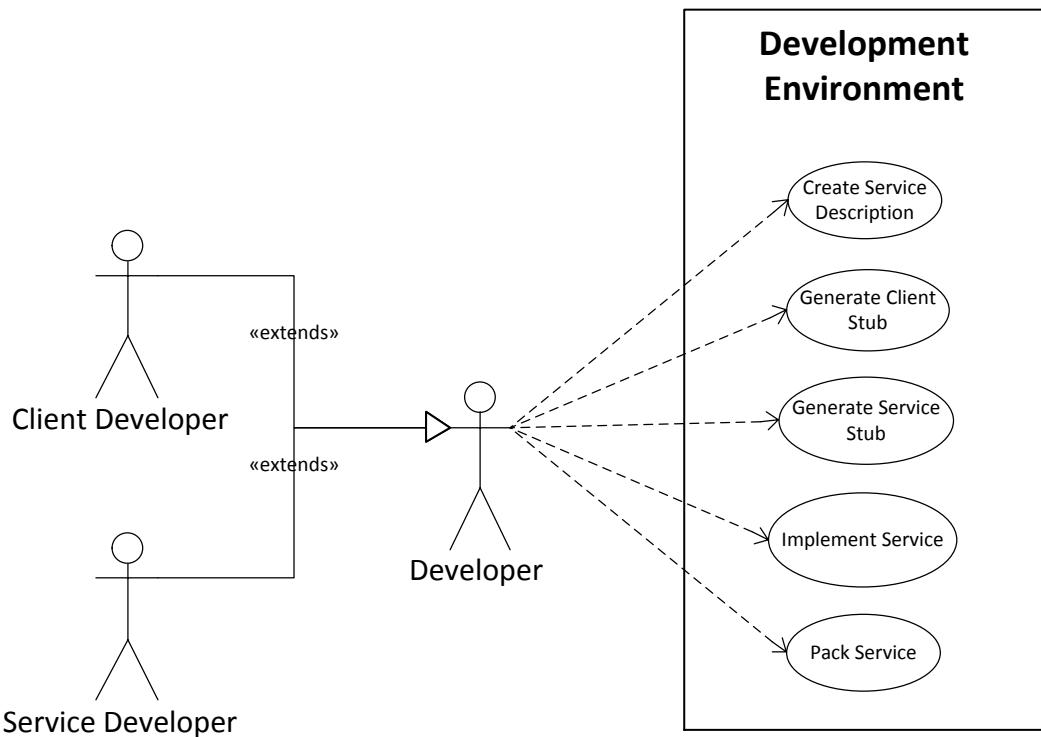


Abbildung 3.6: Use-Case Mobilis-Service Developer

### 3.3 Anforderungen

In diesem Abschnitt sollen die zuvor ermittelten Anwendungsfälle und die aufgezeigten Grenzen der aktuellen Mobilis-Plattform als Grundlage für die Anforderungen an diese Arbeit verwendet werden. Die Anforderungen werden dafür in funktionale und nichtfunktionale Anforderungen unterschieden und nach Prioritäten geordnet.

#### 3.3.1 Funktionale Anforderungen

Die funktionalen Anforderungen an das System werden in die drei Teile *FA-1 Dynamische Serviceumgebung*, *FA-2 Dienstbeschreibungssprache* und *FA-3 Codegenerierung* unterteilt.

##### FA-1 Dynamische Serviceumgebung

In diesem Teil soll das Laufzeitverhalten der Dienste in der Mobilis-Plattform betrachtet werden.

###### FA-1.1 Kompakte und generische Laufzeitumgebung

Der Mobilis-Server soll in eine kompakte Form überführt werden, welche nur noch die nötigsten Funktionen bereitstellt. Der Mobilis-Server soll nur noch eine Art Grundbaustein darstellen, den der Administrator nach belieben anpassen und erweitern kann.

### **3 Anforderungsanalyse**

Dies soll vor allem verhindern, dass unnötige Dienste installiert, oder sogar aktiv sind, aber nie genutzt werden und dennoch Ressourcen verbrauchen.

#### **FA-1.2 De-/Registrierung von Diensten**

Durch die Reduzierung des Mobilis-Server auf die wichtigsten Komponenten, ist ein erweiterter Mechanismus nötig, um neue Dienste zu registrieren und auch wieder zu deregistrieren, da diese Möglichkeit momentan ausschließlich im Code des Mobilis-Server vorgenommen werden kann.

#### **FA-1.3 Discovery und dynamisches Binden an Dienste**

Das Discovery und Binden eines Clients an einen Dienst muss dem Umstand angepasst werden, dass gewünschte Dienste nicht verfügbar sind oder der Client nicht die notwendigen Berechtigungen besitzt einen Dienst zu nutzen. Des Weiteren soll ein dynamisches Binden von Diensten ermöglicht werden, dadurch könnte ein neuer Dienst als Paket vorliegen, welches erst auf Anfrage entpackt und im Mobilis-Server deployed wird.

#### **FA-1.4 Deployment-Unterstützung externer Dienste**

Dem Entwickler soll die Möglichkeit gegeben werden, ohne großen Aufwand, neu entwickelte Dienste auf dem Mobilis-Server auszuführen. Eine Funktion zum Hochladen neuer Dienste auf den Mobilis-Server, oder dem Mobilis-Server mitzuteilen, unter welcher Adresse er neue Dienste auffinden und herunterladen kann, sollten mit in die Plattform integriert werden. Dabei soll der Dienst nach dem Laden sofort zur Nutzung zur Verfügung stehen.

#### **FA-1.5 Steuerung des Lebenszyklus eines Dienstes**

Jedem Teilnehmer, Administrator oder Entwickler, soll die Möglichkeit gegeben werden, den Lebenszyklus eines Dienstes während der Laufzeit zu beeinflussen. Dazu zählt unter anderem das Starten und Stoppen von Diensten, ohne den Mobilis-Server zu beenden. Des Weiteren wäre eine Art Ressourcenfreigabe oder Schlafmodus für einen Dienst denkbar, um Ressourcen zu sparen.

#### **FA-1.6 Einfache Zugriffssteuerung für Services**

Zuletzt soll es ermöglicht werden die Nutzung von Diensten zu reglementieren. Einem Dienstkonsument soll durch Authentifizierung und Autorisierung die Nutzung einzelner Funktionen oder ganzer Dienste gestattet beziehungsweise verweigert werden. Eine Zuweisung der Dienstkonsumenten zu Gruppen und das Definieren von Gruppenrichtlinien wäre eine Art dies umzusetzen.

### **FA-2 Dienstbeschreibungssprache**

Der zweite Teil der funktionalen Anforderungen soll sich mit der Dienstbeschreibungssprache beschäftigen.

#### **FA-2.1 Verwendung XMPP-konformer Standards**

Die Dienstbeschreibung soll sich genau am XMPP-Standard orientieren. Das bedeutet, dass alle verwendeten Elemente in dieser Sprache auf ein korrespondierendes Element im XMPP-Standard abgebildet werden müssen. Zum Beispiel muss jede Art von Nachricht, welche von einem Teilnehmer zu einem anderen gesendet wird, ein Message- oder IQ-Stanza, mit allen notwendigen Attributen auf der XMPP-Seite beschrieben werden.

#### **FA-2.2 Schnittstelle zwischen Dienstanbieter und Dienstnutzer**

Die Dienstbeschreibung muss mindestens die auszutauschenden Nachrichten zwischen Dienstanbieter und Dienstnutzer beschreiben. Weiterhin sollten möglichst viele Metadaten mit in die Beschreibung integriert werden, wie zum Beispiel, den Status den ein Teilnehmer annehmen kann.

#### **FA-2.3 Unterstützung alternativer Transportprotokolle**

Wünschenswert wäre eine Möglichkeit anzugeben, wie die Daten zwischen den Teilnehmern transportiert werden. Sollen etwa die normalen XMPP-XML-Streams oder das HTTP-Protokoll verwendet werden (siehe BOSH).

#### **FA-2.4 Schemavalidierung der Dienstbeschreibung**

Die Dienstbeschreibung soll eine Schemavorgabe nutzen, um stetig eine korrekte Verwendung der Syntax sicherzustellen und Fehler frühzeitig abfangen.

### **FA-3 Codegenerierung**

Im letzten Teil der funktionalen Anforderungen soll auf die Codegenerierung eingegangen werden.

#### **FA-3.1 Unterstützung unterschiedlicher Plattformen**

Die Codegenerierung soll nicht nur auf Java beschränkt bleiben, sondern für verschiedene Plattformen genutzt werden können. Dies ist wichtig, da die Mobilis-Plattform mit vielen verschiedenen Plattformen interagieren soll.

#### **FA-3.2 Einfache Realisierbarkeit neuer Plattformen**

Das Framework zur Codegenerierung sollte möglichst einfach um neue Zielplattformen erweitert werden können, sodass effizient neue Templates für die Entwickler anderer Plattformen erstellt werden können.

#### **FA-3.3 Automatisiertes Test-Framework**

Wünschenswert wäre eine Art Test-Client, welcher optional mit erzeugt werden kann, wenn die Dienstbeschreibung in plattformspezifischen Code transformiert wird. Dies soll dem Entwickler eine einfache Möglichkeit bieten, seine Anwendungslogik auf beiden Seiten schnell und einfach zu testen. So kann zum Beispiel nach der Definition der Dienstbeschreibung der serverspezifische Code erzeugt und die Anwendungslogik implementiert werden. Dadurch soll das Testen, ohne eine komplexe Implementierung eines Clients, gewährleistet werden.

**Tabelle 3.1:** Zusammenfassung der funktionalen Anforderungen

Nr.	Anforderung	Priorität
<b>FA-1</b>	<b>Dynamische Serviceumgebung</b>	
FA-1.1	Kompakte und generische Laufzeitumgebung	Hoch
FA-1.2	De-/Registrierung von Diensten	Hoch
FA-1.3	Discovery und dynamisches Binden an Dienste	Hoch
FA-1.4	Deployment-Unterstützung externer Dienste	Hoch
FA-1.5	Steuerung des Lebenszyklus eines Dienstes	Hoch
FA-1.6	Einfache Zugriffssteuerung für Services	Mittel
<b>FA-2</b>	<b>Dienstbeschreibungssprache</b>	
FA-2.1	Verwendung XMPP-konformer Standards	Hoch
FA-2.2	Schnittstelle zwischen Dienstnutzer und Dienstanbieter	Hoch
FA-2.3	Unterstützung alternativer Transportprotokolle	Mittel
FA-2.4	Schemavalidierung der Dienstbeschreibung	Mittel
<b>FA-3</b>	<b>Codegenerierung</b>	
FA-3.1	Unterstützung unterschiedlicher Plattformen	Hoch
FA-3.2	Einfache Realisierbarkeit neuer Plattformen	Mittel
FA-3.3	Automatisiertes Test-Framework	Niedrig

### 3.3.2 Nichtfunktionale Anforderungen

Neben den funktionalen Anforderungen, die das System erfüllen soll, ist es darüber hinaus notwendig, dass qualitative Bedingungen erfüllt werden.

#### NF-1 Erweiterbarkeit und Austauschbarkeit

Eines der Hauptziele dieser Arbeit ist es, dass der Mobilis-Server während er Laufzeit um beliebige Dienste erweitert und reduziert werden kann. Das bedeutet, dass eine beliebige Komplexität eines Mobilis-Server erreicht werden kann. Bei komplexen Systemen besteht meistens eine sehr hohe Frequentierung, was vor allem die Wartbarkeit einzelner Dienste notwendig macht, aber auch erschwert. Aus diesem Grund sollen einzelne Dienste gewartet werden können, ohne dass das gesamte System in einen Wartungsmodus versetzt werden muss. Zusätzlich zur Wartung soll die Austauschbarkeit eines Dienstes während der Laufzeit möglich sein, um zum Beispiel eine neue Version einzuspielen.

#### NF-2 Interoperabilität

Wie bereits beschrieben, ist eines der Ziele, dass ein Mobilis-Server beliebig erweiterbar sein soll. Dies heißt auch, dass jeder Mobilis-Server unterschiedliche Services anbieten kann. Ein Ziel des serviceorientierten Ansatzes ist es, dass Dienste wiederum von anderen Diensten konsumiert werden können. Demzufolge sollen die Mobilis-Server interoperabel zueinander sein, unabhängig von den Diensten, die sie anbieten.

Des Weiteren soll die Beschreibungssprache so konzipiert werden, dass sie nicht nur auf die Mobilis-Plattform beschränkt ist, sondern für beliebige [XEPs](#), die ein Serviceverhalten anbieten möchten, verwendet werden kann.

**NF-3 Fehlertoleranz und Stabilität**

Durch die Verwendung dynamischer Dienste besteht die Gefahr, dass Dienste (noch) nicht vorhanden sind, wenn sie angefordert werden. Das kann passieren, wenn ein Dienst nicht mehr reagiert oder sich aus unbestimmten Gründen beendet hat. In diesen Fällen muss die Stabilität des restlichen Systems gewährleistet werden und, wenn möglich, eine Fehlerbehandlung durch das System stattfinden. Zum Beispiel könnte das System dem anfragenden Dienstkonsumenten mitteilen, wenn der gewünschte Dienst auf diesem Mobilis-Server nicht registriert wurde. Des Weiteren sollte das System versuchen den Dienst in solchen Fehlerfällen neu zu starten.

**NF-4 Intuitive Nutzerschnittstellen**

Der Entwickler sollte bei der Erstellung der Beschreibungssprache, sowie beim Life cycle Management, durch eine visuelle Benutzeroberfläche unterstützt werden. Während der Life cycle der dynamischen Dienste durch einfache Kontrollfelder in der Oberfläche realisiert werden kann, könnte für die Erstellung der Kommunikationsbeschreibung eine Art Wizard oder grafisches Formular angeboten werden.

Des Weiteren sollte eine Bedienung des Mobilis-Server über eine Shell ermöglicht werden. Dies hat den Vorteil, dass der Mobilis-Server auf Systemen eingesetzt werden kann, die keine grafische Ausgabe ermöglichen. Dieser Aspekt wird deshalb mit angesprochen, da dies gerade bei Serversystemen öfters der Fall ist.

**Tabelle 3.2:** Zusammenfassung der nichtfunktionalen Anforderungen

Nr.	Anforderung	Priorität
NF-1	Erweiterbarkeit und Austauschbarkeit	Hoch
NF-2	Interoperabilität	Hoch
NF-3	Fehlertoleranz und Stabilität	Hoch
NF-4	Intuitive Nutzerschnittstellen	Mittel

## 3.4 Abgrenzungskriterien

Im Folgenden sollen diejenigen Kriterien aufgeführt werden, die nicht durch diese Arbeit abgedeckt werden können.

**AK-1 Code-Generation Framework-unabhängig**

Ein wichtiger Aspekt dieser Arbeit ist es, dem Entwickler ein Werkzeug an die Hand zu geben, mit dessen Hilfe er die verwendeten Nachrichten für das Kommunikationsprotokoll unabhängig von der Plattform beschreiben und unter Verwendung von Code-Generation in plattformspezifischen Code transformieren kann. Hierbei ist es jedoch nicht vorauszusehen, in welcher Art und Weise der Entwickler mit dem XMPP-Server kommuniziert. Dies kann zum Beispiel mit Hilfe eines externen Frameworks wie SMACK [6] oder das MXA [28], sowie mit einer vom Entwickler selbst erstellten Lösung der Fall sein. Diesbezüglich soll lediglich ein Objekt in der plattformspezifischen Sprache erstellt werden, welches jegliche Informationen der Nachricht, wie zum Beispiel

### 3 Anforderungsanalyse

die ID oder den Namespace über Methoden anbietet, so wie es in der aktuellen Version der Mobilis-Plattform mit den XMPP-Beans gehandhabt wird. Der Entwickler ist folglich selbst dafür zuständig eine Adaption dieses Objektes an eine für ihn verwendbare Struktur durchzuführen.

#### AK-2 Keine Modell-Code Synchronisierung

Die Codegenerierung soll in dieser Arbeit lediglich eine “One-Way“-Transformation anbieten. Dies bedeutet, dass jegliche Modifikationen im Code nicht mit der Dienstbeschreibung synchronisiert werden. Die Funktionalität, Code und Dienstbeschreibung synchron zu halten, stellt einen erheblichen Mehraufwand dar, zumal dessen komplette Umsetzung auch fraglich ist. Der Entwickler muss sich somit selbst dafür um eine manuelle Synchronisation kümmern.

**Tabelle 3.3:** Tabellarische Zusammenfassung der Abgrenzungskriterien

Nr.	Kriterium
AK-1	Code-Generation Framework-unabhängig
AK-2	Keine Modell-Code Synchronisierung

## 3.5 Zusammenfassung

In diesem Kapitel wurde zu Beginn in Abschnitt 3.1 eine kurze Einführung in den Entwicklungsprozess in die Mobilis-Plattform gegeben. Dafür wurde auf die Erstellung und Umsetzung des Kommunikationsprotokolls mithilfe der XMPP-Beans eingegangen, die sowohl auf dem Service, als auch auf dem Client die

## 4 Konzeption

In diesem Kapitel werden die Konzepte vorgestellt, die für eine Umsetzung der aus Kapitel 3.3 erstellten Anforderungen notwendig sind.

Einer der Hauptaufgaben der Mobilis-Plattform ist, wie schon angesprochen, die Unterstützung des Entwicklers bei der Erstellung von kollaborativen und mobilen sozialen Anwendungen. Dieser Entwicklung unterliegt in den meisten Fällen ein Entwicklungsprozess, der für die Mobilis-Plattform bereits in Kapitel 3.1 vorgestellt wurde und in dieser Arbeit weiter optimiert wird.

In Abbildung 4.1 ist eine optimierte Variante dargestellt. Neu ist in diesem Prozess, dass der Mobilis-Server nicht mehr heruntergefahren werden muss, um neue Mobilis Services zu installieren und dann zu starten. Die Elemente *Pack Service*, *Upload* und *Installation* gewährleisten eine dynamische Integration neuer Dienste in die Dienstplattform.

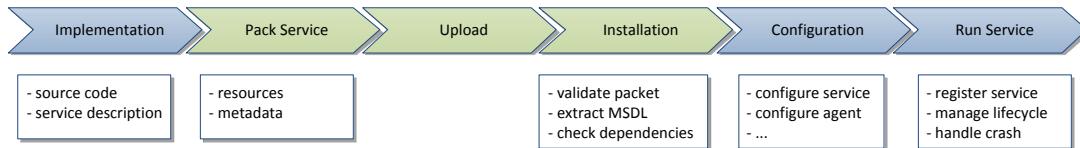


Abbildung 4.1: Ablauf eines neuen Entwicklungsprozesses

Für die Umsetzung dieser Variante ist zunächst eine globale Betrachtung der gesamten Mobilis-Plattform durchzuführen. Im Fokus dieser Betrachtung stehen das dynamische Deployment und die Definition und Integration einer DienstSchnittstelle für die einzelnen Mobilis-Services und Mobilis-Clients. In Abschnitt 4.1 werden dafür verschiedene Ansätze vorgestellt und diskutiert. Darauf aufbauend wird das Konzept des dynamischen Deployments in Abschnitt 4.2 im Detail erläutert. Hier soll besprochen werden, wie die aktuelle Mobilis-Plattform angepasst werden muss, um die vorher ermittelte Architektur umzusetzen. Die Beschreibung der DienstSchnitstelle wird in Abschnitt 4.3 vorgestellt. Dabei werden notwendige Elemente einer Dienstbeschreibung ermittelt, die einen Mobilis-Service und dessen Interaktionen und Abhängigkeiten beschreiben. Ein Vergleich zu [WSDL](#) soll zeigen, ob eventuell bereits existierende Konzepte und Werkzeuge genutzt werden können. Die detaillierte Umsetzung der [XMPP](#)-Beschreibung, sowie die daraus resultierende Codegenerierung sollen in den Abschnitten 4.4 und 4.5 vorgestellt werden.

## 4.1 Architekturkonzept

Das dynamische Integrieren von Mobilis-Services in den Mobilis-Server während der Laufzeit bedarf einer grundlegenden Modifikation der bereits bestehenden Architektur des Mobilis-Servers. In Abschnitt 4.1.1 soll dafür eine allgemeine Betrachtung durchgeführt werden, die einen Überblick über die Zielarchitektur, unabhängig von der Laufzeitumgebung, gibt.

Darauf aufbauen werden in den Abschnitten 4.1.2 und 4.1.3 Ansätze für die Umsetzung der Architektur betrachtet und anschließend in 4.1.4 diskutiert und gegenübergestellt, um eine optimale Zielarchitektur zu konzipieren.

Für ein besseres Verständnis der Konzepte und deren Umsetzung wird an einigen Stellen das Mobilis-Projekt MobilisXHunt [27] als Beispiel herangezogen.

### 4.1.1 Allgemeines Schema

In der ursprünglichen Version der Mobilis-Plattform, wird davon ausgegangen, dass der Entwickler eines Mobilis-Service den Dienst selbst, zum Beispiel einen MobilisXHunt Spiel-Service, , als auch den dazugehörigen Dienstkonsumenten entwickelt. Der Dienstkonsument kann dabei ein Client, zum Beispiel ein MobilisXHunt Spiel-Client oder ein anderer Mobilis-Service, zum Beispiel ein Context-Service, sein. Zwischen einem Dienstanbieter (MobilisXHunt Spiel-Service) und einem Dienstkonsument (MobilisXHunt Spiel-Client) existiert jedoch kein expliziter Dienstvertrag, welcher das Kommunikationsverhalten beider Parteien beschreibt.

In der Zielarchitektur soll es nun möglich sein, dass der Dienstanbieter einen neuen Mobilis-Service oder einen bereits existierenden Mobilis-Service in einer neuen Version installiert. Eine Beschreibung des Kommunikationsverhaltens des neuen Dienstes soll dabei in einer technologieunabhängigen Beschreibungssprache definiert werden. Ein Dienstkonsument soll damit in der Lage sein, diese Beschreibung auszuwerten und daraufhin mit dem Dienst kommunizieren zu können (siehe Abbildung 4.2).

Beispielsweise könnte ein Entwickler A einen MobilisXHunt Spiel-Service erstellen, dessen Kommunikationsverhalten in einer Beschreibung definiert ist. Ein Entwickler B ist mit Hilfe der Beschreibung in der Lage einen MobilisXHunt Spiel-Client für diverse Plattformen, wie zum Beispiel Android<sup>1</sup> oder Windows Mobile Phone<sup>2</sup> zu erstellen. Dies kann, im Gegensatz zur ursprünglichen Mobilis-Plattform, ohne Zugriff auf den Quellcode geschehen, da alle relevanten Informationen in der Beschreibung definiert sind.

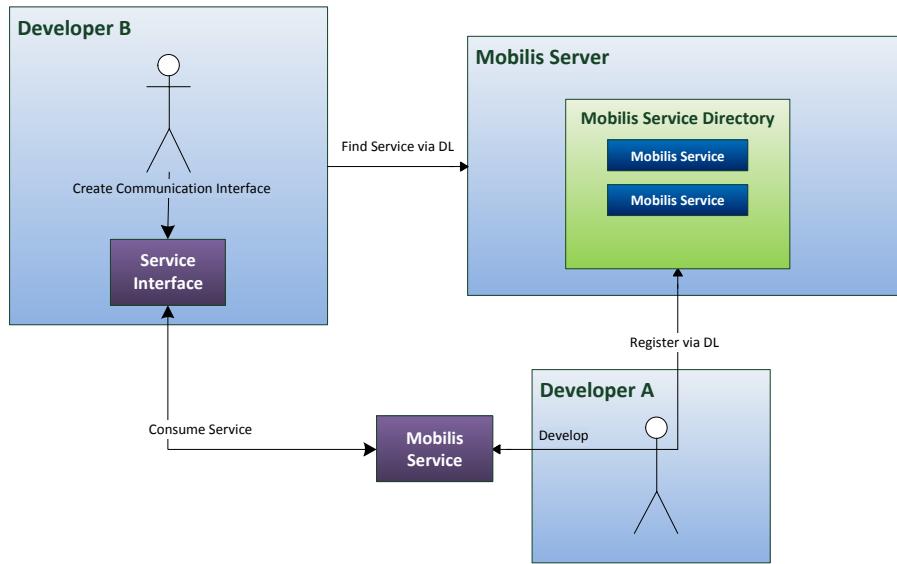
Für die dynamische Integration neuer Dienste in den Mobilis-Server zur Laufzeit werden im Folgenden zwei Architekturen vorgestellt, sowie deren Vor- und Nachteile bezüglich der Umsetzung in dieser Arbeit diskutiert.

Der erste Ansatz beschäftigt sich damit, neue Mobilis-Services als externe Dienste zu erstellen, sodass der Mobilis-Server mithilfe des XMPP-Protokolls mit diesen

---

<sup>1</sup> <http://www.android.com/>

<sup>2</sup> <http://www.microsoft.com/windowsphone/en-us/default.aspx>



**Abbildung 4.2:** Schematische Darstellung der Architektur

kommunizieren kann. Im zweiten Konzept wird der Ansatz betrachtet, einen neuen Mobilis-Service als Plugin, beziehungsweise Modul zum Mobilis-Server hinzuzufügen und wieder zu entfernen.

#### 4.1.2 Ansatz 1: Externe Mobilis-Service-Komponenten

Jeder Mobilis-Service ist in der Lage seine eigene [XMPP](#)-Verbindung aufzubauen, also eine vom Mobilis-Server verschiedene *JID* im [XMPP](#)-Netzwerk zu verwenden. Somit ist ein Mobilis-Service theoretisch in der Lage autark vom restlichen Mobilis-Server lauffähig zu sein (siehe Abbildung 4.3).

Der Mobilis-Server soll in diesem Fall lediglich als Verzeichnisdienst für alle Mobilis-Services dienen, welche sich bei diesem registrieren können. Außerdem stünde der Coordinator-Service weiter zur Verfügung, sodass bereits gestartete Mobilis-Services erfragt, sowie neue Mobilis-Services gestartet werden können.

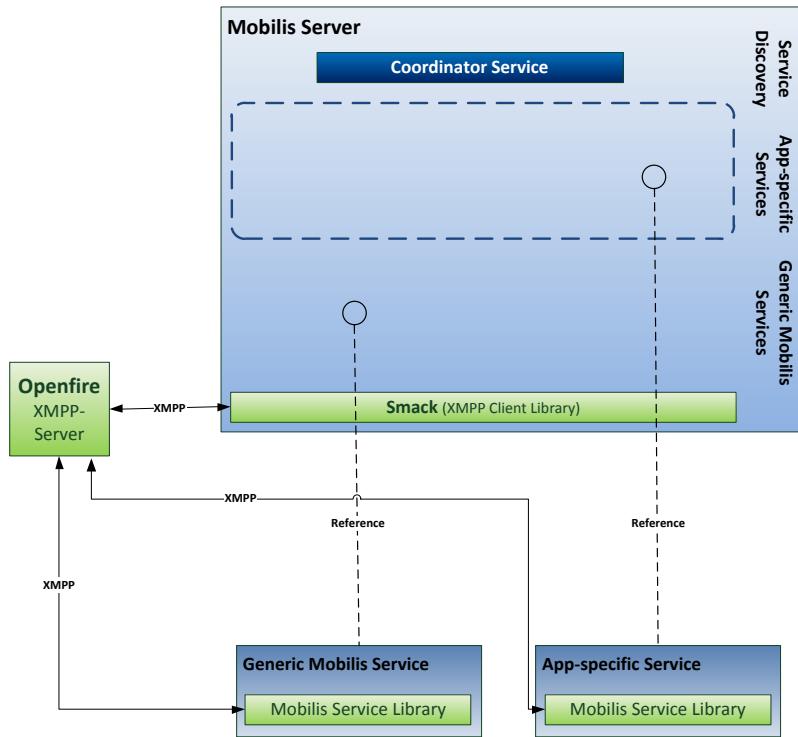
Möchte ein Teilnehmer zum Beispiel ein neues MobilisXHunt Spiel erstellen, so wird dieses als neuer *App-specific Service* umgesetzt. Der Teilnehmer fordert den Coordinator-Service auf, eine neue Instanz eines MobilisXHunt-Service zu starten. Wurde diese Instanz ohne Fehler gestartet, so bekommt der Teilnehmer die *JID* der neuen Service-Instanz zurückgeliefert. Die neue Instanz des MobilisXHunt-Service wird zugleich im Coordinator-Service registriert.

Abbildung 4.3 zeigt, wie eine Auslagerung eines Mobilis-Service realisiert werden könnte. Dabei werden die Grundfunktionalitäten, die ein Service benötigt, in einer Bibliothek zusammengefasst. Diese soll es dem Service ermöglichen im [XMPP](#)-Netzwerk, beziehungsweise mit dem Mobilis-Server, zu kommunizieren.

Der Mobilis-Server besitzt lediglich eine Referenz auf die laufenden Mobilis-Services

## 4 Konzeption

und kann diese in ihrem Lebenszyklus steuern. Die Kommunikation zwischen dem MobilisXHunt Spiel-Client und der neuen MobilisXHunt Spiel-Service-Instanz kann somit unverändert beibehalten werden, da es für den Client unerheblich ist, wo sich die neue Service-Instanz physisch befindet.



**Abbildung 4.3:** Architektur verteilter Mobilis-Services

### Life cycle externer Mobilis-Services

Der Life cycle eines Mobilis-Service könnte dabei wie in Abbildung 4.4 strukturiert werden.

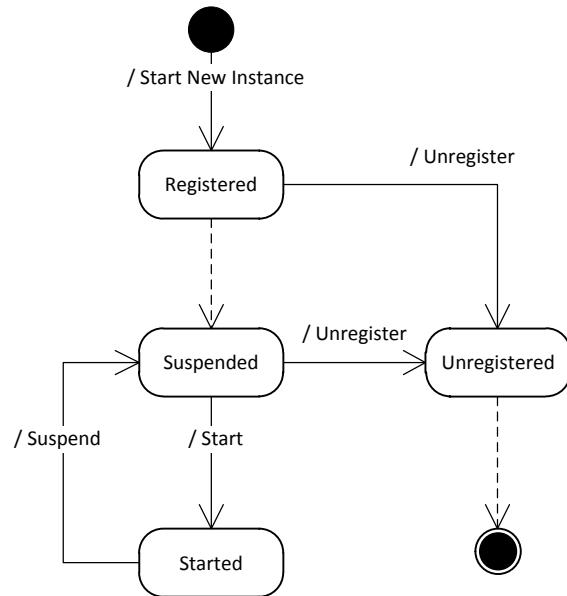
**Registered:** Der Service ist registriert und ist damit für andere [XMPP](#)-Clients erreichbar. Die eigentliche Servicelogik soll hierbei noch nicht gestartet werden (siehe **Suspended**)

**Bound/Started:** Der Service wird von einem Client gestartet und genutzt

**Suspended:** Die Servicelogik ist nicht aktiv, sondern wartet auf ein **Bind/Start** oder **Unregister**. In diesem Zustand sollen möglichst wenig Ressourcen verbraucht werden, indem nur die *Mobilis Service Library* aktiv ist und darauf wartet, dass der Service gestartet oder komplett heruntergefahren wird.

**Unregistered:** Der Service wurde aus dem Mobilis-Server-Register entfernt

**Stopped:** Ein Kompletter Shutdown des Mobilis-Service, inklusive der *Mobilis Service Library*. Dieser Service ist nun nicht mehr erreichbar.

**Abbildung 4.4:** Life cycle externer Mobilis-Services

### Vorteile externer Dienste

Ein bedeutender Vorteil liegt darin, dass Ressourcen, wie Betriebsmittel und lokale Datenhaltung, nicht mehr vom Mobilis-Server verwaltet werden müssen. Dieser kann also sehr kompakt ausfallen, ohne auf die Ressourcenanforderungen der Mobilis-Services achten zu müssen. Lediglich der Coordinator-Service muss vorhanden sein, sodass sich die externen Mobilis-Services registrieren können.

Der Mobilis-Server gewinnt zudem mehr an Stabilität, da jeder Mobilis-Service in einer separaten Umgebung läuft und den Mobilis-Server nicht in seinem Laufzeitverhalten beeinflusst. Sollte der MobilisXHunt Spiel-Service zum Beispiel einen Fehler produzieren und in einer Endlosschleife landen, in welcher er alle Betriebsmittel beansprucht, hat dies keinen Einfluss auf den Mobilis-Server. Der MobilisXHunt Spiel-Service, sowie die darunterliegende Plattform kann neu gestartet werden, ohne dass der Mobilis-Server davon beeinflusst wird.

Zuletzt ist eine weitaus bessere Flexibilität durch eine einfache Austauschbarkeit der Mobilis Services gegeben. Zum Beispiel kann bei Ausfall des MobilisXHunt Spiel-Service ein Fallback-Service, welcher eine genaue Kopie des Anderen ist, den ausgefallenen MobilisXHunt Spiel-Service ersetzen.

### Nachteile externer Dienste

Ein eklatanter Nachteil liegt in der komplexen Verwaltung dieser externen Mobilis-Services. Das Ausfallrisiko wird erhöht, wenn Mobilis-Services verkettet werden, also eine sogenannte Orchestrierung stattfindet. Die erhöhte Gefahr, dass ein "Single-point-of-failure" auftreten kann, muss kompensiert werden, indem zum Beispiel gewährleistet wird, dass Backup-Services zur Verfügung gestellt werden.

## 4 Konzeption

Die Verwaltung der eigenen Ressourcen kann in einem komplexen Geflecht aus Mobilis-Services zu redundanten und asynchronen Daten führen. Diese müssten zumindest aufwendig synchron gehalten werden. Ein Beispiel ist die persistente Speicherung der MobilisXHunt Karteninformationen, die in einer Datenbank gehalten werden. Existieren mehrere MobilisXHunt-Services, von denen jeder seine eigene Datenbank hat, so muss eine Änderung in der Struktur der Informationen in jeder Datenbank einzeln durchgeführt werden. Dasselbe betrifft die XMPP-Beans, die verwendet werden. Diese können unter Umständen mehrfach gespeichert sein, obwohl sie die selben Informationen anbieten.

### 4.1.3 Ansatz 2: Mobilis-Service Module

Der zweite Ansatz Mobilis-Services dynamisch zur Laufzeit in die Umgebung zu integrieren und zu binden, soll sich an OSGi (siehe Grundlagen 2.2.3) orientieren. Hierbei werden einzelne Komponenten in Module verpackt und zur Laufzeit in das System integriert oder wieder entfernt.

Zunächst wird betrachtet, ob OSGi als Laufzeitumgebung verwendet und der Mobilis-Server darauf portiert werden kann. Im Anschluss wird ein Konzept vorgestellt, welches, anstatt OSGi zu verwenden, eine eigene Implementierung einer Laufzeitumgebung verwendet.

#### Verwendung von OSGi als Laufzeitumgebung

OSGi bietet den Vorteil, dass neben den Bundles auch Dienste mit in die Plattform integriert werden können. Diese sollen eine globale Schnittstelle darstellen, welche auch von externen Applikationen verwendet werden kann.

Die normalen, lokalen Dienste, die von OSGi zur Verfügung gestellt werden, beschränken sich lediglich darauf, dass die Kommunikation von Dienstanbieter und Dienstnutzer in der selben Umgebung stattfindet. Nach [4] ist ein Dienst ein Objekt, dass ein Bundle registriert, sodass andere Bundles dieses in derselben virtuellen Maschine aufrufen können. Diese Dienste sind jedoch ungeeignet, um mit Anwendungen in verteilten Systemen zu kommunizieren.

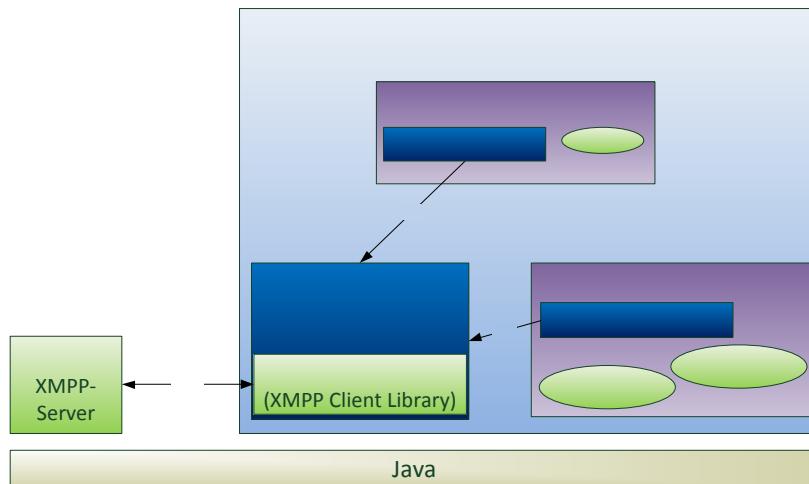
Eine neue Erweiterung sind hingegen die Remote Services. Diese sollen den Umgang mit verteilten Systemen vereinfachen und für typische Problemstellungen Lösungen anbieten. Standardmäßig sind jedoch nur Lösungen für SOAP und RPC integriert. Um XMPP als Protokoll in einem Remote Service zu verwenden, müsste eine Portierung der gesamten XMPP-Umgebung durchgeführt werden.

Eine weitaus einfachere Methode bestünde in der Umwandlung von Mobilis-Services als OSGi-Bundles. Da ein Bundle aber nicht automatisch einem Mobilis-Service gleichgesetzt werden soll, müssen diese Bundles von den anderen abgegrenzt werden oder an einer zentralen Stelle, einem Directory, vermerkt werden. Abbildung 4.5 zeigt, wie so eine Konfiguration aussehen soll.

Da es in der OSGi-Plattform nicht möglich ist, mehrere Instanzen eines Bundles zu

starten, aber in der Mobilis-Plattform ein Mobilis-Service mehrere Instanzen starten kann, stellt dies eine sehr große Einschränkung dar. Die Nutzung von ClassLoadern innerhalb eines Bundles führte zu keinem Erfolg und widerspricht zudem dem ganzen OSGi-Konzept.

Des Weiteren entsteht ein erhöhter Verwaltungsaufwand der Dienste, da zum Einen die Bundles im OSGi-Framework verwaltet werden, für die Nutzung der Mobilis-Services aber noch eine [XMPP](#)-basierte Verwaltung notwendig ist. Jeder Mobilis-Service ist nur über seine [JID](#) ansprechbar, was zudem die lokale Kommunikation überflüssig macht und viele Features von OSGi wären somit ungenutzt.



**Abbildung 4.5:** Mobilis OSGi Übersicht

### Verwendung einer eigenen Laufzeitumgebung

Angelehnt an die OSGi-Plattform, wie bereits in Kapitel 2.2.3 vorgestellt, soll der Mobilis-Server durch Module erweiterbar sein, ohne das Hauptsystem oder andere Dienste in deren Ausführung zu beeinflussen. Der Mobilis-Server soll somit in einer Minimalkonfiguration ausgeliefert werden, was die Konfiguration der anderen Dienste in der initialen Startphase überflüssig macht. Der Mobilis-Server besteht neben den GUI-Elementen aus einem Coordinator Service, der wie bisher für das Discovery und das Erstellen neuer Service-Instanzen zuständig ist.

Die Ausgangsversion des Mobilis-Server bringt bereits eine einfache Laufzeitumgebung für die Mobilis-Services mit. App-specific-Services werden jedoch nur vom Coordinator-Service verwaltet und sind im eigentlichen Mobilis-Server nicht bekannt. Dennoch ist die aktuelle Architektur sehr gut geeignet, um Mobilis-Services auch dynamisch von außen in das System zu integrieren, da dies intern schon in einer sehr einfachen Form getan wird.

Weiterhin können bereits bestehende Persistenz- und Kommunikationskomponenten ohne große Änderungen mit übernommen werden. Der Mobilis-Server behält somit weiter die Kontrolle über die genutzten Ressourcen und über alle laufenden Mobilis-

## 4 Konzeption

Services.

Der Mobilis-Server muss dahingehend weiterentwickelt werden, sodass neue Dienste zum Mobilis-Server hochgeladen werden können und diese Dienste de-/ installiert, sowie de-/ registriert werden. Weiterhin müssen diese Funktionen über das XMPP-Protokoll als Features angeboten werden, sodass diese auch von außen nutzbar sind.

Ein Nachteil den aktuellen Mobilis-Server so zu erweitern, dass Mobilis-Services dynamisch integriert werden können, stellt die Notwendigkeit dar, dass die Mobilis-Services nur auf dem Mobilis-Server lauffähig sind. Ohne diese Plattform ist es dann nicht möglich neue Dienste zu testen oder auf entfernten Plattformen zu hosten. Dies kann jedoch kompensiert werden, indem mehrere Mobilis-Server eingesetzt werden, die jeweils miteinander kommunizieren können.

### 4.1.4 Diskussion

Das Konzept, Mobilis-Services als externe Dienste umzusetzen, ist ein sehr interessanter Ansatz, bringt jedoch einen erheblichen Mehraufwand, wie Replizierung von Ressourcen und die Realisierung von Abhängigkeiten von verteilten Diensten, mit sich und eignet sich somit nicht als Grundlage für diese Arbeit.

OSGi ist ein weit verbreiteter und beliebter Ansatz, Anwendungen in Module zu separieren. OSGi ist jedoch primär für lokale Anwendungen und nicht für verteilte Servicearchitekturen geeignet. Des Weiteren würde der Einsatz von OSGi suggerieren, dass die Mobilis-Plattform eine komplette Unterstützung der OSGi-Spezifikation anbietet. Dies hätte einen ständigen Integrations- und Adoptionsprozess zur Folge. Aber auch die Umsetzung des XMPP-Protokolls auf Basis von OSGi und die damit verbundene XMPP-basierte Verwaltung der einzelnen Mobilis-Services stellt eine nicht unbedeutende Hürde in der Umsetzung dar. Dieser Umstand ergibt sich daraus, dass der gesamte Mobilis-Server und die einzelnen Mobilis-Services in OSGi-Bundles portiert werden müssten.

Durch die Verwendung von Reflection in Java [2] und die bereits einfache Unterstützung dieser in der Ausgangsversion des Mobilis-Servers, liegt die Umsetzung einer eigenen Laufzeitumgebung nahe. Der Mobilis-Server muss lediglich um einige Funktionen erweitert werden, die eine dynamische Integration neuer Mobilis-Services ermöglicht, sowie deren Life cycle Management. Diese Lösung benötigt des Weiteren keine neuen Technologien oder Frameworks, wie zum Beispiel OSGi, in die der Entwickler eingeführt werden müsste.

## 4.2 Dynamische Service-Plattform

Die Verwendung einer eigenen Laufzeitumgebung zur Ausführung der Mobilis-Services erfordert eine Umstrukturierung der Mobilis-Server-Architektur. Der Mobilis-Server muss dafür in der Lage sein, neue Mobilis-Services entgegen zu nehmen, diese zu installieren, zu konfigurieren und deren Life cycle zu verwalten.

In Abschnitt 4.2.1 wird dafür eine neue Architektur vorgestellt, die diese Funktionen ermöglicht. Einen detaillierten Blick auf die einzelnen Komponenten und deren Funktionen werden in den darauffolgenden Abschnitten gegeben. Zuletzt wird in Abschnitt 4.2.3 vorgestellt, wie der Life cycle eines neuen Mobilis-Service aussieht und wie dieser beeinflusst werden kann.

#### 4.2.1 Architektur

In Abbildung 4.6 ist zu sehen, aus welchen Komponenten der neue Mobilis-Server besteht. Die Mobilis-Services werden hierbei in die beiden Kategorien *Dynamic Services* und *System Services* eingeteilt.

**Dynamic Services** sind diejenigen Mobilis-Services, welche während der Laufzeit zum Mobilis-Server hinzugefügt und deployed werden können. Diese Dienste sollen die **System Services** für erweiterte Anpassungen verwenden.

**System Services** sind für die Anpassung von *Dynamic Services* zuständig. Gegenüber den Dynamic Services bieten diese Dienste eine erweiterte Integration- und Zugriffsmöglichkeit in die Verwaltungsstrukturen des Mobilis-Servers.

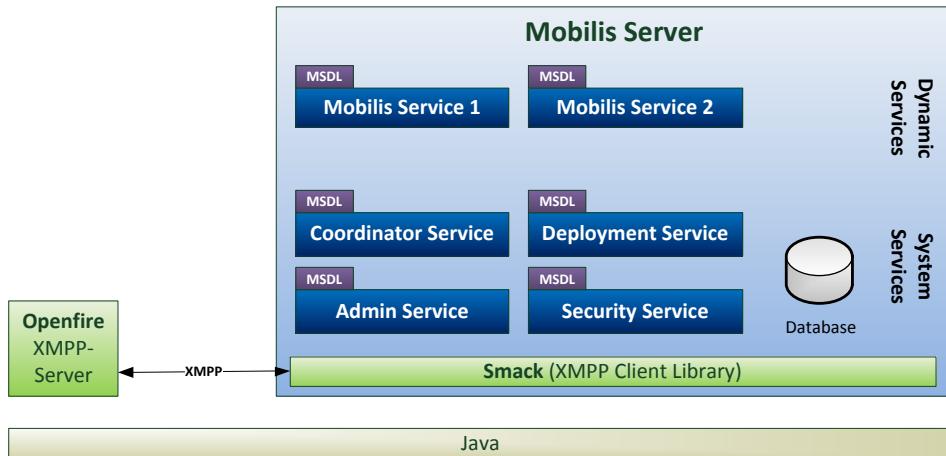


Abbildung 4.6: Übersicht Mobilis PlugIn Plattform

#### System Services

Die *System Services* sind, wie die *Dynamic Services*, ganz normale Mobilis-Services, jedoch bieten diese eine spezielle Funktion an, die einen tieferen Eingriff in den Mobilis-Server notwendig macht. Aus diesem Grund müssen diese Mobilis-Services genau an den Mobilis-Server angepasst werden.

Für eine einfache Umsetzung des dynamischen Deployments im Mobilis-Server sind die folgenden *System Services* notwendig:

**Coordinator Service** Dieser Dienst ist wie gehabt für das Discovery und das Erstellen neuer Service-Instanzen nach [30] verantwortlich. Die App-specific-Services

## 4 Konzeption

sind jedoch nicht mehr als solche vorhanden. Die Verwaltung von mehreren Service-Instanzen wurde in den Mobilis-Container verlagert. Das Discovery wurde um einige Funktionen, wie das Suchen nach Service-Versionen, erweitert. Dabei kann nach einer bestimmten Version, oder nach einem Bereich von Versionen, gesucht werden. Hinzugekommen ist die Möglichkeit eine bereits laufende Service-Instanz von außen zu stoppen.

**Deployment Service** Um Mobilis-Services in gepackter Form hochzuladen und auf dem Mobilis-Server abzulegen ist der *Deployment Service* zuständig. Dabei können neue Dienste entweder via FileStream hochgeladen oder später auch via URL referenziert werden, sodass sich der *Deployment Service* diesen herunterladen kann. Nach einer erfolgreichen Übertragung des Service-Pakets wird der Nutzer darüber informiert.

**Admin Service** Der Life cycle eines Mobilis-Service kann durch den *Admin Service* beeinflusst werden. Mit diesem ist es möglich bereits hochgeladene Mobilis-Service-Pakete zu de-/ installieren, de-/ registrieren, aktualisieren und zu konfigurieren. Wichtig ist hierbei, dass diese Aktionen nur durchgeführt werden können, wenn sich der Mobilis-Container im richtigen Zustand befindet und keine Fehler bei der Durchführung auftreten.

**Security Service** Zur Einschränkung von Zugriffen auf bestimmte Mobilis-Services, dient der *Security Service*. In der Grundkonfiguration kann jeder Dienst von jedem Nutzer genutzt werden. Die Einschränkung erfolgt durch die Zuweisung von Benutzern zu einem Mobilis-Service. Nutzer und später auch Nutzergruppen können angelegt, gelöscht und modifiziert werden. Alle Informationen werden für eine persistente Speicherung in der lokalen Datenbank abgelegt. Ein Mobilis-Service kann somit zu jeder Zeit den Security Service anfragen, ob ein bestimmter Nutzer berechtigt ist diesen Dienst zu verwenden.

Um die Verwendung der *System Services* verständlicher zu machen, sind in Anhang A Beispieldokumente aufgeführt. Der grobe Verlauf ist zudem in Abbildung 4.7 dargestellt. Hier wird gezeigt, wie die Kommunikation zwischen Dienstprovider und den einzelnen *System Services* erfolgen sollte.

Um einen neuen Mobilis-Service zu integrieren ist es wichtig, dass einige Schritte in einer vorgegebenen Reihenfolge durchgeführt werden. Der Deployment-Service muss als erstes darüber informiert werden, dass der Dienstentwickler einen neuen Mobilis-Service hochladen möchte. Wurde dies vom Deployment-Service bestätigt, kann der File-Transfer gestartet werden. War die Übertragung erfolgreich, so wird dies wiederum vom Deployment-Service bestätigt. Erst jetzt ist es möglich den neuen Mobilis-Service auf dem Mobilis-Server zu installieren. Nach der Installation kann der Mobilis-Service jederzeit konfiguriert und die Autorisierungseinstellungen vorgenommen werden. Die Registrierung kann erst durchgeführt werden, wenn die Konfiguration erfolgreich war. Zuletzt ist es möglich neue Service-Instanzen zu starten, wenn der Mobilis-Service registriert wurde.

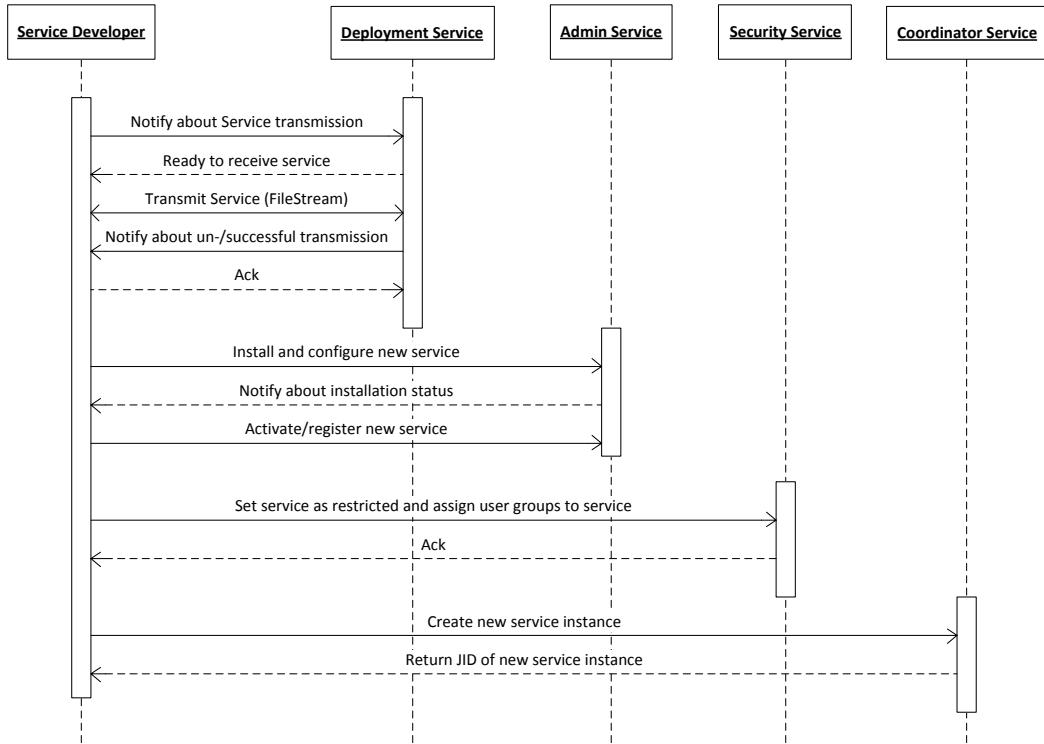


Abbildung 4.7: Mobilis Plugin Erstellungssequenzen

#### 4.2.2 Mobilis-Service-Container

Für die dynamische Verwendung der Mobilis-Services muss eine neue Komponente in die Mobilis-Plattform eingeführt werden. Diese neue Komponente, der Mobilis-Service-Container (kurz Mobilis-Container), nimmt die zentrale Stelle des Mobilis-Service ein und verwaltet einerseits dessen Service-Instanzen und wird andererseits vom *MobilisManager* verwaltet. In Abbildung 4.8 ist das Verhältnis des Mobilis-Containers zu den bereits bestehenden Systemkomponenten dargestellt.

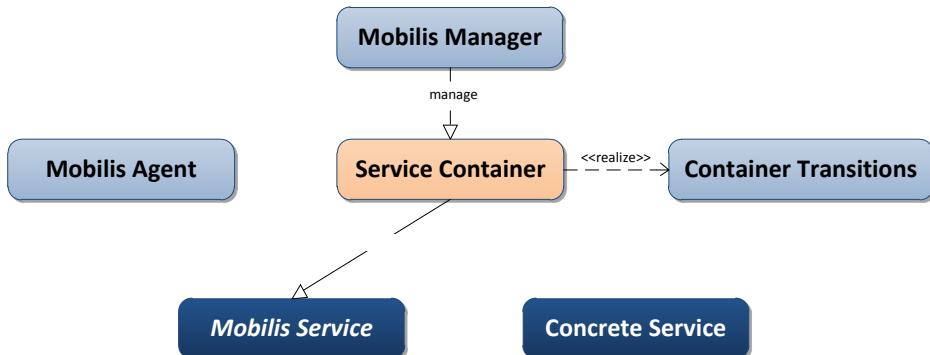
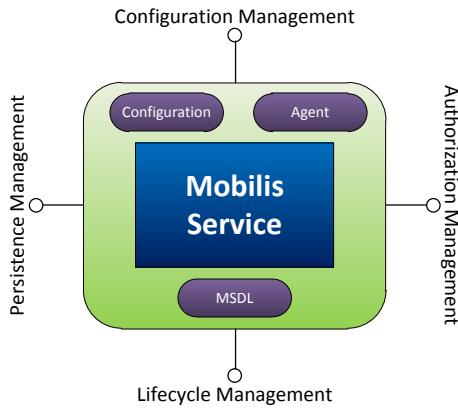


Abbildung 4.8: Klassenschema Mobilis-Container und Mobilis-Service

## 4 Konzeption

Der Mobilis-Container verwaltet, neben den eigentlichen Service-Instanzen, die Dienstbeschreibung, die [XMPP](#)-Einstellungen (die sogenannten Agent-Einstellungen) und weitere beliebig anpassbare Einstellungen. Der Aufbau eines Mobilis-Containers ist in Abbildung 4.9 dargestellt.



**Abbildung 4.9:** Mobilis-Service-Container

Jeder Mobilis-Container wird anhand des Namespace und der Version des zu ihm gehörendem Mobilis-Service identifiziert. Dies ist notwendig, um einen Mobilis-Container innerhalb des Mobilis-Server anzusprechen, beziehungsweise ausfindig zu machen. Des Weiteren wird die [Mobilis Service Description Language \(MSDL\)](#) lokal vorgehalten, sodass diese entweder komplett zurückgegeben werden kann oder vereinzelte Informationen, wie die Abhängigkeiten zu anderen Mobilis-Services, ausgelesen werden können.

Der Mobilis-Container ist zudem in der Lage bestimmte Schnittstellen dem Mobilis-Server anzubieten oder daraus zu verwenden. Außerdem kann der Mobilis-Server den Lebenszyklus eines Mobilis-Container beeinflussen und Konfigurationseinstellungen ändern. Der Mobilis-Container selber ist in der Lage auf Autorisierungsinformationen zuzugreifen, um zu prüfen, ob ein Nutzer berechtigt ist den verwalteten Mobilis-Service zu verwenden oder nicht. Dabei gilt diese Berechtigung für alle Service-Instanzen des Mobilis-Service. Für die Verwendung von Persistenzfunktionen des Mobilis-Server wurde eine weitere Schnittstelle geschaffen. Diese dient den Service-Instanzen zur Verwaltung der eigenen Daten.

Durch die Verwaltung der Mobilis-Container im Mobilis-Server ist weiterhin gegeben, dass der *Coordinator Service* alle aktiven Mobilis-Services auffinden und auslesen kann. Dies ist wichtig, da die Discovery-Schnittstelle somit grundlegend beibehalten werden kann und nur um einige Funktionen erweitert wird, wie das Suchen nach bestimmten Serviceversionen.

Der Mobilis-Container verwendet für das Instanziieren der Mobilis-Service-Instanzen ein Klassentemplate in Kombination mit einem speziellen Klassenlader. So ist es möglich, recht einfach neue Service-Instanzen zu erzeugen, ohne dass das gesamte Service-Paket erst entpackt werden muss. Auch können Mobilis-Services somit ein-

fach ersetzt werden, da sich die Klassen gebündelt in dem Paket befinden und nur bei der Installation wenige Daten lokal gespeichert werden, um einen schnelleren Zugriff darauf zu gewähren.

Zuletzt ist jeder Mobilis-Container in der Lage Events auszulösen, wenn eine Zustandsänderung stattfindet. Diese Events werden im Mobilis-Server gefeuert, sodass jeder Mobilis-Container benachrichtigt wird. Diese sind dafür notwendig, dass die Mobilis-Container registrieren, wenn zum Beispiel ein Mobilis-Service deregistriert oder deinstalliert wird. Ist ein Mobilis-Service A von einem Mobilis-Service B abhängig und der Mobilis-Service B wird deregistriert, wird unter anderem der Mobilis-Service A informiert. Daraufhin wird der Mobilis-Service A deregistriert. Das Feuern der Events ist dabei an den Life cycle des Mobilis-Container und dessen Mobilis-Service-Instanzen gebunden.

### 4.2.3 Mobilis-Service-Lifecycle

In der Ausgangsversion des Mobilis-Servers besitzen die Mobilis-Service-Instanzen bereits einen einfachen Lebenszyklus. Mobilis-Service-Instanzen können gestartet und gestoppt werden. Während eine Instanz innerhalb des Mobilis-Servers gestartet und gestoppt werden kann, ist es von außen lediglich möglich eine neue Instanz mithilfe des Coordinator Service zu starten.

Des Weiteren muss für die Einführung des Mobilis-Container ein neuer Lebenszyklus für diesen vorhanden sein, da somit das eigentliche dynamische Deployment eines Mobilis-Service durchgeführt wird.

#### Mobilis-Service-Instanz-Lifecycle

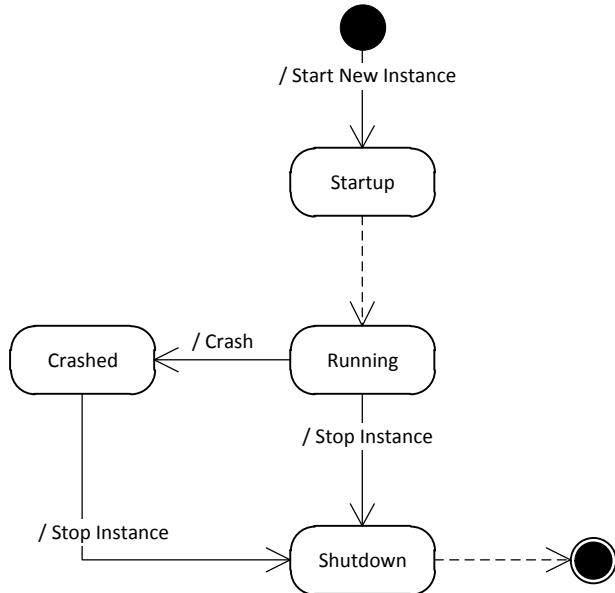
Für die Optimierung der Mobilis-Service-Instanz soll das explizite Stoppen von außen erlaubt werden, da dieses in der Ausgangsversion nur innerhalb des Mobilis-Server durchgeführt werden kann. Dies hat den Vorteil, dass Dienste, die nicht mehr vom Dienstnutzer benötigt werden keine unnötigen Ressourcen verbrauchen.

Des Weiteren wird die Mobilis-Service-Instanz um einen Crash-Zustand erweitert. Dieser ist dafür zuständig fehlerhafte und instabile Mobilis-Service-Instanzen sauber zu beenden und den Fehlervorgang in einer Log zu verzeichnen. Wie in Abbildung 4.10 zu sehen ist, bleibt der Lebenszyklus sonst unverändert, sodass er auch zu den älteren Mobilis-Services kompatibel ist, die einfach den Crash-Zustand ignorieren.

Eine Mobilis-Service-Instanz verfügt über folgende Zustände:

**Startup** Dieser Zustand bleibt unverändert. Hierbei wird von der Mobilis-Service-Instanz lediglich der SMACK-eigene *PacketListener* registriert, der alle notwendigen XMPP-Stanzas der Mobilis-Service-Instanz registriert. Zusätzlich kann der Entwickler die Start-Methode des Mobilis-Service durch eigene Startroutinen erweitern. Sollten während der Startphase Fehler auftreten, wird die Instanz sofort beendet, da grundlegende Probleme mit dem Mobilis-Service bestehen.

**Running** Der Zustand Running wurde formal eingeführt, um zu beschreiben, dass die



**Abbildung 4.10:** Life cycle Mobilis-Service

Mobilis-Service-Instanz erfolgreich gestartet ist und die benötigten Pakete registriert sind. Erst an dieser Stelle ist der Dienst für Dienstkonsumenten nutzbar.

**Shutdown** Der Shutdown-Zustand wurde, wie der Startup-Zustand unverändert übernommen. In diesem Zustand wird der SMACK-eigene *PacketListener* wieder entfernt. Der Entwickler hat auch hierbei wieder die Möglichkeit die Shutdown-Methode um eigene Routinen zu erweitern.

**Crashed** In den Crashed-Zustand gelangt eine Mobilis-Service-Instanz, wenn sich diese während der Laufzeit aufhält. Dafür werden primär Exceptions aufgefangen, die von dieser Instanz geworfen werden, um diese dann in den Crashed-Zustand zu überführen. Hier soll in erster Linie dem Entwickler die Möglichkeit gegeben werden eigene Aufräum-Routinen durchzuführen, sodass reservierte Betriebsmittel wieder freigegeben werden. Anschließend wird die Mobilis-Service-Instanz in den Shutdown-Zustand überführt, sodass diese sich normal beenden kann, was in einem Fehlerfall nicht vorkommen würde. Des Weiteren werden die Vorgänge detailliert in eine Log geschrieben, sodass eine Fehlersuche vereinfacht wird. In einer zweiten Ausbaustufe ist es vorgesehen, die abgestürzte Instanz wieder neu zu starten, insofern der Fehler automatisch reparabel war.

Die Übergänge der Zustände sind wie folgt definiert:

**Start New Instance** Eine neue Mobilis-Service-Instanz wird wie gewohnt über den *Coordinator Service* veranlasst zu starten. Indes werden einige Prüfroutinen gestartet, die zum Beispiel überprüfen, ob der Mobilis-Service nur einmal oder mehrfach gestartet werden kann. Die Eigenschaft `mode` ist dementsprechend entweder als `single` oder `multi` definiert.

**Stop Instance** Dieser Übergang führt die Mobilis-Service-Instanz in den Shutdown-Zustand und kann von jedem anderen Zustand erfolgen.

**Crash** Der Crashed-Übergang wird vom Mobilis-Server selber aufgerufen, sobald sich die Mobilis-Service-Instanz nicht mehr meldet, beziehungsweise eine Exception geworfen wird. Der Entwickler des Mobilis-Service kann diesen Übergang auch manuell herbeiführen, wenn von der Mobilis-Service-Instanz nicht legitime Werte gemessen werden.

### Mobilis-Service-Container-Lifecycle

Durch die Verwaltung des Mobilis-Service und dessen Service-Instanzen durch einen Mobilis-Service-Container, wird für diesen ebenfalls ein Lebenszyklus eingeführt (siehe Abbildung 4.11). Hiermit soll die dynamische Integration eines Mobilis-Service in den Mobilis-Server ermöglicht werden. Die Aufgabe, die einzelnen Service-Instanzen eines *AppSpecificService* zu verwalten, wurde vom *Coordinator Service* in den Mobilis-Container verlagert, sodass der *Coordinator Service* leichter austauschbar ist. Die *System Services* sind dabei in der Lage auf das System zuzugreifen. Dies gilt auch für den *Coordinator Service*, welcher sich alle laufenden Mobilis-Container und damit registrierte Mobilis-Services beschaffen kann.

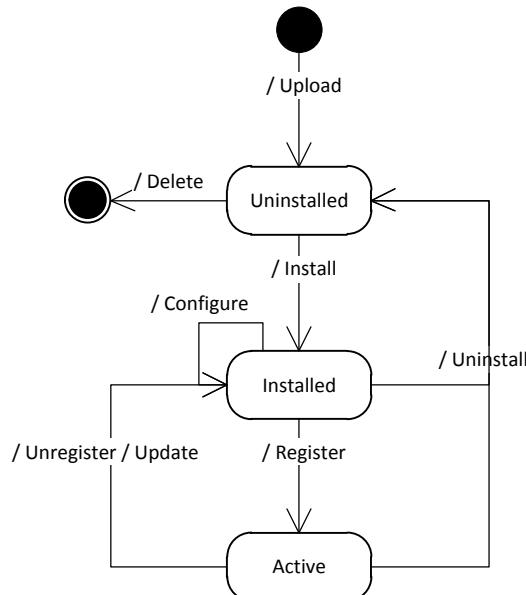


Abbildung 4.11: Lifecycle Mobilis Service Container

Die Zustände eines Mobilis-Container sind wie folgt definiert:

**Uninstalled** Der Mobilis-Service ist in gepackter Form auf dem Mobilis-Server vorhanden und kann installiert werden. In diesem Zustand existiert bereits ein Mobilis-Container, der auf den gepackten Mobilis-Service verweist. Wird das Archiv gelöscht, geht der Mobilis-Container in den Stopp-Zustand und wird aus dem System entfernt.

**Installed** Im Zustand *Installed* ist es möglich den Mobilis-Service zum Beispiel zu konfigurieren, das heißt die XMPP-spezifischen Verbindungseinstellungen zu täglichen, Autorisierungen vorzunehmen und beliebige weitere Einstellungen vorzunehmen. Der Mobilis-Service kann in diesem Zustand wieder deinstalliert werden und damit alle Einstellungen gelöscht werden. In diesem Zustand ist es noch nicht möglich mit dem Mobilis-Server zu kommunizieren, somit können noch keine Service-Instanzen erstellt werden. Dafür muss dieser erst registriert werden. Befand sich der Mobilis-Container bereits im Zustand Active und wurde Service-Instanzen gestartet, laufen diese in diesem Zustand weiter. Dies ist möglich, da jede Service-Instanz ihre eigenen Verbindungseinstellungen verwaltet und somit erst die neuen Service-Instanzen die veränderten Einstellungen übernehmen.

**Active** Der Zustand Active registriert den Mobilis-Container im Mobilis-Server, sodass dieser für die Instanziierung neuer Mobilis-Service-Instanzen genutzt werden kann. Dieser Zustand entspricht dem normalen Zustand eines Mobilis-Service in der Ausgangsversion der Mobilis-Plattform. Um zu verhindern, dass neue Instanzen gestartet werden, muss ein *unregister* durchgeführt. Eine Aktualisierung des Mobilis-Containers erfolgt durch den Übergang *update*. Damit wird der Mobilis-Container wieder in den Zustand *Installed* überführt und das neue Mobilis-Service-Paket eingelesen.

Die Zustände eines Mobilis-Service-Container können durch folgende Übergänge erreicht werden:

**Upload** Der Mobilis-Service wird in gepackter Form zum Mobilis-Server hochgeladen. Es wird dafür ein Mobilis-Container angelegt und das Paket darin referenziert. Diese Mobilis-Container werden in einer separaten Liste aufbewahrt und als sogenannte *Pending Services* behandelt, die auf eine Installation warten.

**Delete** Dieser Übergang löscht das hochgeladene Mobilis-Service-Archiv wieder vollständig aus dem Mobilis-Server, sowie alle Verweise.

**Install** Im Install-Übergang wird zuerst die Dienstbeschreibung extrahiert. Aus dieser Beschreibung wird der Namespace des Mobilis-Service, sowie die Version ausgelesen und lokal gespeichert. In nächsten Schritt wird die Hauptklasse des Mobilis-Service ausgelesen und ein Klassen-Template erzeugt, das als Vorlage für die Mobilis-Service-Instanzen dient. Waren alle Schritte erfolgreich, so wird der Client darüber informiert, wenn nicht wird ein ERROR-IQ als Antwort mit einer Fehlerbeschreibung zurückgesendet.

**Uninstall** Ein Mobilis-Container kann im Zustand *Installed*, sowie *Active* direkt deinstalliert werden. Um dies zu bewerkstelligen, wird für den Übergang ein *unregister* und darauf ein *uninstall* durchgeführt. Wichtig ist, dass durch diesen Übergang alle laufenden Mobilis-Service-Instanzen beendet werden, egal ob diese gerade in Verwendung sind oder nicht. Alle getätigten Einstellungen werden hierbei verworfen. Dies betrifft den Namespace und Version des Mobilis-Service, die Dienstbeschreibung und die Template-Klasse für die Erstellung der Service-Instanzen.

**Register** Der Übergang *Register* verwendet den Namespace des Mobilis-Service und die Version um diesen im Mobilis-Server anzumelden. Der *Coordinator Service*

ist zum Beispiel erst dann in der Lage neue Service-Instanzen zu starten oder diesen Service über das Discovery zu finden. In diesem Vorgang werden zudem die Abhängigkeiten zu anderen Mobilis-Services geprüft. Fehlen Dienste, von denen dieser abhängig ist, so wird eine Fehlermeldung an den Client zurückgesandt und die Registrierung wird abgebrochen.

**Unregister** Der Übergang *Unregister* bewirkt genau das Gegenteil vom Übergang *Register*, sodass dieser Mobilis-Service nicht mehr über den *Coordinator Service* gefunden werden kann und keine neuen Service-Instanzen mehr erstellt werden können.

**Update** Die Ersetzung eines Mobilis-Service durch eine aktuellere Version wird durch den Übergang *Update* vollzogen. Dafür muss bereits eine neu Version des Mobilis-Service auf dem Mobilis-Server vorhanden sein. Der bereits laufende Mobilis-Service wird dafür deregistriert und anschließend neu installiert. Das heißt, dass Namespace und Version, sowie die Dienstbeschreibung ersetzt werden. Unverändert bleiben jedoch die Konfigurationen des Mobilis-Containers. Bereits laufende Instanzen der Vorgängerversion des Mobilis-Service laufen jedoch weiter, sodass diese weiterhin ganz normal angesprochen werden können. Neue Instanzen dieser Vorgängerversion können jedoch nicht mehr gestartet werden.

**Configure** Im Mobilis-Container können beliebige Einstellungen gespeichert werden. Für die Umsetzung dieser Einstellungen wird das in [2.3.2](#) vorgestellte **Data Forms (XEP-0004)** verwendet. Hier werden zum Beispiel alle **XMPP**-spezifischen Einstellungen des Mobilis-Service, die sogenannten Agent-Einstellungen, hinterlegt, die bei der Instantiierung, beziehungsweise der Registrierung verwendet werden.

Für die Verwendung des Mobilis-Container und dessen Lebenszyklus ist es notwendig, dass eine einheitliche Dienstbeschreibung verwendet wird. Diese wird benötigt, um den Namespace und die Version des Mobilis-Service auszulesen, welche als Identifikation des Mobilis-Containers im Mobilis-Server dient. Damit ist es erst möglich unterschiedliche Versionen eines Mobilis-Service laufen zu lassen und Abhängigkeiten zwischen diesen Diensten zu ermöglichen.

## 4.3 Beschreibung XMPP-basierter Dienste

In diesem Abschnitt soll als erstes in [4.3.1](#) analysiert werden, welche Elemente für eine Beschreibung erforderlich sind, um einen Mobilis-Service zu beschreiben. Anhand dieser Analyse wird das Ergebnis in [4.3.2](#) mit dem Schema einer **WSDL** verglichen und erörtert, welche Parallelen bestehen und ob **WSDL** als Dienstbeschreibung infrage käme einen Mobilis-Service zu beschreiben. Zuletzt wird die Beschreibungssprache und deren Elemente anhand eines Beispiels in [4.3.3](#) im Detail erläutert.

### 4.3.1 Elemente einer Dienstbeschreibung für Mobilis-Services

Im Kapitel [2.1](#) wurde bereits erläutert, welche Komponenten eine **SOA** aufweisen muss. Abgesehen von der Dienstbeschreibung, beinhaltet die Mobilis-Plattform bereits alle weiteren Komponenten.

## 4 Konzeption

Eine Dienstbeschreibung in der Mobilis-Plattform ist implizit verfügbar und kann ausgelesen werden, insofern der Zugriff auf den Quellcode besteht. Diese besteht darin zu ermitteln, welche IQs ein Mobilis-Service verstehen kann. Um neue IQs zu verstehen, müssen diese vom Mobilis-Service registriert werden, wobei standardmäßig die XMPP-Beans verwendet werden (siehe Kapitel 2.4).

Das Ziel der Dienstbeschreibung ist es, die native Kommunikation für den Benutzer der Dienstschnittstelle zu vereinfachen und eine formale Beschreibung der Kommunikationspakete zu ermöglichen. In Abbildung 4.12 ist eine abstrakte Sicht auf einen Mobilis-Service und dessen Komponenten wie folgt dargestellt:

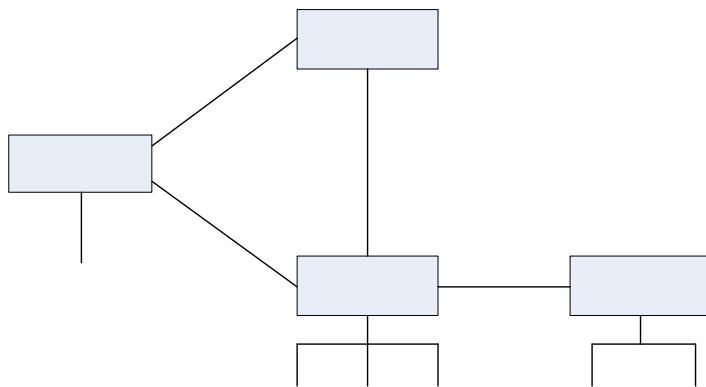


Abbildung 4.12: Elemente der Beschreibungssprache

**Service** Das Service-Element definiert einen Mobilis-Service. In diesem Teil soll festgelegt werden, welchen *Namespace* und welche *Version* der Mobilis-Service hat. Des Weiteren wird eine Liste von Diensten und deren Version vorgehalten, von dem dieser Mobilis-Service abhängig ist. Für die Kommunikation mit Dienstkonsumenten wird eine Menge an Operationen definiert, die der Mobilis-Service anbietet und eine Liste mit Endpunkten (*Endpoint*), durch welche ein Dienst angesprochen werden kann.

**Endpoint** Ein Endpunkt definiert, wie die *Operationen* auf ein bestimmtes Kommunikationsprotokoll umgesetzt werden. Somit kann ein Endpunkt über ein bestimmtes Kommunikationsprotokoll, zum Beispiel **XMP**, die Dienstschnittstelle nach außen hin anbieten. Diese Schnittstelle ist dabei eine Komposition aus verschiedenen *Operationen*.

**Operation** Das Operation-Element definiert den Ablauf und Zusammenhang der Nachrichten, die zum und vom Mobilis-Service gesendet werden. Hierbei sollen verschiedene Nachrichtenmuster wie *in-out*, *out-in*, *in-only* und *out-only*, aus der Sicht des Dienstkonsumenten, unterstützt werden. Zusätzlich können keine, eine oder mehrere Fehlernachrichten definiert werden, die vom Dienstkonsumenten oder vom Dienst gesendet werden. Jeder Operationsteil muss für die Verwendung von Parametern ein *Property*-Element referenzieren, sodass der Datentyp des Parameters bestimmt werden kann.

**Property** Ein *Property*-Element stellt die eigentlichen Daten bereit, die über eine *Operation* angeboten werden. Eine *Property* besteht entweder aus einem einfachen Datentypen, wie String oder Integer, oder aus einem komplexen Datentypen, wie zum Beispiel einem Datentypen Person, welcher aus einfachen Datentypen zusammengesetzt ist.

In der Beschreibung ist es notwendig, die Dienstschnittstelle auf einem abstrakten Niveau zu definieren, sodass diese unabhängig von einer Plattform und dem verwendeten Transportprotokoll ist. Da es möglich sein soll unterschiedliche Transportprotokolle, wie [XMPP](#) oder [BOSH](#) (siehe 2.3.2) zu verwenden, werden diese extra definiert. Zum Schluss muss noch beschrieben werden, wie die Daten der abstrahierten Dienstschnittstelle über ein bestimmtes Transportprotokoll übermittelt werden.

Zur Abgrenzung von anderen Dienstbeschreibungen, wie zum Beispiel der [WSDL](#), wird die Mobilis Dienstbeschreibung als [MSDL](#) bezeichnet.

### 4.3.2 Ein Vergleich zu WSDL

Eine [WSDL](#) beschreibt die Nutzung und den Aufbau von webbasierten Diensten, den sogenannten Webservices. Diese besteht aus einem abstrakten und einem konkreten Teil. Im abstrakten Teil werden die Datentypen und die Operationen, die ein Dienstkonsument verwenden kann unabhängig von der eigentlichen Implementierung beschrieben. Im konkreten Teil werden hingegen die technischen Informationen beschrieben. Dazu zählen das *Binding*, in welchem festgelegt wird, wie die im abstrakten Teil definierten Operationen auf ein festgelegtes Kommunikationsprotokoll angewendet werden. Der Service-Teil bildet die eigentliche Anlaufstelle in der [WSDL](#). Hier werden die Endpunkte festgelegt über die der Webservice angesprochen werden kann und welches *Binding* damit verwendet wird. Des Weiteren wird die Adresse angegeben über die der Webservice erreichbar ist.

Hauptsächlich besteht der Unterschied zwischen einem Webservice und einem Mobilis-Service in der Verwendung des Transportprotokolls, also dem *Binding*. Während Webservices über [HTTP](#) meist ein synchrones Kommunikationsverhalten aufweisen, ist mit [XMPP](#) eine asynchrone Kommunikation zwischen Dienstkonsument und Dienstanbieter möglich. Neben dem Kommunikationsverhalten und der Art, wie Pakete übertragen werden, unterscheidet sich die Adressierung beider Dienstarten voneinander. Die Adressierung eines [XMPP](#)-Dienstes wird dynamisch zur Verbindung durch die Resource festgelegt. Im speziellen Fall von Mobilis werden Mobilis-Services anhand eines Namespace und anhand der Versionsnummer eindeutig identifiziert.

Mit dem in Kapitel 2.3.2 vorgestellten [XEP SOAP Over XMPP \(XEP-0072\)](#) ist es jedoch möglich, [XMPP](#) in SOAP zu überführen. Die Probleme ergeben sich aber daraus, dass SOAP auf synchrone Kommunikation ausgelegt ist und in der Beschreibung ein recht großer Overhead entsteht, wie es im Grundlagenbeispiel gezeigt wurde. Jedes [XMPP](#)-Stanza muss noch einmal in einen SOAP-Body gekapselt und durch einen SOAP-Header erweitert werden. Das [XEP](#) ist zudem nur ein Draft und kann somit noch verändert werden. Es basiert außerdem noch auf dem [WSDL](#)-Standard 1.1. Seit der Definition 2005 ist bis jetzt auch noch keine nutzbare Umsetzung dieser Erweite-

## 4 Konzeption

rung in einem größeren Projekt erfolgt.

In der Gegenüberstellung ist somit zu sehen, dass [WSDL](#) weder nativ, noch durch die [XEP SOAP Over XMPP \(XEP-0072\)](#) verwendet werden kann. Dennoch soll die eigene Umsetzung einer Beschreibungssprache an [WSDL](#) angelehnt werden. Der abstrakte Teil ist komplett ohne Veränderungen wiederverwendbar, sowie die Struktur des konkreten Teils. Mit der Übernahme der Grobstruktur der [WSDL](#) ist es somit auch möglich die hier modifizierte Version der [WSDL](#) wieder zurück zu transformieren. Das hat zur Folge, dass Mobilis-Services mit nur wenigen Änderungen in der Beschreibungssprache auch als Webservices realisiert werden können.

### 4.3.3 Mobilis Service Description Language

Der Aufbau der [MSDL](#) ist an den des [WSDL](#)-Standards 2.0 angelehnt und soll hier an einer Beispiel-[MSDL](#) gezeigt werden. Eine grafische Darstellung des [MSDL](#)-Schemas ist in Anhang [B.2](#) zu finden, sowie die gesamte Beispiel-[MSDL](#) in Anhang [B.3](#).

Zum besseren Verständnis der einzelnen Elemente der [MSDL](#) soll ein Dienst *TreasureHunt* definiert werden. *TreasureHunt* ist ein einfaches *LBS* in dem mehrere Spieler Schätze auf einer Karte anhand ihrer geografischen Positionen finden können. Eine detailliertere Einführung für *TreasureHunt* wird in Kapitel [5.2](#) gegeben. Zur Übersicht sind die Schnittstellen-Operationen des Dienstes in Tabelle [4.1](#) definiert.

**Tabelle 4.1:** Operationen des Dienstes TreasureHunt

Operation	Parameter	Richtung
PickUpTreasure	in:treasureLocation out:treasureValue	Client → Service
GetLocation	in:playerLocation	Service → Client
TreasureCollected	out:playerName out:treasureLocation out:treasureValue	Service → Client (one-way)

Auf die Operationen *PickUpTreasure* und *GetLocation* ist es zudem möglich, dass ein Fehler auftreten kann, der mit berücksichtigt werden soll.

Die Grundlage einer [MSDL](#) ist das *description*-Element, das alle weiteren Elemente beinhaltet. In Abbildung [4.13](#) ist das Grobschema dieses Elementes mit seinen Unterelementen zu sehen. Die Bezeichnung und der Aufbau ist gleich dem der [WSDL](#) und wird in die folgenden Abschnitte eingeteilt:

**types** Dient der Definition der Datentypen und entspricht dem XML-Schema für die Definition von Datentypen

**interface** Für die abstrakte Definition der Operationen und deren Ablauf

**binding** Zur Umsetzung der Operationen aus dem *interface*-Abschnitt auf ein bestimmtes Kommunikationsprotokoll

**service** Definiert den eigentlichen Mobilis-Service

Um zwischen **MSDL**-Elementen und **XMPP**-Elementen zu unterscheiden, wurden die Namespaces `xmlns:msdl` und `xmlns:xmpp` als Attribute definiert.

```
<msdl:description
    targetNamespace=
    xmlns:msdl=
    xmlns:xmpp=
    xmlns:xs=
    xmlns:tns=
    >

    <msdl:types />
    <msdl:interface />
    <msdl:binding />
    <msdl:service />
</msdl:description>
```

**Abbildung 4.13:** Das Element `description` der TreasureHunt-MSDL

### MSDL-Interface

Das `interface`-Element entspricht derselben Definition, wie der des **WSDL**-Interface-Elementes. Hier werden die Operationen des Mobilis-Service in einer abstrakten Weise definiert und deren Kommunikationsablauf festgelegt. Für die Verwendung der Elemente gelten folgende Definitionen:

**operation** Stellt eine abstrakte Operation des Mobilis-Service dar

**attributes** Enthält das `pattern`-Attribut, das die Richtung dieser Operation wie in **WSDL** festlegt

**input** Eingangsparameter, die ein Client angeben muss

**output** Ausgabeparameter, gekapselt in einem Datenobjekt, die ein Client zurückgeliefert bekommt

**fault** (optional) Fehlermeldung die ein Client oder ein Mobilis-Service bekommt, wenn ein Request einen Fehler erzeugt hat

Der Ablauf einer `operation` ist in verschiedenen Varianten möglich. So werden die `pattern` in, out, in-out und out-in unterstützt. Jedes dieser Szenarien kann dabei keinen oder mehrere `fault`-Elemente definieren.

In Abbildung 4.14 ist die Umsetzung des Elements `interafce` auf das TreasureHunt-Beispiel dargestellt. Als Erstes sind die beiden Fehlerarten *PickUpFault* und *GetLocationFault* definiert, die von jedem `operation`-Element verwendet werden können, wie zum Beispiel in der Operation *PickUpTreasure*, die ein `outfault`-Element definiert. Die Kommunikationsrichtung Client → Service ist für die *PickUpTreasure*-Operation mit `http://www.w3.org/ns/wsdl/in-out` definiert, wie im Attribut `pattern` zu sehen ist. Diese Schreibweise wurde vom **WSDL**-Standard übernommen. Die Elemente `input` und `output` verweisen beide auf einen Datentyp, der im `types`-Element definiert ist und die Parameter für diese Operation definiert.

## 4 Konzeption

```
<msdl:interface name="THInterface">

    <msdl:fault name="PickUpFault"
        element="tns:PickUpTreasureFault" />

    <msdl:fault name="GetLocationFault" />

    <msdl:operation name="PickUpTreasure"
        pattern="http://www.w3.org/ns/wsdl/in-out">
        <msdl:input element="tns:PickUpTreasureRequest" />
        <msdl:output element="tns:PickUpTreasureResponse" />
        <msdl:outfault ref="tns:PickUpFault" />
    </msdl:operation>

    <msdl:operation name="GetLocation"
        pattern="http://www.w3.org/ns/wsdl/out-in">
        <msdl:input element="tns:GetLocationResponse" />
        <msdl:output element="tns:GetLocationRequest" />
        <msdl:infault ref="tns:GetLocationFault" />
    </msdl:operation>

    <msdl:operation name="TreasureCollected"
        pattern="http://www.w3.org/ns/wsdl/out-only">
        <msdl:output element="tns:TreasureCollected" />
    </msdl:operation>

</msdl:interface>
```

Abbildung 4.14: Element interface der TreasureHunt-MSDL

### MSDL-Binding

Im **binding**-Element wird das **interface** mit dem Kommunikationsprotokoll verbunden. Auch hier ist die Struktur wieder sehr ähnlich der [WSDL](#) gehalten. Folgende Elemente werden in diesem Abschnitt beschrieben:

**attributes** Allgemeine Attribute, die das Kommunikationsprotokoll definieren

**type** Namespace des zu verwendenden Kommunikationsprotokolls

**name** Name des **binding**

**interface** Name des zu portierenden **interface**-Element

**fault** (optional) Fehlermeldung, die ein Client oder ein Mobilis-Service bekommt, wenn ein Request einen Fehler erzeugt hat

**xmpp:errortype** Typ des Fehlers

**xmpp:errorcondition** Bedingung des Fehlers

**xmpp:errortext** (optional) Fehlertext

**operation** Liste der zu portierenden Operationen aus dem **interface**-Element

**ident** Namespace der Nachricht zur eindeutigen Identifizierung

**input** Eingangsparameter, die ein Client angeben muss

**output** Ausgabeparameter, gekapselt in einem Datenobjekt, die ein Client zurückgeliefert bekommt

**infault/outfault** (optional) Verweist auf einen vorher definiertes **fault**-Element

Die Elemente einer Operation besitzen zusätzlich noch ein Attribut `xmpp:type`-Element, um den Nachrichtentyp festzulegen (SET, GET, RESULT, CHAT). Das `fault`-Element kann zusätzlich durch einen Datentyp erweitert werden, sodass auch komplexere Fehlerantworten erstellt werden können.

Das `binding`-Element ist in dieser Arbeit für die Nutzung des **XMPP**-Protokolls angepasst worden, sodass eine einfache Umwandlung der Daten in **XMPP**-spezifische Beschreibungen ermöglicht werden. Dieses Element kann in einer **MSDL** mehrfach vorkommen, sodass unterschiedliche Endpunkte umgesetzt werden können.

In Abbildung 4.15 ist ein `binding`-Element des TreasureHunt-Beispiels dargestellt. Das `type` ist dabei auf “<http://mobilis.inf.tu-dresden.de/xmpp/>“ festgelegt und beschreibt das Mapping des `interface`-Elements auf reines **XMPP**. Die `fault`-Elemente, wie zum Beispiel das `PickUpFault`, verweisen auf das `interface`-Element und erweitern dieses um **XMPP**-spezifische Attribute, die den Fehler genauer beschreiben. Das `PickUpFault` ist hier beispielsweise auf ein **XMPP**-ErrorType `modify` festgelegt, dessen **XMPP**-ErrorCondition `not-acceptable` und einen optionalen Text “*This Treasure is not valid*“ mit angibt. Somit kann ein **XMPP**-Error einmal genau definiert werden und mehrmals in den `operation`-Elementen verwendet werden.

```

<msdl:binding name="THBinding" interface="tns:THInterface"
  type="http://mobilis.inf.tu-dresden.de/xmpp/">

  <msdl:fault ref="tns:PickUpFault" xmpp:errortype="modify"
    xmpp:errorcondition="not-acceptable"
    xmpp:errortext="This Treasure is not valid" />

  <msdl:fault ref="tns:GetLocationFault" xmpp:errortype="wait"
    xmpp:errorcondition="resource-constraint"
    xmpp:errortext="Waiting for Location" />

  <msdl:operation ref="tns:PickUpTreasure"
    xmpp:ident="treasurehunt:iq:pickuptreasure">
    <msdl:input xmpp:type="set" />
    <msdl:output xmpp:type="result" />
    <msdl:outfault ref="tns:PickUpFault" />
  </msdl:operation>

  <msdl:operation ref="tns:GetLocation"
    xmpp:ident="treasurehunt:iq:getlocation">
    <msdl:input xmpp:type="result" />
    <msdl:output xmpp:type="get" />
    <msdl:infault ref="tns:GetLocationFault" />
  </msdl:operation>

  <msdl:operation ref="tns:TreasureCollected"
    xmpp:ident="treasurehunt:iq:treasurecollected">
    <msdl:output xmpp:type="chat" />
  </msdl:operation>

</msdl:binding>
```

Abbildung 4.15: Element binding der TreasureHunt-MSDL

Die Operation `PickUpTreasure` verweist ebenfalls auf das `operation`-Element in `interface`. Für die Identifizierung des **XMPP**-Stanzas wird das Attribut `ident` hier auf

## 4 Konzeption

den Wert `treasurehunt:iq:pickuptreasure` gesetzt. Die beiden Elemente `input` und `output` definieren in diesem Beispiel ein type-Attribut `set` und `result`, die dem [XMPP](#)-Standard entsprechen.

### MSDL-Service

Das `service`-Element ist der Einstiegspunkt in die Beschreibung. Hier werden allgemeine Attribute des Mobilis-Service beschrieben, sowie die Endpunkte und Abhängigkeiten zu anderen Mobilis-Services definiert. Ein `service`-Element besitzt dabei folgende Kind-Elemente:

**attributes** Definieren allgemeine Attribute des Mobilis-Service

**version** Version des Mobilis-Service als einfachen Integer-Wert

**name** Name des Mobilis-Service

**interface** Interface, das die Operationen dieses Mobilis-Service festlegt

**ident** Namespace des Mobilis-Service zur Identifizierung

**endpoint** Endpunkt, der den Mobilis-Service zur Verfügung stellt

**dependencies** Liste mit Mobilis-Services von denen dieser Service abhängig ist

**svcdep** (optional) Eine Referenz auf einen Mobilis-Service

**version** (optional) Integer-Wert, der eine ganz bestimmte Service-Version voraussetzt

**minVersion** (optional) Integer-Wert, der eine minimale Service-Version voraussetzt (mit `maxVersion` kombinierbar)

**maxVersion** (optional) Integer-Wert, der eine maximale Service-Version voraussetzt (mit `minVersion` kombinierbar)

**ident** Namespace vom abhängigen Mobilis-Service

Die Identifizierung von Mobilis-Services findet nicht wie bei [WSDL](#) über die Adresse (im Fall von [XMPP](#) die [JID](#)) statt, sondern aus einer Kombination aus dem Namespace und der Version des Mobilis-Service.

Die Abhängigkeit von anderen Mobilis-Services ist optional und kann freigelassen werden. Für die Versionsabhängigkeiten gibt es auch die Möglichkeit keines der `version`-Attribute anzugeben, dann wird immer der Mobilis-Service mit der höchsten Version gewählt. Dasselbe gilt für das Intervall von `minVersion` bis `maxVersion`. Liegen hier mehrere Mobilis-Services in dem Intervall wird die höchste Version aus dem Intervall gewählt.

Im TreasureHunt-Beispiel in Abbildung 4.16 wird für die Identifikation des Dienstes der Service-Namespace `http://mobilis.inf.tu-dresden.de#services/TreasureHuntService` als Wert im Attribut `ident` und 1 für das Attribut `version` festgelegt. Beide Attribute sind Pflicht und die Kombination aus beiden muss einmalig sein, um den Dienst später im Mobilis-Server zu identifizieren. Der TreasureHunt-Service ist aus Vorführgründen vom Mobilis-Grouping-Service abhängig, wie das Attribut `ident` im `svcdep`-Element definiert. Der Grouping-Service muss in der Version 1 vorhanden sein, damit der TreasureHunt-Service gestartet werden kann.

```

<msdl:service name="TreasureHunt" interface="tns:THInterface"
  ident="http://mobilis.inf.tu-dresden.de#services/TreasureHuntService"
  version="1">

  <msdl:endpoint name="TPEndpoint" binding="tns:THBinding" />

  <msdl:dependencies>
    <msdl:svcdep
      ident="http://mobilis.inf.tu-dresden.de#services/GroupingService"
      version="1" />
  </msdl:dependencies>

</msdl:service>

```

**Abbildung 4.16:** Element service der TreasureHunt-MSDL

## 4.4 XMPP-Mapping

Für das Mapping der *MSDL* auf *XMPP* wird in diesem Abschnitt das *binding*-Element der *MSDL* betrachtet. Als Beispiel soll hier wieder die *MSDL* des *TreasureHunt*-Service zum besseren Verständnis aus dem vorhergehendem Abschnitt 4.3.3 herangezogen werden, dessen Operationen in Tabelle 4.1 definiert wurden.

Als *type* wurde im *binding* *http://mobilis.inf.tu-dresden.de/xmpp/* festgelegt. Der Teil *xmpp* beschreibt dabei die zu verwendende Übertragungsart, in diesem Fall reines *XMPP*. Neben *xmpp* ist zudem noch *bosh* mit vorgesehen.

Für *XMPP*-spezifische Angaben wird das Präfix *xmpp*, welches im Kopf der *MSDL* mit dem Namespace *http://mobilis.inf.tu-dresden.de/xmpp/* definiert ist, verwendet.

Ein *operation*-Element repräsentiert ein XMPP-Bean, ein XMPP-IQ oder XMPP-Message-Stanza. Da die XMPP-Beans momentan nur IQ-Stanzas unterstützen, werden diese erweitert, sodass eine Verwendung des Message-Stanza möglich ist. Jedem *operation*-Element wird unter anderem ein *xmpp:ident*-Attribut zugewiesen. Dieses ist dem Namespace eines XMPP-Beans gleichzusetzen und dient der Identifizierung und Registrierung der Stanzas im *XMPP*-Netzwerk.

Für die Kind-Elemente *input* und *output* werden der Übersicht halber separate XMPP-Beans erzeugt. Diese beiden Elemente besitzen nur ein *xmpp:type*-Attribut, das den Stanza-Typ angibt. Dieses kann bei einem *input*-IQ-Stanza SET oder GET und bei einem *output*-IQ-Stanza nur RESULT sein. Wird ein Message-Stanza verwendet, so lautet der Typ CHAT (siehe Tabelle 4.2). In Abbildung 4.17 ist zum Beispiel die Umsetzung des *input*-Elements *PickUpTreasure* als *XMPP*-Stanza dargestellt. Die Zuweisung der *type*-Attribute zu den *input*- und *output*-Elementen im Binding ist nötig, da eine automatische Zuweisung und Unterscheidung von SET und GET schwierig, wenn nicht unmöglich ist.

Das *fault*-Element ist etwas komplexer gestaltet und benötigt die Angabe der Attribute *xmpp:errortype*, *xmpp:errorcondition* und *xmpp:errortext*. Diese entsprechen den Standard-Attributen des *XMPP*-ERROR-Typs. In der Abbildung 4.18 ist zu sehen, wie das *PickUpTreasureFault* auf *XMPP* umgesetzt wird. Hierbei finden sich alle er-

## 4 Konzeption

ror-Attribute wieder, so wie sie im [XMPP](#)-Standard definiert sind. Dabei muss darauf geachtet werden, dass diese Namenskonventionen eingehalten werden. Zusätzlich kann eine vorhandene Referenz zu einem Datentyp als *application-specific condition element* umgesetzt werden, so wie es im [XMPP](#)-Standard definiert ist. Im Beispiel verwendet *PickUpTreasureFault* den gleichnamigen Datentypen *PickUpTreasureFault* um die Informationen *ErrorCode* und *Timestamp* mitzusenden.

**Tabelle 4.2:** Mapping der XMPP-Stanzas auf ein operation-Element

Element	IQ-Stanza-Typ	Message-Stanza-Typ
input	SET, GET	CHAT
output	RESULT	CHAT
fault	ERROR	ERROR

```
<iq to="mobilis@openfire/TreasureHuntService" from="player1@openfire/MXA"
    id="mobilis_4056" type="set">
    <query xmlns="treasurehunt:iq:pickuptreasure">
        <TreasureLocation>
            <Latitude>51033880</Latitude>
            <Longitude>13 83! !</Longitude>
        </TreasureLocation>
    </query>
</iq>
```

**Abbildung 4.17:** Beispiel IQ PickUpTreasure SET

```
<iq to="player1@openfire/MXA" from="mobilis@openfire/TreasureHuntService"
    id="mobilis_4056" type="error">
    <query xmlns="treasurehunt:iq:pickuptreasure">
        <TreasureLocation>
            <Latitude>51033880</Latitude>
            <Longitude>13783272</Longitude>
        </TreasureLocation>
    </query>
    <error type="modify">
        <not-acceptable xmlns="urn:ietf:params:xml:ns:xmpp-stanzas" />
        <text xmlns="urn:ietf:params:xml:ns:xmpp-stanzas">
            This Treasure is not valid
        </text>
        <PickUpTreasureFault xmlns="treasurehunt:iq:pickuptreasure#errors">
            <ErrorCode>401</ErrorCode>
            <Timestamp>1326038151485</Timestamp>
        </PickUpTreasureFault>
    </error>
</iq>
```

**Abbildung 4.18:** Beispiel IQ PickUpTreasure ERROR

Die Operation *TreasureCollected* wird, im Gegensatz zu den anderen beiden, als Message-Stanza umgesetzt (siehe Abbildung 4.19), da diese nur in eine Richtung geht und keine Antwort erwartet. Ist hingegen eine leere Antwort verlangt, muss im interface-Abschnitt der [MSDL](#) das *element*-Element eines *input* oder *output*-Elements weggelassen werden.

```
<message to="player1@openfire/MXA" from="mobilis@openfire/TreasureHuntService"
  id="onName_4057" type="chat">
  <query xmlns="treasurehunt:iq:treasurecollected">
    <PlayerName>player1</PlayerName>
    <TreasureLocation>
```

**Abbildung 4.19:** Beispiel Message TreasureCollected

Das Presence-Stanza wird in der Dienstbeschreibung nicht verwendet, da es so konzipiert wurde, dass eine Nachricht an mehrere Empfänger gesendet werden kann, ohne darauf eine Antwort zu erhalten. In einer *Instant Messaging and Presence* Anwendung wird es dafür verwendet einen Aktivitätsstatus an mehrere Teilnehmer zu senden. Da in der Beschreibung des Mobilis-Service jedoch keine transportspezifischen Elemente wie Adressen statisch festgelegt werden, wird das Presence-Stanza an dieser Stelle nicht unterstützt.

## 4.5 Codegenerierung anhand der MSDL

Mit der Erstellung einer Dienstbeschreibung und der Verwendung dieser zum Auslesen von einigen Informationen, ist die Mobilis-Plattform bereits eine vollwertige Dienstplattform. Die Dienstbeschreibung soll aber auch dazu verwendet werden Quellcode zu erzeugen, der die Nutzung des Kommunikationsprotokolls vereinfacht. Dafür werden die definierten Datentypen im `types`-Abschnitt auf Objekte in der jeweiligen Zielsprache transformiert und die Operationen im `interface`-Element auf Methoden derselben Zielsprache.

Eine vollständige Codegenerierung kann nicht im vollem Umfang in dieser Arbeit umgesetzt werden. Diese soll zumindest als ein grundlegendes Konzept realisiert und in einer einfachen Form umgesetzt werden.

Von Vorteil wäre die Nutzung der [WSDL](#)-Tools zur Codegenerierung. Jedoch weigern sich alle gängigen Tools wenigstens die Datentypen zu erstellen, weil in der [MSDL](#) einige Veränderungen im `binding` und `interface`-Element vorgenommen wurden und diese somit nicht mehr dem [WSDL](#)-Schema entspricht.

Im Folgenden soll als Erstes auf die Besonderheiten der Codegenerierung eingegangen werden, die [XMPP](#) als Kommunikationsprotokoll mit sich bringt. Darauf aufbauend wird gezeigt, wie [Extensible Stylesheet Language Transformation \(XSLT\)](#) für die Codegenerierung verwendet werden kann und welche Grenzen diese Methode hat.

### 4.5.1 Besonderheiten bei der Codegenerierung von XM

Dem Entwickler einfache Methoden zur Verfügung zu stellen, die Ein- und Ausgabe-parameter definieren ist aufgrund des asynchronen Kommunikationsverhaltens von

**XMPP** nicht so einfach möglich. Insofern ein Request-Response-Verhalten definiert wurde, soll der Anfragende nicht, wie bei einer **WSDL**-Umsetzung, so lange blockieren bis eine Antwort zurückkommt. Stattdessen soll die Asynchronität des Protokolls auch in die Anwendung übertragen werden. In der Java-Umsetzung wird für diesen Fall immer ein Callback-Objekt mitgegeben, wenn eine Anfrage gestellt wird. Dieses Callback-Objekt wird aufgerufen, sobald eine Antwort im Hintergrund eingetroffen ist.

Andererseits muss auf eingehende Nachrichten, die von der Gegenseite gesendet wurden, reagiert werden können. Zum Beispiel muss die Client-Logik benachrichtigt werden, wenn der Dienst eine Anfrage stellt und eine Antwort erwartet.

Ein zweiter Unterschied besteht in der Adressierung der Gegenstellen, da diese nicht wie bei Webservices von vornherein festgelegt werden, sondern in **XMPP** dynamisch erstellt werden. Deshalb muss darauf geachtet werden, dass neben den Parametern für eine Anfrage immer die Zieladresse (**JID**) mit gesetzt wird, an die das Paket gehen soll.

Zuletzt muss sichergestellt werden, dass alle verwendeten Parameter serialisierbar sind und somit immer eine Objekt-basierte und eine XML-basierte Darstellung der verwendeten Elemente vorhanden ist. Dies ist notwendig um alle Parameter in den Payload des Stanzas einzubetten und diesen wieder auszulesen.

### 4.5.2 Codegenerierung mit XSLT

**XSLT** ermöglicht es das Prinzip der Template-basierten Codegenerierung, wie in Abbildung 4.20(a) dargestellt, umzusetzen. Dafür wird mithilfe einer *Template Engine* ein Datenmodell, unter Verwendung einer Vorlage, in Quellcode transformiert.

**XSLT** ist auf **XML**-basierend und benötigt als Datenmodell eine **XML**-basierte Struktur. Mit einem **XSLT**-Dokument und einem **XSLT**-Prozessor können nun beliebige Ausgabeelemente erzeugt werden. Als Datenmodell soll die in dieser Arbeit entwickelte **MSDL** dienen, um daraus einen Client-Stub zu generieren. Der Service-Stub hingegen ist im Moment nur in der Zielsprache Java nötig.

Voraussetzung für die Verwendung von **XSLT** ist die Version 2.0, da erst ab dieser Version mehrere Ausgabedateien mit nur einer **XSLT**-Datei erzeugt werden können. Dies ist notwendig da Programmiersprachen wie Java nur eine Klasse pro Datei erlauben und die Definition aller XMPP-Beans und Proxy-Elemente in einer Klasse teilweise nicht möglich sind. Andererseits wird das Prinzip der objektorientierten Programmierung damit zunichte gemacht, wohingegen Script-Sprachen wie JavaScript damit keine Probleme haben. In Abbildung 4.20(b) ist zu sehen, wie die Template-basierte Codegenerierung mit **XSLT** umgesetzt wird.

Die Abbildung 4.21 zeigt die Verwendung des generierten Quellcodes im Kontext der Mobilis-Clients und Services. Für die Verwendung der Kommunikationspakete sollen weiterhin die XMPP-Beans verwendet werden. Diese werden schematisch betrachtet in ein *Business Object* und in ein *Communication Object* zerlegt. Das hat den Vorteil, dass die Anwendung des Entwicklers keine Informationen über das Serialisieren in ei-

ne kommunikationsspezifische Form haben muss. Andersherum muss die Schnittstelle, die ein *XMPP*-Bean in ein *XMPP*-Stanza umwandelt keine Informationen über den Aufbau der Nutzdaten haben. Somit kann eine klare Trennung zwischen Anwendungs- und Kommunikationsschicht vollzogen werden.

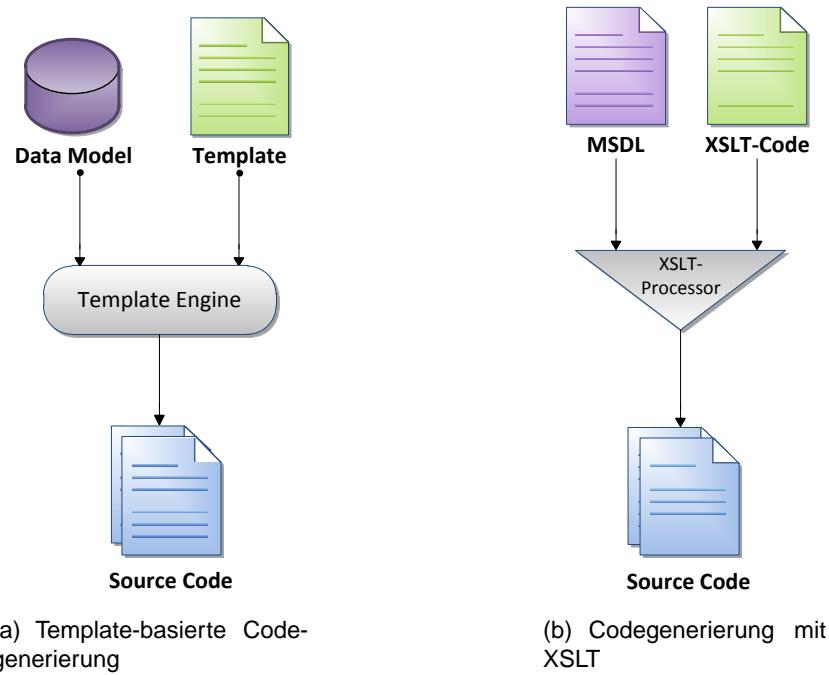


Abbildung 4.20: Template-basierte Codegenerierung

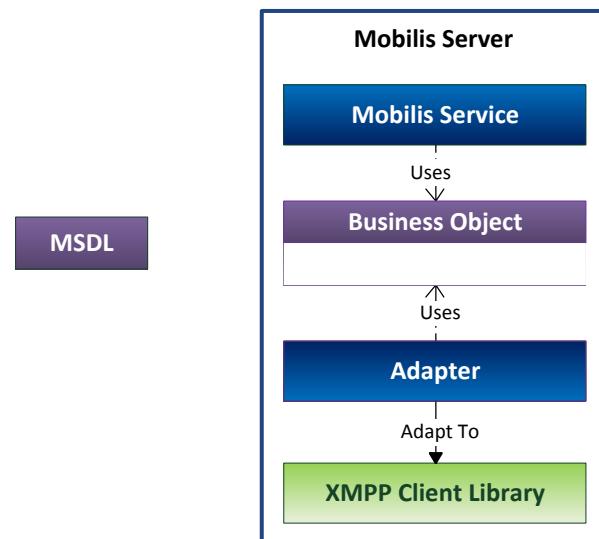


Abbildung 4.21: Codegenerierung Übersicht

## 4 Konzeption

Der Adapter, der ein XMPP-Bean in ein XMPP-Stanza umwandelt, muss jedoch vom Entwickler geschrieben werden, da dieser speziell an die verwendete XMPP-Bibliothek angepasst werden muss, um ein XMPP-Bean in ein XMPP-Stanza umzuwandeln.

Mit Hilfe der Codegenerierung soll nun ermöglicht werden, die konkreten XMPP-Beans und XMPP-Infos zu erstellen. Diese werden dann über eine *Proxy-Klasse* durch Methoden zugänglich gemacht. In Abbildung 4.22 ist das Klassen-Schema dargestellt, das eine Codegenerierung für einen Java-Client erzeugt. Neben der Proxy-Klasse, die für das direkte Versenden von Parametern zur Verfügung steht, bieten die Schnittstellen *IIncoming* und *IOutgoing* die abstrakten Operationen an, die vom Entwickler durch die verwendete XMPP-Bibliothek adaptiert werden müssen.

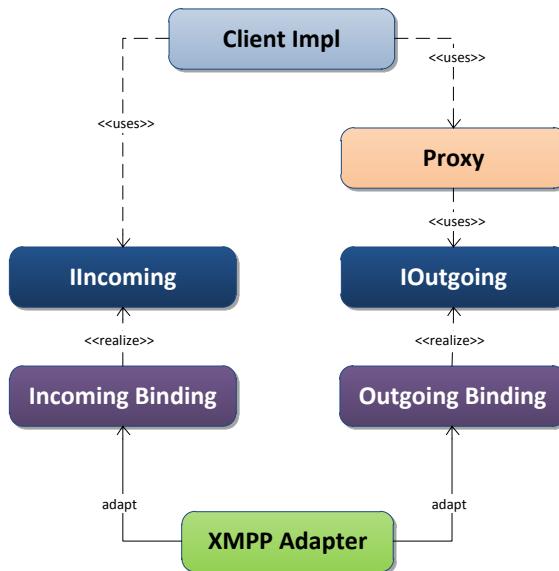


Abbildung 4.22: Codegenerierung Klassen-Schema Java

Zu beachten ist, dass für den Client die Schnittstellen *IIncoming* und *IOutgoing*, sowie die *Proxy-Klasse* anders umgesetzt werden, als die vom Service, da bei diesem der Nachrichtenverlauf genau andersherum stattfindet.

### 4.5.3 Grenzen der Codegenerierung

Die im vorhergehenden Abschnitt beschriebene Codegenerierung besitzt einige Grenzen, sodass nur eine einfache Umsetzung der Codegenerierung möglich ist.

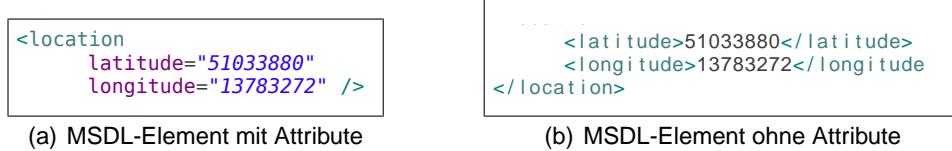
Die *input-* und *output-*Elemente werden als XMPP-Beans umgesetzt. Bei der Codegenerierung wird eine XMPP-Bean-Klasse mit erzeugt, die aber bei der eigentlichen Implementation durch die lokal vorhandene XMPP-Bean-Klasse ersetzt werden muss. Erst dann ist es möglich die bereits erstellte Funktionalität des BeanIQAdapters der Mobilis-Plattform zu verwenden. Dasselbe gilt für die Datentypen, die in den XMPP-Beans verwendet werden. Diese werden, genau wie das XMPP-Bean selber, von XMPP-Info abgeleitet, welches auch durch die lokale Repräsentation ersetzt werden muss.

Ähnlich verhält es sich mit der De- und Serialisierung der Business-Objekte (objektbasiert) in Communication-Objekte (XML-basiert). Diese benötigen für den Vorgang die XML-Pull-Parser-Bibliothek, die lokal in den Classpath eingebunden sein muss. Hier wäre es sinnvoll in einem nächsten Schritt die Bibliothek durch native Verfahren, wie zum Beispiel SAX für Java, zu ersetzen.

Bei Änderungen an der *MSDL* muss das gesamte Code-Skelett neu erzeugt werden. Es ist nicht möglich bereits implementierte Teile mit zu übernehmen, wie bereits in den Abgrenzungskriterien 3.4 festgelegt wurde. Diese implementierten Teile müssen somit manuell kopiert werden.

Die Generierung der einzelnen Datentypen aus dem *types*-Element der *MSDL* wird in dieser Arbeit nur teilweise unterstützt und verlangt eine manuelle Nachjustierung oder Implementierung bei speziellen Datentypen. Einfache Datentypen, wie zum Beispiel *int* oder *string*, werden bereits unterstützt, wohingegen komplexe Datentypen, wie zum Beispiel Datumsangaben, noch nicht unterstützt werden.

Bei der Generierung der Datentypen werden keine Attribute in den *MSDL*-Elementen berücksichtigt, wie in Abbildung 4.23(a) dargestellt. Diese müssen als einfache Elemente definiert werden (siehe Abbildung 4.23(b)). Dies erleichtert die Suche in der *MSDL* um ein Vielfaches, da die Möglichkeiten der Erstellung beschränkt werden.



**Abbildung 4.23:** Aufbau der MSDL-Elemente

## 4.6 Zusammenfassung

In diesem Kapitel wurden zu Beginn in Abschnitt 4.1 verschiedene Architekturansätze betrachtet. Unter den Möglichkeiten Mobilis-Services als externe Dienste zu betreiben und OSGi als Laufzeitumgebung unter den Mobilis-Server zu integrieren, wurde die dritte Variante gewählt, die es ermöglicht den Mobilis-Server um Module zu erweitern, unter Verwendung einer eigenen Laufzeitumgebung.

Das Konzept der eigenen Laufzeitumgebung wurde darauf in Abschnitt 4.2 detailliert erörtert. Dabei wurden die *System Services* definiert, die das dynamische Deployment steuern können. Neben diesen Diensten wurde der *Mobilis Container* eingeführt, der es ermöglicht Mobilis-Services dynamisch zu verwalten und deren Lebenszyklus zu beeinflussen. Die Lebenszyklen des Mobilis-Container und des Mobilis-Service wurden darauf im Detail erläutert.

In Abschnitt 4.3 wurden die benötigten Elemente ermittelt um einen Mobilis-Service

## *4 Konzeption*

formal zu beschreiben. Anhand dieser Analyse und einem Vergleich zu *WSDL* wurde die *MSDL* eingeführt und deren Aufbau erläutert. Diese ist aus der *WSDL* entstanden und lässt sich durch eine Substitution wieder zurück transformieren.

Die Übersetzung der Beschreibung in den *XMPP*-Standard wurde in Abschnitt 4.4 Detail erläutert. Dabei wurde vor allem auf das binding-Element der *MSDL* eingegangen und anhand eines Beispielszenarios die Umsetzung der *MSDL*-Elemente in ein *XMPP*-Stanza gezeigt.

Zuletzt wurde in Abschnitt 4.5 gezeigt, was bei der Codegenerierung zu beachten ist und einige Besonderheiten aufgezeigt, die mit *XMPP* einhergehen. Darauffolgend wurde eine einfache Umsetzung dargestellt, die mit Hilfe der Sprache *XSLT* aus einer *MSDL* Quellcode für die Zielsprache erzeugen kann. Zum Schluss wurden die Grenzen aufgezeigt, die mit dieser Art Code zu generieren, einhergehen.

# 5 Implementierung

In diesem Kapitel soll die Umsetzung der Konzepte aus Kapitel 4 gezeigt werden. Zu Beginn wird ein Überblick über die gesamte Architektur und deren einzelnen Prozesse, in Abschnitt 5.1, gegeben. Diese werden im weiteren Verlauf des Kapitels, anhand des bereits eingeführten Beispiels *TreasureHunt*-Service, aus Kapitel 4.3.3, im Detail erläutert. Das Beispiel *TreasureHunt* wird in Abschnitt 5.2 noch einmal kurz erläutert und das dazugehörige das Kommunikationsprotokoll definiert. Die einzelnen Schritte werden dabei aus der Sicht des Serviceentwicklers betrachtet.

## 5.1 Übersicht

Der Entwicklungsprozess gliedert sich in die folgende Teilprozesse, die ein Entwickler nacheinander durchlaufen muss:

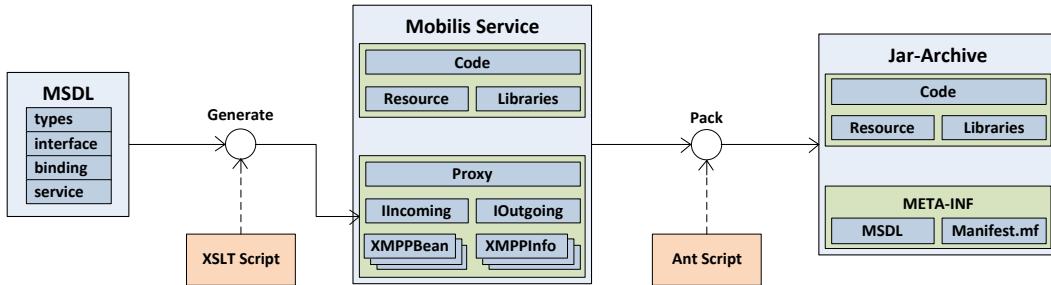
1. Erstellung der Protokoll- und Dienstbeschreibung
2. Generierung des Service-Skeleton anhand der MSDL
3. Implementierung der Servicelogik
4. Packen des Service in ein Jar-Archive
5. Hochladen des Jar-Archive zum Mobilis-Server
6. Life cycle Management des Service (de-/ installieren, konfigurieren, de-/ registrieren, aktualisieren, Instanz starten/stoppen)

Diese Teilprozesse können in zwei Phasen kategorisiert werden. Die Schritte 1 bis 4 stellen die Implementierungsphase dar, deren Zusammenhänge in Abbildung 5.1 dargestellt ist. Die Schritte 5 bis 6 bilden die Managementphase, die den gesamten Life cycle abbildet, wie es bereits ausführlich im Kapitel 4 gezeigt wurde.

## 5.2 Protokoll- und Dienstbeschreibung

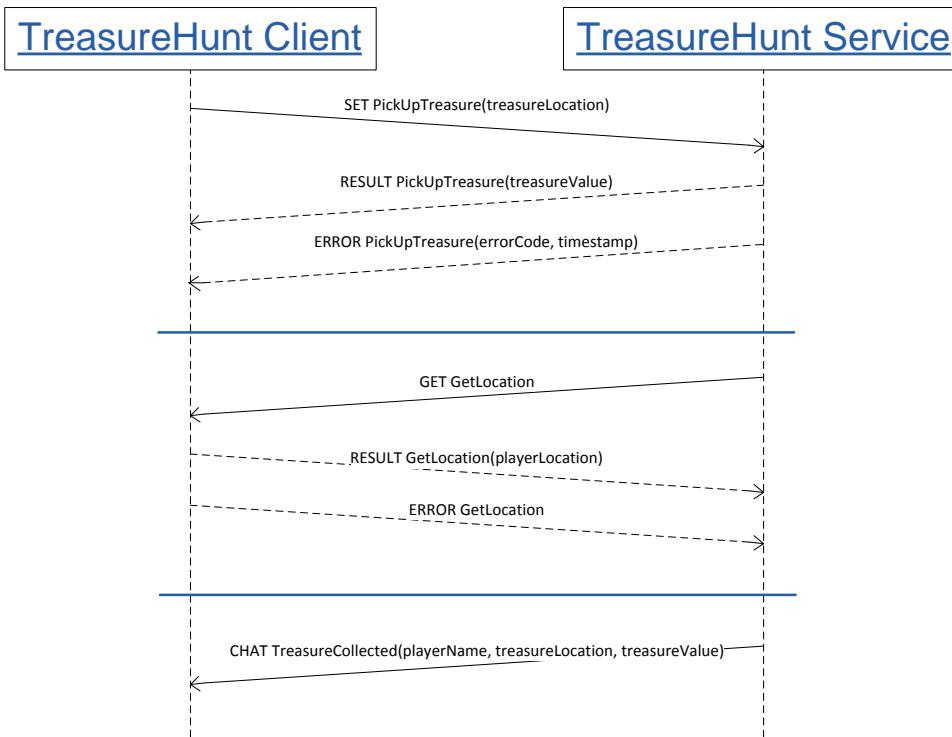
*TreasureHunt* ist eine einfache Umsetzung eines *LBG*, in dem mehrere Spieler auf einer Karte virtuelle Schätze suchen und finden können. Anhand der geografischen Positionen der Spieler ist es somit möglich, einen Schatz an der aktuellen Position aufzunehmen, was durch die Operation *PickUpTreasure* realisiert wird. Der *TreasureHunt*-Service ermittelt dafür in einem periodischen Intervall die Positionen der Spieler mit der Operation *GetLocation*. Hat ein Spieler einen Schatz gefunden, so werden alle anderen Spieler, mithilfe der Operation *TreasureCollected*, darüber informiert. Hierbei erhalten die Spieler Informationen, wo der Schatz gefunden wurde, welcher Spieler

## 5 Implementierung



**Abbildung 5.1:** Übersicht Implementierung von MSDL zum Jar-Archiv

diesen gefunden hat und wie viel dieser Schatz wert ist. Für ein besseres Verständnis sind die einzelnen [XMPP](#)-Kommunikationen in der Abbildung 5.2 dargestellt.



**Abbildung 5.2:** Kommunikation in TreasureHunt

Als erstes müssen für die Erstellung der [MSDL](#) die Operationen des Dienstes beschrieben werden. Die Erstellung der [MSDL](#) kann entweder unter Zuhilfenahme der Schemadatei erstellt werden oder es kann ein beliebiger [WSDL](#) 2.0 Editor verwendet werden, um eine [WSDL](#) zu erstellen. Wird ein [WSDL](#) 2.0 Editor verwendet, so muss

die erstelle [WSDL](#) im Anschluss in eine [MSDL](#) transformiert werden. Eine technische Anleitung für diese Transformation ist in Anhang [B.1](#) aufgeführt.

Auf die Erstellung der [MSDL](#) für den TreasureHunt-Service wird in diesem Abschnitt nicht weiter eingegangen, da diese bereits in Kapitel [4.3.3](#) ausführlich erläutert wurde. Die gesamte [MSDL](#) des TreasureHunt-Service ist in Anhang [B.3](#) zu finden.

## 5.3 Code Generierung

Damit, anhand der erstellten [MSDL](#) verwendbarer Java-Code generiert werden kann, ist es möglich, dass das in dieser Arbeit entwickelte [XSLT](#)-Script verwendet wird. Dafür wurden die Scripte *MSDL2JavaClient.xslt*, für die Generierung des Client-Stubs und *MSDL2JavaService.xslt*, für die Generierung des Server-Skeleton entwickelt.

Eine [XSLT](#) besteht dabei aus einer Menge von Templates, die nacheinander aufgerufen werden, um bestimmte Teile der [MSDL](#) zu analysieren und daraus die benötigten Codezeilen zu generieren. In Abbildung [5.3](#) ist das Main-Template abgebildet, das den Einstieg in die [XSLT](#) bietet und iterativ die einzelnen Teile der [MSDL](#) durchläuft. Templates werden dabei mit dem XSL-Tag `xsl:apply-templates` aufgerufen. Diesem Tag wird, über das Attribut `select`, ein Pfad innerhalb des Eingabedokumentes mitgegeben, der die Ebene bestimmt ab der das Template mit der Bearbeitung anfangen soll. Zum Beispiel beginnen die ersten Templates in Abbildung [5.3](#) an der Wurzel. Zur weiteren Differenzierung der einzelnen Templates können diese in einem bestimmten Modus aufgerufen werden, der mit dem Attribut `mode` festgelegt wird. Mithilfe dieser Funktion ist es möglich, eine Art Funktion zu definieren, wie man es aus der imperativen Programmierung kennt.

```

<xsl:template match="/">
  <xsl:apply-templates select="/" mode="generateProxyClass" />
  <xsl:apply-templates select="/" mode="generateXMPPInfoInterface" />
  <xsl:apply-templates select="/" mode="generateXMPPBeanClass" />

  <xsl:apply-templates select="/" mode="generateIIncomingInterface" />
  <xsl:apply-templates select="/" mode="generateIOutgoingInterface" />

  <xsl:apply-templates select="/" mode="generateXMPPCallbackInterface" />

  <xsl:apply-templates
    select="/msdl:description/msdl:binding/msdl:operation"
    mode="generateTypeBeanClass" />

  <xsl:apply-templates
    select="/msdl:description/msdl:types/xs:schema/xs:complexType"
    mode="generateTypeInfoClass" />
</xsl:template>

```

**Abbildung 5.3:** Main-Template der XSLT zur Codegenerierung

Wichtig für die Verwendung ist, dass die zu erstellenden Code-Dateien in dem Ordner erzeugt werden, der in der `xsl:variable` mit dem Attribut `name="outputFolder"`

## 5 Implementierung

definiert wurde. Dieser Pfad muss physisch vorhanden sein, da [XSLT](#) nicht auf jedem System in der Lage ist, diese Pfade anzulegen.

Neben der bereits erwähnten Struktur aus dem Kapitel [4.5.2](#) werden die bekannten XMPP-Beans mitgeneriert. Diese XMPP-Beans werden aus den Operationsparametern `input` und `output` der [MSDL](#) im Abschnitt `interface` gelesen. Im Beispiel `TreasureHunt` werden für die Operation `PickUpTreasure` (sieh Abbildung [5.4](#)) die XMPP-Beans `PickUpTreasureRequest` und `PickUpTreasureResponse` erstellt. Damit diese XMPP-Beans erstellt werden können, sind die Typinformationen notwendig, die in der [MSDL](#) in Abschnitt `types` definiert sind. Für die Operation `PickUpTreasureRequest` ist diese Definition in Abbildung [5.5](#) dargestellt. Für das korrespondierende XMPP-Bean, das nach der Codegenerierung entsteht, ist in Abbildung [5.6](#) ein Beispiel zu sehen. Alle weiteren Datentypen, die in den XMPP-Beans verwendet werden, sind als XMPP-Info umgesetzt worden, wie zum Beispiel die Eigenschaft `Location` in der Klasse `PickUpTreasureRequest`.

```
<msdl:operation name="PickUpTreasure"
    pattern="http://www.w3.org/ns/wsdl/in-out">
    <msdl:input element="tns:PickUpTreasureRequest" />
    <msdl:output element="tns:PickUpTreasureResponse" />
    <msdl:outfault ref="tns:PickUpFault" />
</msdl:operation>
```

Abbildung 5.4: MSDL-Abschnitt der PickUpTreasure Operation

```
<xss:element name="PickUpTreasureRequest">
    <xss:complexType>
        <xss:sequence>
            <xss:element name="TreasureLocation"
                type="tns:Location" />
        </xss:sequence>
    </xss:complexType>
</xss:element>
```

Abbildung 5.5: MSDL-Abschnitt des Typs PickUpTreasureRequest

Die Fault-Elemente der Beschreibung werden nicht als separate Klassen generiert. Die Erstellung der detaillierten XMPP-Errors ist mit der jeweiligen Klasse möglich, so kann der definierte Fehler `PickUpFault` mit der Funktion `buildPickUpFault` aus der Klasse `PickUpTreasureRequest` erstellt werden. Mit dieser Methode muss der Entwickler die Attribute `errorText`, `errorCondition` und `errorType` nicht mehr selber definieren, da diese aus der erstellten [MSDL](#) verwendet werden. Optinal kann der Fehlertext um einen beliebigen String erweitert werden.

Zuletzt kann für die einfache Handhabung die Proxyklasse verwendet werden, um die jeweiligen XMPP-Beans in einer abstrahierten Form zu erstellen und zu versenden. Die Proxyklasse `TreasureHuntProxy` ist in Abbildung [5.7](#) dargestellt. Hiermit kann der Service die drei anfänglich definierten Methoden nutzen. Wenn eine Antwort erwartet wird, muss ein `IXMPPCallback` definiert werden, an den die Antwort weitergeleitet werden soll. Für jede Methode muss eine `JID` mitgegeben werden, da es immer mehrere

Empfänger geben kann, wie zum Beispiel mehrere Spieler in TreasureHunt, die über das Auffinden eines neuen Schatzes informiert werden sollen. Wenn nur ein Result gesendet werden soll, muss zusätzlich die packetId, auf die sich das Result bezieht, mit angegeben werden. Ist es notwendig, Nachrichten individuell anzupassen, ohne alle Parameter zu verwenden, liefert die Methode getBindingStub ein Interface um beliebige XMPP-Beans versenden zu können.

```
public class PickUpTreasureRequest extends XMPPBean {

    public static final String NAMESPACE = "treasurehunt:iq:pickuptreasure";
    public static final String CHILD_ELEMENT = "PickUpTreasureRequest";

    private Location TreasureLocation = new Location();

    public PickUpTreasureRequest( Location TreasureLocation ) {...}

    public PickUpTreasureRequest(){...}

    @Override
    public void fromXML( XmlPullParser parser ) throws Exception {...}

    @Override
    public String getChildElement() {...}

    @Override
    public String getNamespace() {...}

    @Override
    public XMPPBean clone() {...}

    @Override
    public String payloadToXML() {...}

    public PickUpTreasureRequest buildPickUpFault(String detailedErrorText){...}

    public Location getTreasureLocation() {...}

    public void setTreasureLocation( Location TreasureLocation ) {...}
}
```

**Abbildung 5.6:** Generiertes XMPP-Bean des Typs PickUpTreasureRequest

### Einschränkungen

In der aktuellen Umsetzung bestehen Einschränkungen bezüglich der Codegenerierung. Zum Beispiel werden nicht alle Datentypen unterstützt, die vom XML-Schema vorgesehen sind. Die Datentypen, die unterstützt werden, sowie das dazugehörige Java-Äquivalent sind in Tabelle 5.1 aufgeführt.

Komplexe Datentypen, die von XMPP-Info erben, können innerhalb eines XMPP-Beans nur einmal verwendet werden, da das Parsen nach dem Child-Element bei allen Komplexen Datentypen gleich ist und immer wieder überschrieben wird. Somit ist es nicht möglich, in einem XMPP-Bean eine Eigenschaft *AliceLocation* und *BobLocation* vom selben Typ *Location* zu definieren. Diese müssen durch eine Liste von *Location* rea-

## 5 Implementierung

lisiert werden. Diese Problematik wurde mit den XMPP-Beans übernommen und bestand schon vorher.

```
public class TreasureHuntProxy {  
    private ITreasureHuntOutgoing _bindingStub;  
  
    public TreasureHuntProxy( ITreasureHuntOutgoing bindingStub) {...}  
  
    public ITreasureHuntOutgoing getBindingStub(){  
        return _bindingStub;  
    }  
  
    public void PickUpTreasure( String toJid, String packetId,  
        long TreasureValue ) {...}  
  
    public void GetLocation( String toJid,  
        IXMPPCallback< GetLocationResponse > callback ) {...}  
  
    public void TreasureCollected( String toJid, String PlayerName,  
        Location TreasureLocation, long TreasureValue ) {...}  
}
```

**Abbildung 5.7:** Proxyklasse des TreasureHunt Service

**Tabelle 5.1:** Unterstützte Datentypen der XSLT-Codegenerierung

XML-Typ	Java-Typ	Java Listen-Typ	Initialisierung
int	int	Integer	Integer.MIN_VALUE
long	long	Long	Long.MIN_VALUE
boolean	boolean	Boolean	false
double	double	Double	Double.MIN_VALUE
float	float	Float	Float.MIN_VALUE
byte	byte	Byte	Byte.MIN_VALUE
short	short	Short	Short.MIN_VALUE
string	String	String	null
*[ ]	(Array)List<?>	-	{ }

Beim Parsen der [MSDL](#) werden keine Attribute aus den Typinformationen ausgelesen. Hier muss ein einheitliches Schema eingehalten werden, wie es in der vorgegebenen Schemadatei definiert ist.

Die Klasse `XMPPBean` und das Interface `XMPPInfo` werden bei der Codegenerierung mit erstellt. Für die Mobilis-Service Projekte müssen jedoch die global definierten Klassen aus dem MobilisXMPP-Projekt verwendet werden, da sonst der BeanIQAdapter im Mobilis-Server nicht für die Konvertierung von XMPP-Beans in IQs verwendet werden kann.

Zuletzt wird für das Un-/ Marshalling in den XMPP-Beans, in der Funktion `fromXML` der XML-Pull-Parser benötigt, der auch bei den normalen XMPP-Beans zum Einsatz kommt. Dieser muss also für den Client-Stub mit in das Client-Projekt eingebunden

werden. Für den Service hingegen wird die Bibliothek vom Mobilis-Server mit angeboten und kann implizit genutzt werden.

## 5.4 Deployment eines neuen Mobilis-Service

### 5.4.1 Implementierung

Für die Implementierung eines neuen Mobilis-Service oder einer neuen Version empfiehlt es sich, ein eigenes Projekt zu erstellen. Damit kann sichergestellt werden, dass alle notwendigen Referenzen, wie zum Beispiel benötigte Bibliotheken und Ressourcen, vorhanden sind. Für das Szenario TreasureHunt wird ein gleichnamiges Java-Projekt *TreasureHunt* erstellt. Dessen Paketebene beginnt mit `de.treasurehunt`.

Nachdem der benötigte Code aus der [MSDL](#) erstellt wurde, kann dieser in das neue Projekt eingefügt werden. Hier erhöht eine neue Paketebene wieder die Übersichtlichkeit, wie zum Beispiel ein Paket namens `proxy`, das im Beispiel unterhalb von `de.treasurehunt` erstellt wird. Durch die bereits erwähnten Einschränkungen ist es hier zudem notwendig den XML-Pull-Parser mit in das Projekt einzubinden, damit die Fehlermeldungen behoben werden.

Unterhalb des `src`-Ordners sollte ein Ordner namens `META-INF` angelegt werden, der beim Packen des Archivs mit dem automatisch erstellten Ordner zusammengeführt wird. In diesen Ordner wird die erstellte [MSDL](#) verschoben, sodass diese beim Packen und beim Deployment gefunden werden kann.

Um die Klassen des Mobilis-Server oder weiterer Projekten nutzen zu können, müssen diese Projekte in den Klassenpfad mit eingebunden werden. Das Projekt Mobilis-Server ist hierbei zwingend notwendig, da nur so die Hauptklasse `MobilisService` für den eigenen Service verwendet werden kann. Damit ein eigener Mobilis-Service erstellt werden kann, muss dieser nämlich davon erben können. Weil Bibliotheken bei der Referenzierung nicht mit berücksichtigt werden, müssen des Weiteren die Smack-Bibliotheken `smack` und `smackx` mit in den Klassenpfad eingebunden werden. Diese beiden Bibliotheken, sowie die XML-Pull-Parser-Bibliothek werden jedoch beim Packen vernachlässigt, da diese vom Mobilis-Server bereitgestellt werden.

Nach diesen Schritten sieht die Struktur des *TreasureHuntService*-Projekts wie in Abbildung 5.8 dargestellt aus und besitzt mindestens die Classpath-Einträge, die in Abbildung 5.9 aufgelistet sind.

Zuletzt muss der generierte Code mit Logik gefüllt und an die XMPP-Bibliothek, in diesem Fall Smack, angebunden werden. Hierfür werden die Interfaces `ITreasureHuntOutgoing` und `ITreasureHuntIncoming` verwendet. Die `ITreasureHuntOutgoing`-Instanz wird für die Proxyklasse benötigt, um die XMPP-Beans über die XMPP-Bibliothek versenden zu können. Der Entwickler muss zum Beispiel nur die Methode `PickUpTreasure( String toJid, long TreasureValue )` aufrufen, um ein XMPP-Bean `PickUpTreasureResponse` an den Client zurückzusenden. Die eingehenden XMPP-Beans können an die Instanz `ITreasureHuntIncoming` weitergeleitet werden, um an dieser Stelle auf die eingehenden Pakete reagieren zu können. Dies hat den Vorteil, dass

## 5 Implementierung

durch verschiedene Interfaces unterschiedlich auf die Pakete reagiert werden kann und das Unmarshalling, eines IQs in ein XMPP-Bean, schon vorher erfolgt.

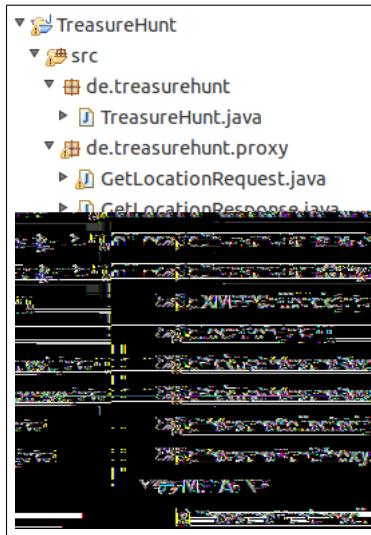


Abbildung 5.8: Struktur des TreasureHunt-Projekts

```
<classpath>
  ...
    <classpathentry kind="lib" path="lib/smack.jar"/>
    <classpathentry kind="lib" path="lib/xpp3-1.1.4c.jar"/>
    <classpathentry kind="lib" path="lib/smackx.jar"/>
    <classpathentry combineaccessrules="false" kind="src"
      path="/MobilisServer"/>
    <classpathentry combineaccessrules="false" kind="src"
      path="/MobilisXMPP"/>
  ...
</classpath>
```

Abbildung 5.9: Classpath TreasureHunt-Service

### 5.4.2 packen

Nachdem der Service um Logik und Ressourcen erweitert wurde, muss dieser in ein Jar-Archiv gepackt werden. Beim Packen ist es wichtig, dass einerseits alle benötigten Classpath-Referenzen richtig gesetzt sind und andererseits alle Ressourcen und Bibliotheken mitgepackt werden. Eclipse bietet hier zwar eine native Unterstützung an, ein Java-Projekt oder Teile davon als Jar-Archiv zu exportieren, packt dabei jedoch nicht die Bibliotheken mit in das Archiv und referenziert diese zudem nicht ausreichend.

Um den Vorgang des Packens zu vereinfachen und alle Anforderungen zu erfüllen, wurde ein Ant-Script erstellt. Dieses muss für jedes Projekt etwas angepasst werden. In Tabelle 5.2 sind dafür alle Variablen aufgeführt, die angepasst werden müssen.

**Tabelle 5.2:** Variablen des Ant-Build-Script für das Packen eines Mobilis-Service

Variable	Beispielwert	Beschreibung
<b>Ordnerstrukturen</b>		
src.dir	src	Pfad zum src-Ordner
src.dirs.extern	basedir/../../MobilisServer/src	Pfad zum src-Ordner von referenzierten Projekten
build.dir	build	Pfad zum temporären Speichern der Class-Files
dist.dir	dist	Ausgabeordner des Jar-Archivs
lib.dir	lib	Pfad zum Ordner der genutzten Bibliotheken
res.dir	resource	Pfad zum Ordner der genutzten Ressourcen
<b>Archiv-Meta-Informationen</b>		
service.package	de.treasurehunt	Paketpfad in dem sich der Mobilis-Service befindet
msdl.filepath	META-INF	Pfad zur <a href="#">MSDL</a> (relativ vom src-Ordner)
service.name	TreasureHunt	Name der Mobilis -Service-Klasse
msdl.filename	TreasureHunt.msdl	Name der <a href="#">MSDL</a> -Datei
<b>Archiv Parameter</b>		
jar.filename	TreasureHuntService_v1.jar	Name des Jar-Archivs
root.packages	de/treasurehunt/**/*.*	Pfad aus welchem die Klassen für das Packen gesucht werden
required.libraries	none	Pfade der benötigten Bibliotheken
required.resources	resource/**/*.*	Pfade der benötigten Ressourcen

Die Struktur des Jar-Archivs, die nach dem Packen des *TreasureHunt*-Service entstand, ist in Abbildung 5.10 aufgeführt. In dem Archiv wurde die Datei *MANIFEST.MF* mitgeneriert. In dieser Datei befinden sich die beiden Attribute *Service-Class*, das den Pfad zum erstellten Mobilis-Service enthält und *MSDL-File*, das den Pfad zur [MSDL](#) enthält. Diese Attribute sind sehr wichtig für das spätere Deployment im Mobilis-Server.

Des Weiteren sind in dem Archiv die Java- und Binärklassen hinterlegt. Es ist auch möglich nur die Binärklassen zu packen und somit den Quellcode zu verbergen. Diese Funktion, sowie weitere für die individuelle Anpassung sind im Ant-Script selber dokumentiert.

Hinsichtlich des Packens muss beachtet werden, dass alle zu packenden Ressourcen, sowie Bibliotheken, in nur einem Hauptordner untergebracht sind, ansonsten muss

## 5 Implementierung

das Ant-Script manuell angepasst werden. Die Ressourcen müssen im Jar-Archiv selber anders angesprochen werden, als in einem herkömmlichen Projekt. Angenommen, die PNG-Grafik "myPic.png" soll geladen werden, die sich im Ordner *resource* befindet, dann muss innerhalb des Codes diese Ressource wie in Abbildung 5.11 geladen werden.

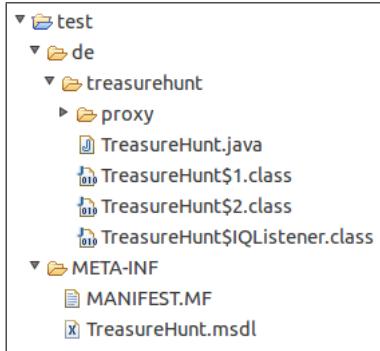


Abbildung 5.10: Jar-Archiv-Struktur des TreasureHunt-Service

```
InputStream is =
Main.class.getClassLoader().getResourceAsStream(
    "resource" + File.separator + "myPic.png");
```

Abbildung 5.11: Laden von Ressourcen aus einem Jar-Archiv

### 5.4.3 Life cycle Management

In Bezug auf das Life cycle Management soll an dieser Stelle nur kurz darauf eingegangen werden, was bei jeder Aktion im Mobilis-Server geschieht. Dafür werden im Folgenden die Aktionen *Hochladen*, *Installation*, *Konfiguration*, *Service-Instanz Starten/Stoppen*, *De-/Registrierung*, *Aktualisierung* und *Deinstallation* betrachtet. Beispiele für die Kommunikationspakete des TreasureHunt-Service sind in Anhang A aufgeführt.

Für das Auftreten von Fehlern beim Life cycle Management wurden für den Mobilis-Server spezielle Exception definiert, sodass eine bessere Fehlerbehandlung ermöglicht werden kann. Wird eine Exception geworfen, so wird diese vom *Admin-Service* abgefangen, in ein XMPP-Error umgewandelt und dem Client als Antwort gesendet. In Tabelle 5.3 sind alle Arten der Exceptions aufgeführt.

Tabelle 5.3: Mobilis Life cycle Exceptions

Exception	Beschreibung
InstallServiceException	Bei der Installation trat ein Fehler auf
RegisterServiceException	Bei der Registrierung trat ein Fehler auf
StartNewServiceInstanceException	Bei der Instantiierung trat ein Fehler auf
UpdateServiceException	Bei der Aktualisierung trat ein Fehler auf

### Hochladen

Für das Hochladen ist auf dem Mobilis-Server der Deployment-Service zuständig. Bevor ein neuer Service hochgeladen wird, muss dem Deployment-Service mittels eines *PrepareServiceUploadBean* mitgeteilt werden, welche Datei im Anschluss per File-Transfer zu erwarten ist. Im Beispiel TreasureHunt lautet der Name der Datei "TreasureHuntService\_v1.jar".

Wurde die Datei erfolgreich übertragen, legt der Deployment-Service diese in einem lokalen, temporären Ordner ab. Sollte sich in diesem Ordner bereits eine Datei mit demselben Namen befinden, so wird die neue Datei um einen aktuellen Zeitstempel erweitert. Somit kann sichergestellt werden, dass auch ein Service mit unterschiedlichen Versionen, zur selben Zeit, unter demselben Namen, auf dem Mobilis-Server vorhanden ist.

Das hochgeladene Archiv wird darauf einem neuen *ServiceContainer* zugewiesen und als sogenannter *PendingService* behandelt. Diese Services werden separat von den anderen aufbewahrt, da sie noch auf eine Installation warten. Vor einer Installation sind weder Namespace noch Version des Service bekannt. Dies erschwert die Identifizierung.

Waren diese Vorgänge erfolgreich, so wird dem Client der Name des Archivs zurückgeliefert, um diesen bei der Installation verwenden zu können.

### Installation

Für die Installation muss das Jar-Archiv *TreasureHunt\_v1.jar* auf dem Mobilis-Server vorhanden sein, sodass mit diesem Dateinamen der Service installiert werden kann.

Bei der Installation werden als Erstes die Pfade zur *MSDL* und zum Mobilis-Service aus der *Manifest.mf*-Datei ausgelesen. Wird die *MSDL* nicht an der vorgegebenen Stelle gefunden, wird das gesamte Jar-Archiv nach einer *MSDL*-Datei durchsucht und die Erste, die gefunden wurde, verwendet. Die *MSDL* wird anschließend lokal in einem temporären Ordner gespeichert, sodass der Auslesevorgang nicht jedes mal wiederholt werden muss. Aus der *MSDL* werden dann schließlich der Namespace und die Version des Service extrahiert und im Service-Container gespeichert.

Im nächsten Schritt wird für die Mobilis-Service-Klasse ein Template angelegt, sodass von dieser Klasse die Service-Instanzen erstellt werden können. Sollte in einem dieser Schritte ein Fehler auftreten oder eine Referenz nicht gefunden werden, so wird dem Client eine Fehlernachricht zurückgeliefert, die eine ausführliche Fehlerbeschreibung enthält.

### Konfiguration

Der Service kann im Zustand *Installed* und *Active* mit dem *ConfigureServiceBean* konfiguriert werden. Zum aktuellen Zeitpunkt ist es nur möglich, die Agent-Einstellungen zu bearbeiten. Die Konfiguration ist aber generisch gehalten, sodass beliebige Parameter darin gespeichert werden können. Die Parameter wurden aus der ehemaligen *MobilisSettings.xml* übernommen und erweitert. In Tabelle 5.4 sind die Agent-

## 5 Implementierung

Informationen aufgeführt, die notwendig sind, um den Service später registrieren zu können.

**Tabelle 5.4:** Parameter für die Agent-Konfiguration

Parameter	Beispielwert	Beschreibung
description	TreasureHunt	Beschreibung des Agent
host	localhost	Adresse des XMPP-Servers
mode	multi	Soll der Service nur einmalig oder mehrfach instantierbar sein
name	treasureHunt	Name des Agent
password	myPass	Xmpp Benutzerpasswort
port	5222	Xmpp Port
resource	TreasureHuntService	Xmpp Ressource
service	openfire	Name des XMPP-Servers
start	ondemand	Startverhalten des Service, wenn der Mobilis-Server gestartet wird
type	[...].MobilisAgent <sup>1</sup>	Agent-Typ
username	me	Xmpp Benutzername

### De-/Registrierung

Damit ein Service, wie der TreasureHunt-Service, vom Coordinator-Service entdeckt und so über ein Service-Discovery gefunden werden kann, muss dieser erst mithilfe des *RegisterServiceBean* registriert werden. Für diesen Aufruf muss der Namespace und die Version des Service angegeben werden. Sollten diese dem Mobilis-Server nicht bekannt sein, so wird eine Fehlermeldung zurückgegeben.

Wenn der Service vorhanden ist, wird im Service-Container überprüft, ob die Agent-Konfiguration vorhanden ist, um den Service im XMPP-Netzwerk anzumelden und ob alle Services, von dem zum Beispiel der TreasureHunt-Service abhängig ist, auch registriert sind. Sobald eine der Voraussetzungen nicht erfüllt wurde, wird eine Fehlermeldung zurückgegeben. Ansonsten wird der Service in der impliziten Service-Registry angemeldet.

Die implizite Service-Registry wird von den Agent-Konfigurationen im MobilisManager, wie schon in der Ausgangsversion, realisiert. Alle Agents die hier verzeichnet sind, können eine Verbindung zum XMPP-Netzwerk herstellen. Die Agents wurden hier lediglich um eine ID zur Identifizierung innerhalb der Settings erweitert. Als Agent-Konfiguration werden die eingestellten Konfigurationen aus dem Service-Container verwendet.

Wurde ein Mobilis-Service registriert, so ist es möglich die *MSDL* des Service anzu fordern. Dafür wurde das *MobilisServiceDiscoveryBean* so angepasst, dass der Client den Namespace und die Version des Service angibt, sowie das Attribut *requestMSDL* auf *true* setzt. Die *MSDL*-Datei wird darauf per File-Transfer übertragen.

---

<sup>1</sup> Ausgeschrieben: de.tudresden.inf.rn.mobilis.server.agents.MobilisAgent

Mithilfe des neuen *UnregisterServiceBean* können Services auch wieder deregistriert werden. Dafür muss wieder der Namespace und die Version des Service übermittelt werden. Nach der Deregistrierung laufen alle Service-Instanzen weiter, aber es können keine neuen mehr gestartet werden.

### Service-Instanz starten/stoppen

Wurde ein Service erfolgreich registriert, so kann er wie zuvor gestartet werden, indem ein *CreateNewServiceInstance* an den *Coordinator-Service* mit dem Namespace des Service gesendet wird. Sind mehrere Versionen unter diesem Namespace registriert, so wird immer eine Instanz des aktuellsten Service gestartet.

Der Befehl *CreateNewServiceInstance* wurde zudem um die Funktionen erweitert, dass auch Services einer bestimmten Version gestartet werden können. Dafür muss das Attribut *serviceVersion* größer "0" gesetzt werden. Ist unter dem angegeben Namespace kein Service dieser Version registriert, so wird eine Fehlermeldung zurückgeliefert. Eine weitere Möglichkeit bietet der Einsatz der beiden Attribute *minVersion* und *maxVersion*, die es ermöglichen den aktuellsten Service in diesem Intervall zu starten. Wird nur eines der Attribute gesetzt, so wird nur dieses Limit des Intervalls berücksichtigt und zum Beispiel nur Service-Instanzen gestartet, die mindestens die Version 5 aufweisen.

Neu ist die Möglichkeit aktuell laufende Service-Instanzen wieder zu stoppen. Dafür wurde das *StopServiceInstanceBean* implementiert. Diesem wird die *JID* des zu stoppenden Service mitgegeben und wenn diese vorhanden ist, wird der Service heruntergefahren.

### Aktualisierung

Für die Aktualisierung eines Service muss das Jar-Archiv der neuen Version bereits lokal vorhanden sein. Mit dem *UpdateServiceBean* kann dieser dann installiert werden. Hierfür müssen der Name des neuen Jar-Archivs, sowie der Namespace und die Version des zu ersetzenen Service angegeben werden.

Vor dem Ersetzen des alten Service wird zunächst ein Backup der alten Jar-Archiv-Referenz erstellt und der Mobilis-Service deregistriert. Daraufhin wird die Referenz des alten Jar-Archivs, mit der des Neuen ersetzt und die bereits beschriebene Installation gestartet. Sollte hierbei ein Fehler auftreten, so wird ein Rollback ausgelöst und der alte Service wird wieder eingespielt und installiert. Der Client wird bei einer fehlgeschlagenen Installation darüber informiert, ob das Rollback erfolgreich war. Zum Schluss werden die Registry-Einträge im Mobilis-Manager aktualisiert.

Service-Instanzen, die noch noch nicht beendet wurden, laufen weiter und können nur noch beendet werden. Die Konfigurationen bleiben währenddessen unverändert und können für den neuen Service verwendet werden. Nach der erfolgreichen Aktualisierung befindet sich der Service im Zustand *Installed* und muss manuell vom Client wieder registriert werden, da erst in dieser Phase die Abhängigkeiten zu anderen Services geprüft werden.

### Deinstallation

Bei der Deinstallation wird der Service zuerst deregistriert und jede noch laufende Service-Instanz beendet. War dies erfolgreich, so werden alle getätigten Einstellungen und Verweise, bis auf die Referenz zum Jar-Archiv, gelöscht.

## 5.5 Weiterführende Aufgaben

In diesem Abschnitt werden die Aufgaben angesprochen, die kurz-, beziehungsweise mittelfristig umgesetzt werden sollten, um das Konzept dieser Arbeit abzurunden und zu vervollständigen.

Einer der wichtigsten Punkte ist die Umstrukturierung des XMPP-Bean, sodass es auch das Message-Stanza unterstützt wird. In der [MSDL](#) ist die Nutzung des Message-Stanza bereits vorgesehen und wird durch die Nachrichtenmuster *in-only* und *out-only* realisiert. Eine konkrete Umsetzung in dem Bean-Layer der Mobilis-Plattform fehlt jedoch. Diese Umstrukturierung zieht eine grundlegende Änderung des gesamten Bean-Layers nach sich und hat somit auch Auswirkungen auf alle bereits erstellen XMPP-Bbeans. Wenn die Änderung erfolgt ist, sollte im Anschluss die Codegenerierung mit angepasst werden. Diese generiert aus den *\*-only* Nachrichtenmuster momentan noch IQ-Stanzas.

Wenn der Mobilis-Server gestartet wird, sind nur die System-Services aktiv, da diese noch nicht als Archive vorliegen. Der Mobilis-Server sollte in Zukunft die Möglichkeit unterstützen, bereits vorhandene und gepackte Services bei einem Systemstart mit zu laden. Dies kann zum Beispiel erreicht werden, indem alle Archive, die beim Systemstart mit geladen werden sollen, in einen bestimmten Ordner verschoben werden. Diese soll der Mobilis-Server dann beim Start überprüfen und die Services automatisch starten. Des Weiteren wäre es von Vorteil, wenn die aktuellen Zustände der bereits integrierten Services, sowie deren Konfigurationen, persistent gespeichert werden. Das hat zur Folge, dass der Mobilis-Server nach einem Neustart wieder an derselben Stelle fortfahren kann, an der er beendet wurde.

Der *Security Service*, der bereits im Konzept mit integriert wurde, ist in dieser Arbeit nicht mehr implementiert worden. Dafür müsste sowohl der Service an sich, sowie das genaue Kommunikationsprotokoll noch definiert werden.

Zuletzt wäre es hilfreich das Discovery weitaus komplexer zu gestalten, sodass nicht nur nach dem Namespace und der Version eines Service gesucht wird, sondern nach komplexeren Eigenschaften. Diese könnten beispielsweise die Operationen sein, die ein Service unterstützt. Der Vorteil liegt darin, dass Clientanwendungen auch andere Serviceversionen verwenden können, wenn diese einen gemeinsamen Nenner an Operationen aufweisen. Eine weitere Eigenschaft könnte das Kommunikationsverhalten hinsichtlich der Antwortzeit eines Service sein, die maximal unterstützt wird. Damit kann ein Client denjenigen Service auswählen, der am besten zu ihm passt.

## 5.6 Zusammenfassung

In diesem Kapitel wurden die implementierungsspezifischen Details dieser Arbeit aufgezeigt und jeweils ein “Best Practice“-Beispiel gegeben.

Der Schwerpunkt dieses Kapitels lag vor allem in der Vorbereitung und Implementierung eines Mobilis-Service. Dazu wurde detailliert erläutert, wie die Codegenerierung funktioniert und aus welchen Hauptelementen diese besteht. Auch das Packen eines Service spielt eine wesentliche Rolle in diesem Kapitel, da einige Anforderungen an das Jar-Archive gestellt werden, zum Beispiel wie die Ressourcen eingebunden werden und um welche Einträge die Manifest-Datei erweitert werden muss.

Über das Life cycle Management eines Mobilis-Service wurde lediglich ein kleiner Einblick gegeben. Hier sind vor allem die Kommunikationsbeispiele in Anhang A relevant, die zeigen, wie der Life cycle gesteuert werden kann.

In Abschnitt 5.5 wurden zum Schluss diejenigen Aufgaben angesprochen, die noch offen sind. Dazu zählen unter anderem die Realisierung von Message-Stanzas im Bean-Layer der Mobilis-Plattform und die Umsetzung des Security-Service für die Beschränkung von Zugriffsrechten.



# 6 Evaluation

Das Konzept und die Umsetzung der neuen Mobilis-Plattform sollen in diesem Kapitel evaluiert werden. Als Grundlage der Bewertung wird das, bereits vorhandene, *LBG* MobilisXHunt auf die neue Mobilis-Struktur portiert. Dabei sollen als Erstes auf die Testumgebung und die verwendeten Werkzeuge eingegangen werden. Darauffolgend werde die Schritte *MSDL*-Erstellung, Codegenerierung und -anbindung, Packen und das Life-cycle Management des Service betrachtet und diskutiert. Auf den Aktualisierungsmechanismus im Life cycle Management soll in diesem Teil besonders eingegangen werden, indem MobilisXHunt erweitert und während der Laufzeit aktualisiert wird.

## 6.1 Testumgebung

Als Betriebssysteme wurden einmal Microsoft Windows 7 Professional und einmal Ubuntu 10.10 für die Entwicklung und Ausführung verwendet. Als *XMPP*-Server kam der Openfire-Server<sup>1</sup> in der Version 3.6.4 zum Einsatz, der auf derselben Maschine wie der Mobilis-Server ausgeführt wurde. Die Entwicklung des Mobilis-Service, sowie der Betrieb des Mobilis-Server, wurden unter der *Java Virtuel Machine (JVM)* 1.5 und 1.6 getestet. Auf der Clientseite wurde Android in der Version 2.1 und 2.2 eingesetzt und in einem lokalen Emulator betrieben.

Als Entwicklungsumgebung wurde für die Programmierung die Eclipse Java EE IDE 3.7<sup>2</sup> eingesetzt. Für die Erstellung der *MSDL* kam die Anwendung Altova XMLSpy 2012<sup>3</sup> in der 30-tägigen Trailversion zum Einsatz. In XMLSpy selber wurde die *MSDL* erst als *WSDL* 2.0 realisiert, sodass der grafische Editor genutzt werden konnte. Durch die in Anhang B.1 definierten Anpassungen wurde die *WSDL* daraufhin in eine *MSDL* transformiert.

Für die Codegenerierung, anhand der *XSLT*, wurden die Anwendungen Altova XML-Spy und EditiX<sup>4</sup>, sowie das Eclipse-Plugin Orangevolt<sup>5</sup> verwendet. Alle Werkzeuge sind dabei in der Lage gewesen mit *XSLT* 2.0 umzugehen. Als *XSLT*-Parser wurde immer der Apache Xerces<sup>6</sup> verwendet, der mit allen in dieser verwendeten *XSLT* verwendeten Befehlen umgehen konnte.

---

<sup>1</sup> <http://www.igniterealtime.org/projects/openfire/>

<sup>2</sup> <http://www.eclipse.org/>

<sup>3</sup> <http://www.altova.com/xmlspy.html>

<sup>4</sup> <http://free.editix.com/>

<sup>5</sup> <http://www.ibm.com/developerworksopensource/library/os-eclipse-orangevolt/index.html>

<sup>6</sup> <http://xerces.apache.org/>

## 6.2 Erstellung der MSDL

Zu Beginn der Erstellung wurden als Erstes die Request- und Response-Elemente im Abschnitt `types` erstellt, welche die XMPP-Beans repräsentieren. Für die komplexen Datentypen, wie zum Beispiel `PlayerInfo`, wurden in der MSDL `complexType`-Tags angelegt. Im zweiten Schritt wurden die Operationen definiert, die genauso benannt wurden, wie die ehemaligen XMPP-Beans. Zum Beispiel wurde `CreateGameBean` als Operation `CreateGame` realisiert, die nun die XMPP-Beans `CreateGameRequest` und `CreateGameResponse` als Request-Response-Parameter verwendet.

Bei der Übertragung der XMPP-Beans mussten die in Java bekannten Map-Typen, wie zum Beispiel die Tickets-Map, aus der XMPP-Info `AreaInfo`, in eine Liste umgewandelt werden. Dafür wurde ein neuer Datentyp `Ticket` angelegt, der nun das Schlüssel-Wertpaar repräsentiert. Somit wurde aus der Map mit dem Schlüssel `TicketId` und dem Wert `TicketName` eine Liste von Ticketobjekten.

Des Weiteren musste der Verlauf des `PlayerExitBean` geändert werden. Dieses konnte ursprünglich vom Server und vom Client aus mit denselben Parametern gesendet werden. Mit diesem XMPP-Bean war es einerseits möglich, dass ein Spieler das Spiel verlassen konnte und andererseits, dass ein Spieler, vom Moderator oder dem Mobilis-Server, aus dem Spiel entfernt werden konnte (siehe Abbildung 6.1). Das `PlayerExitBean` wird jetzt nur noch verwendet, wenn ein Spieler das Spiel verlassen möchte oder der Moderator einen Spieler entfernen möchte. Das Entfernen selber wird dem Spieler nicht mehr mit dem `PlayerExitBean`, sondern mit einem `GameOverBean` signalisiert (siehe Abbildung 6.2).

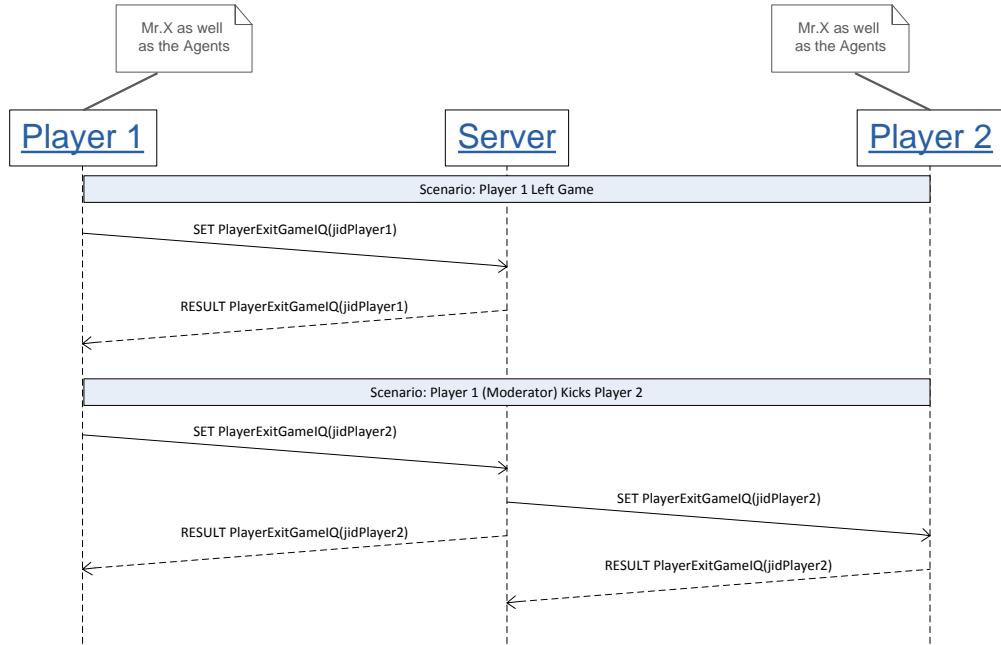
Bei der Umsetzung von Kommunikationsabläufen muss darauf geachtet werden, dass dieselben Nachrichten nicht einmal vom Service und einmal vom Client gleichermaßen gesendet werden können. Diese müssen in separate Operationen verpackt werden und im Binding unterschiedliche Namespaces besitzen.

Mithilfe der Anwendung XMLSpy war die Erstellung, sowie die manuelle Transformation in eine `MSDL`, weitaus schneller und komfortabler als die Erstellung anhand der Schemadatei. In einem direkten Vergleich der `MSDL` des MobilisXHunt-Service, nahm die Schwierigkeit und Komplexität der Erstellung unter Verwendung der Schemadatei sehr schnell zu. Der Vorgang wurde schon nach kurzer Zeit unübersichtlich, da in der XML-Struktur viel umher gesprungen werden musste. Eine visuelle Unterstützung bei der Erstellung wäre für die nahe Zukunft wünschenswert.

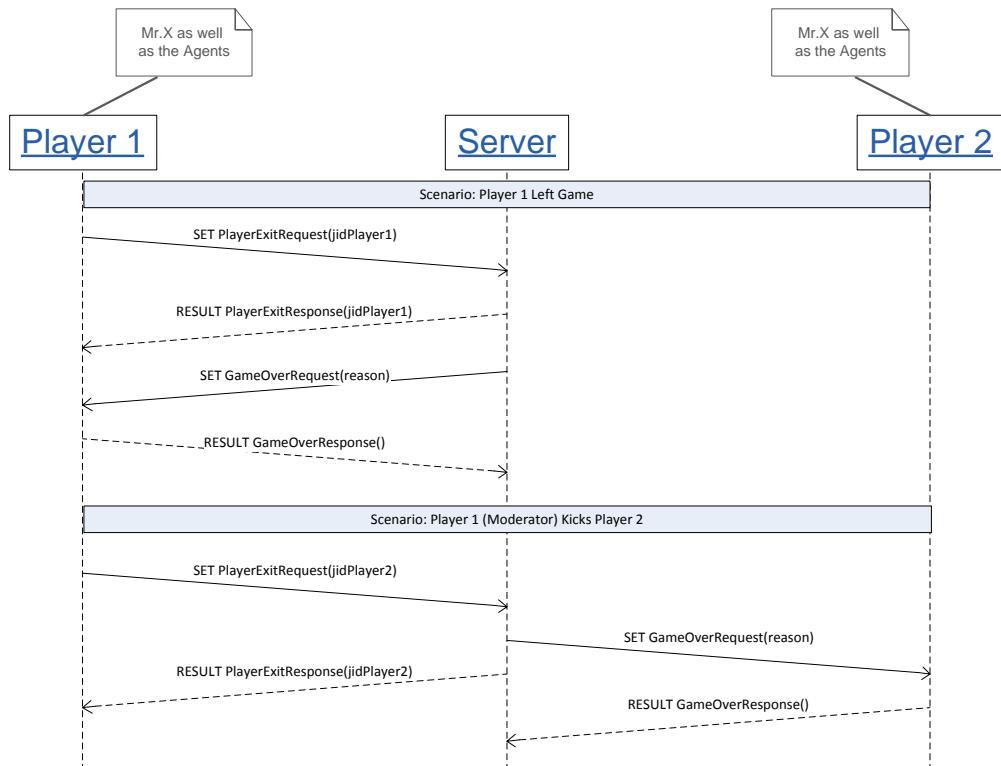
## 6.3 Implementierung des MobilisXHunt-Service

Nachdem die `MSDL` erstellt wurde, kommt der Vorteil der Codegenerierung zu tragen. Circa eine Sekunde dauert die Generierung der 47 Klassen und Interfaces. Der Vorteil dieser Kombination ist vor allem spürbar, wenn ein komplexer Datentyp in einem XMPP-Bean geändert werden soll. Angenommen die Informationen des `TargetBean` sollen durch eine Liste von Targets ersetzt werden, so mussten vorher alle Eigenschaften und Parser-Informationen des `TargetBean` neu geschrieben werden. In der MSDL

### 6.3 Implementierung des MobilisXHunt-Service



**Abbildung 6.1:** Kommunikationsverlauf PlayerExit (alt)



**Abbildung 6.2:** Kommunikationsverlauf PlayerExit (neu)

hingegen müssen lediglich die Elemente in eine neue complexType-Struktur verschoben und eine Referenz vom TargetBean darauf gesetzt werden. Mit dem Hinzufügen

## 6 Evaluation

der Parameter `minOccurs` und `maxOccurs` ist ein neuer Datentyp entstanden, der als Liste verwendet wird. Den Rest übernimmt die Codegenerierung. In diesem konkreten Beispiel führt die Modifizierung in der [MSDL](#) fast 20 mal schneller zum gewünschten Ergebnis als mit der alten Methode.

Ein weiterer wichtiger Aspekt ist die Fehlerminimierung, die durch die Codegenerierung optimiert wird. Die Methoden `fromXML` und `payloadToXML` sind bei einer manuellen Implementierung sehr fehleranfällig. Durch die automatische Generierung wurde dieses Risiko erheblich minimiert, sodass sich der Entwickler nicht mehr darum kümmern muss.

Bei der Umsetzung des MobilisXHunt-Service hat sich jedoch die Aufteilung der XMPP-Beans in Request-Response als ein wenig störend herausgestellt. Aus der [MSDL](#) von MobilisXHunt entstehen aus 16 Operationen 47 Dateien. Aus diesen Dateien sind lediglich die Dateien `MobilisXHuntProxy`, `IMobilisXHuntOutgoing` und `IMobilisXHuntIncoming` von Interesse, da diese für die Anpassung an die Smack-Bibliothek und das Versenden benötigt werden. Die restlichen Dateien können ignoriert werden, da diese implizit verwendet werden und im Normalfall keine manuellen Anpassungen benötigen. Dennoch kann diese Menge an Dateien, gerade in komplexeren Mobilis-Services, unübersichtlich werden und den Code unnötig aufblähen. In Tabelle 6.1 Metriken, wird zum Beispiel die Anzahl der Codezeilen einmal mit und einmal ohne Codegenerierung betrachtet. Für die Bewertung ohne Codegenerierung wurden die früher verwendeten XMPP-Bean-Klassen als Grundlage herangezogen.

Das Interface `IMobilisXHuntIncoming` wurde bei der Umsetzung nicht überall verwendet. Zum Beispiel im Service, wenn nur auf eine Nachricht regiert werden muss, so wie im Zustand `GameStateGameOver`. An dieser Stelle war es unnötig für einen leeren Methodenblock die nicht genutzten Nachrichten zu erstellen.

**Tabelle 6.1:** Codemetriken für die Kommunikation mit und ohne Codegenerierung

Metrik	Ohne Codegenerierung	Mit Codegenerierung
Klassen	18	43
Interfaces	0	4
Codezeilen	2.073	3.615
Codezeilen in Methoden	1.462 (11,1)	2.101 (4,3)
Methoden	486 (10,8)	132 (7,3)
Attribute	52 (2,9)	86 (1,9)

## 6.4 Aktualisierung des MobilisXHunt Service

Eines der wichtigen Ziele dieser Arbeit war es, dass die Aktualisierung eines Mobilis-Service auch während der Laufzeit möglich sein soll und bestehende Service-Instanzen davon nicht beeinflusst werden. Um dies zu testen, wurde MobilisXHunt dahingehend erweitert, dass die Spieler innerhalb der Lobby ihre Tickets tauschen können. Diese Version des MobilisXHunt-Service wird als Version 3 bezeichnet. Für den Test

wurde ein Spiel mit dem MobilisXHunt-Service in der Version 2 erstellt und gestartet. An diesem Spiel haben zwei Spieler teilgenommen. Danach wurde der Service durch die Version 3 aktualisiert. Alle Konfigurationen der alten Version wurden beibehalten und standen somit dem neuen Service zur Verfügung. Nach der Installation und Registrierung der neuen Serviceversion war es wie gewünscht nicht mehr möglich Service-Instanzen der Version 2 zu erstellen. Die Instantiierung der neuen Serviceversion konnte ohne Probleme durchgeführt werden, genauso wie das Testen der neuen Funktion.

Für die Umsetzung der Version 3 des MobilisXHunt-Service wurde die [MSDL](#) um die Operationen *TransferTicket*, zum weitergeben von Tickets und *UpdateTickets*, zur Aktualisierung der Tickets auf der Clientseite, erweitert. Während für den Service ein neues Projekt angelegt wurde, wurde der Client lediglich um die neue Funktionalität erweitert. Der Client ist dabei in der Lage zu prüfen, zu welcher Version des MobilisXHunt-Service er verbunden ist.

In diesem Test konnten zwei Service-Instanzen unterschiedlicher Versionen parallel betrieben und genutzt werden. Des Weiteren wurde in einem weiteren Testdurchlauf der parallele Betrieb beider Serviceversionen erfolgreich durchgeführt, sodass der Client sich aussuchen konnte, welche Version er gerne nutzen möchte. Standardmäßig wurde hierbei immer die neuste Version gewählt.

## 6.5 Zusammenfassung

Anhand der Portierung des MobilisXHunt-Projektes war zu sehen, dass die Anbindung der neuen Kommunikationsstruktur, die durch die Codegenerierung erstellt wurde, teilweise aufwendig ist. Insbesondere, wenn die Methoden zum Versenden von Nachrichten nicht zentral verwendet, sondern überall im Projekt vorhanden sind. Bei einer Neuentwicklung, wie es beim TreasureHunt-Service der Fall war, profitiert der Entwickler aber von dieser Struktur, da Nachrichten somit von einer einheitlichen und zentralen Struktur versendet werden können.

Durch die dynamische Integration von Diensten ist der Mobilis-Server kleiner und kompakter geworden. Das macht es unter anderem möglich diesen schneller auf eine andere Plattform zu verschieben, wie zum Beispiel einer mobilen Plattform. Unter Anpassung der Smack-Bibliothek und der persistenten Datenhaltung ist es durchaus denkbar, dass der Mobilis-Server auf einem Android-Client ausgeführt werden kann.

Wenn man die Anforderungen aus Kapitel 3 betrachtet, so wurden *FA-1 Dynamische Serviceumgebung* und *FA-2 Dienstbeschreibungssprache*, bis auf die Realisierung der Zugriffsbeschränkung, komplett erfüllt. Für den letzten Teil *FA-3 Codegenerierung* war es im Rahmen dieser Arbeit jedoch nicht mehr möglich eine XSLT-Datei für eine zweite Plattform zu entwerfen, sowie ein Test-Framework zu erstellen, um die generierten Codeelemente mit individueller Anwendungslogik automatisiert testen zu können.

Hinsichtlich der nichtfunktionalen Anforderungen konnten alle bis, auf *NF-4 Intuitive Nutzerschnittstellen*, erfüllt werden. Diese Anforderung hat sich im Verlauf dieser Ar-

## *6 Evaluation*

beit jedoch als sehr wichtig herausgestellt um die *MSDL* leichter und strukturierter erstellen zu können und sollte in naher Zukunft realisiert werden.

## 7 Zusammenfassung und Ausblick

Diese Arbeit hat sich das Ziel gesetzt, den Entwicklungsprozess von *Mobile Social Apps* in der Mobilis-Plattform zu optimieren und ein dynamisches Laufzeitverhalten der Servicekomponenten zu realisieren. Dafür wurden in Kapitel 2.5 verwandte Arbeiten betrachtet und die Schwerpunkte eines serviceorientierten Komponentenmodells herausgearbeitet. Anhand dieser Schwerpunkte und der Betrachtung des Entwicklungsprozesses der Mobilis-Plattform in Kapitel 3, wurden die Anforderungen an die neue Mobilis-Plattform definiert, die im wesentlichen zwei Problembereiche abdecken sollen.

Auf der einen Seite ist der Entwicklungsprozess und das Life cycle Management der Dienste in der Ausgangsversion nicht sehr ausgeprägt. Die Mobilis-Services konnten nur in der Implementierungsphase modifiziert und konfiguriert werden, wenn der Mobilis-Server heruntergefahren wurde. Das hatte zur Folge, dass laufende Dienste beendet werden mussten und erst nach der Implementierung wieder gestartet werden konnten. Mit der Einführung der Service-Container in Kapitel 4 ist es nun möglich, Mobilis-Services in einer gepackten Form zum Mobilis-Server hochzuladen und zu deployen. Das gepackte Jar-Archiv enthält alle Klassen, Bibliotheken und Ressourcen, die der Dienst für die Instantiierung benötigt. Dieser Service-Container bietet dem Mobilis-Service damit ein eigenes Life cycle Management, indem das De-/ Installieren, De-/ Registrieren, Konfigurieren und Aktualisieren des Dienstes ermöglicht wird. Diese Aktionen können während der Laufzeit des Mobilis-Servers durchgeführt werden, sodass dieser nicht mehr heruntergefahren werden muss.

Auf der anderen Seite besaßen die Dienste in der Mobilis-Plattform keine explizite Beschreibung, die in einer SOA normalerweise üblich ist. Die Entwicklung des Netzprotokolls war dadurch aufwendig, weil die Dienstbeschreibung anhand des Quellcodes erarbeitet werden musste. Mit der MSDL wurde eine Dienstbeschreibung für die Mobilis-Services eingeführt, die alle notwendigen Komponenten für das Kommunikationsverhalten beschreibt. Des Weiteren wurde diese um eine Codegenerierung, unter Verwendung einer XSLT, erweitert. Dadurch wurden die Fehler bei der manuellen Erstellung minimiert und die Performanz bei Änderungen an dem Kommunikationsprotokoll optimiert. Mit der Codegenerierung hat sich jedoch der Code für die Kommunikation aufgeblättert, wie in Kapitel 6 gezeigt wurde. Der Vorteil dieser Kombination aus Dienstschreibung und Codegenerierung ermöglicht jedoch eine einfache Erstellung der Dienste und Clients für unterschiedliche Plattformen in kürzester Zeit.

Mit dieser Arbeit ist aus der Mobilis-Plattform eine vollständige SOA entstanden. Zudem wurde der gesamte Entwicklungsprozess in der Mobilis-Plattform überarbeitet und konnte auf diesem Weg optimiert werden. Durch diese Umstrukturierung konnte der Mobilis-Server schlanker gestaltet werden, was diesen wiederum interessanter für den mobilen Einsatz macht. Somit ist an einigen Stellen noch ausreichend Entwick-

## 7 Zusammenfassung und Ausblick

lungspotential vorhanden.

Ein erster Ansatz bestünde in der Erweiterung der *MSDL* um zusätzliche Attribute, die das Kommunikationsverhalten erweitern. Dabei sind im Speziellen die Extra-functional Properties gemeint, die schon in [15] definiert und in Kapitel 2.5 mit vorgestellt wurden. Zum Beispiel die Einführung eines Time-out für Antwortzeiten ist gerade im mobilen Bereich ein nützlicher Aspekt. Dies ermöglicht eine einheitliche Implementierung und eine zuverlässigere Kommunikation von Client und Service. Weiterhin wäre auch interessant, welche Elemente der *MSDL* sich bei einer Aktualisierung der Dienstversion geändert haben. Dies ermöglicht einen gewissen Grad an Automatismus bei der späteren Dienstauswahl, sowie die einfachere Unterstützung mehrerer Dienstversionen aus der Sicht der Clientanwendung.

Einen Schritt weiter, in Richtung Automatismus, geht die Umsetzung einer kontext-sensitiven Clientanwendung. Dabei stellt die *MXA*-Anwendung den Ausgangspunkt einer Clientanwendung dar. Der Client könnte mithilfe des *MXA* alle auf einem Mobilis-Server laufenden Mobilis-Services abfragen. Möchte der Client einen Mobilis-Service nutzen, könnte er dies dem Mobilis-Server, über das *MXA*, mitteilen. Anhand der Kontextinformationen des Clients kann der Mobilis-Server eine bestimmte Version eines Mobilis-Service auswählen und eventuell kommunikationsspezifische Attribute einstellen. Dem Client kann dann über einen File-Transfer der bereits kompilierte Mobilis-Client übertragen werden. Dieser muss dann nur noch vom Client installiert werden und steht fortan für die Benutzung zur Verfügung.

Die Realisierung einer *P2P*-Version der Mobilis-Plattform, nach den Vorgaben aus [26], würde den höchsten Grad an Dynamik schaffen. Durch die Konzentration auf Systemdienste und die Integration von zusätzlichen Diensten, ist der Mobilis-Server um ein ganzes Stück kleiner und mobiler geworden. Es ist vorstellbar, dass dieser komplett auf einem Client zum Laufen gebracht werden könnte, um ein *P2P*-Netzwerk zu realisieren. Dabei müsste ein Client einen Super-Peer repräsentieren und die Funktionen des Mobilis-Servers übernehmen, wie sie in der aktuellen Version vorgenommen werden. Dieser Super-Peer sollte möglichst der leistungsstärkste im Netzwerk sein, da dieser weit mehr Nachrichten empfangen und wieder versenden muss als die anderen Peers. In [42] wird das *XEP-174 Serverless Messaging* vorgestellt, dass die Voraussetzungen für eine derartige Architektur mitbringen könnte. Dafür müsste der Mobilis-Server dahingehend weiterentwickelt werden, dass dieser auch die Registrierung externer Mobilis-Services ermöglicht, wie es in Kapitel 4.1.2 angesprochen wurde. Somit könnten zwei verschiedene Mobilis-Server ihre Dienste gegenseitig registrieren und die Dienste des jeweiligen anderen nutzen.

# A System Services Paket-Beispiele

## Deployment Service

```
<iq id="mobilis_18" to="mobilis@xhunt/Deployment" type="set">
    <prepareServiceUpload
        xmlns="http://mobilis.inf.tu-dresden.de#XMPPBeans:admin:prepareServiceUpload">
        <filename>TreasureHuntService_v1.jar</filename>
        <acceptServiceUpload>true</acceptServiceUpload>
    </prepareServiceUpload>
</iq>
```

Abbildung A.1: PrepareServiceUpload SET

```
<iq id="mobilis_18" to="consoleclient@xhunt/JavaClient" from="mobilis@xhunt/Deployment"
    type="result">
    <prepareServiceUpload
        xmlns="http://mobilis.inf.tu-dresden.de#XMPPBeans:admin:prepareServiceUpload">
        <acceptServiceUpload>true</acceptServiceUpload>
    </prepareServiceUpload>
</iq>
```

Abbildung A.2: PrepareServiceUpload RESULT

```
<iq id="mobilis_16" to="consoleclient@xhunt/JavaClient" from="mobilis@xhunt/Deployment"
    type="set">
    <serviceUploadConclusion
        xmlns="http://mobilis.inf.tu-dresden.de#XMPPBean:deployment:serviceUploadConclusion">
        <uploadSuccessful>true</uploadSuccessful>
        <filename>TreasureHuntService_v1.jar</filename>
    </serviceUploadConclusion>
</iq>
```

Abbildung A.3: ServiceUploadConclusion SET

```
<iq id="mobilis_16" to="mobilis@xhunt/Deployment" from="consoleclient@xhunt/JavaClient"
    type="result">
    <serviceUploadConclusion
        xmlns="http://mobilis.inf.tu-dresden.de#XMPPBean:deployment:serviceUploadConclusion" />
</iq>
```

Abbildung A.4: ServiceUploadConclusion RESULT

## Admin Service

```
<iq id="mobilis_21" to="mobilis@xhunt/Admin" type="set">
  <installService
    xmlns="http://mobilis.inf.tu-dresden.de#XMPPBeans:admin:installService">
      <filename>TreasureHuntService_v1.jar</filename>
    </installService>
</iq>
```

Abbildung A.5: InstallService SET

```
<iq id="mobilis_18" to="consoleclient@xhunt/JavaClient" from="mobilis@xhunt/Admin"
  type="result">
  <installService
    xmlns="http://mobilis.inf.tu-dresden.de#XMPPBeans:admin:installService">
      <installationsuccessful>true</installationsuccessful>
    <servicename>http://mobilis.inf.tu-dresden.de#services/TreasureHuntService
    </servicename>
      <serviceversion>1</serviceversion>
    </installService>
</iq>
```

Abbildung A.6: InstallService RESULT

```
<iq id="mobilis_23" to="mobilis@xhunt/Admin" type="set">
  <configureService
    xmlns="http://mobilis.inf.tu-dresden.de#XMPPBeans:admin:configureService">
    <agentConfig>
      <agentConfig>
        <description>TreasureHuntService</description>
        <host>localhost</host>
        <mode>multi</mode>
        <name>TreasureHuntService</name>
        <password>WwmSdzZWf.</password>
        <port>5222</port>
        <resource>TreasureHuntService</resource>
        <service>xhunt</service>
        <start>ondemand</start>
      </agentConfig>
    <type>de.tudresden.inf.rn.mobilis.server.agents.MobilisAgent</type>
    <username>mobilis</username>
  </agentConfig>
  <serviceNamespace>http://mobilis.inf.tu-dresden.de#services/TreasureHuntService
  </serviceNamespace>
  <serviceVersion>1</serviceVersion>
</configureService>
</iq>
```

Abbildung A.7: ConfigureService SET

```

<iq id="mobilis_25" to="consoleclient@xhunt/JavaClient" from="mobilis@xhunt/Admin" type="result">
<configureService xmlns="http://mobilis.inf.tu-dresden.de#beans/admin/configureService" />
</iq>
```

**Abbildung A.8:** ConfigureService RESULT

```

<register
  xmlns="http://mobilis.inf.tu-dresden.de#services/TreasureHuntService" >
  <servicename>TreasureHuntService</servicename>
  <serviceversion>1</serviceversion>
</register>
</iq>
```

**Abbildung A.9:** RegisterService SET

```

<iq id="mobilis_25" to="consoleclient@xhunt/JavaClient" from="mobilis@xhunt/Admin" type="result">
<register
  xmlns="http://mobilis.inf.tu-dresden.de#XMPPBeans:admin:RegisterService" />
</iq>
```

**Abbildung A.10:** RegisterService RESULT

```

<iq id="mobilis_31" to="mobilis@xhunt/Admin" type="set">
<updateService
  xmlns="http://mobilis.inf.tu-dresden.de#XMPPBeans:admin:updateService">
  <filename>TreasureHuntService_v2.jar</filename>
<oldservicename>TreasureHuntService</oldservicename>
  <oldserviceversion>1</oldserviceversion>
</updateService>
</iq>
```

**Abbildung A.11:** UpdateService SET

```

<iq id="mobilis_31" to="consoleclient@xhunt/JavaClient" from="mobilis@xhunt/Admin" type="result">
<updateService
  xmlns="http://mobilis.inf.tu-dresden.de#XMPPBeans:admin:updateService">
<newservicename>TreasureHuntService</newservicename>
  <newserviceversion>2</newserviceversion>
</updateService>
</iq>
```

**Abbildung A.12:** UpdateService RESULT

## A System Services Paket-Beispiele

```
<iq id="mobilis_57" to="mobilis@xhunt/Admin" type="set">
    <unregister
        xmlns="http://mobilis.inf.tu-dresden.de#XMPPBeans:admin:UnregisterService">
        <servicename>http://mobilis.inf.tu-dresden.de#services/TreasureHuntService
        </servicename>
        <serviceversion>1</serviceversion>
    </unregister>
</iq>
```

**Abbildung A.13:** UnregisterService SET

```
<iq id="mobilis_57" to="consoleclient@xhunt/JavaClient" from="mobilis@xhunt/Admin"
    type="result">
    <unregister
        xmlns="http://mobilis.inf.tu-dresden.de#XMPPBeans:admin:UnregisterService" />
</iq>
```

**Abbildung A.14:** UnregisterService RESULT

```
<iq id="mobilis_24" to="mobilis@xhunt/Admin" type="set">
    <uninstall
        xmlns="http://mobilis.inf.tu-dresden.de#XMPPBeans:admin:UninstallService">
        <servicename>http://mobilis.inf.tu-dresden.de#services/TreasureHuntService
        </servicename>
        <serviceversion>2</serviceversion>
    </uninstall>
</iq>
```

**Abbildung A.15:** UninstallService SET

```
<iq id="mobilis_24" to="mobilis@xhunt/Admin" type="result">
    <uninstall
        xmlns="http://mobilis.inf.tu-dresden.de#XMPPBeans:admin:UninstallService" />
</iq>
```

**Abbildung A.16:** UninstallService RESULT

## Coordinator Service

```
<iq id="mobilis_23" to="mobilis@xhunt/Coordinator"
    from="consoleclient@xhunt/JavaClient"
    type="set">
    <stopServiceInstance
        xmlns="http://mobilis.inf.tu-dresden.de#services/CoordinatorService">
        <servicejid>mobilis@xhunt/TreasureHunt_v1#2</servicejid>
    </stopServiceInstance>
</iq>
```

**Abbildung A.17:** StopServiceInstance SET

```
<iq id="mobilis_23" to="consoleclient@xhunt/JavaClient"
from="mobilis@xhunt/Coordinator"
    type="result">
    <stopServiceInstance
        xmlns="http://mobilis.inf.tu-dresden.de#services/CoordinatorService" />
</iq>
```

**Abbildung A.18:** StopServiceInstance RESULT



## B MSDL

### B.1 Transformation WSDL zu MSDL

- Hinzufügen der Namespaces `xmlns:msdl="http://mobilis.inf.tu-dresden.de/msdl/"` und `xmlns:msdl="http://mobilis.inf.tu-dresden.de/xmpp/"` im Abschnitt `description`
- Umbenennung aller `wsdl`-Tags in `msdl`-Tags
- Erweitern der `operation`-Elemente im Abschnitt `binding` um das Attribut `xmpp:ident`, sowie die `input` und `output`-Elemente um das Attribut `type`
- Erweiterung der `fault`-Elemente im Abschnitt `binding` um die Attribute `xmpp:errortype`, `xmpp:errorcondition` und `xmpp:errortext`
- Erweiterung des `service`-Elements um die Attribute `ident` und `version`
- Definition der Service-Abhängigkeiten `dependencies` im Abschnitt `service`

Alle weiteren Attribute wie zum Beispiel das Attribut `address` im Abschnitt `endpoint` sollten für eine bessere Übersichtlichkeit entfernt werden, da diese bei der Codegenerierung nicht mit beachtet werden.

### B.2 Schema

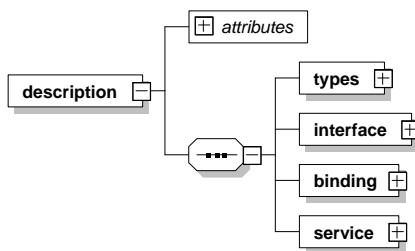
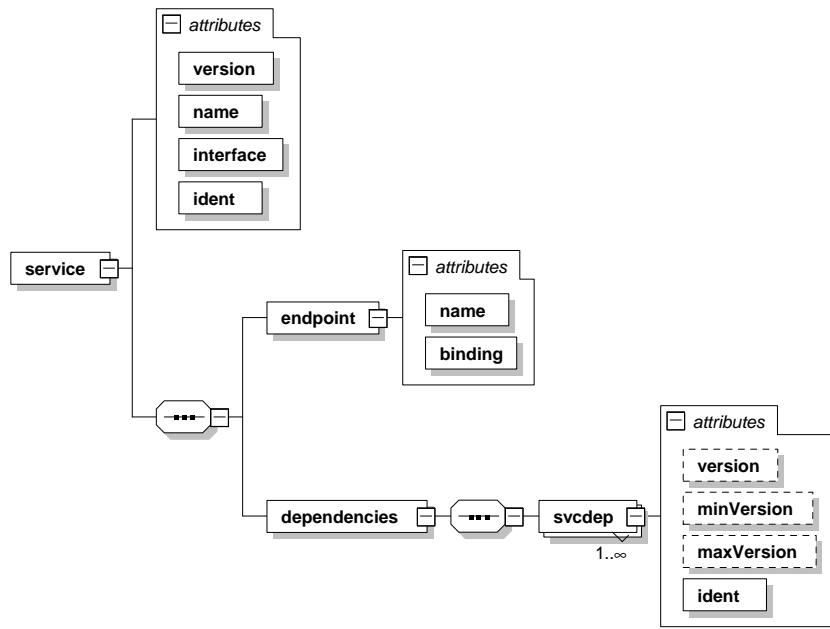
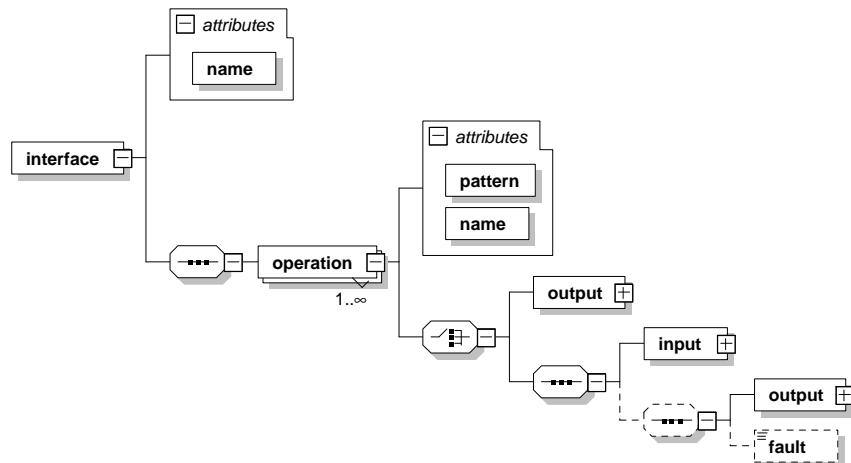


Abbildung B.1: MSDL Schema Grobstruktur

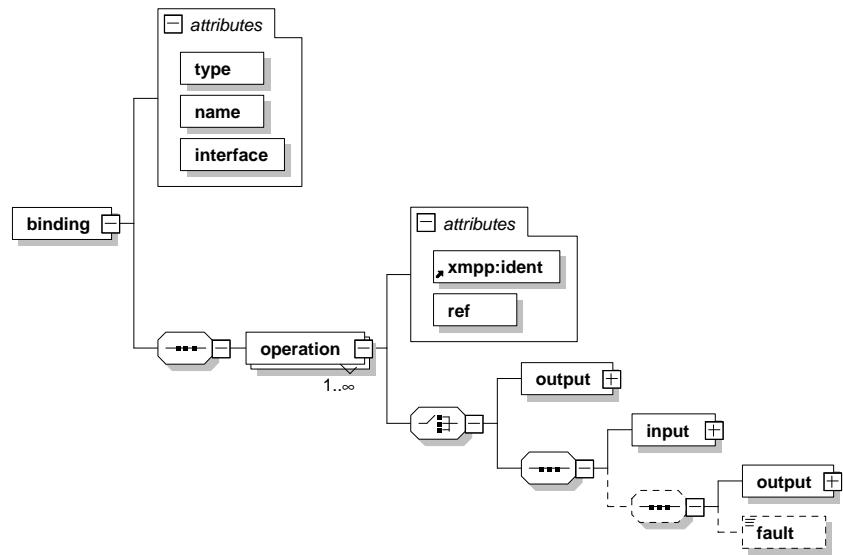
## B MSDL



**Abbildung B.2:** MSDL Schema Abschnitt Service



**Abbildung B.3:** MSDL Schema Abschnitt Interface



**Abbildung B.4:** MSDL Schema Abschnitt Binding

### B.3 MSDL TreasureHunt

```

<?xml version="1.0" encoding="UTF-8"?>
<msdl:description
    targetNamespace="http://mobilis.inf.tu-dresden.de/TreasureHunt"
    xmlns:msdl="http://mobilis.inf.tu-dresden.de/msdl/"
    xmlns:xmpp="http://mobilis.inf.tu-dresden.de/xmpp/"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:tns="http://mobilis.inf.tu-dresden.de/TreasureHunt">

    <msdl:types>
        <xss:schema
            targetNamespace="http://mobilis.inf.tu-dresden.de/TreasureHunt">
                <xss:element name="PickUpTreasureRequest">
                    <xss:complexType>
                        <xss:sequence>
                            <xss:element name="TreasureLocation"
                                type="tns:Location" />
                        </xss:sequence>
                    </xss:complexType>
                </xss:element>
                <xss:element name="PickUpTreasureResponse">
                    <xss:complexType>
                        <xss:sequence>
                            <xss:element name="TreasureValue"
                                type="xs:long" />
                        </xss:sequence>
                    </xss:complexType>
                </xss:element>
                <xss:element name="GetLocationRequest" />
                <xss:element name="GetLocationResponse">
                    <xss:complexType>
                        <xss:sequence>
                            <xss:element name="PlayerLocation"
                                type="tns:Location" />
                        </xss:sequence>
                    </xss:complexType>
                </xss:element>
                <xss:element name="TreasureCollected">
                    <xss:complexType>
                        <xss:sequence>
                            <xss:element name="PlayerName"
                                type="xs:string" />
                            <xss:element name="TreasureLocation"
                                type="tns:Location" />
                            <xss:element name="TreasureValue"
                                type="xs:long" />
                        </xss:sequence>
                    </xss:complexType>
                </xss:element>
                <xss:element name="PickUpTreasureFault">
                    <xss:complexType>
                        <xss:sequence>
                            <xss:element name="ErrorCode" type="xs:int" />
                            <xss:element name="Timestamp" type="xs:long" />
                        </xss:sequence>
                    </xss:complexType>
                </xss:element>
                <xss:complexType name="Location">
                    <xss:sequence>
                        <xss:element name="Latitude" type="xs:long" />
                        <xss:element name="Longitude" type="xs:long" />
                    </xss:sequence>
                </xss:complexType>
            </xss:schema>
        </msdl:types>
    
```

```

<msdl:interface name="THInterface">
    <msdl:fault name="PickUpFault"
        element="tns:PickUpTreasureFault" />
    <msdl:fault name="GetLocationFault" />
    <msdl:operation name="PickUpTreasure"
        pattern="http://www.w3.org/ns/wsdl/in-out">
        <msdl:input element="tns:PickUpTreasureRequest" />
        <msdl:output element="tns:PickUpTreasureResponse" />
        <msdl:outfault ref="tns:PickUpFault" />
    </msdl:operation>
    <msdl:operation name="GetLocation"
        pattern="http://www.w3.org/ns/wsdl/out-in">
        <msdl:input element="tns:GetLocationResponse" />
        <msdl:output element="tns:GetLocationRequest" />
        <msdl:infault ref="tns:GetLocationFault" />
    </msdl:operation>
    <msdl:operation name="TreasureCollected"
        pattern="http://www.w3.org/ns/wsdl/out-only">
        <msdl:output element="tns:TreasureCollected" />
    </msdl:operation>
</msdl:interface>

<msdl:binding name="THBinding" interface="tns:THInterface"
    type="http://mobilis.inf.tu-dresden.de/xmpp/">
    <msdl:fault ref="tns:PickUpFault" xmpp:errortype="modify"
        xmpp:errorcondition="not-acceptable"
        xmpp:errortext="This Treasure is not valid" />
    <msdl:fault ref="tns:GetLocationFault"
        xmpp:errortype="wait"
        xmpp:errorcondition="resource-constraint"
        xmpp:errortext="Waiting for Location" />

    <msdl:operation ref="tns:PickUpTreasure"
        xmpp:ident="treasurehunt:iq:pickuptreasure">
        <msdl:input xmpp:type="set" />
        <msdl:output xmpp:type="result" />
        <msdl:outfault ref="tns:PickUpFault" />
    </msdl:operation>
    <msdl:operation ref="tns:GetLocation"
        xmpp:ident="treasurehunt:iq:getlocation">
        <msdl:input xmpp:type="result" />
        <msdl:output xmpp:type="get" />
        <msdl:infault ref="tns:GetLocationFault" />
    </msdl:operation>
    <msdl:operation ref="tns:TreasureCollected"
        xmpp:ident="treasurehunt:iq:treasurecollected">
        <msdl:output xmpp:type="chat" />
    </msdl:operation>
</msdl:binding>

<msdl:service name="TreasureHunt" interface="tns:THInterface"
    ident="http://mobilis.inf.tu-dresden.de#services/TreasureHuntService"
    version="1">
    <msdl:endpoint name="THEndpoint" binding="tns:THBinding" />

    <msdl:dependencies>
        <msdl:svcddep
            ident="http://mobilis.inf.tu-dresden.de#services/GroupingService"
            version="1" />
    </msdl:dependencies>
</msdl:service>

</msdl:description>

```

## B.4 MSDL Mobilis XHunt

```

<?xml version="1.0" encoding="UTF-8"?>
<msdl:description targetNamespace="http://mobilis.inf.tu-dresden.de/MobilisXHunt"
  xmlns:msdl="http://mobilis.inf.tu-dresden.de/msdl/"
  xmlns:xmpp="http://mobilis.inf.tu-dresden.de/xmpp/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="http://mobilis.inf.tu-dresden.de/MobilisX">

  <msdl:types>
    <xss:schema
      targetNamespace="http://mobilis.inf.tu-dresden.de/MobilisXHunt">
        <xss:complexType name="AreaInfo">
          <xss:sequence>
            <xss:element name="AreaId" type="xs:int"/>
            <xss:element name="AreaName" type="xs:string"/>
            <xss:element name="AreaDescription" type="xs:string"/>
            <xss:element name="Version" type="xs:int"/>
            <xss:element name="Tickets" type="tns:Ticket"
              minOccurs="0" maxOccurs="unbounded"/>
          </xss:sequence>
        </xss:complexType>
        <xss:element name="AreasRequest"/>
        <xss:element name="AreasResponse">
          <xss:complexType>
            <xss:sequence>
              <xss:element name="Areas" type="tns:AreaInfo"
                minOccurs="0" maxOccurs="unbounded"/>
            </xss:sequence>
          </xss:complexType>
        </xss:element>
        <xss:element name="CancelTimerRequest"/>
        <xss:element name="CancelTimerResponse"/>
        <xss:element name="CreateGameRequest">
          <xss:complexType>
            <xss:sequence>
              <xss:element name="AreaId" type="xs:int"/>
              <xss:element name="GameName" type="xs:string"/>
              <xss:element name="GamePassword" type="xs:string"/>
              <xss:element name="CountRounds" type="xs:int"/>
              <xss:element name="MinPlayers" type="xs:int"/>
              <xss:element name="MaxPlayers" type="xs:int"/>
              <xss:element name="StartTimer" type="xs:int"/>
              <xss:element name="TicketsMrX" type="tns:TicketAmount"
                minOccurs="0" maxOccurs="unbounded"/>
              <xss:element name="TicketsAgents" type="tns:TicketAmount"
                minOccurs="0" maxOccurs="unbounded"/>
            </xss:sequence>
          </xss:complexType>
        </xss:element>
        <xss:element name="CreateGameResponse"/>
        <xss:element name="DepartureDataRequest">
          <xss:complexType>
            <xss:sequence>
              <xss:element name="StationId" type="xs:int"/>
            </xss:sequence>
          </xss:complexType>
        </xss:element>
        <xss:element name="DepartureDataResponse">
          <xss:complexType>
            <xss:sequence>
              <xss:element name="Departures" type="tns:DepartureInfo"
                minOccurs="0" maxOccurs="unbounded"/>
            </xss:sequence>
          </xss:complexType>
        </xss:element>
      </xss:schema>
    </msdl:types>
  </msdl:description>

```

```
<xs:complexType name="DepartureInfo">
  <xs:sequence>
    <xs:element name="VehicleId" type="xs:int"/>
    <xs:element name="VehicleName" type="xs:string"/>
    <xs:element name="Direction" type="xs:string"/>
    <xs:element name="TimeLeft" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```

```

<xs:element name="LocationResponse">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="LocationInfo"
                type="tns:LocationInfo"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="PlayerExitRequest">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="Jid" type="xs:string"/>
            <xs:element name="IsSpectator" type="xs:boolean"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="PlayerExitResponse">
<xs:complexType name="PlayerInfo">
    <xs:sequence>
        <xs:element name="Jid" type="xs:string"/>
        <xs:element name="PlayerName" type="xs:string"/>
        <xs:element name="IsModerator" type="xs:boolean"/>
        <xs:element name="IsReady" type="xs:boolean"/>
        <xs:element name="IsReady" type="xs:boolean"/>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="PlayerSnapshotInfo">
    <xs:sequence>
        <xs:element name="PlayerInfo" type="tns:PlayerInfo"/>
        <xs:element name="Location" type="tns:LocationInfo"/>
        <xs:element name="IsTargetFinal" type="xs:boolean"/>
        <xs:element name="TargetId" type="xs:int"/>
        <xs:element name="TargetReached" type="xs:boolean"/>
        <xs:element name="LastStationId" type="xs:int"/>
    </xs:sequence>
</xs:complexType>
<xs:element name="PlayersRequest">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="Players" type="tns:PlayerInfo"
                minOccurs="1" maxOccurs="unbounded"/>
            <xs:element name="Info" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="PlayersResponse"/>
<xs:complexType name="RoundStatusInfo">
    <xs:sequence>
        <xs:element name="PlayerJid" type="xs:string"/>
        <xs:element name="IsTargetFinal" type="xs:boolean"/>
        <xs:element name="TargetId" type="xs:int"/>
        <xs:element name="TargetReached" type="xs:boolean"/>
    </xs:sequence>
</xs:complexType>
<xs:element name="RoundStatusRequest">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="Round" type="xs:int"/>
            <xs:element name="RoundStatusInfos"
                type="tns:RoundStatusInfo" minOccurs="1"
                maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>

```

```

<xs:element name="RoundStatusResponse"/>
<xs:element name="SnapshotRequest">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="GameName" type="xs:string"/>
            <xs:element name="Round" type="xs:int"/>
            <xs:element name="IsRoundStart" type="xs:boolean"/>
            <xs:element name="ShowMrX" type="xs:boolean"/>
            <xs:element name="StartTimer" type="xs:int"/>
            <xs:element name="Tickets" type="tns:TicketAmount"
                minOccurs="0" maxOccurs="unbounded"/>
            <xs:element name="PlayerSnapshots"
                type="tns:PlayerSnapshotInfo" minOccurs="0"
                maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="SnapshotResponse"/>
<xs:element name="StartRoundRequest">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="Round" type="xs:int"/>
            <xs:element name="ShowMrX" type="xs:boolean"/>
            <xs:element name="Tickets" type="tns:TicketAmount"
                minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="StartRoundResponse"/>
<xs:element name="TargetRequest">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="StationId" type="xs:int"/>
            <xs:element name="Round" type="xs:int"/>
            <xs:element name="TicketId" type="xs:int"/>
            <xs:element name="IsFinal" type="xs:boolean"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="TargetResponse">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="TicketId" type="xs:int"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:complexType name="Ticket">
    <xs:sequence>
        <xs:element name="ID" type="xs:int"/>
        <xs:element name="Name" type="xs:string"/>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="TicketAmount">
    <xs:sequence>
        <xs:element name="ID" type="xs:int"/>
        <xs:element name="Amount" type="xs:int"/>
    </xs:sequence>
</xs:complexType>
<xs:element name="UpdatePlayerRequest">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="PlayerInfo"
                type="tns:PlayerInfo"/>
        </xs:sequence>
    </xs:complexType>

```

## B MSVL

```
</xs:element>
<xs:element name="UpdatePlayerResponse">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="Info" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:complexType name="UsedTicketsInfo">
    <xs:sequence>
        <xs:element name="Jid" type="xs:string"/>
        <xs:element name="TicketIds" type="xs:int"
                    minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
</xs:complexType>
<xs:element name="UsedTicketsRequest"/>
<xs:element name="UsedTicketsResponse">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="UsedTickets"
                        type="tns:UsedTicketsInfo"
                        minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
</xs:schema>
</msdl:types>

<msdl:interface name="MXHInterface">
    <msdl:operation name="Areas" pattern="http://www.w3.org/ns/wsdl/in-out">
        <msdl:input element="tns:AreasRequest"/>
        <msdl:output element="tns:AreasResponse"/>
    </msdl:operation>
    <msdl:operation name="CancelStartTimer"
                    pattern="http://www.w3.org/ns/wsdl/in-out">
        <msdl:input element="tns:CancelTimerRequest" />
        <msdl:output element="tns:CancelTimerResponse" />
    </msdl:operation>
    <msdl:operation name="CreateGame"
                    pattern="http://www.w3.org/ns/wsdl/in-out">
        <msdl:input element="tns>CreateGameRequest" />
        <msdl:output element="tns>CreateGameResponse" />
        <msdl:outfault ref="tns:InputDataFault" />
    </msdl:operation>
    <msdl:operation name="DepartureData"
                    pattern="http://www.w3.org/ns/wsdl/in-out">
        <msdl:input element="tns:DepartureDataRequest" />
        <msdl:output element="tns:DepartureDataResponse" />
    </msdl:operation>
    <msdl:operation name="GameDetails"
                    pattern="http://www.w3.org/ns/wsdl/in-out">
        <msdl:input element="tns:GameDetailsRequest" />
        <msdl:output element="tns:GameDetailsResponse" />
    </msdl:operation>
    <msdl:operation name="GameOver"
                    pattern="http://www.w3.org/ns/wsdl/out-in">
        <msdl:input element="tns:GameOverResponse" />
        <msdl:output element="tns:GameOverRequest" />
    </msdl:operation>
    <msdl:operation name="JoinGame"
                    pattern="http://www.w3.org/ns/wsdl/in-out">
        <msdl:input element="tns:JoinGameRequest" />
        <msdl:output element="tns:JoinGameResponse" />
        <msdl:outfault ref="tns:ClosedGameFault" />
    </msdl:operation>
```

```

<msdl:operation name="Location"
    pattern="http://www.w3.org/ns/wsdl/out-in">
    <msdl:input element="tns:LocationResponse" />
    <msdl:output element="tns:LocationRequest" />
</msdl:operation>
<msdl:operation name="PlayerExit"
    pattern="http://www.w3.org/ns/wsdl/in-out">
    <msdl:input element="tns:PlayerExitRequest" />
    <msdl:output element="tns:PlayerExitResponse" />
    <msdl:outfault ref="tns:PermissionFault" />
</msdl:operation>
<msdl:operation name="Players"
    pattern="http://www.w3.org/ns/wsdl/out-in">
    <msdl:input element="tns:PlayersResponse" />
    <msdl:output element="tns:PlayersRequest" />
</msdl:operation>
<msdl:operation name="RoundStatus"
    pattern="http://www.w3.org/ns/wsdl/out-in">
    <msdl:input element="tns:RoundStatusResponse" />
    <msdl:output element="tns:RoundStatusRequest" />
</msdl:operation>
<msdl:operation name="Snapshot"
    pattern="http://www.w3.org/ns/wsdl/out-in">
    <msdl:input element="tns:SnapshotResponse" />
    <msdl:output element="tns:SnapshotRequest" />
</msdl:operation>
<msdl:operation name="StartRound"
    pattern="http://www.w3.org/ns/wsdl/out-in">
    <msdl:input element="tns:StartRoundResponse" />
    <msdl:output element="tns:StartRoundRequest" />
</msdl:operation>
<msdl:operation name="Target"
    pattern="http://www.w3.org/ns/wsdl/in-out">
    <msdl:input element="tns:TargetRequest" />
    <msdl:output element="tns:TargetResponse" />
    <msdl:outfault ref="tns:InvalidTargetFault" />
    <msdl:outfault ref="tns:PlayerSynchronizationFault" />
</msdl:operation>
<msdl:operation name="UpdatePlayer"
    pattern="http://www.w3.org/ns/wsdl/in-out">
    <msdl:input element="tns:UpdatePlayerRequest" />
    <msdl:output element="tns:UpdatePlayerResponse" />
    <msdl:outfault ref="tns:PermissionFault" />
    <msdl:outfault ref="tns:InputDataFault" />
</msdl:operation>
<msdl:operation name="UsedTickets"
    pattern="http://www.w3.org/ns/wsdl/in-out">
    <msdl:input element="tns:UsedTicketsRequest" />
    <msdl:output element="tns:UsedTicketsResponse" />
</msdl:operation>
<msdl:fault name="ClosedGameFault" />
<msdl:fault name="PermissionFault" />
<msdl:fault name="InvalidTargetFault" />
<msdl:fault name="InputDataFault" />
<msdl:fault name="PlayerSynchronizationFault" />
</msdl:interface>

<msdl:binding name="MXHBinding" interface="tns:MXHInterface"
    type="http://mobilis.inf.tu-dresden.de/xmpp/">
    <msdl:operation ref="tns:Areas" xmpp:ident="mobilisxhunt:iq:areas">
        <msdl:input xmpp:type="get" />
        <msdl:output xmpp:type="result" />
    </msdl:operation>

```

## B MSDL

```
<msdl:operation ref="tns:CancelStartTimer"
    xmpp:ident="mobilisxhunt:iq:cancelstarttimer">
    <msdl:input xmpp:type="set" />
    <msdl:output xmpp:type="result" />
</msdl:operation>
<msdl:operation ref="tns>CreateGame"
    xmpp:ident="mobilisxhunt:iq:creategame">
    <msdl:input xmpp:type="set" />
    <msdl:output xmpp:type="result" />
    <msdl:outfault ref="tns:InputDataFault" />
</msdl:operation>
<msdl:operation ref="tns:DepartureData"
    xmpp:ident="mobilisxhunt:iq:departure">
    <msdl:input xmpp:type="get" />
    <msdl:output xmpp:type="result" />
</msdl:operation>
<msdl:operation ref="tns:GameDetails"
    xmpp:ident="mobilisxhunt:iq:gamedetails">
    <msdl:input xmpp:type="get" />
    <msdl:output xmpp:type="result" />
</msdl:operation>
<msdl:operation ref="tns:GameOver"
    xmpp:ident="mobilisxhunt:iq:gameover">
    <msdl:input xmpp:type="result" />
    <msdl:output xmpp:type="set" />
</msdl:operation>
<msdl:operation ref="tns:JoinGame"
    xmpp:ident="mobilisxhunt:iq:joingame">
    <msdl:input xmpp:type="set" />
    <msdl:output xmpp:type="result" />
    <msdl:outfault ref="tns:ClosedGameFault" />
</msdl:operation>
<msdl:operation ref="tns:Location"
    xmpp:ident="mobilisxhunt:iq:location">
    <msdl:input xmpp:type="result" />
    <msdl:output xmpp:type="set" />
</msdl:operation>
<msdl:operation ref="tns:PlayerExit"
    xmpp:ident="mobilisxhunt:iq:playerexit">
    <msdl:input xmpp:type="set" />
    <msdl:output xmpp:type="result" />
    <msdl:outfault ref="tns:PermissionFault" />
</msdl:operation>
<msdl:operation ref="tns:Players"
    xmpp:ident="mobilisxhunt:iq:players">
    <msdl:input xmpp:type="result" />
    <msdl:output xmpp:type="set" />
</msdl:operation>
<msdl:operation ref="tns:RoundStatus"
    xmpp:ident="mobilisxhunt:iq:roundstatus">
    <msdl:input xmpp:type="result" />
    <msdl:output xmpp:type="set" />
</msdl:operation>
<msdl:operation ref="tns:Snapshot"
    xmpp:ident="mobilisxhunt:iq:snapshot">
    <msdl:input xmpp:type="result" />
    <msdl:output xmpp:type="set" />
</msdl:operation>
<msdl:operation ref="tns:StartRound"
    xmpp:ident="mobilisxhunt:iq:startround">
    <msdl:input xmpp:type="result" />
    <msdl:output xmpp:type="set" />
</msdl:operation>
```

```

<msdl:operation ref="tns:Target"
    xmpp:ident="mobilisxhunt:iq:target">
    <msdl:input xmpp:type="set" />
    <msdl:output xmpp:type="result" />
    <msdl:outfault ref="tns:InvalidTargetFault" />
    <msdl:outfault ref="tns:PlayerSynchronizationFault" />
</msdl:operation>
<msdl:operation ref="tns:UpdatePlayer"
    xmpp:ident="mobilisxhunt:iq:updateplayer">
    <msdl:input xmpp:type="set" />
    <msdl:output xmpp:type="result" />
    <msdl:outfault ref="tns:PermissionFault" />
    <msdl:outfault ref="tns:InputDataFault" />
</msdl:operation>
<msdl:operation ref="tns:UsedTickets"
    xmpp:ident="mobilisxhunt:iq:usedtickets">
    <msdl:input xmpp:type="get" />
    <msdl:output xmpp:type="result" />
</msdl:operation>
<msdl:fault ref="tns:ClosedGameFault" xmpp:errortype="cancel"
    xmpp:errorcondition="not-allowed"
    xmpp:errortext="Maximum of Players Reached or Game isn't open." />
<msdl:fault ref="tns:PermissionFault" xmpp:errortype="cancel"
    xmpp:errorcondition="not-allowed"
    xmpp:errortext="You do not have the permission for this action." />
<msdl:fault ref="tns:InvalidTargetFault" xmpp:errortype="modify"
    xmpp:errorcondition="not-acceptable"
    xmpp:errortext="You cannot choose this target." />
<msdl:fault ref="tns:InputDataFault" xmpp:errortype="modify"
    xmpp:errorcondition="not-acceptable"
    xmpp:errortext="Unaccepted data input." />
<msdl:fault ref="tns:PlayerSynchronizationFault"
    xmpp:errortype="cancel" xmpp:errorcondition="not-allowed"
    xmpp:errortext="Your data is inconsistent with the servers data." />
</msdl:binding>

<msdl:service name="MobilisXHunt" interface="tns:MXHInterface"
    ident="http://mobilis.inf.tu-dresden.de#services/MobilisXHuntService"
    version="2">
    <msdl:endpoint name="MXHEndpoint" binding="tns:MXHBinding" />
    <msdl:dependencies />
</msdl:service>

</msdl:description>

```



# **Abkürzungsverzeichnis**

BOSH	Bidirectional-streams Over Synchronous HTTP
CBS	Component-based System
CBSE	Component-based Software Engineering
CIM	Computation Independent Model
COM	Component Object Model
HTTP	Hypertext Transfer Protocol
IM	Instant Messaging
JID	Jabber ID
JVM	Java Virtuel Machine
LBG	Location-based Game
M2M	Maschine-zu-Maschine
MDA	Model-Driven-Architecture
MSDL	Mobilis Service Description Language
MXA	Mobilis XMPP for Android
P2P	Peer-to-Peer
PIM	Platform Independent Model
POJO	Plain-Old-Java-Object
PSM	Platform Specific Model
QoS	Quality of Service
SOA	Serviceorientierte Architektur
TCP	Transmission Control Protocol
UML	Unified Modeling Language
URL	Uniform Resource Locator
WSDL	Web Service Description Language
XEP	XMPP Extension Protocol
XML	Extensible Markup Language
XMPP	Extensible Messaging and Presence Protocol

*Abkürzungsverzeichnis*

XSF	XMPP Standards Foundation
XSLT	Extensible Stylesheet Language Transformation

# Abbildungsverzeichnis

2.1 Übersicht der SOA Komponenten . . . . .	5
2.2 Beispiel Reflection in Java . . . . .	7
2.3 OSGi Plattform Schichten (Quelle: [4]) . . . . .	8
2.4 Lebenszyklus eines OSGi Bundles (Quelle: [4]) . . . . .	8
2.5 XEP Beispiel einer Service Discovery Anfrage . . . . .	11
2.6 XEP Beispiel einer Service Discovery Antwort . . . . .	11
2.7 XEP Beispiel File Transfer anbieten . . . . .	12
2.8 XEP Beispiel File Tranfer akzeptieren . . . . .	13
2.9 XEP Beispiel SOAP Anfrage . . . . .	13
2.10 XEP Beispiel SOAP Antwort . . . . .	14
2.11 Übersicht Mobilis Service Plattform . . . . .	16
2.12 Ein komponentenbasiertes System (Quelle: [15]) . . . . .	17
2.13 Stadien eines Lifecycle einer Komponente (Quelle: [15]) . . . . .	18
2.14 Verlauf einer Komposition nach MDA (Quelle: [43]) . . . . .	20
2.15 Schichten eines serviceorientierten Komponentenmodells (Quelle: [11]) . . . . .	22
2.16 Beschreibung einer Komponente im Gravity Projekt(Quelle: [13]) . . . . .	22
2.17 Abstraktionen in einem serviceorientierten Komponentenmodell(Quelle: [38]) . . . . .	23
2.18 Erweiterung einer iPOJO Komponente (Quelle: [18]) . . . . .	24
2.19 Elemente einer M2M Infrastruktur (Quelle: [10]) . . . . .	25
2.20 Kommunikationsablauf in der kollaborativen Plattform (Quelle: [10]) . . . . .	25
2.21 Die zwei Schichten der P2P-Plattform (Quelle: [26]) . . . . .	25
 3.1 Schema XMPP-Bean (Quelle: [30]) . . . . .	30
3.2 Entwicklungsprozess eines Mobilis-Service . . . . .	31
3.3 XMPP Glue Code im Mobilis Server (Quelle: [30]) . . . . .	31
3.4 Use-Case Mobilis-Service Consumer . . . . .	33
3.5 Use Case Mobilis-Service Provider . . . . .	34
3.6 Use-Case Mobilis-Service Developer . . . . .	35
 4.1 Ablauf eines neuen Entwicklungsprozesses . . . . .	41
4.2 Schematische Darstellung der Architektur . . . . .	43
4.3 Architektur verteilter Mobilis-Services . . . . .	44
4.4 Life cycle externer Mobilis-Services . . . . .	45
4.5 Mobilis OSGi Übersicht . . . . .	47
4.6 Übersicht Mobilis PlugIn Plattform . . . . .	49
4.7 Mobilis PlugIn Erstellungssequenzen . . . . .	51
4.8 Klassenschema Mobilis-Container und Mobilis-Service . . . . .	51
4.9 Mobilis-Service-Container . . . . .	52
4.10 Life cycle Mobilis-Service . . . . .	54
4.11 Lifecycle Mobilis Service Container . . . . .	55

## *Abbildungsverzeichnis*

4.12 Elemente der Beschreibungssprache . . . . .	58
4.13 Das Element <code>description</code> der TreasureHunt-MSDL . . . . .	61
4.14 Element <code>interface</code> der TreasureHunt-MSDL . . . . .	62
4.15 Element <code>binding</code> der TreasureHunt-MSDL . . . . .	63
4.16 Element <code>service</code> der TreasureHunt-MSDL . . . . .	65
4.17 Beispiel IQ PickUpTreasure SET . . . . .	66
4.18 Beispiel IQ PickUpTreasure ERROR . . . . .	66
4.19 Beispiel Message TreasureCollected . . . . .	67
4.20 Template-basierte Codegenerierung . . . . .	69
4.21 Codegenerierung Übersicht . . . . .	69
4.22 Codegenerierung Klassen-Schema Java . . . . .	70
4.23 Aufbau der MSDL-Elemente . . . . .	71
5.1 Übersicht Implementierung von MSDL zum Jar-Archiv . . . . .	74
5.2 Kommunikation in TreasureHunt . . . . .	74
5.3 Main-Template der XSLT zur Codegenerierung . . . . .	75
5.4 MSDL-Abschnitt der PickUpTreasure Operation . . . . .	76
5.5 MSDL-Abschnitt des Typs PickUpTreasureRequest . . . . .	76
5.6 Generiertes XMPP-Bean des Typs PickUpTreasureRequest . . . . .	77
5.7 Proxyklasse des TreasureHunt Service . . . . .	78
5.8 Struktur des TreasureHunt-Projekts . . . . .	80
5.9 Classpath TreasureHunt-Service . . . . .	80
5.10 Jar-Archiv-Struktur des TreasureHunt-Service . . . . .	82
5.11 Laden von Ressourcen aus einem Jar-Archiv . . . . .	82
6.1 Kommunikationsverlauf PlayerExit (alt) . . . . .	91
6.2 Kommunikationsverlauf PlayerExit (neu) . . . . .	91

# **Tabellenverzeichnis**

2.1	Übersicht der Attribute eines XMPP-Stanzas . . . . .	10
2.2	Übersicht der XEPs in der Mobilis-Plattform . . . . .	15
2.3	Gegenüberstellung der Plattformen . . . . .	26
3.1	Zusammenfassung der funktionalen Anforderungen . . . . .	38
3.2	Zusammenfassung der nichtfunktionalen Anforderungen . . . . .	39
3.3	Tabellarische Zusammenfassung der Abgrenzungskriterien . . . . .	40
4.1	Operationen des Dienstes TreasureHunt . . . . .	60
4.2	Mapping der XMPP-Stanzas auf ein <b>operation</b> -Element . . . . .	66
5.1	Unterstützte Datentypen der XSLT-Codegenerierung . . . . .	78
5.2	Variablen des Ant-Build-Script für das Packen eines Mobilis-Service . . . . .	81
5.3	Mobilis Life cycle Exceptions . . . . .	82
5.4	Parameter für die Agent-Konfiguration . . . . .	84
6.1	Codemetriken für die Kommunikation mit und ohne Codegenerierung . . . . .	92



# Literaturverzeichnis

- [1] *About the OSGi Alliance*. <http://www.osgi.org/About/HomePage>. – Zugriff am 18.09.2011
- [2] *Java 6 Reflection Documentation*. <http://download.oracle.com/javase/6/docs/api/index.html>. – Zugriff am 13.11.2011
- [3] *Java SE Security*. <http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136007.html>. – Zugriff am 18.09.2011
- [4] *OSGI 4.3 - Core Specification*. <http://www.osgi.org/Download/Release4V43>. – Zugriff am 18.09.2011
- [5] *OSGi Technology*. [http://www.osgi.org/About/Technology#Standard\\_Services](http://www.osgi.org/About/Technology#Standard_Services). – Zugriff am 18.09.2011
- [6] *Smack API*. <http://www.igniterealtime.org/projects/smack/>. – Zugriff am 20.08.2011
- [7] *XMPP-Core*. <http://xmpp.org/rfcs/rfc3920.html>. – Zugriff am 19.09.2011
- [8] *XMPP Extension Protocols*. <http://xmpp.org/xmpp-protocols/xmpp-extensions/>. – Zugriff am 19.09.2011
- [9] *XMPP Standard Foundation*. <http://xmpp.org/>. – Zugriff am 6.10.2011
- [10] BEN ALAYA, M. ; BAUDIN, V. ; DRIRA, K.: Dynamic Deployment of Collaborative Components in Service-Oriented Architectures. In: *New Technologies of Distributed Systems (NOTERE), 2011 11th Annual International Conference on*, 2011. – ISSN 2162–1896, S. 1 –8
- [11] BROWN, Alan ; JOHNSTON, Simon ; KELLY, Kevin: *Using service-oriented architecture and component-based development to build Web service applications*. <http://www-128.ibm.com/developerworks/rational/library/510.html>. Version: 2002
- [12] CASATI, Fabio ; ILNICKI, Ski ; JIN, LiJie ; KRISHNAMOORTHY, Vasudev ; SHAN, Ming-Chien: Adaptive and Dynamic Service Composition in eFlow. In: WÄNGLER, Benkt (Hrsg.) ; BERGMAN, Lars (Hrsg.): *Advanced Information Systems Engineering* Bd. 1789. Springer Berlin / Heidelberg, 2000. – ISBN 978–3–540–67630–0, S. 13–31
- [13] CERVANTES, Humberto ; HALL, Richard S.: Autonomous Adaptation to Dynamic Availability Using a Service-Oriented Component Model. In: *Proceedings of the 26th International Conference on Software Engineering*. Washington, DC, USA : IEEE Computer Society, 2004 (ICSE '04). – ISBN 0–7695–2163–0, 614–623

## Literaturverzeichnis

- [14] CHINNICI, Roberto ; MOREAU, Jean-Jacques ; RYMAN, Arthur ; WEERAWARANA, Sanjiva: *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language*. 2007
- [15] CRNKOVIC, Ivica ; SENTILLES, Severine ; ANETA, Vulgarakis ; CHAUDRON, Michel R.: A Classification Framework for Software Component Models. In: *Software Engineering, IEEE Transactions on* 37 (2011), sept.-oct., Nr. 5, S. 593 – 615. <http://dx.doi.org/10.1109/TSE.2010.83>. – DOI 10.1109/TSE.2010.83. – ISSN 0098–5589
- [16] DESERTOT, M. ; DONSEZ, D. ; LALANDA, P.: A Dynamic Service-Oriented Implementation for Java EE Servers. In: *Services Computing, 2006. SCC '06. IEEE International Conference on*, 2006, S. 159 –166
- [17] EATMON, Ryan ; HILDEBRAND, Joe ; MILLER, Jeremie ; MULDOWNNEY, Thomas ; SAINT-ANDRE, Peter: XEP-0004: Data Forms / XMPP Standard Foundation. 2007. – Forschungsbericht
- [18] ESCOFFIER, C. ; HALL, R.S. ; LALANDA, P.: iPOJO: an Extensible Service-Oriented Component Framework. In: *Services Computing, 2007. SCC 2007. IEEE International Conference on*, 2007, S. 474 –481
- [19] ESCOFFIER, Clément ; DONSEZ, Didier ; HALL, Richard S.: *Developing an OSGi-like Service Platform for.NET*
- [20] FORNO, Fabio ; SAINT-ANDRE, Peter: XEP-0072: SOAP Over XMPP / XMPP Standard Foundation. 2005. – Forschungsbericht
- [21] HALL, R.S. ; CERVANTES, H.: An OSGi implementation and experience report. In: *Consumer Communications and Networking Conference, 2004. CCNC 2004. First IEEE*, 2004, S. 394 – 399
- [22] HILDEBRAND, Joe ; MILLARD, Peter ; EATMON, Ryan ; SAINT-ANDRE, Peter: XEP-0030: Service Discovery / XMPP Standard Foundation. 2008. – Forschungsbericht
- [23] HOBBS, C. ; BECHA, H. ; AMYOT, D.: Failure Semantics in a SOA Environment. In: *e-Technologies, 2008 International MCETECH Conference on*, 2008, S. 116 –121
- [24] JANSEN, Nikolas: *Design and Implementation of a Web Gateway for Mobile Collaboration Services*. 07 2011
- [25] KARNEGES, Justin: XEP-0291: Service Delegation / XMPP Standard Foundation. 2011. – Forschungsbericht
- [26] KELLERER, Wolfgang ; DESPOTOVIC, Zoran ; MICHEL, Maximilian ; HOFSTATTER, Quirin ; ZOLS, Stefan: Towards a Mobile Peer-to-Peer Service Platform. In: *Applications and the Internet Workshops, 2007. SAINT Workshops 2007. International Symposium on*, 2007, S. 2
- [27] KIEFNER, Danny: Runden-basierte Orts-bezogene Spiele auf mobilen Endgeräten. (2011)

- [28] KOREN, I.: *MXA - Mobilis XMPP for Android.* <http://mobilisplatform.sourceforge.net/mxa.htm>. – Zugriff am 19.08.2011
- [29] LÜBKE, R.: *Tutorial about how to use the Mobilis XMPP Bean Layer.* <http://sourceforge.net/projects/mobilisplatform/files/Tutorial%20Beans.pdf>. – Zugriff am 19.08.2011
- [30] LÜBKE, R.: *Ein Framework zur Entwicklung mobiler Social Software auf Basis von Android.* März 2011
- [31] MELZER, Ingo ; MELZER, Ingo (Hrsg.): *Service-orientierte Architekturen mit Web Services: Konzepte – Standards – Praxis.* 4. Heidelberg : Spektrum, 2010. <http://dx.doi.org/10.1007/978-3-8274-2550-8>. <http://dx.doi.org/10.1007/978-3-8274-2549-2> – ISBN 978-3-8274-2549-2
- [32] MILLARD, Peter ; SAINT-ANDRE, Peter ; MEIJER, Ralph: XEP-0060: Publish-Subscribe / XMPP Standard Foundation. 2010. – Forschungsbericht
- [33] MULDOWNEY, Thomas ; MILLER, Matthew ; KARNEGES, Justin: XEP-0052: File Transfer / XMPP Standard Foundation. Version: 2003. <http://xmpp.org/extensions/xep-0096.html>. 2003. – Forschungsbericht
- [34] PARNAS, D. L.: On the criteria to be used in decomposing systems into modules. In: *Commun. ACM* 15 (1972), December, 1053–1058. <http://dx.doi.org/http://doi.acm.org/10.1145/361598.361623>. – DOI <http://doi.acm.org/10.1145/361598.361623>. – ISSN 0001–0782
- [35] PATERSON, Ian ; SAINT-ANDRE, Peter ; SMITH, Dave ; MOFFITT, Jack: XEP-0124: Bidirectional-streams Over Synchronous HTTP (BOSH) / XMPP Standard Foundation. Version: 2010. <http://xmpp.org/extensions/xep-0124.html>. 2010. – Forschungsbericht
- [36] PEMBERTON, Steven ; AUSTIN, Daniel ; AXELSSON, Jonny: *XHTML 1.0 The Extensible HyperText Markup Language (Second Edition).* 2002
- [37] PETRITSCH, Helmut: Service-Oriented Architecture (SOA) vs. Component Based Architecture / TU Wien. 2006. – Forschungsbericht
- [38] RUDAMETKIN, W. ; TOUSAU, L. ; DONSEZ, D. ; EXERTIER, F.: A Framework for Managing Dynamic Service-Oriented Component Architectures. In: *Services Computing Conference (APSCC), 2010 IEEE Asia-Pacific*, 2010, S. 43 –50
- [39] SAINT-ANDRE, Peter: XEP-0128: Service Discovery Extensions / XMPP Standard Foundation. 2004. – Forschungsbericht
- [40] SAINT-ANDRE, Peter: XEP-0133: Service Administration / XMPP Standard Foundation. 2005. – Forschungsbericht
- [41] SAINT-ANDRE, Peter: XEP-0045: Multi-User Chat / XMPP Standard Foundation. 2008. – Forschungsbericht
- [42] SAINT-ANDRE, Peter: XEP-0174: Serverless Messaging / XMPP Standard Foundation. 2008. – Forschungsbericht

*Literaturverzeichnis*

- [43] SIVASUBRAMANIAN, S.P. ; ILAVARASAN, E. ; VADIVELOU, G.: Dynamic Web Service Composition: Challenges and techniques. In: *Intelligent Agent Multi-Agent Systems, 2009. IAMA 2009. International Conference on*, 2009, S. 1 –8
- [44] STAL, Michael: Web services: beyond component-based computing. In: *Commun. ACM* 45 (2002), October, 71–76. <http://dx.doi.org/http://doi.acm.org/10.1145/570907.570934>. – DOI <http://doi.acm.org/10.1145/570907.570934>. – ISSN 0001–0782
- [45] SÖLLNER, Benjamin: *XMPP-based Media Sharing for Mobile Collaboration with Android Phones*. October 2009
- [46] WÜTHERICH, Gerd ; HARTMANN, Nils ; KOLB, Bernd ; LÜBKEN, Matthias: *Die OSGi-Service-Platform : eine Einführung mit Eclipse Equinox*. 1. Aufl. Heidelberg : dpunkt, 2008

### **Selbständigkeitserklärung**

Hiermit erkläre ich, dass ich die vorliegende Diplomarbeit zum Thema

**Dynamisches Deployment XMPP-basierter Dienste für Mobile Social Apps**

vollkommen selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt, sowie Zitate kenntlich gemacht habe.

Dresden, den 31.01.2012

Unterschrift