A NICE LATEX PAPER TEMPLATE

INSERT COMPLICATED SOUNDING SUBTITLE HERE

YOUR NAME

JULY 2010

You Prof

Your degree
Faculty

Your school

ZUSAMMENFASSUNG

Die Erstellung von modernen Unternehmensanwendungen ist ein komplexes Themengebiet und die dabei verwendeten Technologien und Konzepte sind einem ständigen Wandel unterzogen. In dieser Arbeit soll eine Architektur für diese Art von Anwendungen vorgestellt werden, die diese Komplexität strukturieren und durch verschiedene Implementierungen an einigen Stellen auch reduzieren soll.

Neben der Vorstellung einer Gesamtarchitektur sollen dabei zwei Bereiche dieser Architektur genauer betrachtet werden. Zum einen wird das Konzept des Service Layer beschrieben, und anhand der allgemeinen Eigenschaften von Unternehmensanwendungen eine Reihe von generischen Service Funktionalitäten hergeleitet und implementiert. Zum anderen soll ein zentraler Ausführungsmechanismus entwickelt werden, der es erlaubt die Operationen des Service Layer für den Aufruf über entfernte Methodenaufrufe in verschiedenen Protokollen anzubieten. Anschließend werden drei dieser Protokolle und entsprechende Implementierungen, die diesen gemeinsamen Ausführungsmechanismus verwenden, vorgestellt.

Zum Abschluss der Arbeit werden verschiedene Anwendungsmöglichkeiten für diese Implementierungen sowie mögliche Variationen der Architektur aufgezeigt.

INHALTSVERZEICHNIS

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS

LISTINGS

ACRONYME

API Application Programmers Interface

AOP Aspect Oriented Programming

CRUD Create, Read, Update, Delete

DAO Data Access Object

GORM Grails ORM

HQL Hibernate Query Language

HTML Hypertext Markup Language

HTTP Hypertext Transport Protocol

IDL Interface Description Language

IOC Inversion Of Control

J2EE Java Enterprise Edition

JPA Java Persistence API

JRE Java Runtime Environment

JSON JavaScript Object Notation

JVM Java Virtual Machine

MVC Model View Controller

ORM Object-Relational Mapping

POJO Plain Old Java Object

RDBMS Relational Database Management System

REST Representational State Transfer

RIA Rich Internet Application

RMI Remote Method Invocation

RPC Remote Procedure Call

SGML Standard Generalized Markup Language

SGML Standardized General Markup Language

SLF4J Simple Logging Framework For Java

SQL Structured Query Language

UML Unified Modeling Language

URI Uniform Resource Identifier

URL Uniform Resource Locator

WSDL Werbservice Description Language

XML Extended Markup Language

Teil I

EINLEITUNG

1 MOTIVATION

Moderne Unternehmens- und Webanwendungen zu entwickeln ist ein komplexes Thema und die dabei verwendeten Technologien unterliegen einem ständigen Wandel. Das folgende Kapitel soll diese Anwendungen genauer charakterisieren und den Rahmen sowie den Aufbau dieser Arbeit vorstellen.

1.1 UNTERNEHMENSANWENDUNGEN

Hierfür ist es zuerst notwendig die Art von Anwendungen festzulegen, die im Folgenden näher behandelt werden sollen. Der Begriff Unternehmensanwendung (*Enterprise Applications*) umfasst grundsätzlich eine Art von Software, für die es zwar keine allgemein gültige Definition gibt, die aber einige charakteristische Merkmale aufweist (siehe [Fow02] S. 3).

Grundsätzlich werden in einer Unternehmensanwendung *große Mengen an Daten* verarbeitet und auch *dauerhaft gespeichert*. Für die dauerhafte Speicherung kommen normalerweise eine oder mehrere Datenbanken zum Einsatz und die Menge der Daten erreicht dabei oft einige Terabyte oder mehr. Normalerweise wird auf diese Daten dann auch von *vielen verschiedenen Benutzern*, häufig gleichzeitig, zugegriffen. Je nach Anzahl der Benutzer kann diese Eigenschaft auch besondere Anforderungen an die Skalierbarkeit eines Systems stellen. Durch die Menge der von der Anwendung verarbeiteten Daten ist es auch wahrscheinlich, dass die *Benutzeroberfläche sehr umfangreich* ist und in viele unterschiedliche Ebenen oder Fenster aufgeteilt ist. Weiterhin erfordert die Umsetzung der Anforderungen und Geschäftsprozesse oft eine *komplexe Anwendungslogik*. Das ist vor allem der Fall, wenn Sonderfälle berücksichtigt werden müssen, die nicht direkt einem verallgemeinerten Geschäftsprozess folgen. Ein weiteres Merkmal ist, dass eine Unternehmensanwendung in den meisten Fällen mit anderen Anwendungen kommuniziert. Im Umkehrschluss heißt das auch, dass die Anwendung selbst wahrscheinlich eine Reihe von Schnittstellen bereitstellen muss, die die Kommunikation mit anderen Anwendungen erlauben.

Beispiele für Unternehmensanwendungen sind Warenwirtschaftssysteme, Buchhaltunssysteme, Content Management Systeme oder eine Patientenaktenverwaltung[1]. Da viele dieser Systeme eine Weboberfläche anbieten

1 Keine Unternehmensanwendungen sind: Embedded Systems, Text- oder Bildverarbeitungsprogramme, Betriebssysteme, Compiler oder Videospiele

fallen auch ein Großteil der Webapplikationen unter diese Kategorie. Im weiteren Verlauf sollen die Begriffe *Anwendung* und Unternehmensanwendung gleichbedeutend verwendet werden.

Um der Komplexität bei der Entwicklung von Unternehmensanwendungen entgegen zu wirken wird zur Strukturierung häufig eine *Schichtenarchitektur* verwendet. Eine Schichtenarchitektur erlaubt es, eine Anwendung in logische Schichten zu unterteilen, wobei immer eine übergeordnete Schicht die Schnittstelle der direkt darunterliegenenden Schicht benutzt. Die darunterliegende Schicht stellt diese Schnittstelle zur Verfügung, hat aber selbst keine Kenntnis von den übergeordneten Schichten. Dadurch können die einzelnen Schichten einer Anwendung relativ getrennt voneinander behandelt werden, was die Komplexität deutlich reduziert. In [Fow02] (vgl. S. 20) wird eine Unternehmensanwendung in die drei grundlegenden Schichten *Datasource*, *Application Logic* und *Presentation* unterteilt. Diese Unterteilung soll auch hier für den Aufbau einer Anwendung verwendet werden. Tabelle 1 zeigt eine Übersicht der Schichten und deren Aufgaben.

SCHICHT	BESCHREIBUNG
Presentation / Remoting	Stellt die Anwendungslogik für einen Benutzer bereit. Beispielsweise über eine grafische Benutzeroberfläche oder als Services für entfernte Methodenaufrufe
Application Logic	Beinhaltet die eigentliche Anwendungslogik
Datasource	Stellt die Verbindung zu Datenbanken oder anderen entfernten Systemen her

Tabelle 1: Übersicht über die drei grundlegenden Schichten

Abbildung 1 zeigt die Einordnung einer Unternehmensanwendung mit den vorgestellten Schichten in einem beispielhaften Gesamtzusammenhang. Darin wird die Kommunikation in eine Client und eine Serverseite unterteilt. Auf der Serverseite steht die Anwendung selbst und die entfernten Systeme mit welchen sie kommuniziert, beispielsweise eine Datenbank oder eine andere Unternehmensanwendung. Auf der Clientseite stehen die Anwendungen, die wiederum mit der Anwendung selbst kommunizieren. In diesem Beispiel handelt es sich um eine andere Unternehmensanwendung und einen Webserver, der die Daten der Anwendung für die Darstellung in einem Webbrowser aufbereitet. Diese Aufgabe könnte aber auch von der Anwendung selbst direkt in der Presentation Schicht übernommen werden.

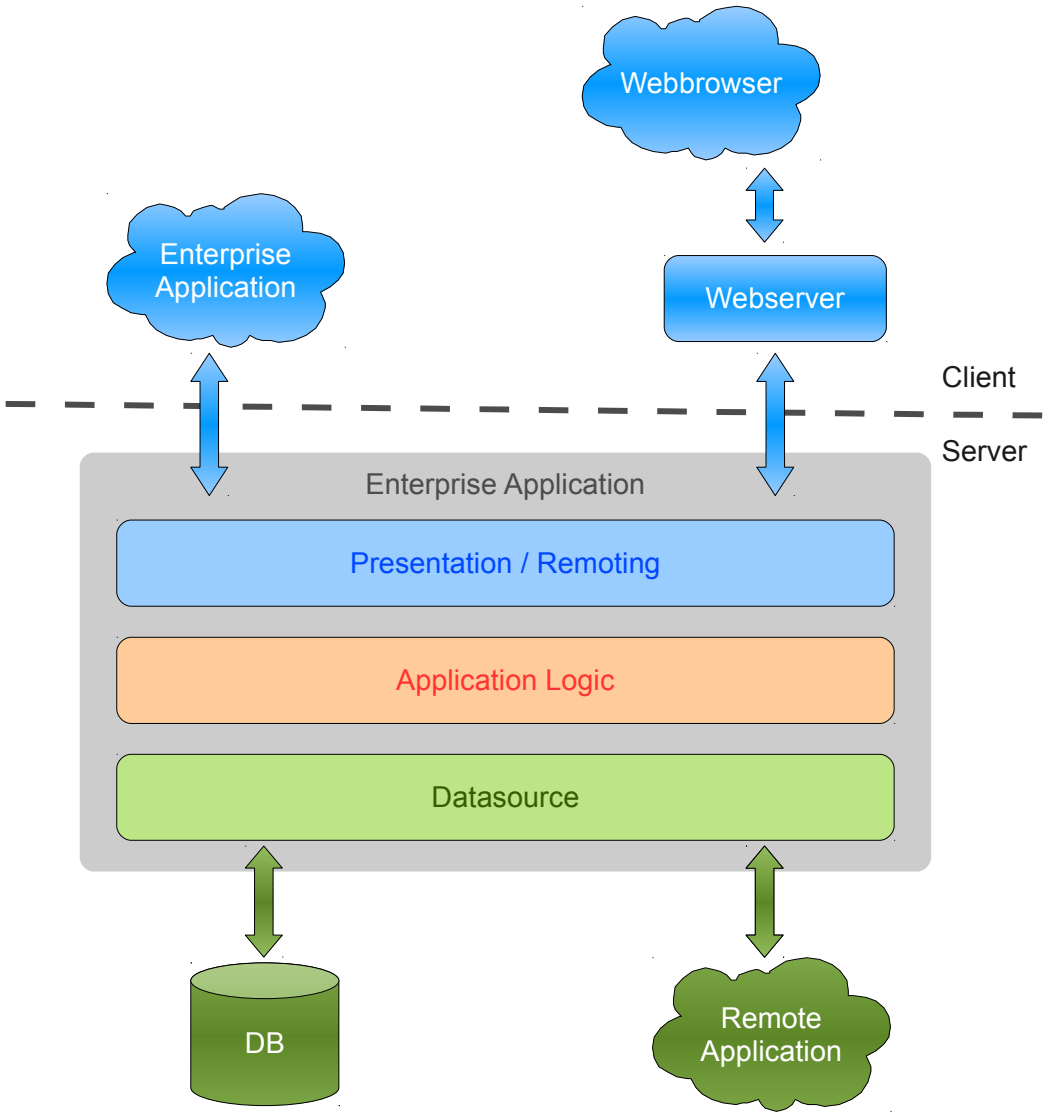

Abbildung 1: Einordnung einer Unternehmensanwendung

1.2 AUFGABEN

Die in dieser Arbeit vorgestellten Implementierungen sind grundsätzlich in zwei Aufgabenbereiche unterteilt.

Im ersten Teil soll auf Grundlage der oben aufgezeigten Merkmale von Unternehmensanwendungen der Service Layer vorgestellt werden. Es handelt sich hierbei um einen Teil der Application Logic Schicht der dazu dient einer übergeordneten Schicht eine eindeutige und einheitliche Schnittstelle zur Anwendungslogik bereitzustellen. Damit haben die verschiedenen Darstellungsmöglichkeiten der Presentation Schicht dann eine einzige gemeinsame Schnittstelle zur Verfügung. Ferner sollen Funktionalitäten dieses Service Layer herausgearbeitet und implementiert werden, die entsprechend den allgemeinen Merkmalen einer Unternehmensanwendung in verschiedenen Anwendungen für die Grundlage eines Service Layer verwendet werden können.

Der zweite Teil befasst sich mit der Presentation Schicht, genauer dem Teil, der die Aufgabe übernimmt die Funktionalitäten des Service Layer über entfernte Methodenaufrufe zur Verfügung zu stellen. Deshalb soll dieser Teil der Presentation Schicht im weiteren Verlauf auch *Remoting* Schicht genannt werden. Aufgabe ist es hier ein abstraktes Modell für die Darstellung von Service Methoden und Methodenaufrufen zu erstellen und einen zentralisierten Ausführungsmechanismus für diese zu entwickeln. Darauf aufbauend sollen nun eine Reihe von Protokollen für entfernte Methodenaufrufe evaluiert werden und die jeweiligen Implementierungen, die den vorher entwickelten zentralen Ausführungsmechanismus verwenden, realisiert werden.

1.3 HERANGEHENSWEISE

Auch wenn der Schwerpunkt dieser Arbeit in der Anwendungslogik (Application Logic) und Remoting Schicht liegt ist es für das Verständnis des Gesamtsystems sinnvoll die Schichtenarchitektur einer Anwendung in ihrer Gesamtheit zu betrachten. Für die hier verwendete Architektur wurden dazu die drei grundlegenden Schichten weiter unterteilt, wobei die einzelnen Komponenten im weiteren Verlauf der Arbeit vorgestellt werden sollen.

Nach der Behandlung wichtiger Grundlagen in Kapitel 2 wird in Kapitel 3 die Schicht des Datasource- und Domain Layer näher vorgestellt. Der Datasource Layer übernimmt dabei die Aufgabe mit anderen entfernten Anwendungen zu kommunizieren während der Domain Layer bereits einen Teil der Anwendungslogik implementiert. Das nachfolgende Kapitel 4 stellt das Konzept des Service Layer näher vor und behandelt dann die Implementierung des ersten Teils der Aufgabenstellung. In Kapitel 5 wird dann der zweite Teil der Aufgabenstellung diskutiert und die Implemen-

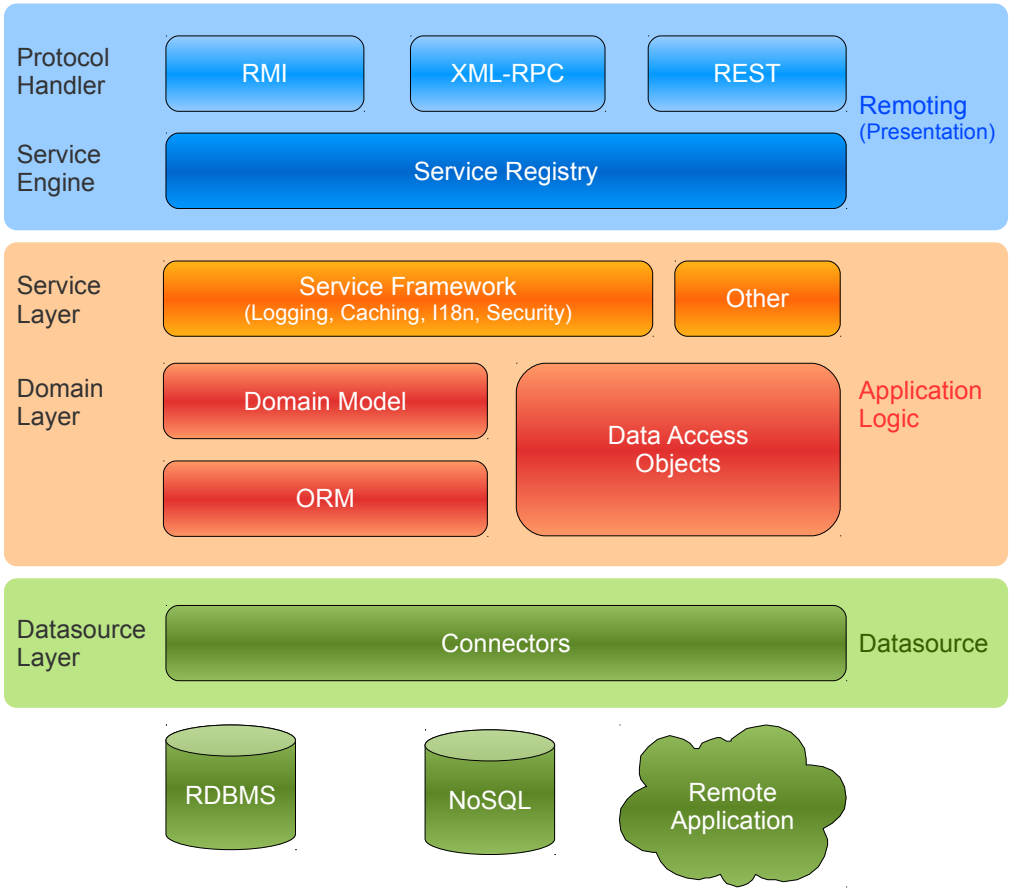

Abbildung 2: Architekturübersicht

tierung der Service Engine als zentralisierter Ausführungsmechanismus für entfernte Methodenaufrufe vorgestellt. Auf dieser Grundlage werden dann drei Referenzimplementierungen von Protokollen für entfernte Methodenaufrufe vorgestellt, die von der Service Engine Gebrauch machen. Kapitel 6 enthält einige Beispiele, wie die vorgestellten Implementierungen im Gesamtzusammenhang verwendet werden können. Abbildung 2 zeigt bereits eine Gesamtübersicht der verwendeten Architektur, wobei die einzelnen Schichten und Module erst im weiteren Verlauf dieser Arbeit vorgestellt werden sollen.

2

GRUNDLAGEN

Im folgenden Kapitel sollen einige wichtige Grundbegriffe und Technologien eingeführt werden, die für das weitere Verständnis der Arbeit wichtig sind. Neben der Festlegung der verwendeten Begriffe wird die Java Plattform und die dort verwendeten Technologien näher behandelt. Anschließend wird das Konzept des Remote Procedure Call (RPC) eingeführt und verschiedene Austauschformate für die Übertragung von Daten in einem Netzwerk erläutert.

Vom Leser werden Kenntnisse in der *Objektorientierten Programmierung* und der dabei verwendeten Terminologie sowie Grundkenntnisse über häufig verwendete *Design Patterns* (siehe [GHJV95] und [Fow02]) erwartet. Da die Implementierung in Java und einer Java sehr ähnlichen Programmiersprache erfolgte, sind gute Java Kenntnisse ebenfalls empfehlenswert. Auch von Grundlagen in Netzwerken, einschließlich grundlegender Web Technologien, wird ausgegangen.

2.1 BEGRIFFE

FRAMEWORK Ein Framework ist in der objektorientierten Programmierung eine Sammlung von Klassen, welche - meist unter der Verwendung verschiedener Design Patterns - einen Rahmen für die Architektur einer Anwendung bieten. Im Unterschied dazu stellt eine Klassenbibliothek lediglich eine Reihe von Klassen bereit, die von einem Entwickler verwendet werden können.

BENUTZER Die folgenden Kapitel beziehen sich auf den Aufbau einer Schichtenarchitektur. Deshalb soll sich in diesem Zusammenhang der Begriff Benutzer immer auf die jeweils übergeordnete Schicht beziehen. Aus Sicht eines Frameworks oder einer Klassenbibliothek ist der Benutzer der Teil einer Anwendung, der diese verwendet. Ein menschlicher Benutzer soll im weiteren Verlauf als Anwender bezeichnet werden.

ASSOZIATIVES ARRAY Assoziative Arrays[1] erlauben die Adressierung von Datenelementen über beliebige, eindeutige Schlüssel (siehe [Wik10a]). Die dafür benötigte Schnittstelle umfasst dabei die Operationen *get(key)*, *put(key, value)* und *remove(key)*. Da assoziative Arrays von den meisten Programmiersprachen unterstützt werden, stellen

1 In verschiedenen Programmiersprachen auch Dictionary, Hash oder Map genannt

sie eine wichtige Grundlage für den Austausch strukturierter Daten dar. Auch im weiteren Verlauf dieser Arbeit wird das Konzept der assoziativen Arrays wiederholt aufgegriffen. Beispiele hierfür sind:

- Java Maps und JavaBeans (in Kapitel 2.2.2)
- NoSQL Datenbanken (in Kapitel 3.1)
- JSON und XML (in Kapitel 2.4.4)
- Anwendungscaches (in Kapitel 4.2.1)
- Java Property Dateien (in Kapitel 4.2.5)

VERTEILT VS. NETZWERK-BASIERT Die Literatur (vgl. [Fie00] S. 24) unterscheidet zwischen verteilten und netzwerkbasierten Systemen. Ein verteiltes System verhält sich gegenüber dem Benutzer wie ein lokales System, wird aber auf mehreren CPUs ausgeführt. Im Gegensatz dazu kommuniziert ein netzwerkbasiertes System zwar über ein Netzwerk, muss dies aber nicht unbedingt für den Benutzer transparent übernehmen. Der Schwerpunkt dieser Arbeit soll auf netzwerkbasierten Systemen liegen.

2.2 JAVA PLATTFORM

Die Java Virtual Machine (JVM) stellt eine Middleware dar, die die plattformunabhängige Ausführung von in entsprechendem Bytecode vorliegenden Programmen ermöglicht (vgl. [Wik10c]). Die JVM ist ein Teil des Java Runtime Environment (JRE) und stellt zusammen mit einer großen Anzahl an Klassenbibliotheken die Java Plattform.

2.2.1 *Programmiersprachen für die JVM*

Zur Einführung der JVM in der Version 1.0 gab es nur einen Bytecode Compiler für die Programmiersprache Java. Mit steigender Verbreitung der Plattform und besserer Performance der JVM wurde es zunehmend interessant, auch andere Programmiersprachen in auf der JVM lauffähigen Bytecode zu übersetzen. Man kann somit unter anderem Programmiersprachen, für die bisher nur Compiler für spezielle Plattformen existierten, eine plattformunabhängige Ausführungsumgebung zur Verfügung stellen oder interpretierte Programmiersprachen durch die Kompilierung in Java Bytecode performanter ausführen, ohne dass die Portabilität verloren geht.

2.2.1.1 *Unterschiedliche Lösungsansätze*

Ein weiteres aktuelles Thema bei Programmiersprachen für die JVM ist die Möglichkeit, durch bestimmte Spracheigenschaften Probleme einfacher und effektiver angehen zu können als es mit Java selbst der Fall ist. So

übernehmen einige Sprachen wie Scala[2] oder Clojure Ansätze aus der funktionalen Programmierung. Dadurch kann unter anderem die Komplexität und Fehleranfälligkeit der nebenläufigen Programmierung reduziert werden, da es, anders als in objektorientierten Programmiersprachen, keinen globalen Zustand gibt, auf den Datenzugriffe aus verschiedenen Threads synchronisiert werden müssen. Auch für die meisten populären dynamischen und interpretierten Sprachen gibt es inzwischen Implementierungen für die JVM. Tabelle 2 zeigt eine Übersicht über einige auf der JVM lauffähigen Programmiersprachen.

NAME	BEMERKUNG	URL
Java	Erste Sprache, OOP	`http://java.sun.com`
Groovy	Dynamisch, OOP	`http://groovy.codehaus.org/`
Scala	OOP / Funktional Hybrid	`http://www.scala-lang.org/`
Clojure	Funktional	`http://clojure.org/`
JRuby	Ruby	`http://jruby.org`
Jython	Python	`http://jython.org`
Rhino	JavaScript	`http://mozilla.org/rhino/`

Tabelle 2: Auswahl einiger Programmiersprachen für die JVM[3]

2.2.1.2 *Groovy*

Eine dieser „neuen" Programmiersprachen, die direkt für die JVM entwickelt wurden, ist Groovy[4]. Es handelt sich hierbei um eine dynamische, objektorientierte Sprache, die Einflüsse aus Programmiersprachen wie Python, Ruby und SmallTalk für Java Entwickler in einer an Java angelehnten Syntax verfügbar macht (siehe [KKLS07]). Dies bedeutet auch, dass der Großteil eines Java Quelltextes gleichzeitig valide Groovy Syntax ist. Dabei besitzt Groovy aber einige zusätzliche Merkmale:

DYNAMISCHE UND STATISCHE TYPISIERUNG Dynamische Typisierung bedeutet, dass der Typ einer Variable zur Laufzeit von deren Wert abhängt. Im Gegensatz dazu wird bei der statischen Typisierung der Typ einer Variable zum Zeitpunkt der Kompilierung festgelegt und kann nur entsprechende Werte annehmen. Groovy unterstützt beide Möglichkeiten, wodurch das jeweils passende Typsystem verwendet werden kann; die statische Typisierung erfolgt identisch mit der in Java. Neben der dynamischen Typisierung erlaubt es Groovy auch, zur

2 Scala steht für *Scalable Language*
3 Quelle: [Wik10d]
4 `http://groovy.codehaus.org/`

Laufzeit neue Methodendefinitionen zu einer Klasse hinzuzufügen oder existierende Definitionen zu ändern.

JAVA INTEGRATION Einer der größten Vorteile von Groovy gegenüber anderen Sprachen für die JVM ist die nahtlose Integration mit einer bereits existierenden Java Codebasis. Da Groovy ebenfalls in Java Bytecode kompiliert wird, ist eine natürliche Verwendung von Groovy Klassen in Java ebenso problemlos möglich wie im umgekehrten Fall.

COLLECTIONS Die Syntax von Groovy sieht eine native Unterstützung für Java Collections vor. Zudem wurden die Java Collection Klassen um eine Reihe nützlicher Methoden erweitert.

CLOSURES Eine Closure ist in Groovy ein anonymer Codeblock[5], dem auch eine Reihe von Parametern übergeben werden können[6]. Sie kann auch für die Ausführung an einer anderen Stelle einer Variablen zugewiesen werden.

OPERATOR OVERLOADING Groovy erlaubt das bisher nicht von Java unterstützte Überladen von Operatoren.

STRING ERWEITERUNGEN Die Unterstützung für Strings wurde in Groovy erweitert. So ist die Referenzierung von Variablennamen in einem String möglich, wodurch die oft unübersichtliche Konkatenation über den + Operator entfällt. Auch Regular Expressions sind ein nativer Bestandteil der Groovy Syntax.

Aufgrund dieser Eigenschaften eignet sich Groovy vor allem sehr gut für die agile Entwicklung von Anwendungen, da viele Probleme mit einem Bruchteil des Quelltextumfangs gelöst werden können, der in Java nötig wäre. Listing 1 zeigt ein Beispiel eines Groovy Unit Tests mit der Verwendung von Closures, Collections und Groovy Strings. Es ist zu beachten, dass dynamische Sprachen, also auch Groovy, grundsätzlich nicht so performant arbeiten, wie statisch getypte Sprachen. Durch die gute Integration von Groovy und Java ist es aber möglich, die Teile einer Anwendung in Java zu implementieren, die performant arbeiten müssen und auf Groovy zurückzugreifen, wenn die Entwicklung beschleunigt und vereinfacht werden soll. Von dieser Möglichkeit wurde auch bei einigen Implementierungen Gebrauch gemacht, die in dieser Arbeit vorgestellt werden sollen.

5 Vergleichbar mit inneren anonymen Klassen in Java
6 Ist kein Parameter angegeben, besitzt eine Closure immer ein implizites Parameter mit dem Namen *it*

```
import java.util.Date;

@Test
void testToUpperCase()
{
        // Use a Java class
        Date date = new Date()
        System.out.println("Todays date is ${date}")
        // Create a collection of strings
        def list = [ "entry1", "entry2", "entry3" ]
        // Puts every entry into uppercase and returns a new list
        def uCaseList = list.collect {
                it.toUpperCase()
        }
        // Print each entry
        uCaseList.each {
                System.out.println("My entry is: ${it}")
        }
        // Compare list
        assert uCaseList == ["ENTRY1", "ENTRY2", "ENTRY3"]
}
```

Listing 1: Beispiel eines Groovy Unit Tests

2.2.2 *JavaBeans*

JavaBeans sind wiederverwendbare Software-Komponenten für Java. Wichtigster Kernpunkt der JavaBean Spezifikation (siehe [Ham97]) sind eine Reihe von Konventionen für die Implementierung von Java Klassen, wodurch diese einfach zu instanziieren sind und leicht in ein übertragbares (serialisierbares) Format gebracht werden können. Damit eine Klasse als JavaBean gilt, muss sie die folgenden Eigenschaften erfüllen (Listing 2 zeigt ein Beispiel für ein JavaBean):

DEFAULT CONSTRUCTOR Die Klasse muss einen öffentlichen Standardkonstruktor definieren.

PROPERTIES In der Objektorientierten Programmierung ist es üblich, dass die Membervariablen einer Klasse nicht nach außen hin sichtbar sind. Für den Zugriff von außen werden deshalb sogenannte *Zugriffsmethoden (Accessor Methods)* implementiert. Dadurch wird die Kapselung der Membervariablen sichergestellt und der Zugriff darauf kann flexibler gestaltet werden. Für die Definition von Zugriffsmethoden sieht die JavaBean Spezifikation eine spezielle Namenskonvention vor. Um eine Eigenschaft zu lesen wird eine Methode nach dem

Schema *Typ getName()* definiert. Für Eigenschaften mit einem boolschen Wert gilt die Signatur *Boolean isName()*. *Name* ist hier der Name der Eigenschaft, und beginnt immer mit einem Großbuchstaben. Zurückgeliefert wird eine Instanz vom Typ dieser Eigenschaft. Für den schreibenden Zugriff muss eine Methode mit der Signatur *void setName(Typ argument)* angelegt werden. Der Parameter argument ist dabei die zu schreibende Eigenschaft. Durch diese Namenskonventionen werden lesende Zugriffsmethoden auch *Getter-* und schreibende Zugriffsmethoden auch *Setter-Methoden* genannt. Eigenschaften, die nur eine Zugriffsmethode für den lesenden Zugriff, besitzen werden *Read-Only* Eigenschaften genannt. Für den selteneren Fall, dass eine Eigenschaft lediglich eine Setter Methode besitzt, nennt man diese *Write-Only* Eigenschaft.

SERIALIZABLE Eine JavaBean Klasse muss das *Serializable* Interface der Java Standardbibliothek implementieren. Es handelt sich hierbei um ein *Marker Interface*, das keinerlei Methoden bereitstellt, sondern eine Klasse lediglich auf eine bestimmte Eigenschaft hin kennzeichnet. In diesem Fall kennzeichnet das *Serializable* Interface, dass der Zustand einer Instanz der implementierenden Klasse auch außerhalb des aktuell ausgeführten Programms dargestellt und auch ohne Informationsverlust von dort wieder hergestellt werden kann. Durch diese Eigenschaft können JavaBeans problemlos in Dateien gespeichert oder über ein Netzwerk übertragen werden.

```
import java.security.NoSuchAlgorithmException;

public class User implements Serializable {
        private String userName;
        private String password;

        public User() {
                this.setPlainPassword("secret"); // Standard password
        }

        public String getUserName() {
                return userName;
        }

        public void setUserName(String userName) {
                this.userName = userName;
        }

        public String getPassword() {
                return password;
        }
```

```
        public void setPasswort(String password) {
                this.password = password;
        }

        // Write only
        public void setPlainPassword(String plainPassword)
                        throws NoSuchAlgorithmException {
                // MD5 encode the plain password
                MessageDigest md = MessageDigest.getInstance("MD5");
                md.update(plainPassword.getBytes(), 0, plainPassword.
                    length());
                // Set the password string to the MD5 hash in HEX
                    values
                String md5Hex = new BigInteger(1, md.digest()).
                    toString(16);
                this.setPassword(md5Hex);
        }
}
```

Listing 2: Beispiel einer JavaBean Klasse

Da der Zugriff auf die *Properties* eines JavaBean durch den jeweiligen Namen erfolgt, können JavaBeans auch als statisches assoziatives Array angesehen werden. Der Name einer Eigenschaft ist der eindeutige Schlüssel und die Menge der möglichen Schlüssel sind die Namen aller Eigenschaften.

2.2.3 *Reflection*

Die *Java Reflection API* (siehe [sun09]) bietet die Möglichkeit, zur Laufzeit Informationen über Methoden und Datenmember von Klassen und Objekten einzuholen. Dadurch ist es möglich Instanzen, von Objekten zu erstellen und Methoden aufzurufen, die erst zur Laufzeit des Programms bekannt sind. Vor allem Frameworks, die über Module erweitert werden können, machen von dieser Möglichkeit häufig Gebrauch. Die Verwendung von Reflection ist grundsätzlich langsamer als eine direkte Ausführung, da Membervariablen und Methoden über Strings angesprochen werden und das entsprechende Aufrufziel deshalb erst aus den Metainformationen der Java Klasse bestimmt werden muss (vgl. [Wik10g]).

Durch die in Kapitel 2.2.2 vorgestellten Namenskonventionen können JavaBeans ebenfalls direkt durch die Reflection API verwendeten werden. Das ist vor allem für Bibliotheken sinnvoll, die allgemein mit JavaBeans

arbeiten, ohne diese bereits zu kennen. Die *Apache Commons Beanutils*[7] Bibliothek bietet hierfür eine Reihe nützlicher Klassen an.

2.2.4 *Spring Framework*

Das Spring Framework[8] ist ein Open Source Anwendungsframework für die Java Plattform zur Entwicklung von Unternehmensanwendungen. Die bisher für diesen Zweck eingesetzte Java Enterprise Edition (J2EE) hatte mit der Zeit eine schwer zu handhabende Komplexität entwickelt, die vor allem bei Entwicklungsbeginn hohe Einstiegshürden legt. Eines der Ziele des Spring Frameworks ist, diese Entwicklung zu vereinfachen, wobei es als Alternative oder zur Ergänzung der J2EE verwendet werden kann. Das Spring Framework ist in verschiedene Module unterteilt, die in Form einer Schichtenarchitektur aufgebaut sind (siehe Abbildung 3).

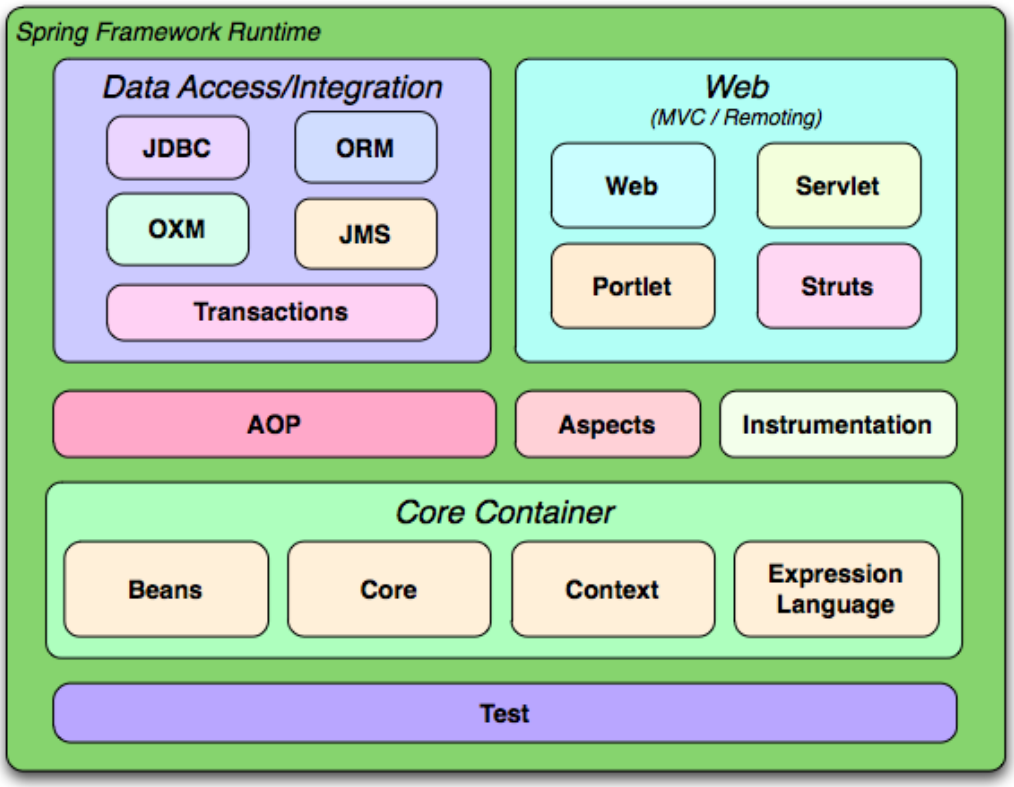

Abbildung 3: Spring Framework Übersicht[9]

7 http://commons.apache.org/beanutils/
8 http://springframework.org
9 Quelle: [Ju09]

2.2.4.1 *Dependency Injection*

Dependency Injection befasst sich mit der grundlegenden Frage, wie man aus einem objektorientierten Klassenmodell die in einer Anwendung benötigten Objekte und deren Abhängigkeiten erstellen kann. Die Bereitstellung eines Dependency Injection Containers ist eine der wichtigsten Aufgaben der Kernkomponenten des Spring Frameworks.

Ein bisher üblicher Ansatz für die Instanziierung der benötigten Objekte ist der des *Factory* Design Patterns. Dort werden die benötigten Konfigurationseinstellungen, oft aus verschiedenen Quellen, gelesen und die entsprechenden Objekte instanziiert, die sich ein Benutzer dann von der Factory holen kann. Ein Nachteil des Factory Patterns ist, dass die gesamte Umsetzung dem Entwickler selbst überlassen ist. Damit ergeben sich zwar viele Freiheiten, da aber eine allgemeine Konvention für die Implementierung fehlt, können auch sehr unflexible Lösungen entstehen. Ein Problem, das dabei häufig auftritt, sind unnötige Abhängigkeiten zwischen Objekten und Klassen, die sich mit ihrer eigenen Instanziierung befassen müssen. Hinzu kommt, dass sich zwar inzwischen die Auszeichnungssprache Extended Markup Language (XML) als Format für Konfigurationsdateien etabliert hat, die einzelnen Dateien in den meisten Fällen aber individuell strukturiert sind.

Im Rahmen des Spring Framework wird für Dependency Injection auch der Begriff Inversion Of Control (IOC) verwendet. IOC bedeutet bei der Verwendung eines Frameworks, dass sich nicht mehr der Benutzer, sondern das Framework mit der Instanziierung der erstellten Klassen befasst. Inzwischen hat sich aber die von Fowler in [Fow04] geprägte und treffende Bezeichnung Dependency Injection durchgesetzt. Für die Definition von Objekten und deren Abhängigkeiten wird eine standardisierte XML Konfigurationsdatei verwendet. Diese Konfiguration wird von dem *Dependency Injection Container* gelesen, der dann mittels Reflection die dort beschriebenen Objekte instanziiert. Die Konfigurationsdatei wird *Context* und die daraus erstellte Objektstruktur *Application Context* genannt. Die Klassen, die in einem Spring Context instanziiert werden, müssen lediglich der schon bei JavaBeans üblichen Konvention der *Properties* folgen, also entsprechende Getter und Setter Methoden bereitstellen. Für eine so definierte Klasse wird auch oft der Begriff Plain Old Java Object (POJO) verwendet, da es sich im Grunde um eine einfache Java Klasse handelt, die lediglich gewisse Konventionen einhält.

Bei der Entwicklung in Java ist es üblich, die Beschreibung einer Schnittstelle durch die Definition eines *Java Interface* von der eigentlichen Implementierung zu trennen. Dadurch können andere Implementierungen verwendet werden, ohne dass die Benutzer des Interface davon Kenntnis benötigen. In Verbindung mit Dependency Injection wird die Verwendung von Interfaces vollständig transparent gestaltet, da die Instanziierung der

passenden Implementierung in der Context Definition, also vollständig unabhängig von dem Objektmodell der Anwendung, vorgenommen wird. Ein vollständiges Beispiel zur Verwendung von Dependency Injection in Spring ist in Anhang A zu finden.

2.2.4.2 *Aspektorientierte Programmierung*

Aspect Oriented Programming (AOP) ergänzt die objektorientierte Programmierung um einen anderen Ansatz zur Modularisierung der Anwendungslogik (vgl. [Ju09] Abschnitt 7.1). In der objektorientierten Programmierung wird die Anwendungslogik in einzelne Klassen unterteilt, die Datenelemente und Methoden besitzen. Im Gegensatz dazu ist das Konzept der aspektorientierten Programmierung die Unterteilung in *Aspekte*, die sogenannte *Cross Cutting Concerns* repräsentieren. Dabei handelt es sich um Funktionalitäten, die nicht direkt einer einzelnen Klasse zugeordnet werden können.

Das ist beispielsweise der Fall, wenn die verschiedenen Methodenaufrufe einer Anwendung protokolliert werden sollen. In der klassischen objektorientierten Programmierung muss zu diesem Zweck in jeder Methode der für die Protokollierung notwendige Aufruf eingefügt werden. Mit der Erweiterung der aspektorientierten Programmierung ist es nun möglich, den Aspekt der Protokollierung von Methodenaufrufen an einer zentralen Stelle, beispielsweise in einer eigenen Klasse, zu definieren. Anschließend wird dieser Aspekt als *Advice* in die Aufrufhierarchie der betroffenen Methoden eingefügt. Dadurch wird der Methodenaufruf an einer festgelegten Stelle unterbrochen und der an dieser Stelle eingefügte Advice ausgeführt. Der Advice kann nun entscheiden, ob der Methodenaufruf weiter ausgeführt werden soll oder nicht. Das Spring Framework unterscheidet vier verschiedene Advice Typen (vgl. [Ju09] Abschnitt 7.1.1):

BEFORE Ein Before Advice wird vor einem Methodenaufruf ausgeführt und kann die Methodenausführung nicht unterbrechen.

AFTER RETURNING Kehrt eine Methode ohne Fehler zum Aufrufer zurück, wird der After Returning Advice aufgerufen.

AFTER THROWING Der After Throwing Advice wird aufgerufen, wenn die betroffene Methode eine Exception wirft.

AROUND Der Around Advice kann Operationen vor und nach dem Aufruf einer Methode ausführen und außerdem entscheiden, ob der eigentliche Methodenaufruf weiter ausgeführt wird oder nicht.

Um einen Advice entsprechend ausführen zu können, erstellt das Spring Framework ein sogenanntes *Proxy* Objekt, das zwischen dem Aufrufer und dem *Zielobjekt* vermittelt. Für den Aufrufer verhält sich das Proxy Objekt

wie das eigentliche Zielobjekt. Der Unterschied bei einem Methodenaufruf soll in Abbildung 4 verdeutlicht werden.

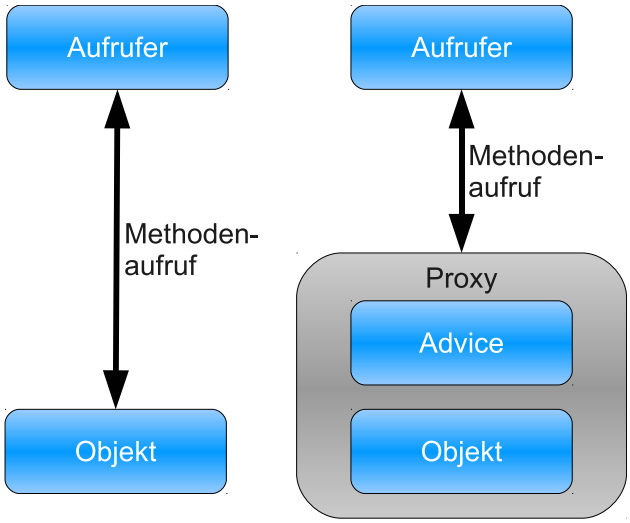

Abbildung 4: Verwendung eines AOP Proxy für einen Methodenaufruf

2.2.4.3 *Weitere Merkmale*

Neben den oben vorgestellten Eigenschaften der Dependency Injection und aspektorientierten Programmierung unterstützt das Spring Framework eine Reihe weiterer Technologien und Architekturprinzipien (entnommen aus [Ju09]):

CONVENTION OVER CONFIGURATION Viele Anwendungsframeworks auf der Java Plattform sind sehr flexibel gestaltet und benötigen deshalb umfangreiche und detaillierte Konfigurationseinstellungen, um bei der Entwicklung einer konkreten Anwendung verwendet werden zu können. Im Gegensatz dazu sieht Convention Over Configuration vor, dass der Konfigurationsaufwand durch sinnvolle Standardeinstellungen und Namenskonventionen so gering wie möglich gehalten werden kann. Es ist ein bei Entwicklern inzwischen sehr beliebtes Prinzip. Deshalb soll auch in den hier vorgestellten Implementierungen wenn möglich nach dem Convention Over Configuration Prinzip vorgegangen werden.

SECURITY MODUL Spring Security ist ein Unterprojekt des Spring Framework für die Absicherung von Anwendungen, die auf dem Spring Framework basieren. Die verschiedenen Merkmale und Funktionen von Spring Security werden in Abschnitt 4.2.3 näher vorgestellt.

VEREINFACHTE ORM NUTZUNG Das Spring Framework bietet außerdem Klassen an, die die Nutzung von verschiedenen Object-Relational

Mapping (ORM) Frameworks vereinheitlichen und vereinfachen. ORM Technologien werden in Kapitel 3.2.3 genauer behandelt.

REMOTE ANBINDUNG Durch den Dependency Injection Container kann das Spring Framework die Arbeit mit verschiedenen RPC Protokollen deutlich vereinfachen. Eine Möglichkeit für die Verwendung der bereitgestellten Klassen ist in Abschnitt 5.2.1 zu finden.

WEB MVC FRAMEWORK Eine Weboberfläche ist eine Möglichkeit für die Darstellung im Presentation Layer einer Anwendung. Hierfür bietet das Spring Framework ein Modul, das auf Basis der anderen Spring Framework Funktionalitäten die Erstellung von Webapplikationen in Java unterstützt.

2.3 REMOTE PROCEDURE CALL

Ein RPC, oder auch entfernter Prozeduraufruf, erlaubt die Ausführung von Prozeduren und Methoden in einem anderen Adressraum. Die Adressräume können sich auch auf physikalisch unterschiedlichen Systemen befinden, die über ein Netzwerk verbunden sind. Die Grundidee des RPC geht auf [Whi75] vom 14. Januar 1976 zurück. Darin wird die Notwendigkeit eines generischen Protokolls beschrieben, das den Aufruf von Operationen mit mehr als einem Parameter auf einem anderen Host des ARPANET[10] ermöglicht.

Das RPC Modell selbst beschreibt Sun Microsystems im Jahr 1988 in [Mic88] und [Sri95]. So soll ein RPC ähnlich wie ein lokaler Methodenaufruf arbeiten. Bei einem lokalen Aufruf stellt der Aufrufer eine Liste von Argumenten zur Verfügung und übergibt diese an eine Methode oder Funktion. Anschließend übernimmt diese den weiteren Kontrollfluss des Programms und kehrt am Ende wieder mit einem Ergebnis zum Aufrufer zurück.

Im Gegensatz dazu erstreckt sich der Aufruf eines RPC über zwei Prozesse, nämlich den Aufrufenden- und den Serverprozess. Der aufrufende Prozess (Client) sendet eine Nachricht mit den Aufrufparametern an den Serverprozess (Server) und wartet auf eine Antwortnachricht. Der Server befindet sich bis dahin im wartenden Zustand. Trifft eine Nachricht ein, wird sie gelesen, die entsprechenden Operationen ausgeführt und das Ergebnis zurückgesendet. Bei einem synchronen RPC handelt es sich somit um eine direkte Client-Server Kommunikation. Das RPC Modell schreibt nicht vor, dass die Ausführung synchron erfolgen muss, auch eine asynchrone Ausführung ist möglich. Dadurch muss der Client nach dem Aufruf des RPC nicht warten, bis die Antwort des Servers eintrifft, sondern kann andere Aufgaben ausführen. Liegt das Ergebnis vor, wird der Client

10 Anfang 1976 bestand das ARPANET aus 75 Host Systemen

darüber informiert[11]. Im weiteren Verlauf dieser Arbeit sollen synchrone RPC Protokolle behandelt werden.

2.3.1 *Herausforderungen*

Im Vergleich zu lokalen Methodenaufrufen sind bei entfernten Methodenaufrufen verschiedene zusätzliche Faktoren zu berücksichtigen:

FEHLERQUELLEN Neben Fehlern, die auch bei lokalen Methodenaufrufen auftreten können, muss sich der Client mit Problemen befassen, die durch die bei einem RPC notwendige Kommunikation und die Ausführung in zwei getrennten Ausführungsumgebungen auftreten können. So kann es vorkommen, dass der Serverprozess gar nicht oder in einer unerwarteten Form antwortet, die Verbindung während der Kommunikation abbricht, oder das Netzwerk nicht erreichbar ist.

KOMMUNIKATIONSOVERHEAD Für die Kommunikation mit einem anderen Prozess muss eine Nachricht in eine Form gebracht werden, die über das Netzwerk übertragen werden kann. Nach der Übertragung müssen die empfangenen Daten dann auf der Serverseite so konvertiert werden, dass der Serverprozess damit arbeiten kann. All diese Schritte sind mit zusätzlichem Aufwand verbunden weshalb ein RPC deutlich langsamer als ein lokaler Methodenaufruf ist.

KEIN GLOBALER ZUSTAND Da der Serverprozess nicht auf den Adressraum des Client zugreifen kann, hat er keine direkten Informationen über dessen globalen Zustand. Für die Bereitstellung der benötigten Informationen wird zwischen *zustandsloser* und *zustandsbehafteter* Kommunikation unterschieden. Bei einer zustandslosen Kommunikation muss der Client mit jedem RPC sämtliche Informationen übertragen, die der Serverprozess benötigt, um diesen bearbeiten zu können. Bei einer zustandsbehafteten Kommunikation authentifiziert sich ein Client zu Beginn bei dem Serverprozess. Die dabei gewonnene Information kann bei nachfolgenden Anfragen dazu verwendet werden, den Client eindeutig zu identifizieren. Dadurch kann ein serverseitiger globaler Zustand für alle Anfragen eines Client geschaffen werden.

BESCHREIBUNG Die von einem Server bereitgestellten Methoden müssen in einer für den Client verständlichen Form beschrieben werden. Für diesen Zweck bieten einige RPC Protokolle jeweils eine eigene Interface Description Language (IDL) an.

11 Beispiel einer asynchronen Kommunikation ist der Empfang und Versand von E-Mails

LOKALISIERUNG Neben der Beschreibung der bereitgestellten Methoden muss ein Client auch wissen, wo und wie diese zu finden sind. Oft werden hierfür Namensdienste verwendet, wo einem Client über einen ihm bekannten Namen die notwendigen Informationen bereitgestellt werden, mit welchen er sich dann mit dem gewünschten Server verbinden kann.

AUTHENTIFIZIERUNG Werden entfernte Methodenaufrufe über ein unsicheres Netzwerk übertragen, muss sich ein Client gegenüber dem Server authentifizieren.

2.3.2 *Methoden- und Funktionsaufrufe*

Das RPC Prinzip wurde erstmals zu einem Zeitpunkt beschrieben, als die prozedurale Programmierung das vorherrschende Programmierparadigma war. Die zu übertragenden Informationen bestanden dabei aus Prozedurname und den Parametern. Durch die Entwicklung hin zur objektorientierten Programmierung änderten sich auch die Anforderungen der bei einem RPC zu übertragenenen Informationen. Bei einem entfernten Methodenaufruf ist es notwendig, neben dem Methodennamen und den Paramtern auch das Objekt zu identifizieren bzw. zu übertragen, auf dem diese Methode ausgeführt werden soll. Die Probleme, die sich dabei für ein RPC Protokoll ergeben, das objektorientiert, transparent und bidirektional arbeiten soll, werden in [WWWK94] näher erläutert und sollen hier nicht in allgemeiner Form behandelt werden. Herausforderungen einzelner Implementierungen werden bei der Vorstellung des jeweiligen RPC Protokolls in Kapitel 5.2 näher erläutert.

2.3.3 *Webservices*

Eine besondere Form eines RPC sind Webservices. Formell wurden Webservices von einer Arbeitsgruppe des W3C[12] Konsortium wie folgt definiert (Übersetzung frei nach [MBF+04] Abschnitt 1.4):

Ein Webservice ist ein Softwaresystem, das die Maschine zu Maschine-Kommunikation unterschiedlicher Plattformen unterstützt. Die Schnittstelle ist in einem maschinenlesbaren Format beschrieben (speziell in der Werbservice Description Language (WSDL)). Andere Systeme kommunizieren mit einem Webservice über diese Schnittstellenbeschreibung unter der Verwendung von SOAP[13] Nachrichten, die HTTP als Übertra-

12 http://www.w3.org

13 SOAP stand früher für die Abkürzung *Simple Object Access Protocol*. Da die Spezifikation in der Version 1.2 letzendlich aber alles andere als „Simple“ ausfiel wurde entschieden SOAP als Namen und nicht mehr als Akronym zu verwenden

gungsprotokoll und XML sowie andere Web Standards für den Austausch verwenden.

Die Webservice Technologien um SOAP und die verschiedenen Erweiterungen haben sich zu einem sehr komplexen Themengebiet entwickelt und sollen im Rahmen dieser Arbeit nicht weiter behandelt werden. Ein anderer Webservice Ansatz in Form von Representational State Transfer (REST)[14], der momentan vor allem in Webanwendungen wachsenden Zuspruch findet, wird in Abschnitt 5.2.3 vorgestellt. Das XML-RPC Protokoll, das eine Art Vorgänger von SOAP darstellt und lediglich einfache Prozeduraufrufe erlaubt, wird in Abschnitt 5.2.2 beschrieben.

2.4 AUSTAUSCHFORMATE

Um Daten über ein Netzwerk übertragen oder in einer Datei speichern zu können, müssen sie in einer Form vorliegen, die die Übertragung und Wiederherstellung dieser Daten verlustfrei ermöglicht. Die Konvertierung in ein entsprechendes Format wird *Serialisierung* genannt, der umgekehrte Fall *Deserialisierung*. Dabei können die gleichen Daten in verschiedenen Formaten dargestellt werden, von denen einige in folgenden Abschnitten beschrieben werden.

2.4.1 *Binärformate*

Eine direkte Möglichkeit der Übertragung ist es, die vorhandenen Daten, so wie sie im lokalen Adressraum der Anwendung vorliegen, in binärer Form zu übertragen. Der dafür notwendige Aufwand ist äußerst gering und eine Serialisierung der Daten nicht notwendig. Ein direktes Abbild des Speichers kann aber oft nur von der Programmiersprache, in der die Anwendung entwickelt wurde, interpretiert werden. Deshalb sind native binäre Austauschformate sehr anwendungsspezifisch und nicht portabel.

Allgemeine binäre Austauschformate, mit denen sich beliebige Daten portabel übertragen lassen, sind häufig mit einer komplexen Spezifikation verbunden, da neben der Strukturierung der Daten auch festgelegt werden muss, in welcher Form diese kodiert werden. Dadurch kann der notwendige Serialisierungs- und Deserialisierungsaufwand dann deutlich höher ausfallen.

2.4.2 *Extended Markup Language*

Bei dem immer größer werdenden Umfang vernetzter Informationen in verschiedenen Formaten wird die digitale Erfassung, Kategorisierung und Durchsuchung dieser Daten deutlich erschwert. Viele Austauschformate

14 Für eine genauere Einordnung siehe [MBF+04] Abschnitt 3.1.3

berücksichtigen auch nicht die Anforderungen, die sich an Portabilität und Interoperabilität ergeben, wenn die Daten in einem heterogenen Netzwerk verwendet werden sollen. Deshalb wurden verschieden Auszeichnungssprachen (*Markup Language*) entwickelt, die eine allgemeine textuelle Darstellung strukturierter Informationen erlauben. Der erste Standard für die Definition von Auszeichnungssprachen wurde im Jahr 1986 in Form der Standardized General Markup Language (SGML) [ISO86] verabschiedet. Aufgrund seiner hohen Flexibilität ist SGML aber auch sehr komplex.

Eine durch SGML beschrieben Auszeichnungssprache ist die Hypertext Markup Language (HTML) (aktuell ist die Version 4.01, siehe [JRH99]). Sie bildet die Grundlage des heutigen World Wide Web. Dabei wurde versucht HTML für Entwickler einfach zugänglich zu machen und durch eine einfachere Syntax die Komplexität der Webbrowser möglichst gering zu halten.

Neben der Weiterentwicklung von HTML wurde gleichzeitig daran gearbeitet, eine allgemeine Auszeichnungssprache zu entwerfen, die einen Großteil der Möglichkeiten der SGML bietet, aber einfacher zu benutzen ist. Als Ergebnis veröffentlichte eine Arbeitsgruppe des W3C Konsortiums im Februar 1998 den ersten Vorschlag der XML (aktuell ist die Version 5, siehe [BPM+08]). XML stellt eine Untermenge der SGML dar, deckt aber trotzdem einen Großteil der SGML Anwendungsfälle ab und soll in erster Linie dem portablen Datenaustausch im World Wide Web dienen. Durch das standardisierte Format ist grundsätzlich jeder Teilnehmer in der Lage ein XML Dokument zu lesen. Mit XML lassen sich aber nicht nur Online Inhalte sondern auch jede andere Form von Daten strukturieren, übertragen und speichern. Viele aktuelle Dateiformate setzen deshalb auf XML, und auch für die Konfiguration von Anwendungen hat sich XML inzwischen als de facto Standard durchgesetzt. Durch entsprechende Transformationsregeln ist es oft problemlos möglich, zwischen verschiedenen XML Formaten zu konvertieren.

Listing 3 zeigt ein Beispiel für ein XML Dokument, angelehnt an das JavaBean Beispiel aus Kapitel 2.2.2.

```
<?xml version="1.0" encoding="utf-8"?>
<users>
        <user>
                <username>User1</username>
                <password>e52d98c459819a11775936d8dfbb7929</password>
        </user>
        <user>
                <username>User2</username>
                <password>e54cfb3714f76cedd4b27889e1f6a174</password>
        </user>
</users>
```

Listing 3: Beispiel eines XML Dokuments

2.4.3 *JavaScript Object Notation*

JavaScript Object Notation (JSON) ist ein Textformat für die Serialisierung strukturierter Daten [Cro06]. Es handelt sich dabei um eine Untermenge der Syntaxdefinition der ECMAScript Programmiersprache. JSON definiert die primitiven Typen *String*, *Number*, *Boolean*, *Null* und die zusammengesetzten Typen *Object* und *Array*. In ECMAScript entspricht der Typ *Object* einem assoziativen Array mit einem eindeutigen String als Schlüssel. Ein Array ist eine geordnete Menge, die aus keinem oder mehreren beliebigen primitiven Typen, Objekten oder Arrays bestehen kann.

Vor allem in Anwendungen, die in ECMAScript Dialekten wie *JavaScript* oder *ActionScript* (Flash) entwickelt werden, ist JSON als Austauschformat äußerst beliebt. Dies ist vor allem darauf zurückzuführen, dass es sich bei JSON Daten um Quelltext handelt, der direkt interpretiert werden kann und als Ergebnis die entsprechenden Objekte zurückliefert. Aber auch für viele andere Programmiersprachen existieren inzwischen JSON Parser und Generatoren, da eine kompakte Darstellung möglich ist und die Serialisierung und Deserialisierung mit deutlich weniger Aufwand verbunden ist, als es beispielsweise bei XML der Fall ist.

ECMAScript ist eine dynamisch typisierte Programmiersprache, weshalb es auch für JSON keine Schemadefinition gibt. Es handelt sich aber um eine sichere Untermenge von ECMAScript. Das heißt, dass es sich bei validem JSON zwar um ausführbaren Quelltext handelt, dieser aber nur die oben beschriebenen Datenelemente enthalten kann. Zuweisungen, Funktionsdeklarationen und Funktionsaufrufe sind nicht möglich.

```
{
        "users" :
        [
         {
                 "username" : "User1",
```

```
                "password" : "e52d98c459819a11775936d8dfbb7929"
        },
        {
                "username" : "User2",
                "password" : "e54cfb3714f76cedd4b27889e1f6a174"
        }
       ]
}
```

Beispiel einer Benutzerliste in JSON

2.4.4 *Äquivalenzen zu assoziativen Arrays*

Auch JSON und XML lassen sich in deserialisierter Form als assoziatives Array darstellen. Die folgende Beispiele sollen die Äquivalenz von einem JavaBean, einer Java HashMap und der entsprechenden serialisierten Form in XML und JSON aufzeigen.

Als Grundlage wird das JavaBean Beispiel aus Kapitel 2.2.2 verwendet. Der Java Quelltextauszug in Listing 5 instanziiert ein User JavaBean und eine Java Map mit identischen Datenelementen:

```
// User JavaBean
User userBean = new User();
userBean.setUsername("User");
// MD5(secret) = 5ebe2294ecd0e0f08eab7690d2a6ee69
userBean.setPassword("5ebe2294ecd0e0f08eab7690d2a6ee69");

// Java Map
Map<String, Object> userMap = new HashMap<String, Object>();
userMap.put("username", "User");
userMap.put("password", "5ebe2294ecd0e0f08eab7690d2a6ee69");
```

Listing 5: JavaBean und äquivalente Java Map

Listing 6 zeigt die nach XML serialisierte Darstellung dieser Daten. Durch die hohe Flexibilität von XML wäre auch eine anderer Aufbau möglich.

```
<?xml version="1.0" encoding="UTF-8"?>
<user>
        <username>User</username>
        <password>5ebe2294ecd0e0f08eab7690d2a6ee69</password>
</user>
```

Listing 6: XML Darstellung eines JavaBean

Das folgenden Listing 7 zeigt eine äquivalente Darstellung dieses Beans in JSON:

```
{
        "username" : "User",
        "password" : "5ebe2294ecd0e0f08eab7690d2a6ee69"
}
```

Listing 7: JSON Darstellung eines JavaBean

Die Äquivalenz von JavaBeans und assoziativen Arrays nimmt in Kapitel 5.1 eine wichtige Rolle für die Konvertierung von Methodenparametern ein. In Kapitel 5.2.3 werden für Daten in Form eines JavaBean unter anderem Darstellungsmöglichkeiten für XML und JSON implementiert.

Teil II

KONZEPTION UND REALISIERUNG

3

DATASOURCE- UND DOMAIN LAYER

In diesem Kapitel sollen der Datasource- und Domain Layer der Gesamtarchitektur vorgestellt werden. Bei der Definition des Datasource Layer wird dabei kurz auf die Systeme eingegangen, mit denen eine Anwendung üblicherweise kommuniziert. Anschließend werden die drei Kompenenten des Domain Layer und deren Aufgaben näher diskutiert.

3.1 DATASOURCE LAYER

Der Datasource Layer stellt die unterste Schicht einer Anwendung dar und dient dazu, eine Verbindung mit anderen Systemen herzustellen, die Aufgaben übernehmen, die nicht von der Anwendung selbst ausgeführt werden (vgl. [Fow02] S. 20). In den meisten Fällen handelt es sich bei diesen Systemen um eine oder mehrere Datenbanken, die zur permanenten Speicherung von Daten dienen. Aber auch andere, für die Weiterverarbeitung der Daten zuständige Anwendungen können angesprochen werden. Die für die Verbindung notwendigen Bibliotheken sollen im folgenden allgemein als *Connector* bezeichnet werden. Mögliche entfernte Systeme für die Kommunikation sind:

RELATIONALE DATENBANKEN Die meisten Anwendungen kommunizieren primär mit einem Relational Database Management System (RDBMS), das die dauerhafte Speicherung der Daten in einer relationalen Tabellenstruktur übernimmt. Der Datasource Layer stellt hierfür eine programmiersprachenspezifische Schnittstellentechnologie zur Verfügung[1], über die mit der Abfragesprache Structured Query Language (SQL) mit einem oder mehreren RDBMS kommuniziert werden kann. Die Vorteile von RDBMS sind eine solide und zuverlässige technische Grundlage sowie die Tatsache, dass ein Großteil der Entwickler bereits mit SQL vertraut ist.

NOSQL DATENBANKEN Treten aber gleichzeitig ein hohes Anfragevolumen und sehr große Datenmengen auf, kann die Skalierbarkeit von RDBMS an ihre Grenzen stoßen. Auch kann es sein, dass eine relationale Tabellenstruktur nicht immer die optimale Lösung für die Speicherung von Daten darstellt. Für diese Zwecke wurden Datenbanklösungen entwickelt, die Anfang 2009 erstmals allgemein unter dem Begriff der NoSQL Datenbanken zusammengefasst wurden

1 In Java wird hierfür JDBC verwendet

(siehe [Wik10f]). Um eine bessere Skalierbarkeit und Flexibilität zu erreichen, wird bei vielen dieser Datenbanken auf eine feste Schemadefinition verzichtet. Auch eine Abfragesprache wie SQL ist meist nicht vorhanden. Um Abfragen durchzuführen gibt es die Möglichkeit des *Query-by-example*. Dabei wird ein Beispielobjekt an die Datenbank übergeben, die alle dazu passenden Objekte zurückliefert. Auch bieten manche Datenbanken die Möglichkeit Suchalgorithmen in einer Programmiersprache zu implementieren, die nativ auf der Datenbank ausgeführt werden können. In einfachen Ausprägungen können NoSQL Datenbanken auch lediglich ein verteiltes assoziatives Array zur Verfügung stellen, in denen gespeicherte Elemente durch einen bekannten eindeutigen Bezeichner referenziert werden. Tabelle 3 zeigt eine Auswahl einiger NoSQL Datenbanken, die vor allem in großen Webanwendungen häufiger zum Einsatz kommen.

ENTFERNTE ANWENDUNGEN Entfernte Anwendungen (*Remote Application*) übernehmen Aufgaben, die nicht in der Anwendung selbst implementiert wurden. So kann beispielsweise ein Online Shop mit einem Warenwirtschaftssystem kommunizieren, das Bestellungen bearbeitet und verbucht. Die Kommunikation mit anderen entfernten Anwendungen erfolgt üblicherweise über ein RPC Protokoll. Dabei stellt der Datasource Layer eine Klassenbibliothek für das entsprechende Protokoll zur Verfügung, die dann von übergeordneten Schichten verwendet wird.

NAME	URL	BEMERKUNG
Db4O	`http://www.db4o.com`	Objektorientiert für Java und .NET
CouchDb	`http://couchdb.apache.org`	Dokumentorientiert
Neo4J	`http://neo4j.org`	Speichert Objektgraphen
Memcachedb	`http://memcachedb.org/`	Verwendet Memcached Protokoll
Cassandra	`http://incubator.apache.org/cassandra/`	Tabellenorientiert

Tabelle 3: Auswahl einiger NoSQL Datenbanken[2]

Abbildung 5 zeigt die Einordnung des Datasource Layer in die in Kapitel 1 vorgestellte Gesamtarchitektur.

2 Quelle: [Wik10f]

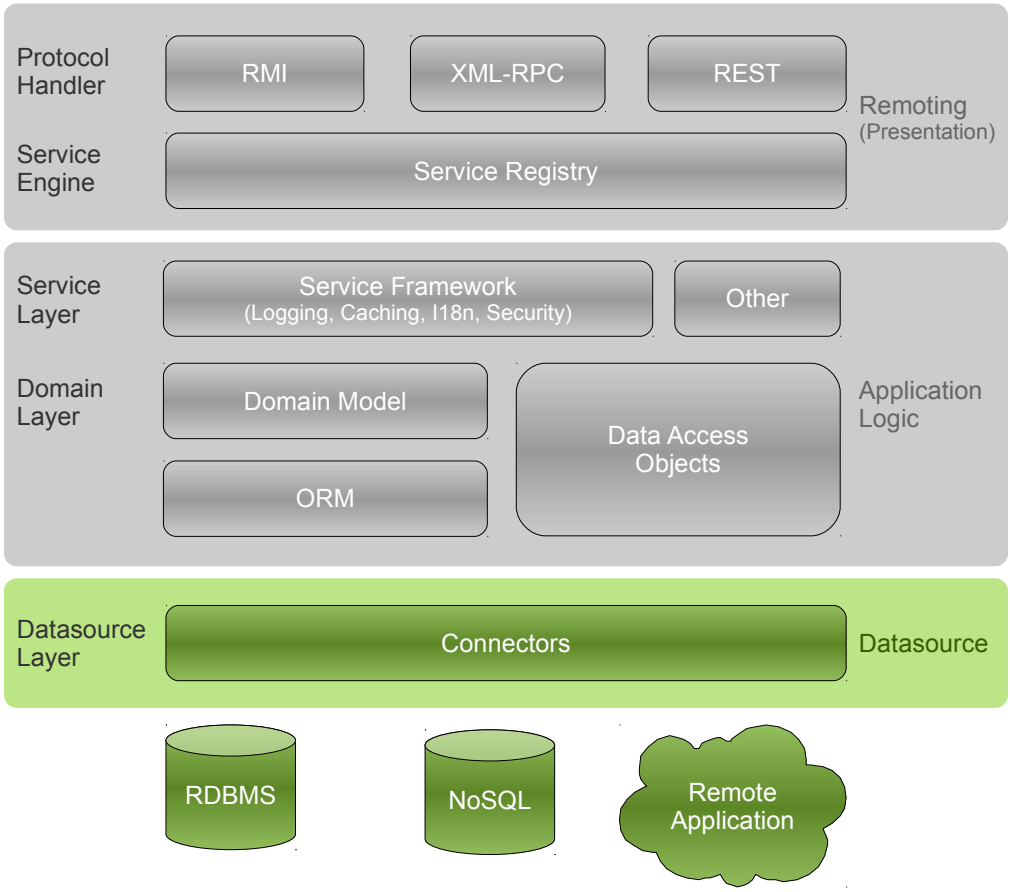

Abbildung 5: Einordnung des Datasource Layer

3.2 DOMAIN LAYER

Die dem Datasource Layer übergeordnete Schicht ist der Domain Layer. Er beinhaltet einen Großteil der Anwendungslogik und die dafür notwendige Kommunikation mit dem Datasource Layer. Daran sind verschiedene Komponenten beteiligt, die nachfolgend vorgestellt werden sollen.

3.2.1 *Domain Model*

Das Domain Model ist die Beschreibung aller in einer Anwendung verwendeten Daten, deren Abhängigkkeiten voneinander und den Operationen, die darauf ausgeführt werden (siehe [Sto06]). Der Entwurf des Domain Model ermöglicht eine konzeptionelle Sicht auf das Gesamtsystem und hilft dabei, das Verständnis für die gewünschten Funktionalitäten und das Datenmodell der Anwendung herzustellen und zu verifizieren. In modernen Programmiersprachen stellt das Domain Model dann eine objektorientierte Abbildung dieses Datenmodells dar. Umgesetzt wird die Beschreibung des Domain Models oft als Unified Modeling Language (UML) Klassendia-

gram, woraus sich dann problemlos die Klassenstruktur in einer beliebigen objektorientierten Programmiersprache erstellen lässt. Analog zu der Terminologie der objektorientierten Programmierung soll im weiteren Verlauf die Definition einer einzelnen Klasse des Domain Models als *Domain Klasse* und eine Instanz dieser Klasse als *Domain Objekt* bezeichnet werden.

Für die Implementierung der Anwendungslogik gibt es zwei unterschiedliche Ansätze (vgl. [Fow02] S. 116 - 124). Zum einen ist es möglich, diese direkt in den Domain Klassen zu implementieren. Das hat den Vorteil, das alle Daten und Operationen an einer Stelle gekapselt werden und direkt ersichtlich sind. Ein Nachteil ergibt sich daraus dann, wenn die Domain Klassen sehr viel Anwendungslogik enthalten und dadurch sehr unübersichtlich werden können. Hinzu kommt, dass eine Domain Klasse dadurch nur schwer in anderen Anwendungen oder anderen Teilen der Anwendung wiederverwendet werden kann. Oft ist es auch nicht möglich, auf diese Art Operationen zu implementieren, die auf verschiedenen Domain Klassen arbeiten. Die andere Möglichkeit besteht darin, Domain Klassen teilweise oder vollständig als reine Datencontainer zu benutzen und die Anwendungslogik in den in Kapitel 4 näher vorgestellten Service Layer auszulagern. Auch wenn die notwendige Vereinfachung die tatsächlichen Vorteile eines Domain Model nicht optimal veranschaulicht, soll Abbildung 6 die Idee des Domain Model näher illustrieren.

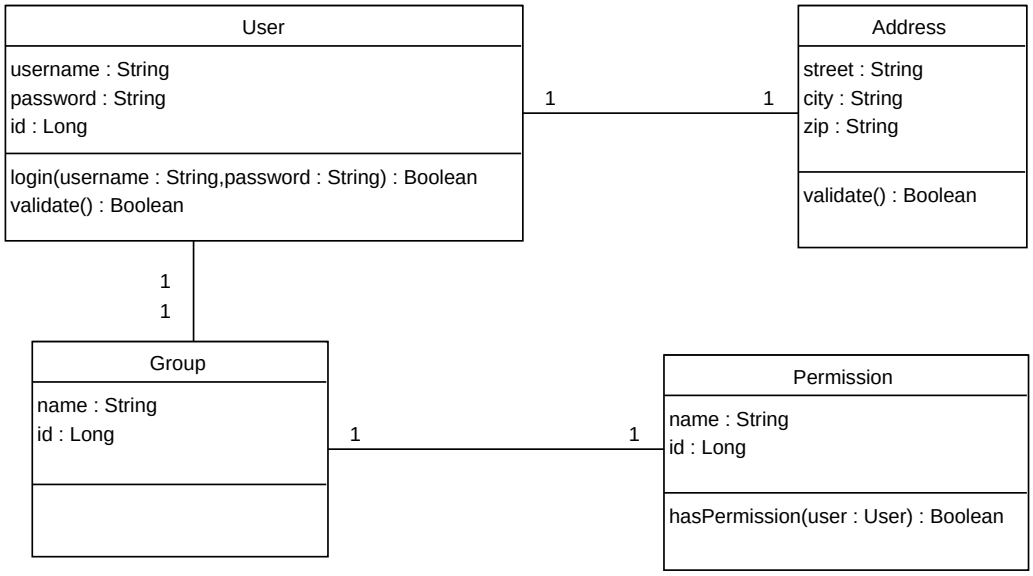

Abbildung 6: Beispiel eines einfachen Domain Model UML Diagramms

Für beide Implementierungsansätze ist es aber weiterhin sinnvoll, Funktionalitäten, die die Konsistenz des Domain Models sicherstellen, den Domain Klassen selbst zu überlassen. Darunter fällt etwa die Prüfung auf die Einhaltung bestimmter Geschäftsregeln, deren Nichtbeachtung einen

unzulässigen Zustand des gesamten Domain Models zur Folge hätte [3]. Da diese Art der Validierung meist sehr stark von den Daten der Domain Objekte abhängig ist, sollte sie ebenfalls an dieser Stelle implementiert werden.

3.2.2 *Data Access Objects*

Auch wenn das Domain Model die Anwendungslogik enthält, ist es oft sinnvoll, die Domain Objekte von der Verbindung zum Datasource Layer zu entkoppeln. Das bedeutet dann, dass der Datasource Layer verändert werden kann, beispielsweise durch einen Wechsel von einem RDBMS zu einer NoSQL Datenbank, ohne das die Domain Objekte angepasst werden müssen. Für diesen Zweck werden *Data Access Objects* (für die Java Spezifikation siehe [sun02]) verwendet, die auf den Domain Objekten arbeiten und mit dem Datasource Layer kommunizieren. Meist existiert eine DAO Implementierung für jeweils eine Domain Klasse. Ein DAO kann als Vermittler zwischen dem Datasource Layer, dem Domain Model und dem Benutzer, also der Schicht über dem Domain Layer, gesehen werden. Dieser Benutzer arbeitet weiterhin mit Domain Objekten, hat aber zusätzlich in den verschiedenen DAO Klassen Methoden zur Verfügung, die Domain Objekte aus dem Datasource Layer lesen. Die Implementierung eines DAO übernimmt in diesen Methoden dann die Kommunikation mit dem Datasource Layer und die Erstellung passender Domain Objekte.

3.2.3 *Object-Relational Mapping*

Die häufigste Aufgabe eines DAO ist die Instanziierung von Domain Objekten aus den Ergebnissen von SQL Abfragen auf einer relationalen Datenbank. Die objektorientierte Darstellung des Domain Model lässt sich nämlich nur sehr selten direkt in eine relationalen Tabellenstruktur übertragen. Für die Konvertierung der Darstellungen gibt es verschiedene Lösungsmöglichkeiten. Die verschiedenen Ansätze werden in [Fow02] Kapitel 10 - 14 näher behandelt. Hier soll der verwendete Ansatz der Object-Relational Mapping Frameworks näher behandelt werden.

Grundsätzlich müssen die Ergebnisse einer SQL Abfrage an das RDBMS auf die entsprechenden Domain Objekte übertragen werden, was sich am einfachsten direkt programmatisch in einem DAO lösen lässt. Mit der Weiterentwicklung objektorientierter Technologien wurde dieses Vorgehen weiter abstrahiert und automatisiert. Technologien und Bibliotheken, die diese Aufgabe übernehmen, werden allgemein unter dem Begriff des ORM zusammengefasst. ORM Frameworks stellen einen Vermittler zwischen dem

3 Einfache Beispiele sind die Validierung einer E-Mail Adresse oder sicherzustellen, dass ein Benutzername nur einmal vorkommt

Domain Model und dem verwendeten RDBMS dar. Sie automatisieren somit die Aufgaben eines DAO. Ein DAO kommuniziert bei der Verwendung eines ORM dann mit dem Application Programmers Interface (API) des ORM.

3.2.3.1 *Abbildung auf eine relationale Datenbank*

Ein Großteil der aktuellen ORM Bibliotheken übernimmt die Generierung des Datenbankschemas aus dem Domain Model. Dadurch, und da der Benutzer des Domain Layer nur noch mit den Domain Objekten selbst arbeitet, ist oft ein transparenter Austausch des verwendeten RDBMS möglich. Für die Zuordnung von Domain Klassen in ein relationales Datenbankschema wenden die meisten ORM Bibliotheken die folgenden Regeln an (vgl. [Wik09]):

KLASSEN Eine Domain Klasse stellt eine Tabelle in der Datenbank dar. Lässt sich der Wert eines Datenmembers direkt in einer Spalte speichern, wird diese angelegt. Das ist bei allen Datentypen, die direkt durch das RDBMS unterstützt werden, der Fall. Üblicherweise wird zusätzlich noch eine Spalte mit einem eindeutigen Primärschlüssel angelegt.

REFERENZEN Hat eine Domain Klasse eine Referenz auf eine andere Klasse, so wird dies durch eine Fremd- und Primärschlüssel Abhängigkeit in der Datenbank dargestellt. Dabei gibt es verschiedene Arten von Beziehungen zu berücksichtigen. Es ist häufig der Fall, dass sich diese nicht direkt über das programmiersprachenspezifische Domain Model ableiten lassen und deshalb durch zusätzliche Metainformationen, meist in Form einer externen Konfigurationsdatei oder Vermerken im Quelltext, definiert werden müssen. Die umzusetzenden Referenzen bestehen dann aus

- One to one
- Many to one
- One to many
- Many to many

Beziehungen, wobei für eine Many-To-Many Beziehung eine zusätzliche Tabelle angelegt wird.

3.2.3.2 *ORM Bibliotheken*

Inzwischen existieren ORM Frameworks für alle gängigen Programmiersprachen. Für die Programmiersprache Java wurde im Mai 2006 die Java Persistence API (JPA) veröffentlicht (siehe [DK06]), die eine einheitliche Schnittstelle für die Verwendung von ORM Bibliotheken bietet. Beliebteste Implementierung der JPA ist das Open Source Projekt Hibernate[4]. Es bietet

4 http://www.hibernate.org

neben Unterstützung für einen Großteil der aktuellen RDBMS auch eine eigene, Hibernate Query Language (HQL) genannte Abfragesprache. HQL ist an SQL angelehnt, wird aber an das objektorientierte Domain Model gerichtet. Dadurch sind auch komplexe Anfragen an das Domain Model möglich, es bleibt aber gleichzeitig die Unabhängigkeit von einem RDBMS erhalten. Die folgende Tabelle 4 zeigt eine Auswahl von ORM Bibliotheken für verschiedene Programmiersprachen.

NAME	SPRACHE	BEMERKUNG
Hibernate	Java	Open Source, JPA
Toplink[1]	Java	Oracle, JPA
Doctrine[2]	PHP	Ab PHP 5.2.3+
LINQ to SQL[3]	.NET	Teil des .NET Frameworks
nHibernate[4]	.NET	Hibernate für .NET
Django[5]	Python	Webframework mit ORM
Active Record[6]	Ruby	Teil von Ruby On Rails

Tabelle 4: ORM Bibliotheken verschiedener Programmiersprachen[7]

1 http://www.oracle.com/technology/products/ias/toplink/index.html
2 http://www.doctrine-project.org/
3 http://msdn.microsoft.com/de-de/library/bb386976.aspx
4 https://www.hibernate.org/343.html
5 http://www.djangoproject.com/
6 http://rubyonrails.org/
7 Quelle: [Wik10e]

3.2.4 *Einordnung*

Die Komponenten des Domain Layer sind, wie Abbildung 7 zeigt, als Schicht direkt über dem Datasource Layer und bereits in der Schicht der Anwendungslogik einzuorden. Durch Verwendung der DAO Komponenten kann das Domain Model vom Datasource Layer unabhängig gehalten werden. Ein DAO kann direkt mit dem Datasource Layer kommunizieren oder auch das API eines ORM benutzen. Für Benutzer des Domain Layer sind nur das Domain Model und die Methoden der verschiedenen DAO Implementierungen sichtbar.

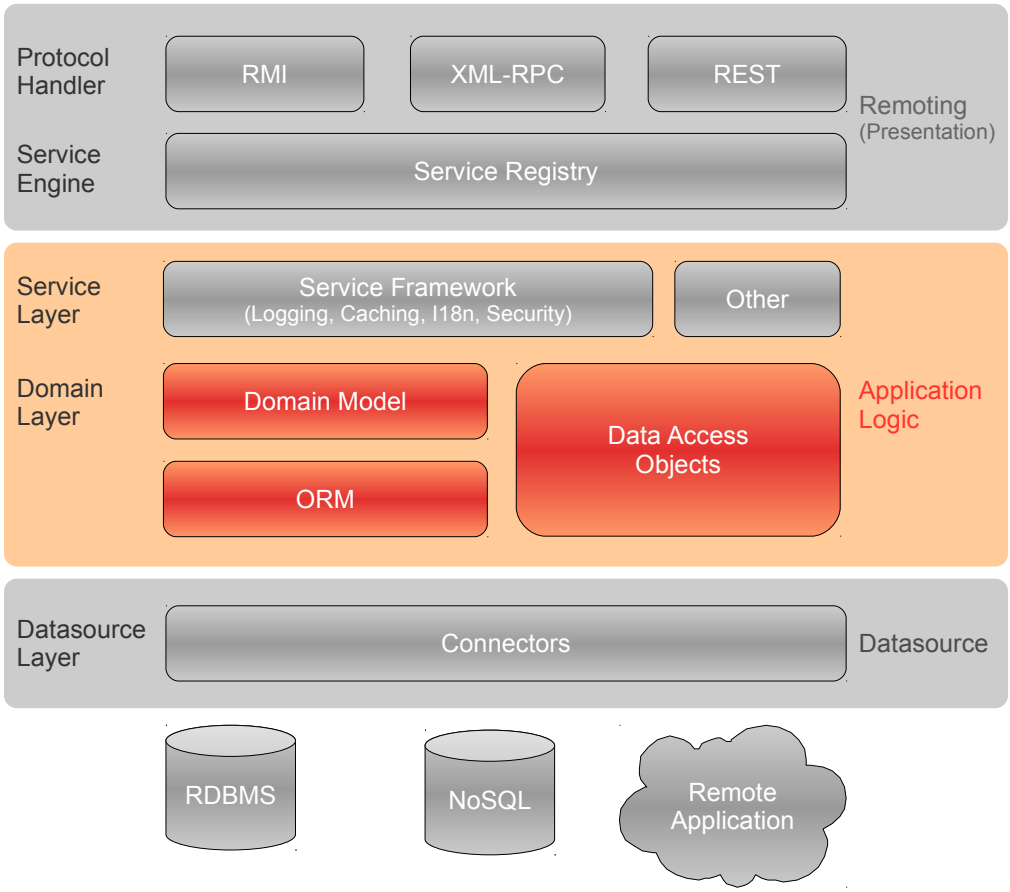

Abbildung 7: Einordnung des Domain Layer

4 SERVICE LAYER

In diesem Kapitel sollen das Konzept des Service Layer als gemeinsame Schnittstelle einer Anwendung für alle übergeordneten Schichten besprochen werden. Im zweiten Teil des Kapitels werden dann einige Implementierungen von Service Funktionalitäten vorgestellt, die in verschiedenen Anwendungsfällen einsetzbar und wiederverwendbar sind.

4.1 DEFINITION

Der Presentation Layer einer Anwendung bietet die Möglichkeit, die Anwendungslogik auf verschiedenen Wegen verfügbar zu machen. So zum Beispiel über eine grafische Anwenderoberfläche, eine Webanwendung, eine Kommandozeilenanwendung oder einen Dateneinspieler sowie über verschiedene RPC Protokolle[1]. Um eine Duplizierung von Anwendungslogik zu vermeiden ist es sinnvoll, für diese Möglichkeiten eine einzige gemeinsame Schnittstelle der Anwendung zu definieren, die auch als *Service Layer* (vgl. [Fow02] S. 133) bezeichnet wird. Für den Benutzer des Service Layer werden somit alle verfügbaren Operationen der Anwendung an einer zentralen Stelle und mit einer möglichst einfachen Schnittstelle definiert.

Einzuordnen ist der Service Layer in der Schicht der Anwendungslogik über dem Domain Layer, wie Abbildung 8 zeigt.

4.1.1 *Implementierungmöglichkeiten*

Analog zu dem in Abschnitt 3.2.1 beschriebenen Möglichkeiten für die Implementierung von Anwendungslogik im Domain Model gibt es auch für den Service Layer zwei verschiedene Implementierungsansätze (vgl. [Fow02] S. 134).

Bei der *Domain Model Facade* implementieren die Service Layer Klassen selbst keine Anwendungslogik, sondern dienen lediglich als Vermittler zwischen dem Domain Model und dem Benutzer des Service Layer. Dadurch muss sich der Benutzer nicht mit der Verwendung des möglicherweise sehr komplexen Domain Model befassen, sondern kann die meist deutlich einfacher strukturierte Schnittstelle des Service Layer verwenden.

Durch die Einführung des Service Layer als Schicht über dem Domain Model bietet sich aber auch die Möglichkeit an, die Anwendungslogik

1 Der dafür hergeleitete Remoting Layer wird in Kapitel 5 näher behandelt

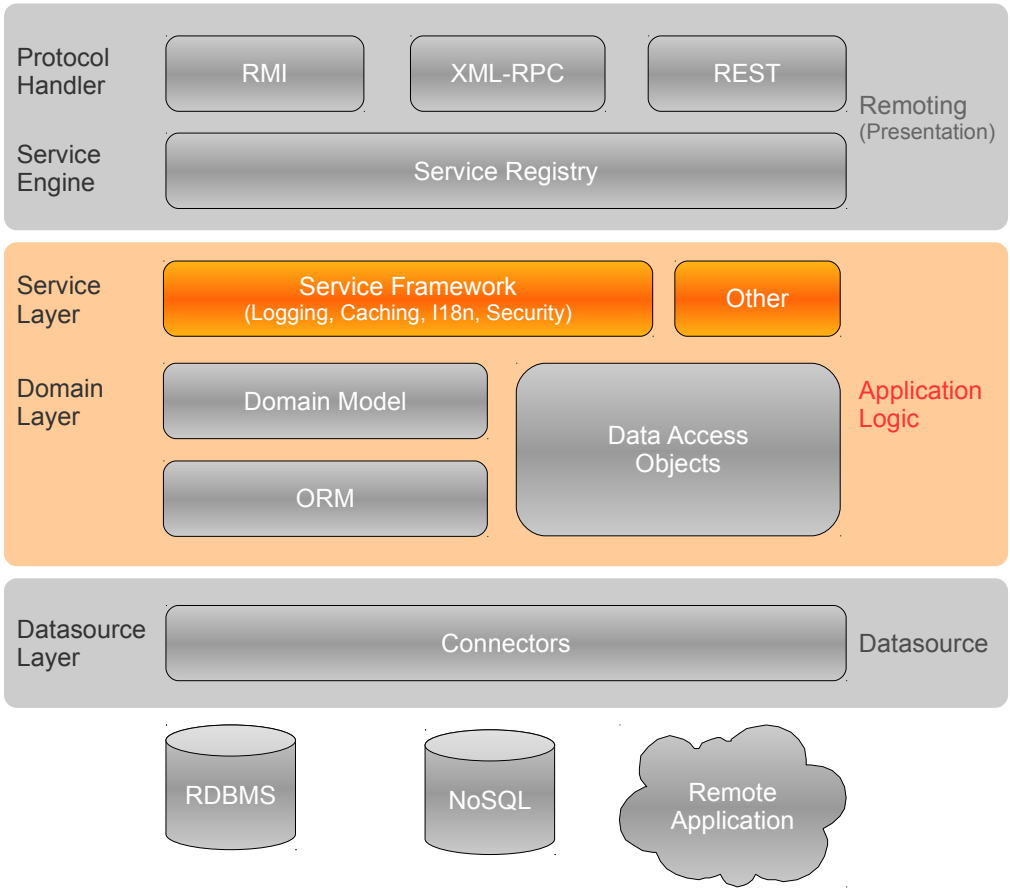

Abbildung 8: Einordnung des Service Layer

ebenfalls entsprechend zu organisieren. Hierbei unterscheidet der *Operation Script* Ansatz zwischen der allgemeinen Anwendungslogik, die in den Service Klassen implementiert wird, und Anwendungslogik, die weiterhin in Domain Klassen gehalten werden sollte und deshalb an diese weitergeleitet wird. Die Implementierungen einzelner Service Klassen fassen dabei zusamengehörige Bereiche der Anwendungslogik zusammen.

4.1.2 *Umsetzung*

Die für den Service Layer benötigten Methoden sind relativ einfach herzuleiten, da sie sich direkt nach den Anforderungen der direkt übergeordneten Schichten richten. Die wichtigste übergeordnete Schicht ist hier normalerweise eine *Anwenderoberfläche*, weshalb sich die Service Layer Methoden in erster Linie nach den Operationen ausrichten, die dort benötigt werden. Auch wenn die dafür notwendige Anwendungslogik sehr komplex sein kann, lassen sie sich aus Sicht der Schnittstelle in den meisten Fällen auf einfache Create, Read, Update, Delete (CRUD) Funktionalitäten reduzieren. Trotz der Ausrichtung der Service Methoden nach der überge-

ordneten Schicht sollte die Implementierung des Service Layer weiterhin unabhängig davon erfolgen. Das heißt, dass der Service Layer selbst keine Kenntnis diese Schichten benötigen sollte.

Schwieriger gestaltet sich die Abstraktion in passende Service Layer Klassen. Für kleinere Anwendungen kann es ausreichend sein, eine einzige Service Layer Klasse zu implementieren, die nach der Anwendung selbst benannt ist und alle notwendigen Operationen bereitstellt. Bei größeren Anwendungen gibt es die Möglichkeit, für verschiedene Teilbereiche der Anwendung jeweils eine Service Klasse zu implementieren. Ebenso ist es möglich, verschiedene Teilbereiche des Domain Model in je einer Service Klasse zu implementieren.

Auch wenn die tatsächliche Implementierung der Anwendungslogik in einer Service Klasse erfolgt, sollte die Definition der Service Layer Schnittstelle in Form von *Service Interfaces* erfolgen. Dafür wird ein Java Interface erstellt, das dann von einer Service Klasse implementiert wird. In Verbindung mit Dependency Injection muss der Benutzer dadurch lediglich Kenntniss über das Service Interface haben und kann dieses verwenden, während die implementierende Klasse problemlos ausgetauscht werden kann. Ist die Implementierung einer Service Klasse von einem anderen Service abhängig, sollte auch dort das entsprechende Interface verwendet werden.

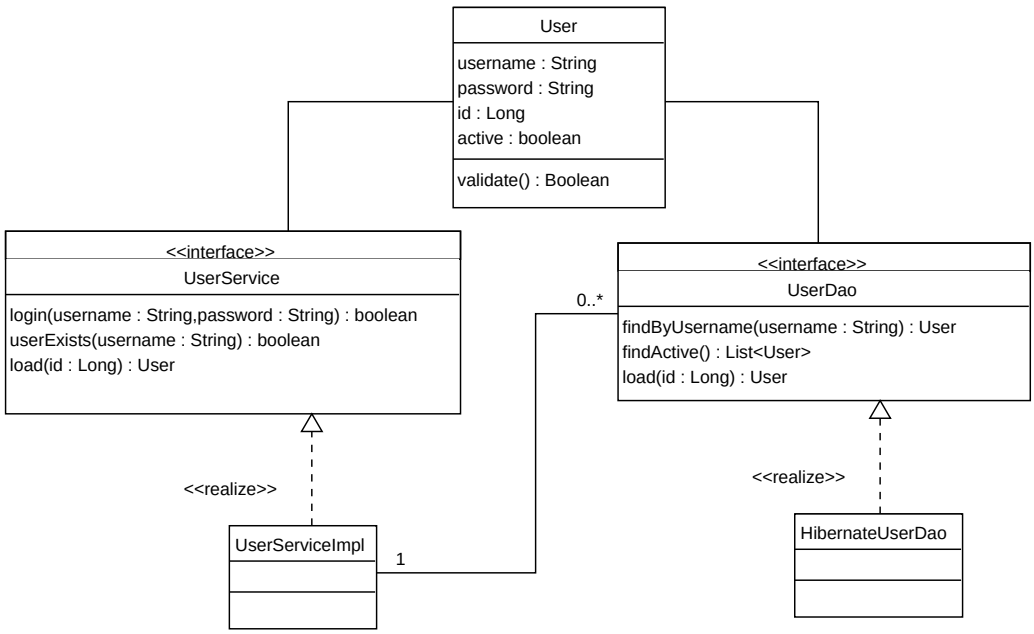

Abbildung 9: Beispiel eines Service Layer Interfaces und einem DAO

4.2 IMPLEMENTIERUNGEN

Durch die in Abschnitt 1 vorgestellten Eigenschaften einer Anwendung stellen sich auch verschiedene Anforderungen an die Implementierung des Service Layer. Viele der benötigten Funktionalitäten werden in verschiedenen Anwendungen in einer ähnlichen Form benötigt und eignen sich deshalb sehr gut für eine generische Implementierung in Form einer Klassenbibliothek. Deshalb sollen in den folgenden Abschnitten Service Interfaces und entsprechende Implementierungen vorgestellt werden, die einige dieser Aufgaben übernehmen.

4.2.1 *Anwendungscaches*

Anwendungscaches sind Zwischenspeicher, die Daten, die der Anwendung bereits vorlagen, speichern, um sie bei einem neuerlichen Zugriff schneller zur Verfügung stellen zu können. Das ist vor allem in Anwendungen sinnvoll, die eine hohe Zahl an Anfragen verarbeiten müssen und sich die bereitgetellten Daten im Verhältnis dazu selten ändern. Entsprechend der Definition von Unternehmensanwendungen ist das sehr häufig der Fall. Kommuniziert die Anwendung mit anderen entfernten Systemen, kommt hinzu, dass die direkte Abfrage von Daten meist mit mehr Aufwand verbunden ist, als es bei dem Zugriff auf einen Cache der Fall ist. Dazu zählt grundsätzlich jede Abfrage, die eine Netzwerkkommunikation benötigt, da schon die Übertragung der Daten eine gewisse Zeit in Anspruch nimmt. Ist die entfernte Anwendung eine Datenbank, sollte ebenfalls jede überflüssige Abfrage vermieden werden, um die Ressourcen des Datenbankservers für die tatsächlich anfallende Arbeitslast zur Verfügung zu haben. Das gilt auch, wenn durch die entfernte Anwendung andere umfangreiche Berechnungen vorgenommen werden und die Wahrscheinlichkeit hoch ist, dass das Ergebnis mehrmals abgefragt wird. Da die Kommunikation mit anderen entfernten Anwendungen auch mit zusätzlichen Kosten verbunden sein kann, lassen sich durch einen sinnvoll eingesetzten Cache auch Kosten sparen.

4.2.1.1 *Verdrängungsstrategien*

Auf der anderen Seite wird von einem Cache wiederum ein schneller Datenspeicher benötigt, der ebenfalls mit zusätzlichen Kosten verbunden ist. Hier muss ein passender Kompromiss zwischen Kosten und Größe des Cache Speichers gefunden werden. Dies führt in den meisten Fällen dazu, dass der verwendete Cache deutlich kleiner ausfällt als die Gesamtheit der Daten die eine Anwendung zur Verfügung stellen kann. Werden neue Daten in den Cache gelegt, wenn dieser bereits voll ist, können verschiedene Verdrängungsstrategien zur Anwendung kommen (vgl. [Wik10b]):

OPTIMAL Auch *Belady's Verfahren* genannt, ist eine optimale Verdrängungsstrategie, die immer das Element aus dem Cache entfernt, auf das in der Zukunft am seltensten zugegriffen wird. Das ist aber nur möglich, wenn der gesamte Ablauf eines Programms im Voraus bekannt ist. Da dies normalerweise in keiner Anwendung der Fall ist, kann diese Verdrängungsstrategie nicht direkt implementiert werden. Ein optimaler Cache eignet sich aber gut als Vergleich für andere Verdrängungsstrategien.

FIRST IN FIRST OUT (FIFO) Das erste Element das in den Cache eingefügt worden ist wird auch als erstes verdrängt. Die dafür notwendige Datenstruktur ist sehr einfach aufgebaut und kann sehr performant implementiert werden. Eine FIFO Verdrängungsstrategie liefert aber nur in seltenen Fällen gute Ergebnisse.

LEAST RECENTLY USED (LRU) Das Element, auf das am längsten nicht zugegriffen wurde, wird aus dem Cache verdrängt. Hierbei muss eine Datenstruktur benutzt werden, die den Zeitpunkt des letzten Zugriffs auf ein Cache Element speichert und danach sortiert. Das erste Element in dieser Liste wird dann als erstes verdängt.

LEAST FREQUENTLY USED (LFU) Das am seltensten gelesene Element wird aus dem Cache verdrängt. Entsprechend einem LRU Cache wird eine Datenstruktur benutzt, die nach der Anzahl der Zugriffe sortiert ist. Auch hier wird dann das erste Element der Liste zuerst verdrängt.

Die Optimierung der Cache Verdrängungsstrategie ist sehr anwendungsspezifisch und erfordert umfangreiche Tests und Benchmarks. Da der Programmablauf in einem Benchmark im Voraus bekannt ist, kann man hier einen optimalen Cache als Vergleich zu der aktuellen Verdrängungsstrategie heranführen. Die Kriterien für die Bewertung der Verdrängungsstrategie sind *Cache Hits*, *Cache Misses* und *False Positives*. Ein Cache Hit tritt auf, wenn das gesuchte Element im Cache gefunden wurde, ein Cache Miss, wenn nicht. Bei einem Cache Miss muss das gewünschte Ergebnis dann von der Anwendung selbst geliefert und in den Cache gelegt werden. False Positives treten bei einem Cache Hit auf, wenn das gespeicherte Element nicht mit dem eigentlichen Ergebnis der Anwendung konsistent ist. Das ist der Fall, wenn sich der Zustand der Anwendung geändert hat, aber die betroffenen Elemente im Cache nicht entfernt wurden. Hier kommt es auch auf die Anwendung selbst an, ob False Positives für einen bestimmten Zeitraum akzeptabel sind oder nicht.

4.2.1.2 *Cache Arten*

In netzwerkbasierten Anwendungen wird zwischen *lokalen* und *verteilten* Caches unterschieden. Um eine möglichst einfache Schnittstelle zur Verfügung zu stellen, werden Anwendungscaches als eine Art assoziatives Array behandelt, wobei der Schlüssel ein eindeutiger Bezeichner[2] des zu cachenden Elements und der Wert das zu speichernde Element selbst ist. Bei einem lokalen Cache werden die Cache Elemente im Arbeitsspeicher oder auf der Festplatte des Systems gespeichert, auf dem auch die Anwendung selbst ausgeführt wird. Meist existiert eine für die Programmiersprache native Schnittstelle, wodurch die Einbindung in eine Anwendung entsprechend einfach ist. Ein verteilter Cache wird über ein für diese Aufgabe passendes Netzwerkprotokoll angesprochen und kann wiederum mit anderen entfernten Caches kommunizieren.

4.2.1.3 *Implementierung*

Durch die Key-Value Store Schnittstelle der Cache Implementierungen muss das Cache Service Interface lediglich die dafür üblichen Operationen *get(key)*, *put(key, value)* und *remove(key)* bereitstellen. Um die Generierung eindeutiger Schlüssel zu vereinfachen, wurde außerdem ein *Region* Parameter eingeführt, das den Cache in virtuelle Namensräume aufteilt oder - je nach Implementierung - dazu dient, verschiedene Caches anzusprechen. Wird das Cache Service Interfaces konsequent im gesamten Service Layer verwendet, ist ein problemloser Wechsel zu einem anderen Cache System möglich.

Bei der Verwendung eines Cache müssen die betroffenen Stellen den Service Implementierungen angepasst werden. Da die Anwendung ausschließlich über die Schnittstelle des Service Layer verwendet wird, wäre es grundsätzlich möglich, alle Ergebnisse der Methodenaufrufe in einem Cache zwischenzuspeichern. Aus einem Service Layer Methodenaufruf ließe sich problemlos ein eindeutiger Schlüssel generieren, unter dem das Ergebnis im Cache abgelegt wird. Anhand der Methodenaufrufe ist aber meist nicht erkennbar, ob und wie der Zustand der Anwendung geändert wurde. Deshalb ist es kaum möglich festzustellen, welche Elemente im Cache gelöscht werden müssten um False Positives zu vermeiden. Aus diesem Grund bleibt die Verwendung des Cache Service weiterhin dem Entwickler überlassen. In dem Pseudocode Beispiel in Listing 8 wird die Verwendung des Cache Service Interface illustriert.

2 Hier wird meist ein Hash Wert oder ein anderer eindeutiger String verwendet

```
function getData()
begin
        if cacheService.exists(data_key) then
                data = cacheService.get(data_key)
        else
                data = getFromDataSourceLayer()
                cacheService.put(data_key, data)
        return data
end

function updateData(data)
begin
        updateInDataSourceLayer(data)
        cacheService.remove(data_key)
end
```

Listing 8: Pseudocode für die Verwendung des Cache Service

Das Cache Service Interface wurde für das lokale Cache System EhCache und den verteilten Cache Memcached implementiert.

EHCACHE EhCache[3] ist grundsätzlich ein lokaler Cache für Java, der im Arbeitsspeicher der Anwendung arbeitet, aber auch Daten auf der Festplatte ablegen kann. Durch die Möglichkeit, die Cache Inhalte über ein Netzwerk zu replizieren, ist auch die Verwendung als verteilter Cache möglich. Für die Verwendung als verteiltert Cache bietet der EhCache Hersteller Terracotta[4] auch kommerzielle Lösungen an.

MEMCACHED Memcached[5] ist ein verteilter Cache, der die Daten normalerweise im Arbeitsspeicher der Cache Server ablegt. Memcached ist einfach zu verwenden und hat sehr gute Skalierungseigenschaften, weshalb es vor allem bei großen Webapplikationen eingesetzt wird um die Datenbanken zu entlasten. So betrieb das soziale Netzwerk Facebook[6] Ende 2008 ein aus 800 Servern bestehendes Memcached Cluster, das insgesamt 28 TB RAM als Cache zur Verfügung stellte (siehe [Saa08]).

4.2.2 *Domain Model Zugriff*

Eine der wichtigsten Aufgaben des Service Layer ist es, dem Benutzer Zugriff auf die im Domain Model verarbeiteten Daten zu ermöglichen. Auch

3 http://ehcache.org/
4 http://www.terracotta.org/
5 http://memcached.org/
6 http://facebook.com

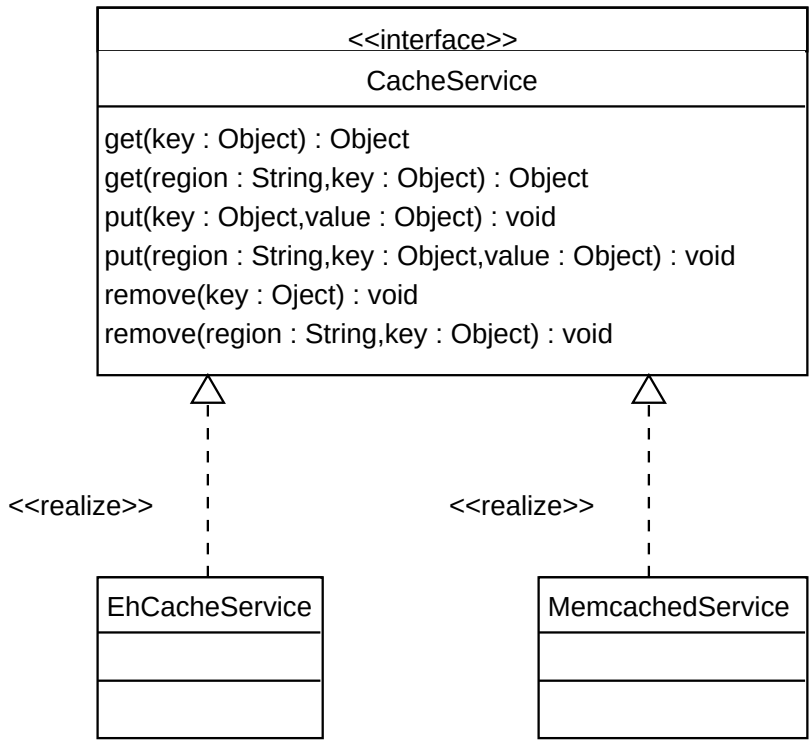

Abbildung 10: UML Diagramm des Cache Service und Implementierungen

wenn die Verarbeitung dieser Daten in der Anwendung selbst mit komplexer Anwendungslogik verbunden sein kann, handelt es sich bei den nach außen hin benötigten Methoden um grundlegende CRUD Funktionalitäten. Die Hauptaufgabe eines Service zur Domain Model Verwaltung ist also das *Erstellen*, *Lesen*, *Aktualisieren* und *Löschen* von Domain Objekten.

In den meisten Fällen kommt zur dauerhaften Speicherung des Domain Models ein RDBMS zum Einsatz, häufig in Verbindung mit einem ORM Framework. In diesem Fall sind die Domain Objekte auch gleichzeitig Entitäten, die in einem RDBMS abgelegt werden. Die CRUD Funktionen auf einem Domain Objekt werden dann normalerweise in den entsprechenden Aufruf auf der Datenbank umgesetzt.

ERSTELLEN UND AKTUALISIEREN Beide Operationen sind sich meist sehr ähnlich. Die Entscheidung, ob ein übertragenes Domain Objekt angelegt oder aktualisiert werden muss, wird normalerweise anhand des eindeutigen Bezeichners getroffen, sofern dieser vorhanden ist. Ist er nicht gesetzt, muss das Domain Objekt neu angelegt werden, ansonsten wird das Domain Objekt mit dem entsprechenden Bezeichner aktualisiert. Bei beiden Operationen sollte immer eine Validierung aller betroffenen Domain Objekte vorgenommen werden.

LESEN Im einfachsten Fall wird ein Domain Objekt anhand seines eindeutigen Bezeichners gelesen. Auch die Auflistung von Domain Objekten anhand verschiedener Auswahlkriterien muss ermöglicht werden.

LÖSCHEN Entfernt ein Domain Objekt mit einem eindeutigen Bezeichner.

4.2.2.1 *Grails ORM*

Grails ORM (GORM) ist eine Erweiterung des ORM Frameworks Hibernate für die Programmiersprache Groovy. Es ist Teil des Webframeworks *Grails*[7], kann aber inzwischen auch unabhängig davon benutzt werden. Deshalb soll hier auf die Details von Grails selbst nicht weiter eingegangen werden. Durch die dynamischen Eigenschaften von Groovy entstehen einige neue Möglichkeiten den Zugriff auf Domain Objekte, die von Hibernate verwaltet und in einem RDBMS abgelegt werden, zu vereinfachen. So erstellt GORM automatisch verschiedene CRUD Methoden, die direkt auf den Domain Objekten verwendet werden können und übernimmt somit auch Aufgaben der DAO Klassen. Methoden, die zum Lesen des Domain Objekts verwendet werden, sind hierbei statische Methoden auf der Domain Klasse. Methoden zum Speichern und Löschen werden auf dem betroffenen Domain Objekt selbst aufgerufen. Listing 9 zeigt ein Beispiel für die Verwendung von GORM.

```
@Entity
class User
{
        String username
        String password
        Boolean active = true
}

User testUser = new User(username:"test",password:"test123")
// Save in database
test.save()
// Load user with id 1
User savedUser = User.load(1)
// Delete user
savedUser.delete()
// Find all users with username test
List<User> users = User.findByUsername("test")
// Find all active users
List<User> active = User.findByActive(true)
```

Listing 9: Beispiel für die Verwendung des GORM

7 http://grails.org

4.2.2.2 *Implementierung*

In der Implementierung des Domain Model Services konnten durch die gute gegenseitige Integration von Groovy und Java die Vorteile einer dynamischen Sprache mit denen der statischen Typisierung optimal miteinander verbunden werden. Ein Service Interface muss als eindeutige, statische Schnittstelle definiert werden, weshalb es nicht möglich ist, die dynamisch durch GORM erstellten Methoden direkt anzubieten. Die Beschreibung einer Service Schnittstelle erfolgt deshalb weiterhin als Java Interface. Da Groovy aber ein Java Interface zur Laufzeit implementieren kann, ist es möglich, Aufrufe auf dieses Interface an die von GORM erstellten Methoden weiterzuleiten. Somit wird die Anforderung des Service Layer an ein eindeutiges Interface erfüllt, es ist aber gleichzeitig möglich, die dynamisch erstellten Methoden zu verwenden. Diese Möglichkeit bleibt auch bestehen, wenn Groovy nicht direkt zur Implementierung des Service Layer verwendet wird[8]. Der Entwickler muss lediglich das gewünschte Service Interface definieren, das den GORM Konventionen für die Benennung der dynamischen Methoden folgt, wie in Listing 10 gezeigt.

```
public interface UserService
{
        // User.findAll() with paging
        public List<User> findAll(Pager pager);
        // usr.save()
        public void save(User usr);
        // User.load(id)
        public User load(Long id);
        // User.findyByUsername(username)
        public User findByUsername(String username);
        // usr.delete()
        public void delete(User usr);
        // User.findByActive(active)
        public List<User> findByActive(boolean active);
}
```

Listing 10: Beispiel eines Domain Model Service Interface

Listing 11 zeigt ein Beispiel, für die Spring Context Definition zur Verwendung des Service. Eine Klasse, die das Service Interface implementiert ist nicht notwendig.

```
<bean name="userService" class="org.feathry.service.domain.
    DynamicGormService">
        <constructor-arg value="org.feathry.example.UserService" />
</bean>
```

8 Die Groovy Klassenbibliotheken müssen aber zur Verfügung stehen

Listing 11: Verwendung des Domain Model Service in einem Spring Context

Werden Listen von Domain Objekten ausgelesen, möchte man die zurückgelieferten Elemente oft nach verschiedenen Kriterien sortieren und die Größe der zurückgelieferten Liste beschränken und navigierbar machen. Dieser Vorgang wird auch *Paging* genannt. Die Implementierung des Paging stellt einen gewissen Zusatzaufwand dar, da jede Methode, die eine Liste von Domain Objekten zurückliefert entsprechend angepasst werden muss. Um die Arbeit damit etwas zu erleichtern, wurde eine *Pager* Klasse implementiert, die Informationen darüber enthält, welcher Bereich der Auflistung angefordert wird und nach welchen Kriterien die Elemente sortiert werden sollen. Für Service Methoden die an eine GORM Methode weiterleiten und mit einem Pager als Parameter definiert werden, wird das Paging automatisch vorgenommen. Soll Paging auch in anderen Methoden angewendet werden, muss das Pager Objekt dort programmatisch eingebunden werden. Dies kann man entweder direkt auf Ebene der Datenbankabfrage oder im Nachhinein auf der zurückgelieferten Liste der Domain Objekte umsetzen.

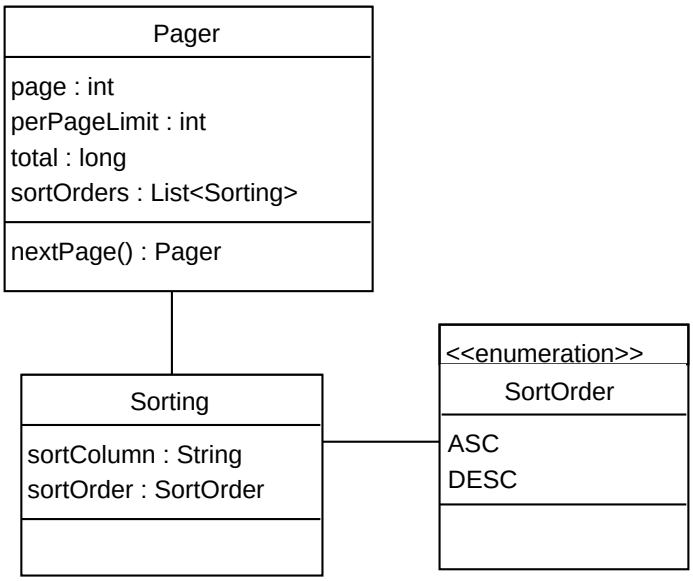

Abbildung 11: UML Diagramm der Paging Klassen

4.2.3 *Benutzerverwaltung*

Da eine Anwendung fast ausnahmslos von mehreren Benutzern verwendet wird, müssen sie in irgendeiner Form eindeutig identifizierbar sein. Auch hier stellt der Service Layer wieder einen guten Ansatzpunkt dar, um die Aufgaben der Authentifizierung und Autorisierung zu übernehmen.

AUTHENTIFIZIERUNG Authentifizierung ist der Vorgang, einen Benutzer eindeutig zu identifizieren und sicherzustellen, dass es sich um den behaupteten Teilnehmer handelt. Grundsätzlich wird zwischen drei verschiedenen Arten der Authentifizierung (vgl. [Kru04] S. 295) unterschieden.

- Bei der Authentifizierung durch *Wissen* handelt es sich üblicherweise um die Kombination aus Benutzername und Passwort.
- Ein Teilnehmer authentifiziert sich durch *Besitz*, wenn er ein Objekt mit den entsprechenden Authentizitätsinformationen besitzt, beispielsweise eine Chipkarte.
- Ferner ist es möglich, dass sich ein Teilnehmer durch *unveränderliche Eigenschaften*, wie zum Beispiel die biometrische Eigenschaft eines Fingerabdrucks, authentifizieren kann.
- Auch eine Kombination der verschiedenen Authentifizierungsmechanismen ist möglich.

AUTORISIERUNG Berechtigungen werden von einem Administrator der Anwendung vergeben und bestimmen, ob ein authentifizierter Teilnehmer auf bestimmte Domain Objekte und Service Methoden zugreifen darf (vgl. [Kru04] S. 292). Der Vorgang der Überprüfung, ob ein Teilnehmer die benötigten Berechtigungen hat, wird *Autorisierung* genannt. Berechtigungen auf Domain Objekten hängen in den meisten Fällen direkt mit den darauf auszuführenden CRUD Operationen zusammen. So dürfen manche Benutzer Domain Objekte lesen, erstellen und verändern, während andere lediglich lesenden Zugriff darauf haben.

Die sichere Beschaffung und Übertragung der für die Authentifizierung eines Benutzers notwendigen Informationen zum Service Layer ist weiterhin Aufgabe des Benutzers.

4.2.3.1 *Spring Security*

Spring Security (vormals Acegi Security) ist ein Unterprojekt des Spring Frameworks, das umfangreiche Möglichkeiten für die Authentifizierung und Autorisierung von Benutzern bietet. Einige nennenswerte Protokolle zur Benutzerauthentifizierung sind:

LDAP LDAP ist eine vor allem in großen Unternehmen häufig genutzte Möglichkeit, auf zentral gespeicherte Benutzerinformationen zugreifen zu können.

HTTP AUTHENTIFIZIERUNG Auch die im Hypertext Transport Protocol (HTTP) Protokoll spezifizierten Authentifizierungsmechanismen werden unterstützt.

OPENID OpenID ist ein dezentralisierter, offener Standard für die Benutzerauthentifizierung. An einem Authentifizierungsvorgang beteiligt sind die Anwendung selbst, der zu authentifzierende Benutzer und ein OpenID Anbieter, bei dem der Benutzer seine Zugangsinformationen gespeichert hat.

USERSERVICE Spring Security bietet auch die Möglichkeit, ein eigenes UserService Interface zu implementieren. Dadurch kann die Anwendung die für die Authentifizierung notwendigen Informationen selbst bereitstellen.

Hat sich ein Benutzer erfolgreich authentifiziert, legt Spring Security diese Informationen in einem *SecurityContext* ab, der dann in der gesamten Anwendung verfügbar ist.

Ist der Context erstellt, also der Benutzer authentifiziert, bietet Spring Security verschiedene Möglichkeiten zur Autorisierung der vom Benutzer duchgeführten Aktionen. Unterstützt wird eine *rollenbasierte Authorisierung* (vgl. [PSA01]), auf dessen Grundlage Service Methoden oder einzelne Domain Objekte geschützt werden können. Die Grundidee der rollenbasierten Authorisierung ist, dass Berechtigungen mit Rollen[9] verbunden sind und den Benutzern dann diese Rollen zugewiesen werden.

4.2.3.2 *Implementierung*

Der Service zur Benutzerverwaltung übernimmt zwei Aufgaben. Zum Einen die Administration von Benutzern und Rollen, zum Anderen die Authentifizierung und Authorisierung eines Benutzers. Neben der Erstellung und Implementierung des entsprechenden Service Interface ist es deshalb auch notwendig, von GORM verwaltete Domain Klassen für Benutzer und Rollen zu erstellen. Abbildung 12 zeigt das entsprechende Domain Model und die Interfaces, die aus dem Spring Security Framework verwendet wurden.

Da der Service Layer unabhängig von seinen Benutzern als eigene Schicht arbeiten soll, übernimmt er keine implementierungsspezifischen Aufgaben des verwendeten Authentifizierungsmechanismus. Ist der Benutzer beispielsweise eine Weboberfläche, muss die Umwandlung von der dort verwendeten HTTP Authentifzierung in einen Security Service Methodenaufruf bereits dort erfolgen. Abbildung 13 zeigt die Methoden des Security Service Interface.

4.2.4 *Logging und Monitoring*

Dadurch, dass eine Unternehmensanwendung von vielen Benutzern verwendet wird und im Normalfall auch mit anderen entfernten Anwen-

9 Mögliche Rollen sind beispielsweise Administrator, Manager, User usw.

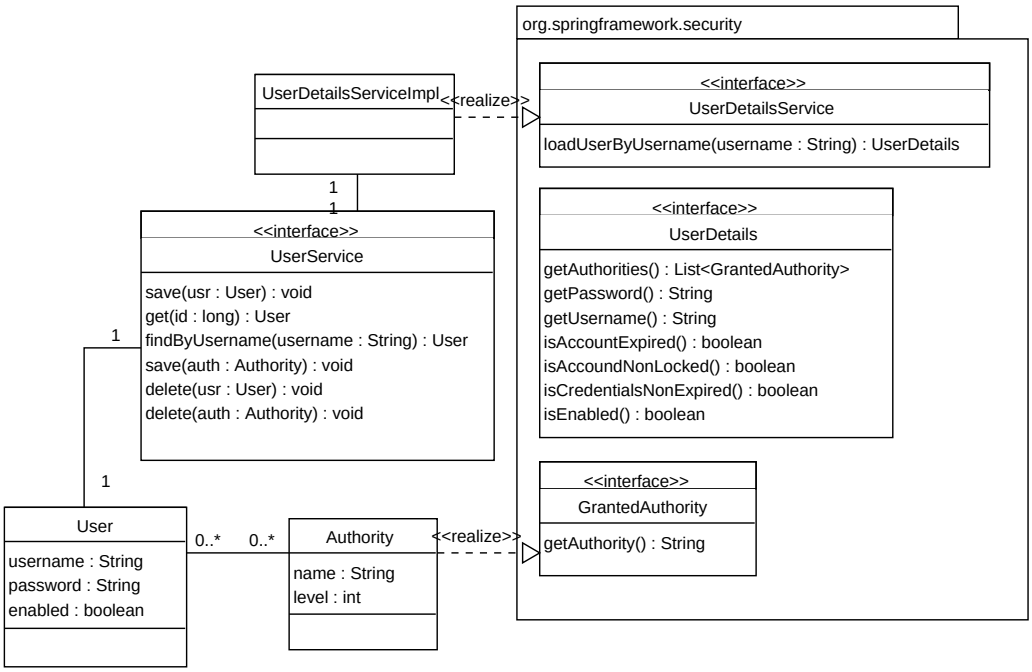

Abbildung 12: Benutzer und Rollen Domain Model mit Spring Security

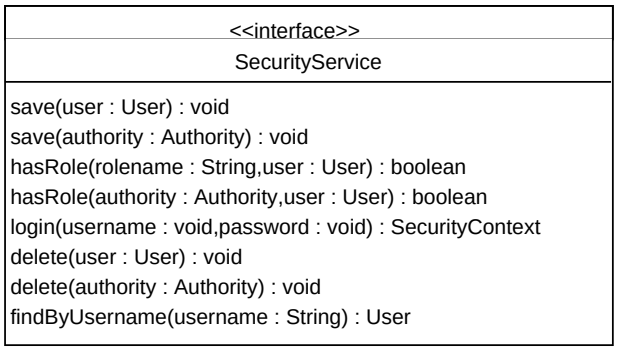

Abbildung 13: Security Service Interface

dungen kommuniziert, können Fehler auftreten, die nicht immer direkt nachvollziehbar sind. Deshalb ist es sinnvoll, den Programmablauf zu protokollieren, um Fehler leichter zu lokalisieren und auch das Nutzungs- und Laufzeitverhalten besser nachvollziehen zu können. Dies wird allgemein *Logging* genannt. In lokalen Anwendungen können Debugger, die eine Übersicht über den Zustand der Anwendung ermöglichen, für die Fehlersuche verwendet werden. In verteilten Systemen, die über entfernte Methodenaufrufe kommunizieren, ist diese Möglichkeit der Fehlersuche nicht mehr in dieser Form realisierbar. Hier können aber Statusmeldungen durch Logging hilfreich sein, um den Fehler auch in solchen Anwendungen besser eingrenzen zu können und die Administratoren über aufgetretene Fehler zu informieren. Das Logging der Ausführungszeiten einzelner

Methoden kann außerdem dabei helfen die Laufzeit der Anwendung zu optimieren. Auch die Auswertung des Benutzerverhaltens ist eine wichtige Aufgabe des Loggings. Dadurch lassen sich Rückschlüsse führen wie die Anwendung tatsächlich verwendet wird und verbessert werden kann.

Hinzu kommt der Trend, dass die Bezahlung für die Benutzung einer Anwendung individuell nach dem tatsächlichen Nutzungsvolumen erfolgt (vgl. [Bra09]). Um dies zu ermöglichen bietet sich der zentrale Logging Mechanismus ebenfalls an. Durch die Auswertung der protokollierten Benutzeraktivitäten lässt sich dann eine nutzungsbezogene Rechnung stellen.

4.2.4.1 *Logging Frameworks für Java*

Es gibt eine Vielzahl an Logging Frameworks für Java, die aber meist ähnlichen Ansätzen folgen. So werden mehrere Log Level[10] unterstützt, für die Nachrichten bzw. Exception Objekte protokolliert werden können. Auch ist es möglich, Logger an verschiedene Ausgabesysteme anzubinden. Normalerweise handelt es sich dabei um eine Datei, aber auch die Anbindung an eine Datenbank oder die Versendung von Nachrichten bei einem bestimmten Log-Ereignis sind möglich. Idealerweise soll ein Framework nicht direkt von der tatsächlichen Implementierung eines der vielen Java Logging Frameworks abhängig sein, sondern diese Entscheidung dem Benutzer überlassen. Zu diesem Zweck wurde das Simple Logging Framework For Java (SLF4J)[11] entwickelt, das entsprechend dem *Facade Design Pattern* ein Logging Interface bereitstellt und Methodenaufrufe an die vom Benutzer gewählte Logging Implementierung weiterleitet.

4.2.4.2 *Implementierung*

Die Implementierung des Logging Service stellt grundsätzlich die Methoden dieses Interfaces zur Verfügung. Zusätzlich wurde der Logging Mechanismus um sogenannte *Actions* erweitert. Diese repräsentieren jeweils Ereignisse, die ausgeführt werden, sobald sie durch den Logging Service protokolliert werden. Dadurch kann die unmittelbare Rechnungsstellung von Benutzeraktionen ebenso realisiert werden wie die Ausführung von bestimmten Wiederherstellungsoperationen bei einem kritischen Fehler. Zu Optimierung der Laufzeit können Log Meldungen und Actions außerdem durch den verwendeten Cache Service gepuffert werden, um zu einem späteren Zeitpunkt ausgeführt oder geschrieben zu werden.

10 Üblich sind FATAL, ERROR, WARN, INFO und DEBUG
11 http://www.slf4j.org

4.2.5 *Internationalisierung*

Hat die Anwendung eine sehr große Zahl an Anwendern, kann es vorkommen, dass eine Anpassung an unterschiedliche Kulturzonen und Sprachen der Anwender benötigt wird. Das ist vor allem bei öffentlich zugänglichen Webanwendungen der Fall, die auf einen internationalen Markt abzielen. Die Erfahrung hat gezeigt, dass der Aufwand für die Internationalisierung oft unterschätzt wird. Der Service Layer als gemeinsame Anwendungsschnittstelle kann hierbei ein guter Ansatz sein, um die Anwendung internationalisierbar zu machen und den damit verbundenen Mehraufwand minimal zu halten.

4.2.5.1 *Internationalisierung in Java und Spring*

Bei der Übersetzung in andere Sprachen ist es üblich, Strings, die in verschiedenen Sprachen vorkommen können, durch einen eindeutigen Schlüssel zu ersetzen. Ist die gewünschte Sprache bekannt, wird dieser Schlüssel dann durch den entsprechenden Sprachtext ersetzt. Für die Internationalisierung bieten die Java Standardbibliothek und das Spring Framework verschiedene Klassen an, die die Arbeit damit erleichtern:

LOCALE Locale[12] Objekte stellen eine geografische, politische oder kulturelle Region dar. Sie werden entsprechend den ISO Abkürzungen für Sprachen[13] und Länder[14] repräsentiert. Operationen, die aufgrund des übergebenen Locale Objekts regions- und sprachspezifische Ergebnisse zurückliefern, werden auch locale-sensitiv genannt. Locale Objekte werden meist nicht instanziiert, sondern von der JVM bereitgestellt, die eine Liste der installierten Sprachen und den entsprechenden Locale Objekten mitführt.

FORMAT Format[15] ist eine abstrakte Basisklasse um locale-sensitive Informationen zu formatieren. Für die gängigen Anwendungsfälle stellt die Java Standardbibliothek die folgenden Implementierungen bereit:

- *DateFormat* für die Formatierung von Datumsangaben
- *NumberFormat* für die Formatierung von Zahlen unter Berücksichtigung der für die Locale üblichen Trennzeichen
- *MessageFormat* um Nachrichten sprachunabhängig zusammenzusetzen und dann entsprechend einer *Locale* zu formatieren

12 http://java.sun.com/javase/6/docs/api/java/util/Locale.html
13 http://www.loc.gov/standards/iso639-2/englangn.html
14 http://www.iso.ch/iso/en/prods-services/iso3166ma/02iso-3166-code-lists/list-en1.html
15 http://java.sun.com/javase/6/docs/api/java/text/Format.html

RESOURCEBUNDLE Ein ResourceBundle[16] ist eine abstrakte Basisklasse, die lokalisierte Objekte bereitstellt. Auf die Objekte kann über die Kombination aus einem eindeutigen String (Schlüssel) und dem Locale Objekt für die gewünschte Sprache zugegriffen werden.

MESSAGESOURCE Eine MessageSource[17] ist ein vom Spring Framework zur Verfügung gestelltes Interface, das den Zugriff auf internationalisierte Nachrichten vereinfacht. Die bereitgestellten Implementierungen verwenden das oben beschriebene *ResourceBundle* um die Nachrichten aufzulösen und das entsprechende *MessageFormat* um diese zu parameterisieren.

In der Standardimplementierung benutzt ein *ResourceBundle* die von Java unterstützten *Property* Dateien. Für jeweils eine unterstützte Locale wird eine Datei angelegt, die als Konvention das Kürzel der entsprechenden Locale am Ende des Dateinamens enthält. In dieser Datei erfolgt dann die Zuordnung von einem Schlüssel zu der passenden Übersetzung mit Platzhaltern für Inhalte, die dynamisch eingefügt werden. Listing 12 zeigt ein entsprechendes Beispiel.

```
# messages_en.properties
button.cancel=Cancel
button.ok=Ok
button.update=Update
message.uploadError=Error uploading file {0}

# messages_de.properties
button.cancel=Abbrechen
button.ok=Ok
button.update=Aktualisieren
message.uploadError=Fehler beim Hochladen von {0}
```

Listing 12: Beispiel für die Internationalisierung mit Properties

4.2.5.2 *Implementierung*

Das Interface des I18n[18] Service fasst nun die Schnittstellen der oben vorgestellten Klassen zusammen und bindet sie in das Gesamtkonzept des Service Layer ein.

So ist es möglich, in Verbindung mit dem Security Service die bevorzugte Locale eines Benutzers zu bestimmen. Diese Information kann entweder bereits beim Anlegen eines Benutzers gespeichert werden oder dynamisch

16 http://java.sun.com/javase/6/docs/api/java/util/ResourceBundle.html

17 http://static.springsource.org/spring/docs/3.0.x/javadoc-api/org/springframework/context/MessageSource.html

18 I18n ist die allgemein verwendete Abkürzung für *Internationalization*, weil zwischen dem ersten und letzten Buchstaben 18 weitere Buchstaben stehen

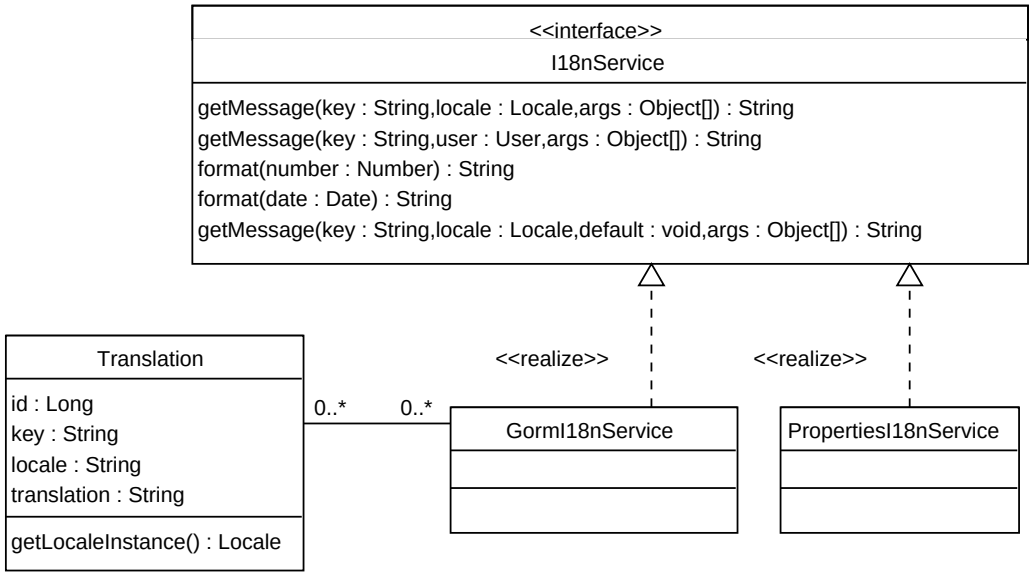

Abbildung 14: Interface und Domain Klassen des I18n Service

in der über dem Service Layer liegenden Schicht bestimmt werden, beispielsweise durch die Liste der bevorzugten Sprachen im Header einer HTTP Anfrage.

Wenn sich die Inhalte der übersetzten Texte häufig ändern kann es sehr unflexibel sein, diese in Java Property Dateien zu speichern, da man direkten Zugriff auf das Dateisystem benötigt. Deshalb unterstützt der I18N Service auch eine Domain Klasse zur Speicherung von Übersetzungstexten, die über den Domain Model Service in der Datenbank gespeichert wird. Damit ist es deutlich einfacher möglich, diese Texte zu aktualisieren, beispielsweise über das Administrationsmenü der Anwendung. Abbildung 14 zeigt das Service Interface und die Domain Klasse für Übersetzungstexte. Da auch die Übersetzungen über einen eindeutigen Schlüssel zugeordnet werden und sich relativ selten ändern, eignet sich der Cache Service sehr gut um diese für einen schnellen Zugriff zu speichern.

5 REMOTING LAYER

Eine besondere Form des Presentation Layer ist der Remoting Layer, der die Aufgabe übernimmt eine Anwendung über verscheidene RPC Protokolle anzubinden. In diesem Kapitel soll zuerst ein zentralisierter Ausführungsmechanismus vorgestellt werden, der eine verallgemeinerte Form eines RPC Aufrufs annehmen kann. Anschließend werden konkrete RPC Protokolle und deren Implementierungen besprochen, die einen Aufruf in diese Form umwandeln können.

5.1 SERVICE ENGINE

Im Service Layer werden eine Reihe von Java Interfaces bereitgestellt, welche die nach außen hin sichtbare Schnittstelle einer Anwendung definieren. Da die Service Methoden so kompakt wie möglich gehalten werden sollten und nur die Methoden bereitgestellt werden, die auch tatsächlich von den Benutzern verwendet werden, eignen sich diese Interfaces auch als Schnittstelle für entfernte Methodenaufrufe (vgl. [Fow02] S. 135).

Zu diesem Zweck existieren in Java eine Reihe von Bibliotheken, die verschiedene RPC Protokolle implementieren. Jede Bibliothek verfolgt dabei ein eigenes Konzept, wie ein Service in dem entsprechendem RPC Protokoll zur Verfügung gestellt wird. Möchte man eine Schnittstelle über verschiedene Protokolle zur Verfügung stellen ist es notwendig, jede Bibliothek individuell zu konfigurieren, obwohl die dafür notwendigen Informationen jeweils identisch sind. Gleichzeitig fehlt ein zentraler Ausführungsmechanismus, da jede RPC Bibliothek die Ausführung der aufgerufenen Methoden selbst übernimmt.

Diesen Nachteilen soll die Zwischenschicht der Service Engine entgegenwirken, indem sie die Verwaltung aller vorhandenen Services und die zentrale Ausführung von Methoden übernimmt, ohne selbst ein RPC Protokoll zu implementieren. Implementierungen eines Protokolls (anschließend *Protocol Handler* genannt) sind dann die der Service Engine übergeordneten Schichten. Sie müssen lediglich die Konvertierung eines konkreten RPC in eine allgemeine Form übernehmen, die dann an die Service Engine zur Ausführung übergeben werden kann.

Einzuordnen ist die Service Engine als Zwischenschicht direkt über dem Service Layer, befindet sich aber bereits in der vormals eingeführten Remoting Schicht (siehe Abbildung 15).

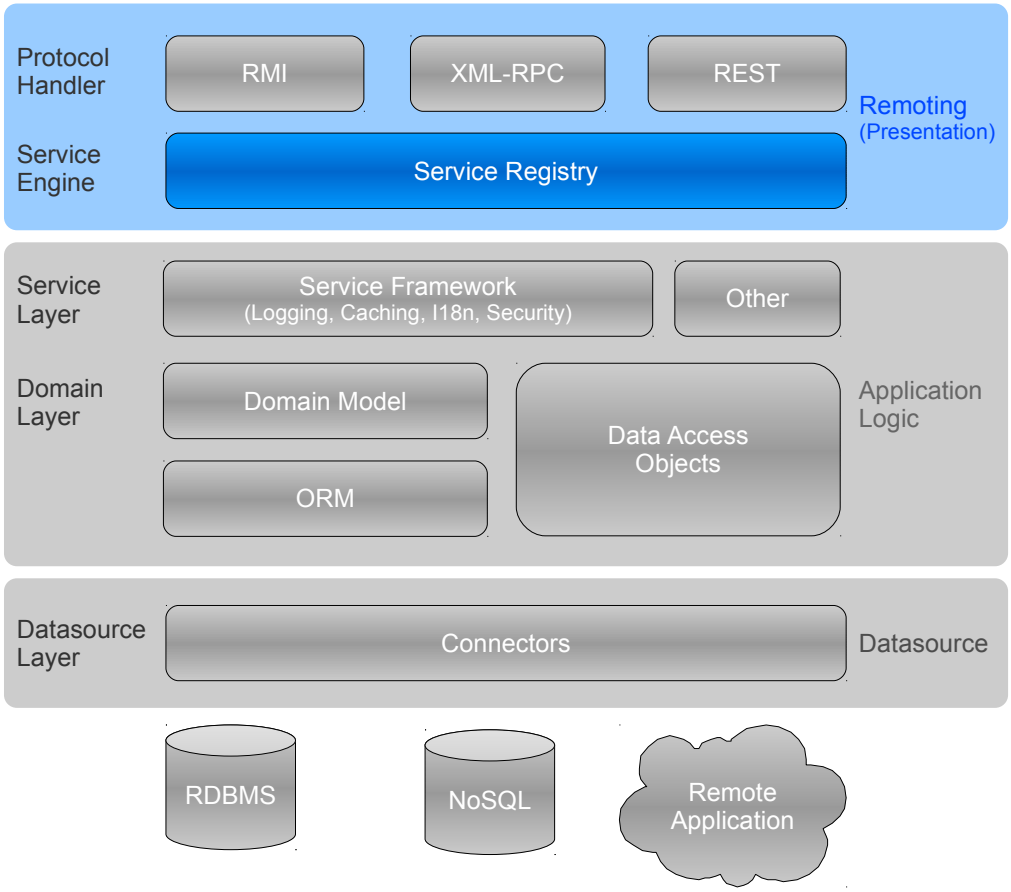

Abbildung 15: Einordnung der Service Engine

5.1.1 *Remote Schnittstelle*

Das Konzept des Service Layer sieht vor, dass sich die Service Methoden direkt nach den Anforderungen der Benutzer richten. Als direkt übergeordnete Schicht ist auch die Service Engine ein Benutzer des Service Layer und kann deshalb ebenfalls auf die Erstellung der Service Layer Schnittstelle Einfluss nehmen. Dabei richten sich die Anforderungen der Service Engine wiederum nach den Protocol Handlern und den RPC Protokollen, die diese implementieren. Obwohl der Service Layer keine direkten Abhängigkeiten von seinen übergeordneten Schichten haben sollte, wirken sich die unterstützten RPC Protokolle somit trotzdem indirekt auf die Implementierung des Service Layer aus.

Nicht alle RPC Protokolle stellen dabei den gleichen Funktionsumfang zur Verfügung. Das erste wichtige Unterscheidungsmerkmal ist, ob ein RPC mit Objekten und Methodenaufrufen arbeitet oder einfache Prozeduraufrufe übertragen werden. Im letzteren Fall muss der Service Layer entsprechend so gestaltet werden, dass die einzelnen Service Methoden wie getrennte Prozeduren ausgeführt werden können, die nicht zu einem

Objekt gehören. Ein weiterer zu beachtender Punkt ist, ob ein RPC Protokoll zustandsgebunden oder zustandslos arbeitet. Zustandsgebunden heißt auch, dass ein Client über verschiedene Anfragen das gleiche Service Objekt zur Verfügung hat, während bei einem zustandslosen Protokoll jede Anfrage unabhängig von allen anderen Anfragen und ohne einen globalen Zustand bearbeitet wird[1].

Bei dem Entwurf und der Implementierung der Service Interfaces, die über die Service Engine angebunden werden sollen, muss also der „kleinste gemeinsame Nenner" zwischen den anzubietenden RPC Protokollen gefunden werden. In Abschnitt 5.2 werden diese Einschränkungen an den dort vorgestellten RPC Protokollen näher analysiert.

5.1.2 *Service Registry*

Der zentrale Bestandteil der Service Engine ist das Interface der *Service Regitry*. Die Service Registry übernimmt die Verwaltung der Service Interfaces, die Instanziierung entsprechender Objekte, sowie die zentrale Ausführung von Methoden auf diesen Objekten. Jedem Service Interface ist hierbei ein eindeutiger Name zugewiesen. Wie die Services verwaltet und instanziiert werden, bleibt dabei der Klasse überlassen, die das Service Registry Interface implementiert.

Da andere Teile der Implementierung bereits auf dem Spring Framework basieren, hat es sich angeboten eine Service Registry zu implementieren, die auf dem *Application Context* des Dependency Injection Containers basiert. Die Instanziierung und Benennung von Service Objekten wird hier bereits bei der Erstellung des Spring Context vorgenommen. Die Implementierung des Service Registry Interface in Form der *ApplicationContextRegistry* stellt nun alle Objekte aus einem Application Context als Service Objekte bereit, sofern sie nach einer festgelegten Namenskonvention benannt sind. In der Standardeinstellung werden alle Objekte, deren Name auf „Service" endet, über die Service Registry verfügbar gemacht. Da diese Endung in der Service Registry eine redundante Information darstellen würde, wird sie entfernt um daraus den eindeutigen Namen für den bereitgestellten Service zu generieren. Beispielsweise ist das Context Objekt „userService" dann in der Service Registry unter dem Namen „user" verfügbar. Die Zuordnung von Objekten aus dem Application Context zu einem Service Interface muss manuell vorgenommen werden, da ein Objekt auch mehrere Service Interfaces implementieren kann. Die für die Ausführung von Methoden wichtigen Bestandteile der Service Registry sollen in den folgenden Abschnitten näher erläutert werden.

1 Das bedeutet normalerweise, dass für jede Anfrage ein neues Service Objekt erzeugt wird

5.1.3 *Interne RPC Darstellung*

Wie bereits erwähnt ist es nicht die Aufgabe der Service Engine, ein konkretes RPC Protokoll zu implementieren. Das bedeutet, dass die übergeordnete Schicht der Service Engine einen Methodenaufruf in einer verallgemeinerten Form zur Verfügung stellen muss. Da diese Schicht in erster Linie aus verschiedenen RPC Protocol Handlern besteht, die ebenfalls in Java implementiert werden, bietet sich für die allgemeine Darstellungsform eines Methodenaufrufs deshalb die Implementierung einer entsprechenden Java Klasse an. Diese Klasse muss alle Informationen enthalten, die für die Ausführung benötigt werden:

- Den Namen des Service in der Service Registry
- Den Namen der aufzurufenden Methode
- Eine Liste der Methodenparameter
- Eine Liste der Parameter Typen
- Informationen über den serverseitigen Zustand des Client

In Java lässt sich der Typ eines Parameters zur Laufzeit anhand der Instanz des Parameter Objekts bestimmten, muss deshalb also nicht getrennt gespeichert werden. Der serverseitige Zustand eines Client wird im folgenden Abschnitt näher behandelt.

Die *RemoteProcedureCall* Klasse wurde als unveränderliches Klasse mit den oben genannten Datenelementen implementiert. Ein RemoteProcedureCall Objekt dient lediglich als Transferobjekt zwischen der Service Engine und der übergeordneten Schicht. Die Validierung und eventuell notwendige Konvertierung von Parametern erfolgt dann an zentraler Stelle in der Service Registry und wird in Abschnitt 5.1.5 genauer betrachtet.

5.1.4 *Sessions*

Bei einem RPC wird zwischen zustandslosen und zustandsbehafteten Protokollen unterschieden, was auch bei der Ausführung von Methoden in der Service Registry beachtet werden muss. Für die zustandslose Kommunikation kann für jede Anfrage auf der Service Registry ein neues Service Objekt erstellt werden, auf dem die übergebene Methode ausgeführt wird. Im Gegensatz dazu muss sich ein Client bei einer zustandsbehafteten Kommunikation zu Beginn bei dem Server authentifizieren. Nach einer erfolgreichen Authentifizierung kann der Server alle nachfolgende Anfragen des Client diesem wieder eindeutig zuordnen. Das bedeutet auch, dass der Client üblicherweise erwartet, für alle folgenden Anfragen die jeweils gleiche Instanz eines Service Objekts zur Verfügung zu haben.

Der serverseitige Zustand von Objekten, die über die zustandsgebundene Kommunikation einem einzigen Client zugeordnet sind, soll nachfolgend *Session* genannt werden.

Wie für die allgemeine Form eines RPC die RemoteProcedureCall Klasse entwickelt wurde, ist es auch für Sessions notwendig einen Ansatz zu implementieren, der unabhängig von einem bestimmten RPC Protokoll arbeitet. Da die Umsetzung von Sessions in verschiedenen Protokollen unterschiedlich gehandhabt wird, bleibt die Erstellung und Zuordnung einer Session dem jeweiligen Protocol Handler überlassen, während die Verwaltung an zentraler Stelle in der Service Registry erfolgt. Ein Session Objekt repräsentiert jeweils den serverseitigen Zustand eines bestimmten Clients. Zur Zuordnung und Identifikation besitzt jedes Session Objekt einen eindeutigen Bezeichner (*Session ID*). Aufgabe des Protocol Handler ist es nun, Anfragen eines Client mit der entsprechenden Session ID zu verbinden, so dass die Service Registry den korrekten serverseitigen Zustand für die Ausführung der Anfrage auswählen kann.

Auch für die Implementierung von Sessions bietet das Spring Framework eine gute Grundlage. Bei der Definition eines Objekts im Context kann angegeben werden, über welchen Zeitraum (*Scope*) eine Objektinstanz im Application Context zur Verfügung stehen soll. Standardmäßig unterscheidet Spring hierbei zwischen *Singleton* und *Prototype*. Ein Singleton Objekt wird bei der Initialisierung des Application Context erstellt und ist über die gesamte Laufzeit der Anwendung verfügbar. Wird eine Referenz auf das Objekt angefordert handelt es sich also immer um die gleiche Instanz. Im Gegensatz dazu wird ein Prototype Objekt für jede angeforderte Referenz neu erstellt. Durch die hohe Flexibilität des Spring Frameworks ist es auch möglich eigene Scopes für Objekte zu definieren. Diese Möglichkeit wurde bei der Implementierung verwendet um einen Scope zu erstellen, der an jeweils ein Session Objekt der Service Registry gebunden ist. Möchte man also ein Objekt aus dem Spring Context für jeweils einen Benutzer verfügbar machen, muss es im Context mit diesem Scope definiert werden.

5.1.5 *Methodenausführung*

Nachdem nun ein allgemeines Transferobjekt für einen RPC in Form der RemoteProcedureCall Klasse implementiert wurde und auch zustandsgebundene Protokolle mit Hilfe der Session Umgebung in der Service Engine einen serverseitigen Zustand erhalten, ist die Service Registry nun in der Lage, die Informationen aus einem RemoteProcedureCall Objekt in einen Methodenaufruf auf einem Service Objekt umzusetzen.

Den Benutzern der Service Engine werden bei der Erstellung eines passenden RemoteProcedureCall Objekts bewusst verschiedene Möglichkeiten gegeben, in welcher Form die Methodenparameter übergeben werden kön-

nen. Dadurch ist die Verwendung der Service Engine weniger komplex und die Konvertierung und Validierung von Parametern erfolgt an zentraler Stelle in der Service Registry. Eine wichtige Rolle spielt dabei die in Kapitel 2.4.4 behandelte Äquivalenz zwischen JavaBeans und Assoziativen Arrays. Viele RPC Protokolle benutzen für die Übertragung von Objekten ein Format, dass sich relativ leicht in ein assoziatives Array (Map) deserialisieren lässt[2]. Damit nicht jeder Protocol Handler die Umwandlung in ein passendes JavaBean selbst übernehmen muss, wird die Konvertierung von einem assoziativen Array in ein JavaBean von der Service Registry vorgenommen. Hierfür wurde ein Algorithmus implementiert, der bei einem gegebenen assoziativen Array aus einer Reihe von JavaBean Klassen die passendste Implementierung auswählt, diese instanziiert und die Properties des JavaBean mit den entsprechenden Werten der Map füllt. Die ausgewählte JavaBean Klasse ist diejenige, deren Eigenschaften am genauesten mit den Schlüsseln der übergebenen Map übereinstimmen. Für die Suche der auszuführenden Methode ergibts sich dann das folgende Vorgehen:

1. Im ersten Schritt muss das Service Objekt bestimmt werden. Über den Namen des Service wird es entweder in der Service Registry neu erstellt oder, wenn eine Session Umgebung vorliegt und das Objekt mit dem gegebenen Namen dort vorhanden ist, aus einer Session geholt.

2. Daraufhin wird überprüft, ob das Service Interface eine Methode besitzt, die bereits mit dem Namen und der exakten Signatur des übergebenen RemoteProcedureCall Objekts übereinstimmt. Aufgrund einer Beschränkung in der Java Reflection muss Polymorphie bei den Parametern manuell geprüft werden[3]. Wurde eine Methode gefunden, wird sie auf dem Service Objekt ausgeführt.

3. Wurde keine passende Methode gefunden, werden alle Methoden mit dem Namen und der Anzahl von Parametern des RemoteProcedureCall Objekts gesucht. Nun wird für jeden Parameter überprüft, ob es sich um ein assoziatives Array handelt. Ist dies der Fall, wird unter allen Parameter Typen der Methoden an dieser Stelle das passendste JavaBean instanziiert. Mit diesen neu gewonnenen Parametern wird nun erneut Schritt zwei ausgeführt.

4. Wurde keine Methode gefunden, wird eine NoSuchMethodError Exception geworfen. Als Erweiterung der Ausführungsvorgangs hat ein Service Objekt noch die Möglichkeit grundsätzlich alle Methodenaufrufe behandeln zu können, indem es das *Interceptable* Interface

2 Beispielsweise JSON
3 http://stackoverflow.com/questions/2169497/unexpected-class-getmethod-behaviour

implementiert. Wurde in den Schritten zwei und drei keine passende Methode gefunden, wird der Methodenaufruf an dieses Interface übergeben.

5.1.6 *Übersicht*

Das UML Diagramm in Abbildung 16 zeigt die wichtigsten an der Service Engine beteiligten Interfaces. Die Schnittstelle der Service Engine zu übergeordneten Schichten besteht aus dem ServiceRegistry und dem Session Interface sowie der RemoteProcedureCall Klasse. Das ProtocolHandler Interface wird von Klassen implementiert, die eine Instanz einer Service Registry benötigen.

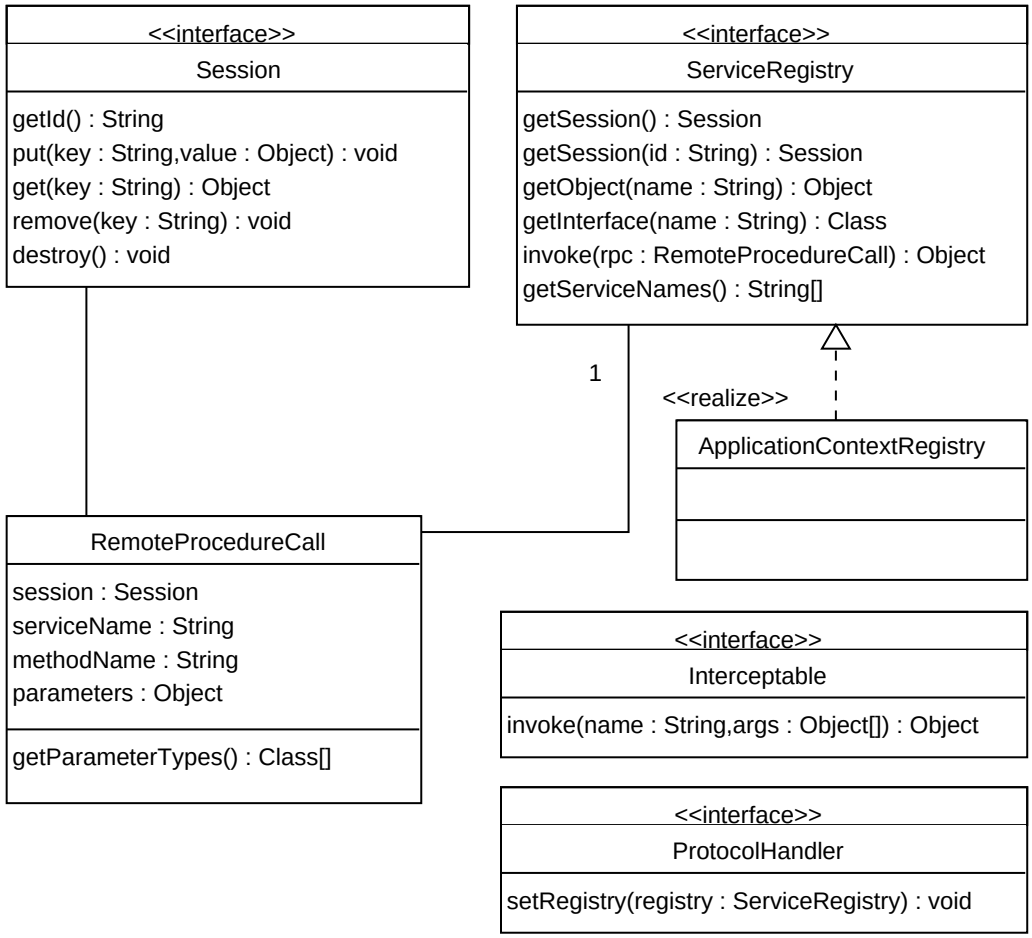

Abbildung 16: UML Diagramm der wichtigsten Komponenten der Service Engine

5.2 PROTOCOL HANDLER

Die Service Engine stellt über die Schnittstelle der Service Registry eine Möglichkeit zur Verfügung, einen RPC in der allgemeinen Form des *RemoteProcedureCall* Objekts ausführen zu können. Es ist nun die Aufgabe der *Protocol Handler*, die tatsächliche Implementierung der verschiedenen RPC Protokolle bereitzustellen und der Service Registry in Form dieses Objekts zur Ausführung zu übergeben. Die Aufgaben eines Protocol Handler umfassen somit:

- Beschreibung der Java Interfaces in der IDL des implementierten RPC Protokolls, soweit notwendig. Für die meisten RPC Protokolle beinhaltet das Java Service Interface bereits alle Informationen, die für die Schnittstellenbeschreibung notwendig sind.
- Bereitstellung eines Namensdienstes zur Auffindung der Services einer Service Registry, soweit nowendig.
- Annahme von Anfragen in einem spezifischen RPC Protokoll.
- Initialisierung einer zu dem RPC Protokoll gehörigen Session Umgebung.
- Deserialisierung von Methodenaufrufen und Umwandlung in ein RemoteProcedureCall Objekt.
- Übergabe des RemoteProcedureCall Objekts an die Service Registry zur Ausführung des Methodenaufrufs.
- Serialisierung des Ergebnisses in das Austauschformat des RPC Protokolls und Übertragung dieser Daten.
- Serialisierung von Fehlern, die während der Methodenausführung aufgetreten sind.

In der Gesamtarchitektur sind die Protocol Handler im Remoting Layer über der Service Engine einzuordnen (siehe Abbildung 17). Es ist die Schicht des Gesamtsystems, mit der andere entfernte System direkt kommunizieren.

Es ist weiterhin möglich, die Implementierungen vorhandener RPC Protokolle zu verwenden, so lange die Möglichkeit besteht, Methodenaufrufe in die abstrahierte Form des RemoteProcedureCall Objekts zu konvertieren und an die Service Registry zu übergeben. In diesem Kapitel sollen drei unterschiedliche RPC Protokolle und deren Implementierungen eines Protocol Handler vorgestellt werden. Die ausgewählten Protokolle verfolgen jeweils unterschiedliche Ansätze bezüglich serverseitigem Zustand und der Art von Methoden- bzw. Prozeduraufrufen. Deshalb sind sie

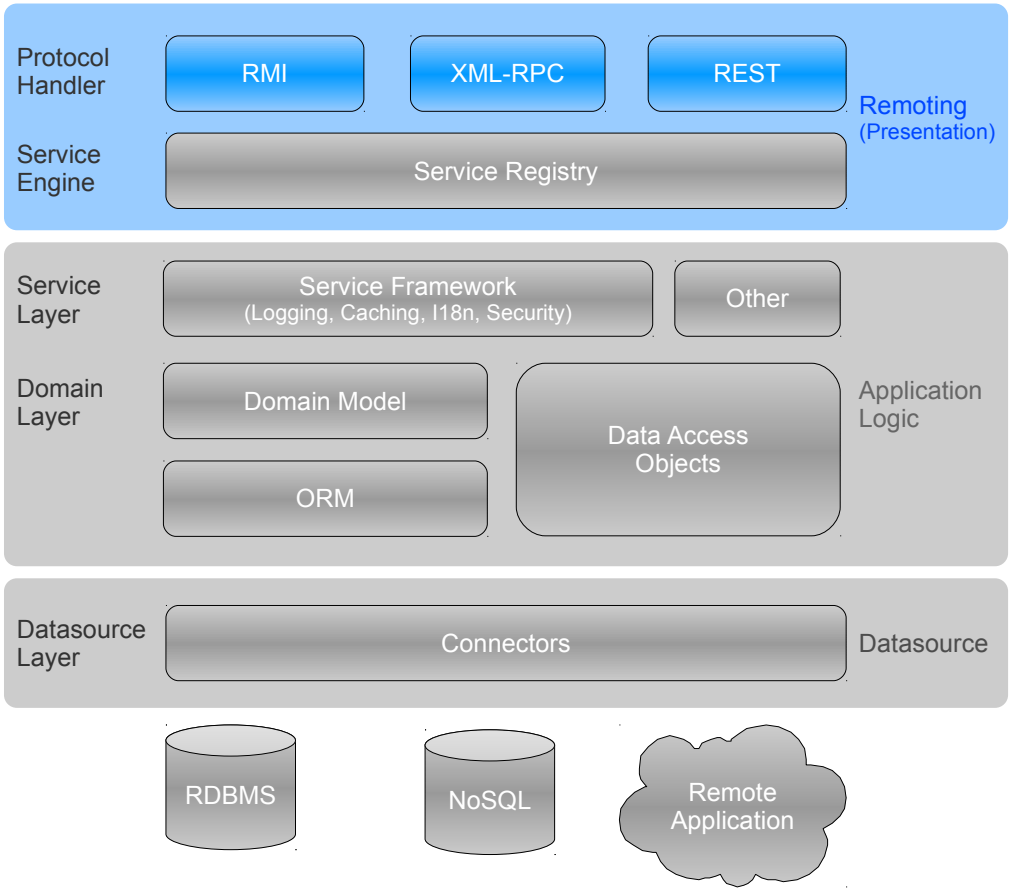

Abbildung 17: Einordnung der Protocol Handler

gut geeignet, die Vorteile, die durch die Verwendung der Service Engine entstehen, aufzuzeigen.

5.2.1 *Remote Method Invocation*

Java Remote Method Invocation (RMI) (siehe [sun]) ist die Implementierung eines binären RPC Protokolls für den Austausch von Objekten zwischen verschiedenen, auch entfernten, Instanzen einer JVM. Es setzt dabei stark auf die von der JVM unterstützten Serialisierung von Objekten. RMI ist zustandsgebunden, wodurch auch auf entfernten Objekten Methodenaufrufe ausgeführt werden können. An der für RMI notwendigen Infrastruktur sind folgende Komponenten beteiligt (siehe [Wik10h]):

REMOTE INTERFACE Das Remote Interface ist ein Java Interface, das dem Client bekannt ist und die auf dem Server implementierten Methoden beschreibt. Dadurch wird das Problem, die vom Server bereitgestellten Methoden zu beschreiben, auf eine für Java natürliche Art und Weise gelöst.

REMOTE OBJECT Das Remote Object ist eine Instanz der Klasse, die das Remote Interface auf der Serverseite implementiert.

RMI REGISTRY Die RMI Registry ist ein Namensdienst, der für das Auffinden der vom Server bereitgestellten Remote Objekte zuständig ist. Jeweils einem implementierten Remote Interface wird dabei ein eindeutiger Name zugewiesen.

REMOTE REFERENCE Erkundigt sich der Client bei der RMI Registry nach dem Remote Object für einen bestimmten Namen, bekommt er eine Referenz auf das serverseitige Remote Object zurück.

Ist die Verbindung von einem Client zu einem Remote Object auf dem Server hergestellt, erlaubt RMI eine transparente Kommunikation. Auch Klassendefinitionen, die auf der jeweils anderen Teilnehmerseite nicht vorhanden sind, werden dynamisch nachgeladen. Tritt einer der bei einem RPC zusätzlichen Fehlerfälle auf, wird eine *RemoteException* geworfen. Abbildung 18 zeigt den Ablauf der RMI Kommunikation und den beteiligten Komponenten.

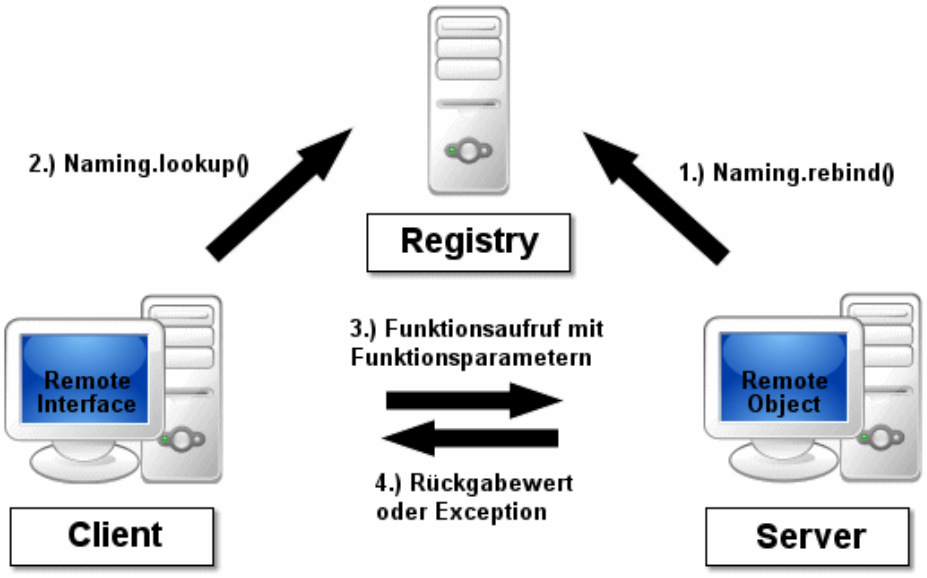

Abbildung 18: Kommunkationsablauf bei RMI[4]

5.2.1.1 *Spring und RMI*

Das Spring Framework vereinfacht die Nutzung von RMI erheblich (vgl. [Ju09] Abschnitt 19.2). So können in einem Kontext definierte Objekte

4 Quelle: [Wik10h], Public Domain

ebenfalls über eine Context Konfiguration als RMI Services exportiert werden, wenn sie das zu exportierende Remote Interface implementieren.

Da das Spring Framework durch Dependency Injection bereits die Verwendung von Interfaces unterstützt, muss im Spring Context der Clientseite dann lediglich ein *Proxy* definiert werden, der das Remote Interface für den Client implementiert und mit dem im Context angegeben RMI Server kommuniziert. Für Klassen des Client, die das Remote Interface verwenden, wird dann dieser Proxy injiziert.

Listings 13 und 14 zeigen eine Spring Context Konfiguration für die Verwendung des in Abschnitt 4.2.5 vorgestellten I18NService über RMI.

```
<beans>
        <bean name="i18nService" class="org.feathry.service.i18n.
            GormI18nService">
            <!-- ... -->
        </bean>

        <bean class="org.springframework.remoting.rmi.
            RmiServiceExporter">
            <property name="serviceName" value="I18nService"/>
            <property name="service" ref="i18nService"/>
            <property name="serviceInterface" value="org.feathry.
                service.i18n.I18nService"/>
        </bean>
</beans>
```

Listing 13: Spring und RMI - Server Context

```
<beans>
        <bean name="anotherBean" class="example.AnotherBean">
            <property name="i18nService" ref="i18nService" />
        </bean>

        <bean id="i18nService" class="org.springframework.remoting.
            rmi.RmiProxyFactoryBean">
            <property name="serviceUrl" value="rmi://hostname:1099/
                I18nService" />
            <property name="serviceInterface" value="org.feathry.
                service.i18n.I18nService" />
        </bean>
</beans>
```

Listing 14: Spring und RMI - Client Context

5.2.1.2 *Implementierung*

Da Spring bereits eine sehr gute Unterstützung für RMI besitzt ist es sinnvoll die dafür verwendeten Klassen für die Implementierung eines

RMI Protocol Handler zu erweitern. Die *RmiProtocolHandler* Klasse erstellt so für jeden in der Service Registry vorhandenen Service eine Instanz des Spring *RmiServiceExporter* mit den passenden Parametern für Service Name, Service Instanz und Service Interface. Dadurch sind bereits alle Service Interfaces über RMI auffindbar.

Schwieriger gestaltet sich nun die Bereitstellung des passenden Remote Object. Der Client ruft über das ihm bekannte Remote Interface eine Methode auf einem serverseitigen Remote Object auf. Auf dem Remote Object selbst verhält sich dieser Aufruf genauso wie ein lokaler Methodenaufruf. Die Aufgabe des Protocol Handler ist es aber, diesen Aufruf in ein entsprechendes *RemoteProcedureCall* Objekt umzuwandeln. Um diese Umwandlung zu realisieren, wird ein Service Interface von einem AOP Proxy[5] implementiert, der dann als Remote Object dient. Durch einen *Around Advice* für alle Methodenaufrufe auf diesem Proxy ist es nun möglich, die Informationen des Methodenaufrufs zu extrahieren und in ein *RemoteProcedureCall* Objekt umzuwandeln, das dann an die Service Registry übergeben werden kann. Auch die Identifikation der Session ist bei diesem Vorgehen problemlos möglich, da ein Remote Object, also hier das Proxy Objekt, immer direkt einem Client zugeordnet ist.

5.2.2 *XML-RPC*

XML-RPC ist ein zustandsloses RPC Protokoll, das XML als Austauschformat und HTTP als Übertragungsprotokoll verwendet (siehe [Win99]). Übertragen werden Prozeduraufrufe, wobei die Struktur bewusst sehr einfach gehalten wird, weshalb das Protokoll sowohl auf Server- als auch auf Clientseite leicht zu implementieren ist.

5.2.2.1 *Spezifikation*

Als Parameter- und Rückgabewerte werden sieben primitive und zwei zusammengesetzte Typen unterstützt. Ein Typ wird jeweils von einem <*value*> XML Tag umschlossen. Die primitiven XML-RPC Typen sind in Tabelle 5 aufgelistet. Der zusammengesetzte Typ *Struct* ist das XML-RPC Äquivalent eines assoziativen Arrays, während der Typ *Array* eine geordnete Menge von beliebigen anderen XML-RPC Typen enthält. Im Fehlerfall wird ein *Fault* Objekt übertragen, das ein Struct Element mit den Schlüsseln *faultCode* und *faultString* enthält.

Da XML-RPC HTTP als Übertragungsprotokoll verwendet, erfolgt die Lokalisierung über einen eindeutigen HTTP Uniform Resource Identifier (URI), wohin alle Prozeduraufrufe gerichtet werden. Die XML-RPC Spezifikati-

5 Siehe Abschnitt 2.2.4.2

6 In den weiteren Beispielen wurden die umschließenden XML Tags aus Platzgründen weg gelassen

TYP	XML TAG	BEISPIEL
Integer	<i4> oder <int>	<int>33</int>[6]
Double	<double>	3.1415
Date/Time	<dateTime.iso8601>	20100223T19:34:56
Boolean	<boolean>	1, 0, true, false
String	<string>	Text
Base64 Binärdaten	<base64>	VGV4dA==

Tabelle 5: Primitive Typen des XML-RPC Protokolls

on sieht nicht vor, die zur Verfügung gestellten Prozeduren automatisch zu beschreiben. Die Beschreibung erfolgt meist in Form einer API Dokumentation. Eine inoffizielle aber häufig verwendete Erweiterung zur Schnittstellen Beschreibung ist die *XML-RPC Introspection* (siehe [Heno7]).

Ein vollständiges Beispiel einer XML-RPC Kommunikation mit dem Aufbau von Struct, Array und Fault Elementen sowie den dazugehörigen HTTP Headern ist in Anhang A zu finden.

5.2.2.2 *Implementierung*

Für Java existieren eine Reihe von Bibliotheken, die XML-RPC Anfragen verarbeiten können. Deshalb bot es sich, an eine passende Implementierung für die Umsetzung des XML-RPC Protocol Handler zu verwenden. Voraussetzung ist, dass die Bibliothek die Möglichkeit bietet, eine eigene Implementierung für die Bearbeitung eines deserialisierten XML-RPC zur Verfügung zu stellen. Nach der Evaluierung verschiedener Bibliotheken fiel die Entscheidung dabei auf die *Redstone XML-RPC Library*[7], da sie performant arbeitet, klar strukturiert ist und eigene Erweiterungen zulässt. Für die Realisierung musste dann lediglich ein Interface implementiert werden, das bei einem XML-RPC Aufruf mit den entsprechenden Methodennamen und Parametern aufgerufen wird. Dort wird dann das entsprechende RemoteProcedureCall Objekt erstellt und an die Service Registry weitergegeben.

5.2.3 *Representational State Transfer*

Der Begriff REST wurde von Roy T. Fielding im Jahr 2000 in [Fieoo] (S. 76 - 106) geprägt und beschreibt einen Software Architekturstil für verteilte Hypermedia Systeme. Hypermedia (vgl. [Fieoo] S. 68) bedeutet die Verwendung von *Multimedia-Formaten* und *Hypertext*[8] zur Übertragung von

7 http://xmlrpc.sourceforge.net/
8 Beispielsweise in Form von HTML oder XML

Informationen, wobei in einem Hypertext-Dokument wiederum durch *Hyperlinks* auf andere Hypermedia Inhalte verwiesen werden kann. Verteilte Hypermedia Systeme erlauben es, diese Inhalte auch von einem entfernten System zu beziehen.

Bei REST handelt es sich um einen Architekturstil, also eine Richtlinie, wie Anwendungen, die die für REST festgelegten Kriterien erfüllen, strukturiert werden können. Somit ist REST selbst implementierungs- und technologieunabhängig. Das bekannteste Beispiel einer Umsetzung des REST Architekturstils ist das HTTP Protokoll, das im nachfolgenden Abschnitt genauer behandelt wird. Anwendungen, die der REST Architektur folgen, werden auch als *RESTful* bezeichnet. Damit eine Implementierung dem REST Architekturstil, folgt stellt Fielding verschiedene Bedingungen auf (siehe [Fieoo] S. 76 - 85):

CLIENT-SERVER Durch die Verwendung einer Client-Server Architektur ist eine einfache Trennung der Aufgabenbereiche (*Separation Of Concerns*) für beide Teilnehmer möglich. Dabei ist der Client für die Darstellung zuständig, während der Server die Datenhaltung und die Anwendungslogik übernimmt. Verschiedene Clients können dabei die vom Server bereitgestellten Daten in unterschiedlichen Benutzeroberflächen bereitstellen und verarbeiten, während die serverseitige Anwendungslogik gleichzeitig vereinfacht werden kann. Ein weiterer Vorteil ist, dass beide Seiten unabhängig voneinander weiterentwickelt werden können, so lange sie das entsprechende Protokoll und Austauschformat einhalten.

ZUSTANDSLOSIGKEIT Die Kommunikation zwischen Client und Server erfolgt zustandslos. Das heißt, dass ein Client mit jeder Anfrage alle für die Bearbeitung der Anfrage notwendigen Informationen übertragen muss. Dadurch verringert sich die Komplexität der Serverseite weiter, da jede Anfrage unabhängig von einem globalen Zustand bearbeitet werden kann. Gleichzeitig wird die Skalierbarkeit erhöht, da jede Anfrage von einer beliebigen Serverinstanz bearbeitet werden kann.

CACHE Im Hinblick auf die Netzwerkauslastung kann sich eine zustandslose Kommunikation negativ auswirken, da viele Informationen mehrfach übertragen werden müssen. Deshalb sieht REST die Verwendung von Caches vor. Ist die Antwort auf eine Anfrage des Client als cachebar gekennzeichnet, kann er diese in einem *clientseitigem Cache* über einen bestimmten Zeitraum als Antwort für erneute, identische Anfragen verwenden. Auch die Verwendung von transparenten Caches (*Proxies*) ist möglich.

WOHLDEFINIERTE OPERATIONEN REST sieht eine Reihe wohldefinierter Operationen vor, die auf Hypermedia Inhalten ausgeführt werden

können. Die Identifikation und Beschreibung dieser Operationen wird im folgenden Abschnitt näher besprochen.

SCHICHTENARCHITEKTUR Die schon in Kapitel 1 vorgestellten Vorteile einer Schichtenarchitektur finden auch in REST Anwendung. In Verbindung mit der hier verwendeten einheitlichen Schnittstelle kann die nach außen hin sichtbare Komplexität eines Gesamtsystems erheblich reduziert werden.

CODE-ON-DEMAND Code-On-Demand ist eine optionale Eigenschaft, die es erlaubt, einen Client um Funktionalitäten zu erweitern, die vom Server in Form eines Skripts oder Applets zur Verfügung gestellt werden.

5.2.3.1 *Ressourcen*

Ein zentraler Bestandteil der REST Architektur sind *Ressourcen*. Ressourcen können jede Art von Hypermedia Informationen sein, die eindeutig addressierbar sind. Beispiele (vgl. [Fieoo] S. 88) sind ein Dokument, eine Bilddatei, Listen von anderen Ressourcen, dynamische Services (z.B. Wetterinformationen) oder Domain Objekte mit einem eindeutigen Bezeichner. Bei verteilten Hypermedia Systemen geschieht die Addressierung in Form eines URI. In [BLFM05], der Definition des URI, wird ein einheitliches Addressierungsschema für Ressourcen festgelegt. Beispiele sind (vgl. [BLFM05] Abschnitt 1.1.2):

- http://tools.ietf.org/html/rfc3986
- jdbc:mysql://host:port/database
- tel:+4989123456
- rmi://host:1099/serviceName

Vor allem in Verbindung mit dem HTTP Protokoll wird oft der Begriff des Uniform Resource Locator (URL) verwendet. Die URL Definition ist eine Untermenge der aktuelleren URI Spezifikation, weshalb im weiteren Verlauf der technisch korrekte Begriff der URI verwendet werden soll.

Nachdem nun Ressourcen eindeutig identifiziert werden können, müssen die darauf auszuführenden Operationen definiert werden. In den meisten Fällen entsprechen diese Operationen direkt einer CRUD Schnittstelle.

5.2.3.2 *HTTP*

REST wurde aus den Design- und Implementierungsentscheidungen hergeleitet, die bei der Entwicklung des HTTP Protokolls maßgeblich waren.

Bei HTTP handelt es sich somit um die am weitesten verbreitete und faktisch einzige (vgl. [Til09]) Implementierung des REST Architekturstils. Im Unterschied zu klassischen Webservices, die zwar HTTP als Übertragungsprotokoll verwenden, aber eine weitere Zwischenschicht für die Nachrichtenübermittlung verwenden, verfolgt die REST Umsetzung für HTTP den Ansatz, dass das HTTP Protokoll selbst bereits alle dafür notwendigen Eigenschaften besitzt.

HTTP METHODEN Als Operationen, die auf allen Ressourcen ausgeführt werden können (siehe [FGM+99] Abschnitt 9), legt HTTP *GET*, *POST*, *PUT* und *DELETE* sowie *OPTIONS* und *HEAD* fest:

- GET ist eine „sichere“, nur lesende Methode, die die in der URI festgelegte Ressource in einem vom Client gewählten Austauschformat zurückliefert. „Sicher“ bedeutet hier, dass eine GET Operation nur lesend auf die Daten zugreift und in keinem Fall eine Veränderung auf der Serverseite bewirkt.
- PUT überträgt die Repräsentation einer Ressource, die an der von der URI angegebenen Stelle angelegt werden soll. Die PUT Operation ist „idempotent“. Das bedeutet, dass sich mehrere identische PUT Operationen genauso auswirken wie eine einzige.
- DELETE löscht die von einer URI angegebene Ressource.
- POST aktualisiert eine vorhandene Ressource. Außerdem können durch POST Operationen auch andere Methoden abgebildet werden. Über die tatsächliche Semantik einer POST Operationen entscheidet dabei die Serverseite.
- OPTIONS gibt Informationen über die vom Server unterstützten Methoden. Auch hierbei handelt es sich um eine sichere Operation.
- HEAD ist identisch zu einer GET Operation, außer dass lediglich der HTTP Header der Antwort gesendet wird.

STATUSCODES Im Antwortheader jeder Anfrage gibt der Server verschiedene Statuscodes zurück, die den Client darüber informieren, ob die Anfrage erfolgreich bearbeitet wurde bzw. welche Fehler aufgetreten sind. Die dabei verwendeten Statuscodes sind in verschiedene Bereiche (siehe [FGM+99] Abschnitt 10) unterteilt, die in der folgenden Tabelle 6 kurz beschrieben werden.

CONTENT NEGOTIATION Eine weitere für REST wichtige Eigenschaft ist die Unterstützung von verschiedenen Austauschformaten (*Representations*) für eine Ressource. Zu diesem Zweck unterstützt HTTP die sogenannte

BEREICH	BEDEUTUNG
1XX	Informationen
2XX	Anfrage erfolgreich ausgeführt und bearbeitet
3XX	Weiterleitung, weitere Aktionen von Clientseite erwartet
4XX	Client Fehler, fehlerhafte Anfrage
5XX	Server Fehler, Anfrage kann nicht bearbeitet werden

Tabelle 6: HTTP Statuscode Bereiche

Content Negotiation. Dadurch kann ein Client bei einer Anfrage im *Accept* Teil des Headers eine Liste der gewünschten Austauschformate angeben. Der Server antwortet dann mit einer Repräsentation der Ressource in dem ersten Austauschformat aus der Client Liste, dass er zur Verfügung stellen kann oder mit einem *406 Not Acceptable* Fehlercode, wenn die Darstellung der Ressource in keinem erwarteten Format möglich ist.

BEISPIEL Das folgende Beispiel soll eine HTTP Kommunikation veranschaulichen, die die verschiedenen Methoden in Verbindung mit Content Negotation verwendet und dem REST Architekturstil folgt.

Ein Client sendet einen PUT Request für die Ressource http://example.com/users mit der XML Repräsentation einer Ressource an den Server:

```
PUT /users HTTP/1.1
Host: example.com
Content-Type: text/xml
Content-Length: 77
Accept: application/xml,application/json
Accept-Charset: ISO-8859-1,utf-8

<user>
        <username>UserX</username>
        <password>supersecret</password>
</user>
```

Listing 15: HTTP PUT Request

Daraufhin antwortet der Server mit einer passenden Statusmeldung, die im Feld *Content Location* die URI der neu angelegten Ressource angibt:

```
HTTP/1.1 201 Created
Date: Fri, 26 Feb 2010 11:39:07 GMT
Server: Service Engine HTTP Connector
Content-Location: http://example.com/users/2
```

Listing 16: HTTP PUT Antwort

Nun kann ein anderer Client diese Resource in einem beliebigen vom Server unterstützten Austauschformat anfordern:

```
GET /users/2 HTTP/1.1
Host: example.com
Accept: application/json
Accept-Charset: utf-8,ISO-8859-1
```

Listing 17: HTTP GET Request

Und bekommt die entsprechende Antwort:

```
HTTP/1.1 200 OK
Date: Fri, 26 Feb 2010 13:14:07 GMT
Server: Service Engine HTTP Connector
Content-Type: application/json; charset=utf-8
Content-Length: 55

{
        "username" : "UserX",
        "password" : "supersecret"
}
```

Listing 18: HTTP GET Antwort

5.2.3.3 *Implementierung*

REST ist kein RPC Protokoll (vgl. [Fieoo] S. 141), da das zentrale Element nicht Methodenaufrufe sondern Ressourcen sind, die über eine gemeinsame, eindeutige Schnittstelle angesprochen werden können. Viele Frameworks, die eine REST Unterstützung über HTTP anbieten, benötigen deshalb zusätzliche Konfigurationseinstellungen, um die Zuordnung von Service Methoden zu Ressourcen und HTTP Methoden vornehmen zu können. Eine andere häufig verwendete Möglichkeit ist ein Interface, das mit HTTP Request und Response Objekten arbeitet.

Die erste Möglichkeit erfordert zusätzlichen Konfigurationsaufwand, was gegen das bei der Implementierung verfolgte Prinzip der *Convention Over Configuration* verstoßen würde. Der zweite Ansatz würde durch die protokollspezifischen Request und Response Objekte im Service Layer eine Abhängigkeit zu einer übergeordneten Schicht herstellen. Deshalb wurde bei der Implementierung des REST Protocol Handler ein anderer Ansatz gewählt. Da das Java Interface eines Service bereits im Service Layer und bei der Implementierung der Service Engine eine zentrale Rolle spielte, soll es auch hier für die Zuordnung zu Ressourcen und HTTP Methoden dienen. Gleichzeitig soll die Unabhängigkeit des Service Layer zu seinen übergeordneten Schichten erhalten bleiben. Dazu wurde ein generisches Java Interface erstellt (siehe Listing 19), von dem alle Service Interfaces

ableiten müssen, die eine Ressource über REST und HTTP verfügbar machen wollen:

```
public interface Resource<T>
{
        // GET
        public T get(Serializable id);
        // PUT
        public void save(T arg) throws NotSupportedException;
        // DELETE
        public void delete(T arg) throws NotSupportedException;
        // GET
        public List<T> getAll() throws NotSupportedException;
}
```

Listing 19: Genersiches Resource Interface

Der generische Parameter T stellt dabei die Ressource dar. Da die Namen der Interface Methoden den GORM Konventionen folgen, ist eine direkte Verwendung des Domain Model Service möglich. Um eine Domain Klasse über REST verfügbar zu machen, muss dann lediglich ein von dem Resource Interface mit dieser Domain Klasse als Parameter abgeleitetes Service Interface erstellt werden. Listing 20 zeigt ein abgeleitetes Interface am Beispiel des User JavaBeans aus Kapitel 2.2.2.

```
public interface UserService extends Resource<User>
{
        public User findByUsername(String username);
}
```

Listing 20: Von Resource abgeleitetes UserService Interface

Durch die Implementierung des *Resource Interface* können die HTTP Methoden auf einer Ressource eindeutig einer entsprechenden Service Methode zugewiesen werden. Der für die URI verwendete Name ist der Name des Service, im Fall der Spring Application Context basierten Service Registry also der Name des Interface ohne die „Service“ Endung.

Um das Service Interface weiterhin unabhängig von jeder übergeordneten Schicht zu halten müssen Content Negotiation und Serialisierung beziehungsweise Deserialisierung von Parametern bereits direkt im REST Protocol Handler erfolgen. Für die Aufgabe der Serialisierung und Deserialisierung von Objekten sind Klassen zuständig, die das *Representation* Interface implementieren. Dadurch können neue Austauschformate problemlos in den REST Protocol Handler eingebunden werden. Im aktuellen Stand der Implementierung werden XML und JSON unterstützt. Aber auch beliebige weitere Formate wie beispielsweise *PDF*, *CSV* oder *Excel* können so problemlos implementiert werden. Abbildung 19 zeigt eine Übersicht der Komponenten der REST Protocol Handler.

Auch eine Representation Implementierung für *HTML* befindet sich in Entwicklung. Viele Webframeworks bieten zwar inzwischen eine Unterstützung für REST an, überlassen die Umsetzung des Architekturstils aber weiterhin zum Großteil dem Entwickler. Das hat dazu geführt, dass viele Anwendungen als RESTful bezeichnet werden, obwohl sie REST nicht der Definition entsprechend umsetzen. Der Vorteil gegenüber den Ansätzen klassischer Webframeworks bei der Implementierung des Representation Interfaces für HTML ist, dass HTML lediglich als andere Darstellungsform der Daten behandelt wird. Dadurch muss die Generierung von HTML Seiten bereits durch die geltenden Rahmenbedingungen dem REST Architekturstil folgen.

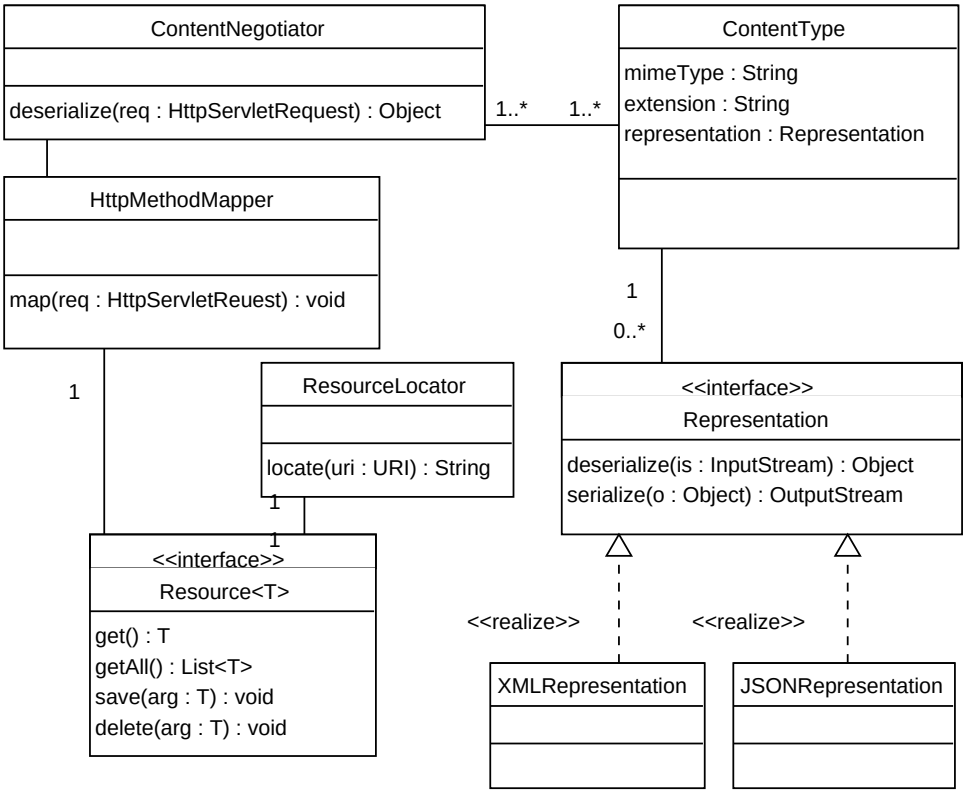

Abbildung 19: UML Übersicht des REST Protocol Handler

5.2.4 *Übersicht*

Die folgenden Tabellen 7, 8 und 9 bieten einen Überblick über die wichtigsten Eigenschaften und Unterschiede der im Verlauf dieses Kapitels vorgestellten RPC Protokolle.

5.2.4.1 *RMI*

EIGENSCHAFT	BESCHREIBUNG
SCHNITTSTELLENBESCHREIBUNG	Java Interface
LOKALISIERUNG	RMI URI
AUSTAUSCHFORMAT	Binär (Java Objektserialisierung)
KOMMUNIKATION	Zustandsgebunden
AUFRUFFORM	Objektorientiert
FEHLERFÄLLE	Java Exceptions (RemoteException)

Tabelle 7: Eigenschaften von RMI

5.2.4.2 *XML-RPC*

EIGENSCHAFT	BESCHREIBUNG
SCHNITTSTELLENBESCHREIBUNG	Dokumentation, Introspection
LOKALISIERUNG	Eindeutige HTTP URI
AUSTAUSCHFORMAT	XML
KOMMUNIKATION	Zustandslos
AUFRUFFORM	Prozedural
FEHLERFÄLLE	Fault Objekte

Tabelle 8: Eigenschaften von XML-RPC

5.2.4.3 *REST über HTTP*

EIGENSCHAFT	BESCHREIBUNG
SCHNITTSTELLENBESCHREIBUNG	Methoden aus der HTTP Spezifikation
LOKALISIERUNG	URI einer Ressource (z.B. in Hypertext Dokumenten)
AUSTAUSCHFORMAT	Entsprechend Content Negotiation
KOMMUNIKATION	Zustandslos
AUFRUFFORM	HTTP Methoden auf Ressourcen
FEHLERFÄLLE	HTTP Fehlercodes

Tabelle 9: Eigenschaften von REST

Teil III

GESAMTKONZEPT

6 NUZTUNGSMÖGLICHKEITEN

In diesem Kapitel soll ein kurzer Überblick über verschiedene Anwendungsmöglichkeiten der in dieser Arbeit vorgestellten Service Engine und der Service Implementierungen gegeben werden. Grundsätzlich sind auch andere Anwendungsszenarios denkbar, denn vor allem in der Integration mit anderen Anwendungen hat die Service Engine als protokollunabhängige Ausführungsschicht erhebliche Vorteile. So können bei Bedarf jederzeit neue Protocol Handler angebunden werden, ohne dass die Anwendungslogik angepasst werden muss.

6.1 SERVICE KLASSENBIBLIOTHEK

In Kapitel 4 wurde eine Art des Presentation Layer vorgestellt, der die Anwendungslogik des Service Layer über verschiedene RPC Protokolle verfügbar macht und deshalb Remoting Layer genannt wurde. Entsprechend dem Prinzip einer Schichtenarchitektur sind aber auch andere Formen des Presentation Layer möglich. Dabei können die in Kapitel 4 vorgestellten Service Implementierungen weiterhin verwendet werden. Eine Möglichkeit wäre beispielsweise ein Presentation Layer in Form einer Weboberfläche. Dabei sind die Service Implementierungen eine Grundlage für den Service Layer der Webapplikation.

Da die Service Implementierungen bereits stark von den Möglichkeiten des Spring Framework gebrauch machen, bietet sich das Model View Controller (MVC) Modul des Spring Frameworks an, das die Erstellung von Webapplikationen auf Basis des Spring Frameworks erlaubt. Auch die Verwendung des Groovy Webframeworks Grails wäre denkbar, da es ebenfalls auf dem Spring MVC Framework aufbaut und einige der Services in Groovy implementiert wurden. So lange die Groovy Laufzeitbibliotheken für die Service Implementierungen verfügbar sind, ist eine direkte Verwendung von Groovy aber nicht notwendig. Die nachfolgende Abbildung 20 zeigt eine Möglichkeit für die Architektur einer Webapplikatione, die die Service Implementierungen und das Spring MVC Modul verwendet.

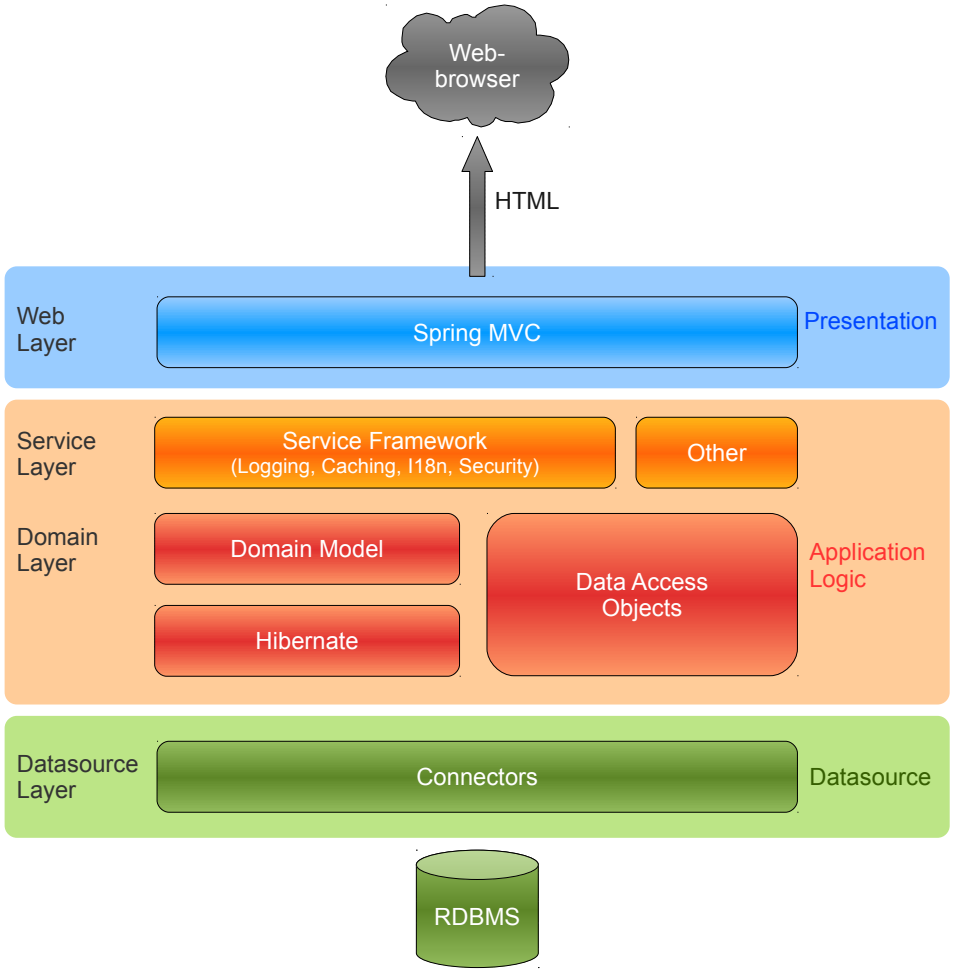

Abbildung 20: Verwendung der Service Implementierungen in einer Spring MVC Webanwendung

6.2 RICH INTERNET APPLICATIONS

Eine Rich Internet Application (RIA) ist eine Webanwendung, die ähnlich wie eine klassische Desktopapplikation zu bedienen ist (siehe [Wik10i]). Typischerweise wird für die Verwendung dieser Anwendungen ein Browser Plugin[1] benötigt, aber auch klassische Webapplikationen, die die Benutzeroberfläche zum Großteil durch JavaScript dynamisch gestalten fallen inzwischen in diese Kategorie. Eine RIA übernimmt die Aufgabe der Darstellung von Daten und kommuniziert dann mit einem Webserver, der für die Datenhaltung zuständig ist.

Da viele RIA Frontends in ECMAScript Derivaten wie JavaScript oder ActionScript[2] entwickelt werden, ist die Kombination aus REST mit HTTP als Übertragungsprotokoll und JSON als Austauschformat für die Kommunikation mit dem Server sehr beliebt. Aus diesem Grund bietet es sich an, die Service Engine in Verbindung mit dem REST Protocol Handler und den in Kapitel 4 vorgestellten Service Funktionalitäten zu verwenden. Durch diesen Aufbau lässt sich schnell eine serverseitige Grundlage für die Verwendung in RIA Anwendungen schaffen.

Ein weiterer Vorteil der Service Engine ist dabei, dass andere Anwendungen weiterhin über ihr bevorzugtes Protokoll kommunizieren können. So ist für eine Java Anwendung die Kommunikation über RMI deutlich einfacher und performanter und sollte deshalb wenn möglich auch verwendet werden. Abbildung 21 zeigt eine Übersicht, wie eine RIA mit der Service Engine kommuniziert und wie eine Java Anwendung passend angebunden werden kann.

1 Beispielsweise für Java, Microsoft Silverlight oder Adobe Flash
2 Verwendet in Adobe Flash, Adobe Flex und Adobeb Air

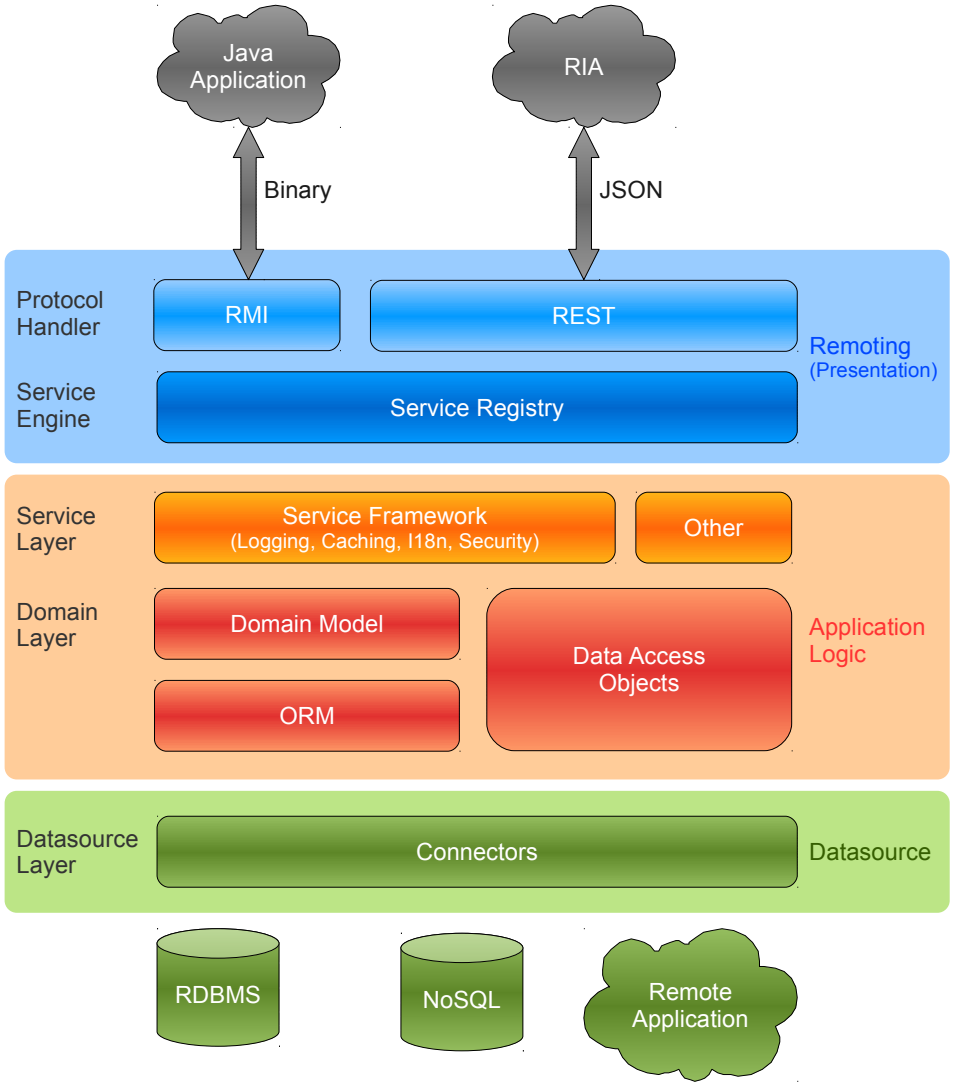

Abbildung 21: Übersicht für die Verwendung mit Rich Internet Applications

6.3 GATEWAY

Grundsätzlich stellt eine Anwendung die Verbindung mit anderen entfernten Systemen im Datasource Layer her. Die DAO Implementierungen übernehmen dann die Aufgabe über diese Verbindung Anfragen an ein entferntes System zu stellen und die Ergebnisse in das Domain Model der Anwendung zu konvertieren, wo es weiter verarbeitet wird. In manchen Fällen kann es aber auch sinnvoll sein, die Methoden des Entfernten Systems direkt als Service Interface zur Verfügung zu stellen. Die Implementierung dieses Interface leitet dann alle Methodenaufrufe direkt an das entfernte System weiter und übernimmt somit die Aufgabe einer Gateway. Eine Gateway arbeitet hierbei als Vermittler zwischen dem aufrufenden Client und einem anderen entfernten System, wobei der Client lediglich Kenntnis über das System benötigt mit dem er direkt kommuniziert, in diesem Fall also der Service Engine und dem Service Interface. Der Vorteil hierbei ist, dass ein Client ein beliebiges von den verwendeten Protocol Handlern angebotenes RPC Protokoll verwenden kann. Somit kann ein entferntes System über Protokolle angebunden werden, die von diesem nicht direkt unterstützt werden.

Die Implementierung einer verallgemeinerten Gateway Funktionalität befindet sich noch in einem sehr frühen Stadium, weshalb ein Entwickler momentan einen Großteil der Konvertierungsarbeit selbst in der entsprechenden Service Implementierung übernehmen muss. Abbildung 22 ordnet die Gateway Funktionalität in die Gesamtarchitektur einer Anwendung ein.

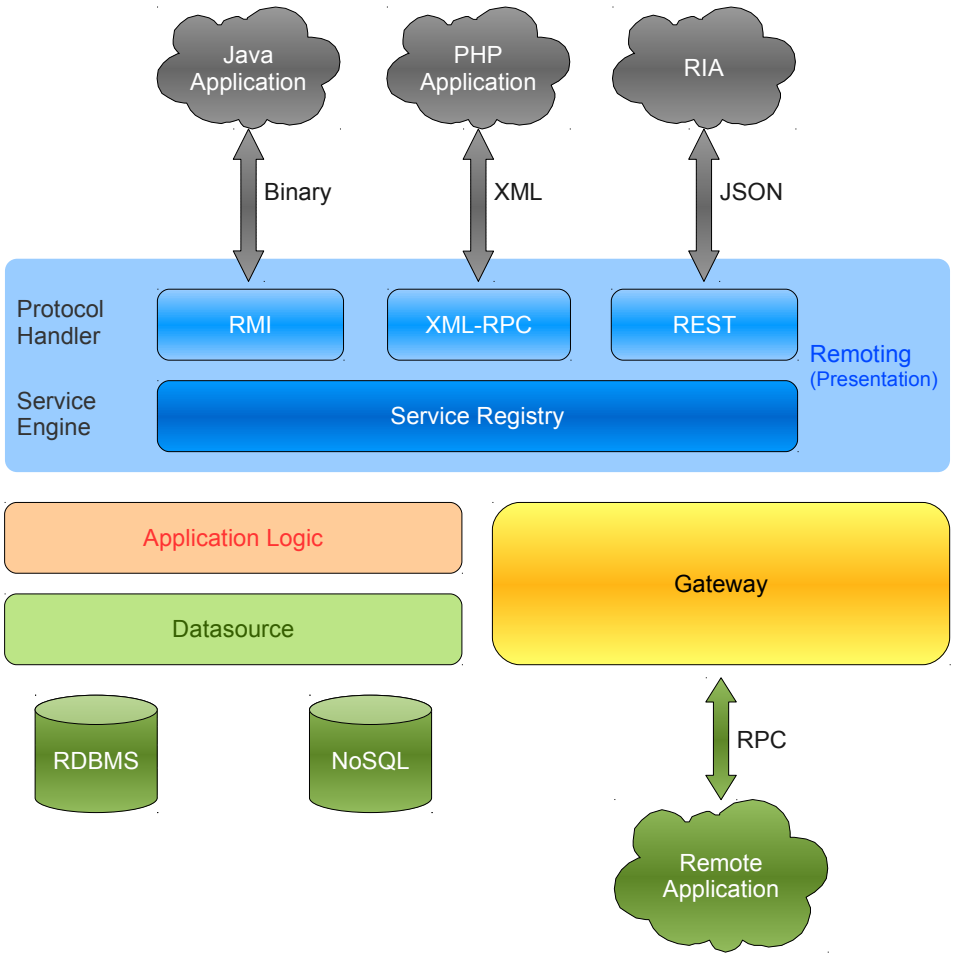

Abbildung 22: Einordnung der Gateway Funktionalität

7 AUSBLICK

7.1 ENTWICKLUNGS- UND TESTUMGEBUNG

Die Entwicklung erfolgte mit *Eclipse*[1] als Entwicklungsumgebung und *Ubuntu Linux*[2] 9.10 (64 bit Desktop Edition) als Betriebssystem. Für die Verwaltung des Projekts wurde das Online Projektmanagementsystem *Redmine*[3] und das Versionsverwaltungssystem *Subversion*[4] auf einem virtuellen Server in einem Frankfurter Rechenzentrum installiert. Dieses System soll auch für die Weiterentwicklung und eventuelle Veröffentlichungen dienen. In Eclipse selbst wurden verschiedene Plugins verwendet:

SPRING IDE Spring IDE ist ein Eclipse Plugin, das die Arbeit mit dem Spring Framework unterstützt. Dazu gehören automatische Vervollständigung von Klassennamen und Eigenschaften in einer Context Definition sowie weitere Möglichkeiten zur Verwaltung des Context. Spring IDE wurde inzwischen in die Springsource Tool Suite [5] integriert.

GROOVY ECLIPSE PLUGIN Für die Entwicklung in Groovy wurde das Groovy Eclipse Plugin[6] verwendet. Dadurch kann mit Groovy oder Java/Groovy Projekten ähnlich gearbeitet werden, wie schon aus der Java Entwicklung in Eclipse gewohnt.

IVYDE Ivy bietet die Möglichkeit, verschiedene Versionen von Bibliotheken und deren Abhängigkeiten von öffentlichen und nichtöffentlichen Repository Servern[7] zu beziehen. Dafür wird in einem Eclipse Projekt eine XML Konfigurationsdatei angelegt, in der alle Abhängigkeiten dieses Projekts definiert werden. Ivy bezieht nun diese Bibliotheken von den Repository Servern und macht sie in einem Projekt verfügbar. Ein weiterer Vorteil dieser Vorgehensweise ist, dass es nicht mehr notwendig ist, die verwendeten Java Bibliotheken mit dem Projekt in ein Versionskontrollsystem einchecken zu müssen.

1 http://eclipse.org
2 http://www.ubuntu.com
3 http://www.redmine.org
4 http://subversion.apache.org
5 http://www.springsource.com/products/sts
6 http://groovy.codehaus.org/Eclipse+Plugin
7 Bekanntester öffentlich verfügbarer Server ist http://mvnrepository.com

REDMINE MYLYN CONNECTOR Der Redmine Mylyn Connector[8] verbindet die unter Eclipse verfügbare Aufgabenverwaltung Mylyn mit dem Projektmanagementsystem Redmine. Dadurch kann die Verwaltung von Aufgaben und Fehlerberichten in einem Projekt zum Großteil direkt in Eclipse vorgenommen werden.

SUBCLIPSE Subclipse[9] ist ein Eclipse Plugin für die Versionsverwaltung mit Subversion.

Für die Tests einzelner Implementierungen wurden umfangreiche Unit Tests unter Verwendung von *JUnit*[10] erstellt. Auch hierfür existiert eine Unterstützung des Spring Frameworks. Für Tests über ein Netzwerk wurde auf dem lokalen System mit Hilfe der Virtualisierungslösung *Virtualbox*[11] ein *Debian*[12] 5.0 System installiert. In der virtuellen Maschine wurde die Verbindung zu dem in Abschnitt 5.2.2 vorgestellten XML-RPC Protocol Handler mit einer PHP Implementierung des XML-RPC Protokolls[13] getestet. Zum Testen des REST Protocol Handlers aus Abschnitt 5.2.3 wurde die Java Anwendung *Rest Client*[14] verwendet. Da bei diesem Testaufbau beide Teilnehmer auf dem gleichen physikalischen System arbeiteten, waren keine aussagekräftigen Lasttests möglich.

7.2 ERWEITERUNGSMÖGLICHKEITEN

Auch wenn die Ziele dieser Arbeit eine Grundlage für einen Service Layer zu erstellen und durch die Service Engine einen protokollunabhängigen Ausführungsmechanismus für entfernte Methodenaufrufe zu implementieren, erreicht wurden, lässt der modulare Aufbau des Gesamtsystems viele Erweiterungsmöglichkeiten zu.

Durch die Referenzimplementierungen der Protocol Handler für RMI, XML-RPC und REST werden zwar bereits einige häufig verwendete Protokolle unterstützt, aber vor allem in diesem Bereich besteht einiges Potential für Erweiterungen. Nachdem in der Service Engine die Darstellung von zustandsgebundenen und zustandslosen synchronen RPC Aufrufen abstrahiert wurde, wäre der nächste Schritt eine allgemeine Unterstützung für asynchrone entfernte Methodenaufrufe zu implementieren. Dadurch ist dann beispielsweise die Unterstützung des Webservice Protokolls SOAP in der Version 1.2 möglich.

Auch die Ausarbeitung von Details in den vorhandenen Implementierungen ist an manchen Stellen möglich. So ist es geplant die in Kapitel 4

8 http://sourceforge.net/projects/redmin-mylyncon/
9 http://subclipse.tigris.org
10 http://www.junit.org
11 http://www.virtualbox.org
12 http://www.debian.org
13 http://phpxmlrpc.sourceforge.net/
14 http://code.google.com/p/rest-client/

vorgestellten Service Implementierungen zu einem Framework zu erweitern, das den Aufbau des Service Layer einer Anwendung vorgibt. Dazu gehört auch eine bessere Integration der Services untereinander, da diese momentan noch manuell in der Spring Context Definition vorgenommen werden muss. Ein weiterer Punkt ist die Optimierung der HTTP Umsetzung (entsprechend der Spezifikation in [FGM+99]) des REST Protokoll Handler und die Fertigstellung der Presentation Implementierung für HTML.

7.3 FAZIT

Die Erstellung von modernen Unternehmens- und Webanwendungen ist ein komplexes Themengebiet, das einem ständigen Wandel unterzogen ist. Diese Arbeit und die vorgestellten Implementierungen sollen dazu beitragen diese Komplexität an einigen Stellen etwas zu reduzieren und zu strukturieren.

Dafür wurden im Einleitungskapitel die Definition und die Herausforderungen von modernen Unternehmensanwendungen herausgearbeitet und die Aufgabenstellung dieser Arbeit sowie die Herangehensweise anhand der verwendeten Schichtenarchitektur vorgestellt.

Aufgabe war es zum einen das Konzept des Service Layer zu evaluieren und anhand der Eigenschaften von Unternehmensanwendungen eine Reihe von Services zu implementieren, die einen Teil dieser Aufgabe übernehmen. Diese Implementierungen können somit in verschiedenen Anwendungen die Grundlage für einen Service Layer bilden. Zum anderen sollte ein zentraler Ausführungsmechanismus entwickelt werden, der die Grundlage dafür bietet, die Methoden dieses Service Layer über entfernte Methodenaufrufe zur Verfügung zu stellen. Darauf aufbauend sollten anschließend Reihe von RPC Protokollen für entfernte Methodenaufrufe evaluiert werden und die jeweiligen Implementierungen, die diesen Ausführungsmechanismus verwenden, realisiert werden.

Ein Grundlagenkapitel erstellte einen Überblick über die verwendeten Technologien, Konzepte und Begriffe, die für das weitere Verständnis wichtig waren. Ein Schwerpunkt lag dabei auf der Java Plattform und den dort möglichen Programmiersprachen sowie auf verschiedenen Java Technologien, unter anderem dem Spring Framework. Weiterhin wurde das Konzept des entfernten Methodenaufrufs (RPC) sowie verschiedene Möglichkeiten für den Datenaustausch behandelt.

Im weiteren Verlauf wurde der Datasource Layer, der dazu dient die Verbindung zu anderen Anwendungen herzustellen, als unterste Schicht der Architektur vorgestellt. Darauf aufbauend wurden die verschiedenen Komponenten des Domain Layer näher betrachtet, die bereits einen Teil der Anwendungslogik übernehmen. Für die weitere Implementierung von Anwendungslogik wurde anschließend das Konzept des Service Layer eingeführt, der eine gemeinsame Schnittstelle einer Anwendung an ei-

ner zentralen Stelle definiert. In diesem Zusammenhang wurden dann einige Service Implementierungen vorgestellt, die aufgrund der allgemeinen Anforderungen von Unternehmensanwendungen in verschiedenen Anwendungen zum Einsatz kommen können.

Das nachfolgende Kapitel beschäftigte sich mit dem Remoting Layer, der dazu dient den Service Layer einer Anwendung über verschiedene RPC Protokolle anzubinden. Hier wurde die Implementierung der Service Engine vorgestellt, die die Aufgabe übernimmt die abstrahierte Form eines Methodenaufrufs zu verarbeiten und auf einem entsprechenden Service auszuführen. Durch diese allgemeine Form eines Methodenaufrufs ist es nun möglich verschiedene RPC Protokolle über den zentralen Ausführungsmechanismus der Service Engine anzubinden. Dazu wurden in den folgenden Abschnitten Referenzimplementierungen für RMI, XML-RPC und REST behandelt. Weiterhin wurden dann im Rahmen des Gesamtkonzeptes verschiedene Anwendungsfälle gezeigt, wie diese Implementierungen genutzt werden können.

Zusammenfassend lässt sich sagen, dass beide Teile der Aufgabenstellung erfolgreich umgesetzt wurden. Zum einen wurde eine Reihe von Services hergeleitet und implementiert, die verschiedene häufige Anforderungen bei der Entwicklung einer Unternehmensanwendung in einer generischen Form umsetzen und deshalb leicht wiederverwendbar sind. Der Vorteil dieser Implementierungen und der Verwendung eines Service Layer ist, dass sie unabhängig von der Präsentationsschicht einer Anwendung arbeiten. Dies steht im Unterschied zu einem Großteil der bisher verfügbaren Frameworks, da diese in den meisten Fällen mit einer konkreten Präsentationsschicht arbeiten[15].

Zum anderen wurde eine Anwendungsschicht in Form der Service Engine entwickelt, die eine zentrale Ausführungsumgebung für Anfragen von verschiedenen RPC Protokollen auf den Services des Service Layer bietet. Die Service Engine verwendet dabei eine allgemeine Darstellung eines RPC und implementiert selbst kein konkretes Protokoll. Diese allgemeine Darstellung wird dann von der übergeordneten Schicht in Form verschiedener Protocol Handler, die ein konkretes RPC Protokoll implementieren, verwendet. An den Referenzimplementierungen für RMI, XML-RPC und REST wurde nun gezeigt, dass auch Protokolle mit unterschiedlichen Eigenschaften und Voraussetzungen an die Service Engine angebunden werden können. Auch hier existierten bisher nur Bibliotheken, die ein konkretes RPC Protokoll implementieren und auch direkt die Aufgabe der Ausführung übernehmen, was die Unterstützung mehrerer RPC Protokolle deutlich komplizierter gestaltete.

15 Beispielsweise Webframeworks, GUI Frameworks etc.

Teil IV

ANHANG

A

CODEBEISPIELE

A.1 SPRING FRAMEWORK

Die folgenden Codebeispiele sollen die Verwendung des Dependency Injection Container des Spring Frameworks zeigen. Zu diesem Zweck soll das bisher verwendete User JavaBean als Grundlage verwendet werden:

```
import java.security.NoSuchAlgorithmException;

public class User implements Serializable {
        private String userName;
        private String password;

        public User() {
                this.setPlainPassword("secret"); // Standard password
        }

        public String getUserName() {
                return userName;
        }

        public void setUserName(String userName) {
                this.userName = userName;
        }

        public String getPassword() {
                return password;
        }

        public void setPassword(String password) {
                this.password = password;
        }

        // Write only
        public void setPlainPassword(String plainPassword)
                        throws NoSuchAlgorithmException {
                // MD5 encode the plain password
                MessageDigest md = MessageDigest.getInstance("MD5");
                md.update(plainPassword.getBytes(), 0, plainPassword.
                    length());
                // Set the password string to the MD5 hash in HEX
                    values
```

```
                String md5Hex = new BigInteger(1, md.digest()).
                    toString(16);
                this.setPassword(md5Hex);
        }

        public boolean equals(Object other) {
                return this.getClass().equals(other.getClass()) &&
                        ((User)other).getUsername().equals(this.
                            getUsername()) &&
                        ((User)other).getPassword().equals(this.
                            getPassword());
        }
}
```

Listing 21: User Domain Objekt (User.java)

Anschließend wird ein Service Interface für dieses User Domain Objekt erstellt, das die einfache Authentifizierung eines Benutzers anhang von Benutzername und Passwort ermöglicht.

```
package com.example;

public interface UserService
{
        public User login(String username, String password);
}
```

Listing 22: Beispiel eines Service Interface (UserService.java)

Dieses Interface wird nun in einer sehr einfachen Form implementiert. Die Implementierung erwartet eine Liste von Benutzer Objekten und überprüft bei einem Aufruf der login() Methode, ob die übergebenen Authentifizierungsdaten mit einem User Objekt übereinstimmen.

```
package com.example;

import java.util.List;

public class SimpleUserService implements UserService
{
        private List<User> users;

        @Override
        public User login(String username, String password)
        {
                User loginUser = new User();
                loginUser.setUsername(username);
                // MD5 encodes password
                loginUser.setPlainPassword(password);
```

```
                for(User user : users)
                {
                        // Users can now be compared using equals
                        if(loginUser.equals(user)) {
                                return user; // Return logged in user
                        }
                }
                throw new SecurityException("Could not find user!") ;
        }

        public void setUsers(List<User> users)
        {
                this.users = users;
        }

        public List<User> getUsers()
        {
                return users;
        }

}
```

Listing 23: Einfache Implementierung des User Service Interface (SimpleUserService.java)

Nun muss eine Spring Context Definition erstellt werden, die die SimpleUserService Implementierung instanziiert und dieser eine Reihe von User Objekten übergibt.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.springframework.org/schema/
            beans
                http://www.springframework.org/schema/beans/spring-
                    beans-3.0.xsd">

        <bean name="userService" class="com.example.SimpleUserService
            ">
                <property name="users">
                        <list>
                                <ref bean="user1" />
                                <ref bean="user2" />
                        </list>
                </property>
        </bean>
```

```
        <bean name="user1" class="com.example.User">
                <property name="username" value="User1" />
                <property name="plainPassword" value="secret1" />
        </bean>

        <bean name="user2" class="com.example.User">
                <property name="username" value="User2" />
                <property name="plainPassword" value="secret2" />
        </bean>
</beans>
```

Listing 24: Spring Context Definition für den SimpleUSerService (applicationContext.xml)

Wird der dieser Context nun in einem Hauptprogramm in den Dependency Injection Container gelesen können die dort definierten Objekte anhand der Interfaces benutzt werden, die diese implementieren. Das folgende Listing 25 zeigt die Verwendung in einem einfachen Java Kommandozeilenprogramm.

```
package com.example;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.
    ClassPathXmlApplicationContext;

public class Bootstrap
{
        public static void main(String[] args)
        {
                ApplicationContext context = new
                    ClassPathXmlApplicationContext("applicationContext
                    .xml");
                UserService userService = context.getBean("
                    userService", UserService.class);
                assert(userService.login("User2", "secret2") != null)
                    ;
        }
}
```

Listing 25: Verwendung des Dependency Injection Containers (Bootstrap.java)

A.2 XML-RPC

Das in Abschnit A.1 erstellte UserService Interface wird hier auch verwendet um einen Aufruf über das XML-RPC Protokoll zu illustrieren. Ein Methodenaufruf auf dieses Interface hat die folgende Form:

```
POST /xml-rpc HTTP/1.0
User-Agent: Remote Application X
Host: example.com
Content-Type: text/xml
Content-length: 224

<?xml version="1.0"?>
<methodCall>
        <methodName>userService.login</methodName>
        <params>
                <param>
                        <value><string>User2</string></value>
                        <value><string>secret2</string></value>
                </param>
        </params>
</methodCall>
```

Listing 26: XML-RPC Methodenaufruf

Wurde der richtige Benutzer gefunden wird ein Struct Objekt, das den entsprechenden Benutzer repräsentiert, zurückgeliefert.

```
HTTP/1.1 200 OK
Connection: close
Content-Length: 305
Content-Type: text/xml
Date: Fri, 11 Mar 2010 23:59:59 GMT
Server: Service Engine HTTP Connector

<?xml version="1.0"?>
<methodResponse>
        <params><param><value>
                <struct>
                        <member>
                                <name>username</name>
                                <value><string>User2</string></value>
                        </member>
                        <member>
                                <name>password</name>
                                <value><string>
                                e54cfb3714f76cedd4b27889e1f6a174
                                </string></value>
                        </member>
```

```
                </struct>
        </value></param></params>
</methodResponse>
```

Listing 27: Antwort auf einen XML-RPC Methodenaufruf

Wurde der entsprechende Benutzer nicht gefunden wird ein Fehlerobjekt in der folgenden Form übertragen.

```
HTTP/1.1 200 OK
Connection: close
Content-Length: 323
Content-Type: text/xml
Date: Fri, 11 Mar 2010 23:59:59 GMT
Server: Service Engine HTTP Connector

<?xml version="1.0"?>
<methodResponse>
        <fault>
                <value><struct>
                        <member>
                                <name>faultCode</name>
                                <value><int>9999</int></value>
                        </member>
                        <member>
                                <name>faultString</name>
                                <value><string>
                                Could not find user!
                                </string></value>
                        </member>
                </struct></value>
        </fault>
</methodResponse>
```

Listing 28: XML-RPC Antwort bei einem Fehler

[BLFM05] Berners-Lee, T. ; Fielding, R. ; Masinter, L.: *Uniform Resource Identifier (URI): Generic Syntax*. RFC 3986 (Standard). Januar 2005 (Request for Comments). – http://www.ietf.org/rfc/rfc3986.txt (Cited on page 71.)

[BPM+08] Bray, Tim ; Paoli, Jean ; Maler, Eve ; Yergeau, François ; Sperberg-McQueen, C. M.: Extensible Markup Language (XML) 1.0 (Fifth Edition) / W3C. 2008. – W3C Recommendation (Cited on page 24.)

[Bra09] Brandon, Tina: *SaaS Billing, Cloud Computing Monetization and Recurring Billing Payment Processing*. http://www.articlesbase.com/finance-articles. November 2009. – SC 1418342 (Cited on page 53.)

[Cro06] Crockford, D.: *The application/json Media Type for JavaScript Object Notation (JSON)*. RFC 4627 (Informational). Juli 2006 (Request for Comments). – http://www.ietf.org/rfc/rfc4627.txt (Cited on page 25.)

[DK06] DeMichiel, Linda ; Keith, Michael: JSR 220: Enterprise JavaBeansTM, Java Persistence API / Sun Microsystems. 2006. – Forschungsbericht (Cited on page 36.)

[FGM+99] Fielding, R. ; Gettys, J. ; Mogul, J. ; Frystyk, H. ; Masinter, L. ; Leach, P. ; Berners-Lee, T.: *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616 (Draft Standard). Juni 1999 (Request for Comments). – http://www.ietf.org/rfc/rfc2616.txt (Cited on pages 72 and 89.)

[Fie00] Fielding, Roy T.: *Architectural Styles and the Design of Network-based Software Architectures*, University of California, Irvine, Diss., 2000. – http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf (Cited on pages 10, 69, 70, 71, and 74.)

[Fow02] Fowler, Martin: *Patterns of Enterprise Application Architecture*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2002. – ISBN 0321127420 (Cited on pages 3, 4, 9, 31, 34, 35, 39, and 57.)

[Fow04] Fowler, Martin. *Inversion of Control Containers and the Dependency Injection pattern*. http://martinfowler.com/articles/injection.html. Januar 2004 (Cited on page 17.)

[GHJV95] Gamma, Erich ; Helm, Richard ; Johnson, Ralph E. ; Vlissides, John: *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, March 1995 (Cited on page 9.)

[Ham97] Hamilton, Graham. *JavaBeans specification*. http://java.sun.com/javase/technologies/desktop/javabeans/docs/spec.html. August 1997 (Cited on page 13.)

[Hen07] Henderson, Bryan. *XML-RPC Introspection*. http://xmlrpc-c.sourceforge.net/introspection.html. November 2007 (Cited on page 69.)

[ISO86] ISO: *ISO 8879:1986: Information processing — Text and office systems — Standard Generalized Markup Language (SGML)*. International Organization for Standardization, August 1986. – 155 S (Cited on page 24.)

[JRH99] Jacobs, Ian ; Raggett, David ; Hors, Arnaud L.: HTML 4.01 Specification / W3C. 1999. – W3C Recommendation (Cited on page 24.)

[Ju09] Johnson, Rod ; u.a., Juergen H. *Spring Framework 3.0 Reference Documentation*. http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/. 2009 (Cited on pages 16, 18, 19, and 66.)

[KKLS07] Koenig, Dierk ; King, Paul ; Laforge, Guillaume ; Skeet, Jon: *Groovy in Action*. Greenwich, CT, USA : Manning Publications Co., 2007. – ISBN 1932394842 (Cited on page 11.)

[Kru04] Kruth, Wilhelm: *IT Grundlagenwissen - Kompaktwissen Informationstechnik für Datenschutz- und Security-Management*. 2. Frechen, Deutschland : DATAKONTEXT-FACHVERLAH GmbH, 2004 (Cited on page 50.)

[MBF+04] McCabe, Francis ; Booth, David ; Ferris, Christopher ; Orchard, David ; Champion, Mike ; Newcomer, Eric ; Haas, Hugo: Web Services Architecture / W3C. 2004. – W3C Note (Cited on pages 22 and 23.)

[Mic88] Microsystems, Sun: *RPC: Remote Procedure Call Protocol specification: Version 2*. RFC 1057 (Informational). Juni 1988 (Request for Comments). – http://www.ietf.org/rfc/rfc1057.txt (Cited on page 20.)

[PSA01] PARK, Joon S. ; SANDHU, Ravi ; AHN, Gail-Joon: Role-based access control on the web. In: *ACM Trans. Inf. Syst. Secur.* 4 (2001), Nr. 1, S. 37–71 (Cited on page 51.)

[Saa08] SAAB, Paul. *Scaling memcached at Facebook*. http://www.facebook.com/note.php?note_id=39391378919. Dezember 2008 (Cited on page 45.)

[Sri95] SRINIVASAN, R.: *RPC: Remote Procedure Call Protocol Specification Version 2*. RFC 1831 (Proposed Standard). August 1995 (Request for Comments). – http://www.ietf.org/rfc/rfc1831.txt (Cited on page 20.)

[Sto06] STOYANOVA, Nadezhda: *Domain Model*. http://kaul.inf.fh-bonn-rhein-sieg.de/wiki2/index.php?title=Domain_Model&oldid=3253. Dezember 2006. – [Online; Stand 01. März 2010] (Cited on page 33.)

[sun] (Cited on page 65.)
Java Remote Method Invocation Whitepaper. http://java.sun.com/javase/technologies/core/basic/rmi/whitepaper/index.jsp

[sun02] (Cited on page 35.)
Core J2EE Patterns - Data Access Object. http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html. 2002

[sun09] (Cited on page 15.)
Java Reflection API. http://java.sun.com/javase/6/docs/api/java/lang/reflect/package-summary.html. 2009

[Til09] TILKOV, Stefan: REST - Der bessere Web Service? In: *Javamagazin* (2009), Nr. 1, S. 74 – 80 (Cited on page 72.)

[Whi75] WHITE, J.E.: *High-level framework for network-based resource sharing*. RFC 707. Dezember 1975 (Request for Comments). – http://www.ietf.org/rfc/rfc707.txt (Cited on page 20.)

[Wik09] WIKIPEDIA: *Objektrelationale Abbildung — Wikipedia, Die freie Enzyklopädie*. http://de.wikipedia.org/w/index.php?title=Objektrelationale_Abbildung&oldid=64500679. 2009. – [Online; Stand 7. März 2010] (Cited on page 36.)

[Wik10a] WIKIPEDIA: *Associative array — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Associative_array&oldid=338560020. 2010. – [Online; accessed 1-March-2010] (Cited on page 9.)

[Wik10b] WIKIPEDIA: *Cache algorithms — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Cache_algorithms&oldid=342709946. 2010. – [Online; Stand 20. Februar 2010] (Cited on page 42.)

[Wik10c] WIKIPEDIA: *Java Virtual Machine — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Java_Virtual_Machine&oldid=347523883. 2010. – [Online; accessed 5-March-2010] (Cited on page 10.)

[Wik10d] WIKIPEDIA: *List of JVM languages — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=List_of_JVM_languages&oldid=342539250. 2010. – [Online; accessed 28-February-2010] (Cited on page 11.)

[Wik10e] WIKIPEDIA: *List of object-relational mapping software — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=List_of_object-relational_mapping_software&oldid=347165713. 2010. – [Online; accessed 1-March-2010] (Cited on page 37.)

[Wik10f] WIKIPEDIA: *NoSQL — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=NoSQL&oldid=347311249. 2010. – [Online; accessed 3-March-2010] (Cited on page 32.)

[Wik10g] WIKIPEDIA: *Reflexion (Programmierung) — Wikipedia, Die freie Enzyklopädie*. http://de.wikipedia.org/w/index.php?title=Reflexion_(Programmierung)&oldid=69865549. 2010. – [Online; Stand 28. Februar 2010] (Cited on page 15.)

[Wik10h] WIKIPEDIA: *Remote Method Invocation — Wikipedia, Die freie Enzyklopädie*. http://de.wikipedia.org/w/index.php?title=Remote_Method_Invocation&oldid=70184266. 2010. – [Online; Stand 23. Februar 2010] (Cited on pages 65 and 66.)

[Wik10i] WIKIPEDIA: *Rich Internet application — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Rich_Internet_application&oldid=348365799. 2010. – [Online; accessed 8-March-2010] (Cited on page 83.)

[Win99] WINER, Dave. *XML-RPC Specification*. http://www.xmlrpc.com/spec. Juni 1999 (Cited on page 68.)

[WWWK94] WALDO, Jim ; WYANT, Geoff ; WOLLRATH, Ann ; KENDALL, Sam. *A Note on Distributed Computing*. http://research.sun.com/techrep/1994/smli_tr-94-29.pdf. November 1994 (Cited on page 22.)