Hochschule für angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Andreas Lehmann

Ein exemplarischer Vergleich von Ruby und Java
anhand einer Webanwendung

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Andreas Lehmann

Ein exemplarischer Vergleich von Ruby und Java anhand einer Webanwendung

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Angewandte Informatik
am Studiendepartment Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. Olaf Zukunft
Zweitgutachter : Prof. Dr. Wolfgang Gerken

Abgegeben am 26. März 2008

Andreas Lehmann

Thema der Bachelorarbeit

Ein exemplarischer Vergleich von Ruby und Java anhand einer Webanwendung

Stichworte

Ruby, Ruby on Rails, Active Record, JavaServer Faces, Framework, Metriken

Kurzzusammenfassung

In dieser Arbeit werden zwei Webanwendungen mit unterschiedlichen Technologien entwickelt. Der fachliche Rahmen beider Anwendungen reduziert sich auf eine Benutzerverwaltung. Für eine Anwendung wird das Framework „Ruby on Rails“ eingesetzt. Das Framework wird detailliert vorgestellt. Zweite Anwendung wird mit dem Framework „JavaServer Faces“ implementiert. Dieses Framework wird nur überblicksartig präsentiert. Anschließend werden die entwickelte Anwendungen auf ihre Performance getestet und auf den Entwicklungsaufwand untersucht. Dafür werden entsprechende Metriken entwickelt und vorgestellt. Die Messergebnisse werden miteinander verglichen und ausgewertet.

Andreas Lehmann

Title of the paper

An examplary comparison of Ruby with Java on the basis of a webapplication

Keywords

Ruby, Ruby on Rails, Active Record , JavaServer Faces, Framework, Metrics

Abstract

This thesis demonstrates the development of two web applications with different technologies. The two applications a user administration. For one of the applications the framework „Ruby on Rails“ is used. This framework is described in detail. The second application is implemented with the framework „JavaServer Faces“. The demonstration of this framework is given in the way of an overview. Finally the developed applications are tested for their performance and are analysed for their development effort. For this purpose appropriate are developed and described. The measure results are compared to each other and evaluated.

Danksagung

An dieser Stelle möchte ich meinen Dank denjenigen aussprechen, die mir bei der Realisierung dieser Arbeit zur Seite standen.

Als erstes möchte ich mich bei meinem Betreuer, Prof. Dr. Olaf Zukunft für seine Unterstützung während meiner Bachelorarbeit bedanken.

Des Weiteren möchte ich mich bei einem weiteren hilfsbereiten Menschen bedanken: Frau Angelika Rusalowski.

Ohne diese beiden Menschen wäre diese Arbeit nicht zu Stande gekommen.

Andreas Lehmann, März 2008

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

1 Einleitung

„Das Abenteuer beginnt."

1.1 Motivation

Heute ist der PC sowie das Internet aus unserem Leben nicht mehr weg zu denken. Im Jahre 1969 diente ein militärisches Projekt des US-Verteidigungsministeriums namens „ARPA"-*Advanced Research Project Agency* der Entstehung des Internets. Es wurde zur Vernetzung von Universitäten und Forschungseinrichtungen benutzt, mit der Absicht die knappen Rechenkapazitäten erst in USA dann weltweit zu nutzen. Seitdem hat sich das Internet permanent aus der technischen Hinsicht weiterentwickelt und ist erheblich gewachsen.

Eine genaue Anzahl der Teilnehmer im Internet kann heute nicht exakt bestimmt werden, da es viele Mobile-Geräte mit einem Internetzugang gibt, deren Benutzer sich jederzeit an- und abmelden können. „Laut IWS hatten im März 2007 etwa 16,9 Prozent der Weltbevölkerung Zugang zum Internet. Laut EITO nutzen Anfang 2007 1,13 Milliarden Menschen das Internet. In Deutschland verfügen ungefähr 68 Prozent der Erwachsenen über einen Internetanschluss. Etwa 80 Prozent der deutschen Jugendlichen (10-13 Jahre) nutzen das Internet. Etwa 60 Prozent aller Deutschen nutzen regelmäßig das Internet, Tendenz steigend um 2-3 Prozent jährlich"[W01]-10.10.07.

Mit der Entstehung des „World Wide Web" im Jahre 1989 gab es die Möglichkeit statische Informationen im „HTML"-Format von einem Rechner - „Server" übers

Internet zu holen und mit Hilfe eines Web-Browsers auf dem Rechner des Benutzers - „Client“ darzustellen.

Die steigenden Anforderungen des Benutzers haben zur Weiterentwicklung des Web's beigetragen. Angefangen mit der einfachen Darstellung eines Textes oder Dokumentes entwickelte sich das Web zu den interaktiven Webanwendungen, wie Information-Portale, Online-Shops und Online-Büchereien. Für die Firmen wurde das Internet zu einem großen Markt mit Kunden aus aller Welt. Die Anbieter konnten sich durch das Internet dem Verbraucher präsentieren. Heute haben nicht nur große Unternehmen einen eigenen Internet-Auftritt, sondern auch Selbstständige und Privatpersonen.

Dadurch wurde die Softwareentwicklung im Allgemeinen beeinflusst und wurde unmittelbar in die Richtung der Webanwendungsentwicklung gelenkt. So wie die Firmen unterschiedlich sind, so wollen sie auch unterschiedlich auffällig im Internet auftreten, um neue Kunden zu beeindrucken und anschließend zu gewinnen. Durch die neuen Anforderungen an eine Webanwendung haben sich auch verschiedene Technologien entwickelt, mit denen diese gebaut werden kann.

Einige Technologien basieren auf den schon bekannten, sich bewerten Programmiersprachen, wie „Java“, andere hingegen auf den neuen Programmiersprachen. Dazu gehört zum Beispiel „Ruby“. Neue Technologien versprechen dem Softwareentwickler (weiter auch Entwickler) oft einfachere Handhabung des Programmierens, sowie effizientere Lösungen für bekannte Probleme. Aber auch Lösungswege für neue Probleme, die mit den steigenden Anforderungen des Benutzers entstehen, werden schneller gefunden.

1.2 Aufgabenstellung

In dieser Arbeit werden zwei Webanwendungen mit gleicher Funktionalität unter Verwendung verschiedener Technologien gebaut. Eine Anwendung wird in Java entwickelt, wobei für die Darstellung der Web-Oberfläche die „JavaServer Faces“ Technologie verwendet wird. Die Kommunikation mit der Persistenzschicht geschieht mittels JDBC. Für die zweite Anwendung kommt das Framework „Ruby on Rails“ zum Einsatz, welches auf Ruby basiert. Das Framework liefert ein Mechanismus - „ActiveRecords“ mit, der die relationale Persistenzschicht und die objektorientierte Geschäftslogik der Anwendung in „Ruby on Rails“ miteinander verbindet. Die Funktionsweise dieser Anwendungen wird genau untersucht und detailliert erläutert. Danach wird der Entwicklungsaufwand für die beiden Anwendungen ermittelt. Anschließend wird die Performance der beiden Webanwendungen gemessen und miteinander verglichen.

1.3 Aufbau der Arbeit

Im Kapitel 2 wird es eine Einführung in die Programmiersprache Ruby und in deren Entstehungsgeschichte geben. Zur Verdeutlichung der Sprache wird der Text mit einigen Code-Beispielen bereichert. Das Kapitel 3 widmet sich dem Framework „Ruby on Rails“, es erklärt genau die Funktionsweise von „ActiveRecords“. Im darauf folgenden Kapitel 4 wird eine, auf Java basierende Web-Technologie „JSF“ beschrieben. Anschließend werden im Kapitel 5 einige Vergleichsmethoden in der Softwareentwicklung dargestellt. Die Metriken, die in diese Arbeit eingeflossen sind, werden einzeln beschrieben. Das Kapitel 6 beschäftigt sich mit dem Design der beiden Testanwendungen, die eine Benutzerverwaltung realisieren sollen. Die Realisierung der Anwendungen kann im Kapitel 7 nachvollzogen werden. Des Weiteren wird dem Leser die Entwicklungsumgebung vorgestellt und deren Auswahl begründet. Im Kapitel 8 wird beschrieben, wie der Versuch aufgebaut, durchgeführt und ausgewertet wird. Folglich werden die Ergebnisse erfasst und Schlüsse gezogen. Das Fazit sowie ein Ausblick für die beiden Web-Technologien kommen abschließend im Kapitel 9.

1.4 Rahmen der Arbeit

Im Rahmen dieser Arbeit wird vorausgesetzt, dass es dem Leser folgende Themen bekannt sind: Objektorientierte Programmierung (OOP), Client-Server-Architektur, da diese nicht genau erklärt werden.

Eine Webanwendung beschreibt eine Client-Server-Architektur mit einer TCP/IP Verbindung zum gegenseitigen Informationsaustausch. Als Kommunikationsprotokoll kommt „HTTP“ zum Einsatz (vgl. Abbildung 1 auf der nächsten Seite). Auf der Serverseite befinden sich Geschäfts- und Anwendungslogik, sowie Datenbank zum persistenten Speichern der Daten und der Web-Server. Clientseitig wird ein Web-Browser benutzt, der seinerseits persistente Daten (HTTP-Cookie) intern verwaltet.

Der Benutzer startet die Webanwendung, indem er die URL des Web-Servers (sei gleich der Adresse der Internetseite gesetzt) in einem Web-Browser eingibt, damit sendet er die erste Anfrage (HTTP-Request) an den Web-Server. Der Web-Server nimmt diese entgegen und leitet die an ein Programm weiter. Das Programm generiert daraufhin den HTML-Quellcode einer Webseite, welche dann vom Web-Server als die Antwort (HTTP-Response) an den Client zurück gesendet wird. Diese Webseite ist die grafische Oberfläche der Webanwendung, über die die Daten an die Geschäftslogik der Anwendung geliefert werden. Dort werden diese abgearbeitet und dem Anwender als Ergebnis zum Sehen gestellt.

Dieser Prozess läuft in einem Zyklus ab, der Anfrage-Antwort-Zyklus genannt wird und ist in der Abbildung 1 auf der nächsten Seite dargestellt. Aus technischen Gründen wird die Anwendung nicht ins Internet gestellt, sondern lokal entwickelt.

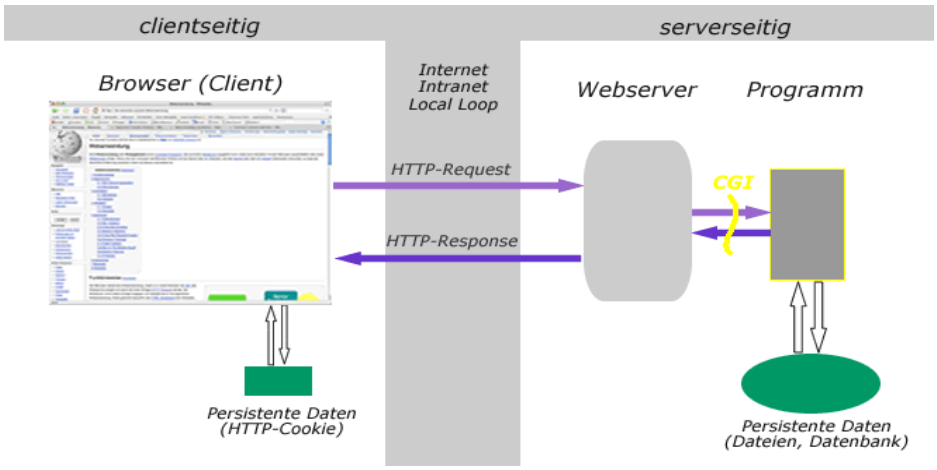


Abbildung 1: Datenfluss bei einer Webanwendung [W10] - (22.03.07)

2 Ruby

„Rubin unter den Programmiersprachen."

2.1 Hintergrund

Obwohl es in dieser Arbeit zum größten Teil um das Framework „Ruby on Rails“ geht, muss deutlich gemacht werden, worauf es eigentlich basiert. Der grundlegende Baustein des Frameworks ist die Programmiersprache Ruby. In diesem Kapitel wird es nur eine Einführung in Ruby gegeben. Eine komplette Betrachtung der Sprache würde den Rahmen der Arbeit sprengen.

Dennoch sollen die Unterschiede zu den anderen Programmiersprachen verdeutlicht werden, unter anderem anhand kleiner Beispielen. Um den späteren Gedanken und den Zusammenhängen im Verlaufe dieser Arbeit folgen zu können, darf auf dieses Kapitel nicht verzichtet werden. Im Endeffekt stellt dieses Kapitel kein Nachschlagewerk zu Ruby. Daher wird auf den „Leitfaden der Pragmatischen Programmierer“ [LPP] verwiesen.

2.2 Überblick

Als Vater von Ruby gilt der Japaner Yukihiro „Matz“ Matsumoto, der seit 24 Februar 1993 in Japan die Programmiersprache entwickelt. Die Sprache sollte die Vorteile von Perl, Python und Smalltalk vereinen. Der Name Ruby (auf deutsch Rubin) wurde von dem Namen der Sprache Perl (auf deutsch Perle) beeinflusst. Der Name inspirierte den Japaner, den Namen eines Edelsteins für seine Sprache zu verwenden.

Abbildung 2: Yukihiro Matsumoto [W11] - (22.03.08)

Heute wird Ruby als OpenSource-Project weiter gepflegt. Erst ab dem Jahr 2000 wurde die Sprache in Europa und Amerika durch nichtjapanische Literatur bekannt. Die frage „Was ist Ruby ?“ kann folgendermaßen beantwortet werden:

„Nimm eine richtig objekt-orientierte Sprache, etwa Smalltalk. Schmeiße die ungewöhnliche Syntax raus und benutze einen etwas konventionelleren, datei-basierten Quell-Code. Nun füge ein gutes Maß an Flexibilität und Komfort solcher Sprachen wie Python oder Perl hinzu“. [LPP]

Somit ergibt sich eine interpretierte, objekt-orientierte Scriptsprache. Der Programmcode wird zur Laufzeit von einem Interpreter übersetzt. Das ist das Standardverfahren für Programme aller Art. Es gibt eine Implementierung eines Ruby-Interpreters in Java. Sie ermöglicht die Interaktion von Ruby und Java in beiden Richtungen. Das Ganze läuft auf der JVM - „*Java Virtual Machine“* (sieh „Jruby“).

2.3 Grundlagen

2.3.1 Ein Beispiel

Das Programm „Hello World !“ ist der bekannteste Programmcode dieser Welt. Um das auf dem Bildschirm ausgeben zu können, wird in Ruby eine Zeile benötigt:

```
puts „Hello World!“
```
[1]

Eine `main`-Methode oder eine Klasse sind für diese Aufgabe nicht notwendig. Dabei ist `Hello World!` ein Objekt der Klasse `String`.

2.3.2 Objektorientierung

Im Gegensatz zu den anderen Programmiersprachen muss Ruby komplett objektorientiert programmiert werden. In Ruby gibt es nur Objekte. Die Zahl `8` repräsentiert ein Objekt der Klasse `Fixnum`, die Zahl `15.4` gehört der Klasse `Float` an. Wahrheitswert `true` ist eine Instanz der Klasse `TrueClass`, `nil` (Nullzeiger) und `self` (in Java `this`) besitzen ebenso Objekteigenschaften. Die Klasse `String` schließt Zeichenketten in Hochkommata wie `"Hello World"` ein.

Auf den ersten Blick stellen komplexere Ausdrücke wie Arrays oder Zahlenfolgen (Ranges) keine Ausnahmen dar. Dementsprechend sind das auch Objekte. Sogar die Codeblöcke können als Objekte beschrieben werden (`Closures`).

Diese Besonderheit der Sprache lässt die typischen Merkmale der Objektorientierung wie Kapselung, Generalisierung/Spezialisierung und Methodenaufrufe auf allen Repräsentanten von Ruby implementieren.

Das heißt, der Methodenaufruf: `10.next` liefert das Objekt `"11"` der Klasse `FixNum` zurück. In der Tabelle 1 auf der Nächsten Seite gibt es dazu einige Code-Beispiele.

1 - ausprobieren im interaktiven Ruby-Interpreter http://tryruby.hobix.com/(1.10.07)

Klasse	Eingabe	Ausgabe
Fixnum	1.next	2
	5.to_s	"5"
	1.class	Fixnum
Float	5.7.inspect	"5.7"
	5.7.to_i	5
	5.7.round	6
	7.5 / 3	2.5
String	"Hello".reverse	"olleH"
	"Ruby".size	4
	"Ruby".size.to_s	"4"
	"Ruby".type	String
	"R"+"u"+"b"+"y"	"Ruby"
	"a" * 5	"aaaaa"
Array	[6,3,2,5,4].size	5
	[6,3,2,5,4].sort.to_s	"23456"
	[4,2].empty?	false
NilClass	nil.object_id	4
Object	self.inspect	"main"
	self.object_id	67382520
Hash	{"Stadt"=>"Hamburg"}.size	1
	{"Stadt"=>"Hamburg"}.invert	{"Hamburg"=>"Stadt"}
Range	(1..10).last	10
	(`a`..`c`).to_a.inspect	["a","b","c"]

Tabelle 1: Ruby-Code-Beispiele

2.3.3 Typisierung

In Programmiersprachen wird die Typisierung nach Kriterien unterschieden. Das Hauptziel der Typisierung ist es, die Laufzeitfehler zu vermeiden.

Starke Typisierung

Starke Typisierung (strong typing) bezeichnet ein Schema der Typisierung von den Programmiersprachen. In der Literatur wird auch von der strengen Typisierung gesprochen. Bei dieser Typisierung bleibt die einmal durchgeführte Bindung des Datentypen an die Variable bestehen. Die Datentypen wie Ganz- und Gleitkommazahlen müssen voneinander unterschieden werden. Starke Typisierung bring den Vorteil, wenn Typkonvertierungen wie integer -> float explizit durchgeführt

werden müssen. Der Laufzeitaufwand, der eine solche Konvertierung darstellt, ist im Programm direkt zu sehen.

Eine Programmiersprache ist dann stark typisiert, wenn:

1. Datentypen an Variablennamen anstatt an diskreten Werten geknüpft sind.
2. Typüberprüfungen zur Compile-Zeit stattfinden.
3. implizite Typkonvertierungen verboten sind.
4. Typkonvertierungen explizit durchgeführt werden müssen.
5. die Sprache keine Mechanismen besitzt, um das Typ-System zu umgehen, etwa type casts (Typumwandlungen) in C.
6. es ein komplexes, fein abgestuftes System an Typen mit Sub-Typen gibt.
7. der Datentyp eines Objektes fix ist und sich während der gesamten Lebensdauer des Objektes nicht verändern kann.
8. das Typ-System das Laufzeitverhalten eines Programms garantieren kann.

Schwache Typisierung

Eine Programmiersprache heißt schwach typisiert (weak typing), wenn in einem Programm der Sprache einige Größen zu mehreren Datentypen gehören können. So können die Programmteile allgemein geschrieben werden und arbeiten dann z.B mit Fließkommazahlen und Integer problemlos. Das vereinfacht die Integration von verschiedenen Modulen einer Anwendung.

Einige Eigenschaften von schwach typisierten Programmiersprachen sind:

1. Datentypen sind an diskrete Werte gebunden.
2. Typüberprüfungen finden zur Laufzeit statt.
3. Implizite Typkonvertierungen sind erlaubt.
4. Das Typ-System kann das Laufzeitverhalten eines Programms nicht garantieren.
5. Datentyp einer Variablen kann sich zur Laufzeit ändern.

Statische und dynamische Typisierung

Es gibt zwei Möglichkeiten den Typ einer Variablen festzulegen. Die statische Typisierung (static typing) legt den Typ schon beim Erzeugen des Quellcodes vor dem Kompilieren fest, während die dynamische Typisierung (dynamic typing) geschieht erst zur Laufzeit, was aber gewisse Laufzeitnachteile mit sich bringt und die Fehlersuche erschwert.

Ruby Typisierung

Ruby ist stark, dynamisch typisiert. Dies erlaubt zum Beispiel, den Variablentyp

innerhalb eines Scopes nach der Deklaration zu ändern. In Java ist das nicht möglich. Der Vergleich der Programmiersprachen, bezogen auf deren Typisierung, ist in der Tabelle 2 dargestellt.

Typisierung	***schwach***	***stark***
statisch	C	C++,Java
dynamisch	PHP	**Ruby**

Tabelle 2: Vergleich der Typisierung

Duck Typing

Java ist im Gegensatz zu Ruby statisch typisiert. Somit muss vor der Bytecodeerzeugung eine Typüberprüfung durchgeführt werden. Es wird durch Interfaces sichergestellt, ob der Methodenaufruf auch dann erfolgen kann, wenn der Typ einer Variablen erst zur Laufzeit bekannt wird. In diesem Fall muss der Compiler prüfen, ob die betroffenen Klassen das Interface schon implementiert haben.

Ruby geht mit dieser Problematik anders um, indem sie einen Mechanismus namens „Duck Typing“ verwendet. Dieser Mechanismus erlaubt Ruby, sich mehr auf das Verhalten und die Möglichkeiten eines Objektes (sprich: Methoden) und nicht auf den Typ bzw. Klassen- oder Interfacenamen dieses Objektes zu verlassen. Folgendes Beispiel macht deutlich, dass es keine Rolle spielt, welche Daten-Typen der `connect`-Methode übergeben werden. Dazu muss lediglich die „+“-Methode implementiert werden.

```
class Assembly
      def connect (var1, var2)
            var1 + var2
      end
end
line = Assembly.new
line.connect(1,2)
-> 3
line.connect("The", " Ruby")
-> "The Ruby"
```

2.4 Iteratoren

In Ruby lassen sich die Iteratoren mit der Funktionalität nach Wünschen des Programmierers frei gestalten. Iteratoren, deren Verwendung sehr intuitiv ist, heben sich von der Implementierung in anderen Sprachen erheblich ab. Um einen Text mehrmals

auf dem Bildschirm anzeigen zu lassen, wird das Problem in meisten Sprachen in Teilprobleme, wie folgt, zerlegt:

- Anzeigen des Textes
- Wiederholung

Zweites Problem wird durch Realisierung eines Zählers gelöst. Dieser wird durch Parameter initialisiert, inkrementiert und kontrolliert. Beispielsweise in Java:

```
for(int i=0; i<=3; i++){
      System.out.println("I love this!");
}
```

Der Text „I love this"! wird 3 Mal ausgegeben. Wenn der Text irgendwo später im Programm 10 Mal in Erscheinung treten soll, muss dafür ein weiterer Zähler nach dem selben Muster initialisiert, inkrementiert und kontrolliert werden.

Ruby dagegen kümmert sich selbst um den Zähler, was den Programmieraufwand verringert. Der Entwickler entscheidet nun, was er mit dem Zähler macht.

```
3.times do
      puts "I love this!"
end

10.times do
      puts "It cannot be easier!"
end
```

Somit bleibt der Zähler vom Programmierer fast vollständig in der Methode „`times`" verborgen. Der Entwickler beschäftigt sich nicht mit dem Zähler, sondern konzentriert sich auf die eigentliche Aufgabe – wiederholte Ausgabe eines Textes. Im nächsten Beispiel wird die Handhabung einer Liste bzw. eines Arrays demonstriert. Eine Methode soll auf jedes Element der Liste angewandt werden. Dafür muss in Java die Länge des Arrays abgefragt werden, um dem Zähler mit richtigen Parametern aufbauen zu können. Danach wird das Element mittels eines Iterators ausgelesen, entsprechende Methode aufgerufen und der Zähler inkrementiert. Das passiert solange, bis der Zähler gleiche Größe wie die Arraylänge hat.

```
bookCounter = books.lenght;
for(i=0; i<bookCounter; i++){
      Book oneBook = books[i];
      oneBook.getTitel();
}
```

In Ruby wird für die selbe Aufgabe ein Iterator genannt. Die Anzahl der Durchläufe über die Liste und die Umgangsweise mit Listenelementen weiß Ruby zur Laufzeit selbst.

```
books.each do |oneBook|
      oneBook.getTitel
end
```

Das Schlüsselwort `each` teilt dem Array `books`, dass über alle Elemente iteriert werden soll. Das jeweilige Element wird im Iterator gehalten, die Variable `oneBook` beinhaltet bei jedem Durchlauf ein neues Element. Der Programmierer braucht nur die Methode `getTitel` anzuwenden, das Durchlaufen erledigt Ruby selbstständig.

2.5 Symbole

„Viele Ruby-Anfänger kämpfen damit, zu verstehen, was Symbole sind und wofür sie benutzt werden können. Symbole sind keine Strings und sind am einfachsten als Identitäten (IDs) zu verstehen. Bei einem Symbol ist wichtig, wer es ist, nicht was es ist“. [RLFU]

In allen Sprachen werden trotz der gleichen Bedeutung für zwei Zeichenketten „Hallo“ und „Hallo“ zwei unterschiedliche `String`-Objekte erzeugt. In Ruby ist es nicht anders. Die Symbole hingegen werden anders gehandhabt. Gleichnamige Symbole referenzieren auf das gleiche `Symbol`-Objekt. Sie Zeigen auf dasselbe Objekt im Speicher (sind identisch). Dieses Verfahren stellt nichts Neues dar und ist schon aus den anderen Programmiersprachen als „Referenzbildung“ bekannt.

Symbole werden mit einem Doppelpunkt vor einer Zeichenkette gekennzeichnet. Das Beispiel veranschaulicht die Handhabung der Symbole:

```
# das ist ein Ruby-Kommenter
# object_id gibt die ID-Nummer des Objektes aus
"Hallo".object_id -> 72909980
"Hallo".object_id -> 72909970

# das ist ein Ruby-Sybol
:hallo.object_id -> 233298
:hallo.object_id -> 233298
```

Symbole werden oft als Schlüsselwörter bei den assoziativen Arrays bei der

Parameterübergabe verwendet. Sie dienen zur Identifizierung der Parameter. Dadurch spielt die Reihenfolge der Parameter keine Rolle mehr, wobei die beiden Aufrufe das gleiche Ergebnis liefern:

```
connect(:param1 =>"Die Sohne",:param2 =>"scheint")
-> „Die Sohne scheint"

connect(:param2 =>"scheint",:param1 =>"Die Sohne")
-> „Die Sohne scheint"
```

Genauso können die Symbole auch für einmalig vorkommende Entitäten verwendet werden. Was in C oder Java mit einer Enumeration `enum` beschrieben wird, wird in Ruby mit Hilfe von Symbolen beschrieben. So wird sichergestellt, dass die Elemente tatsächlich singulär bleiben.

```
karten_farben = [:kreuz,:pik,:herz,:karo]
karten_bilder = [:ass,:koenig,:damme,:bubbe]
```

Das bring den Vorteil, dass der Platz im Hauptspeicher durch das einmalige Referenzieren eines Symbols gespart werden kann.

2.6 Erweiterbarkeit

Zu den großen Stärken von Ruby gehört seine Erweiterbarkeit. Wie das von den meisten Sprachen bekannt ist, lassen sich die bereits programmierten Klassen nicht mehr verändern. Durch Vererbung können diese Klassen in der abgeleiteten Klasse natürlich ergänzt werden. In Java müsste die Klasse `ArrayList` in eine Unterklasse `MyArrayList` abgeleitet und dort mit eigenen Methoden spezialisiert werden.

In Ruby sind alle Klassen jederzeit erweiterbar. Auch die Klassen aus der Ruby-Bibliothek sind davon nicht ausgeschlossen. Sogar der Zeitpunkt der Erweiterung steht offen. Eine Möglichkeit ist, die Klassen erst zur Laufzeit um Methoden oder Attribute zu erweitern. Ein Beispiel dazu ist auf der Nächsten dargestellt.

```
class Fixnum
      def seconds
            self
      end
      def minutes
            self * 60
      end
      def hours
            self * 60 * 60
      end
      def days
            self * 60 * 60 * 24
      end
end

puts Time.now
-> Tue Nov 13 15:00:10 +0000 2007
puts Time.now + 10.minutes
-> Tue Nov 13 15:10:10 +0000 2007
puts Time.now + 24.hours
-> Wed Nov 14 15:10:10 +0000 2007
puts Time.now – 7.days
-> Tue Nov 06 15:10:10 +0000 2007
```

Die Hilfsmethoden `seconds`, `minutes`, `hours`, `days` liefern ein entsprechendes Ergebnis ausgehend von einer Zahl, sprich der Aufruf `6.seconds` ergibt 6 Sekunden, `7.days` errechnet aus einer `7` - 7 Tage. Die Implementierung der Methoden ist recht anschaulich und unterscheiden sich von den anderen Programmiersprachen wenig. Entscheidend ist, dass in Ruby ein Datentyp, in dem Fall die Klasse `Fixnum`, um diese Methoden erweitert werden kann, währen in anderen Sprachen dafür extra Funktionen geschrieben werden müssen. Schließlich können so sehr intuitive Befehlskombinationen erstellt werden:

```
10.euro.to_dollar
-> 14.33
5.years.ago
lock_for_shop_in(1000.meter)
"book".many
->"books"
```

Weitere Flexibilität von Ruby bringen die so genannten „Singleton Methods“ ein. Das bedeutet, einzelne Objekt-Instanzen können um beliebige Methoden erweitert werden, die sie aber mit anderen Objekten derselben Klasse nicht teilen. Ein Beispiel dazu gibt es auf der Nächsten Seite.

```
class Hund
      def bellt
            puts "Wau Wau"
      end
end

grosserHund = Hund.new
kleinerHund = Hund.new
grosserHund.bellt
-> "Wau Wau"
kleinerHund.bellt
-> "Wau Wau"

def kleinerHund.bellt
      puts "Taf Taf"
end

kleinerHund.bellt
-> "Taf Taf"
grosserHund.bellt
-> "Wau Wau"
neuerHund = Hund.new
neuerHund.bellt
-> "Wau Wau"
```

2.7 Zusammenfassung

Ruby ist so ausgelegt, dass der Programmierer nicht mit der komplizierten Syntax konfrontiert wird, sondern sich auf die eigentlichen Aufgaben konzentriert. Der Programmierer wird von Ruby durch zahlreiche Methoden für gelöste Probleme unterstützt, die dazu noch dynamisch genutzt und erweitert werden können.

Durch seine einfache Syntax ähnelt Ruby der natürlichen Sprache sehr, was die Befehle wie `if waren_korb.empty?` oder `5.megabytes` sofort zu verstehen erlaubt.

Nicht zu vergessen bleibt aber, dass mit Ruby auch unlesbarer Code erzeugt werden kann. Die Konzepte und Ansätze sind am Anfang für einen Umsteiger von einer traditionellen Programmiersprache wie C++, Java oder PHP ungewohnt. Um die Vorteile der Sprache nutzen zu können, ist eine gewisse Einarbeitung notwendig.

3 Ruby on Rails

„Die Effiziente Web-Entwicklung."

3.1 Hintergrund

Am Anfang der Ära der Web-Entwicklung wurden einerseits Programmiersprachen wie Perl, Phyton, Java und Ruby, die aus der Zeit vor der Entwicklung des Web's stammen, eingesetzt. Andererseits entstanden völlig neue Technologien, die sich auf die Entwicklung der Webanwendungen spezialisierten. Die Vertreter dieser Sparta waren z.B PHP (Hypertext Preprocessor) und JSP (Java Server Pages). Die Entwicklung von großen, dynamischen Webanwendungen wurde durch eine hohe Anzahl an involvierten Sprachen immer aufwändiger und komplizierter. So musste den Entwicklern geholfen werden. Die Hilfe kam in Form von Web-Frameworks, die wiederkehrende Probleme lösen und das Ergebnis dem Entwickler durch die Bibliotheken und wiederverwendbaren Komponenten bereit stellen.

„Ein Framework (engl. für „Rahmenstruktur, Fachwerk") ist ein Programmiergerüst, welches in der Softwaretechnik insbesondere im Rahmen der objektorientierten Softwareentwicklung sowie bei komponentenbasierten Entwicklungsansätzen verwendet wird". [W06]-25.02.08

Ein Framework stellt den Rahmen zur Verfügung, innerhalb dessen der Programmierer eine Anwendung erstellt. Frameworks verwenden in der Regel Entwurfsmuster, durch

die auch die Struktur der individuellen Anwendung beeinflusst wird. Beispiel eines Entwurfsmusters ist MVC, wird im Weiteren erklärt. Ein Framework gibt somit in der Regel die Anwendungsarchitektur vor und definiert insbesondere den Kontrollfluss der Anwendung und die Schnittstellen für die konkreten Klassen, die vom Programmierer erstellt und registriert werden müssen.

„Für die Programmiersprache Ruby existierte lange kein entsprechendes Framework." [RRE]

Im Jahre 2004 wurde der erste Baustein für ein Ruby-Web-Framework gelegt. Ein dänischer Programmierer David Heinemeier Hansson stellte die erste Beta-Version von „Ruby on Rails" der Öffentlichkeit vor. Seitdem wurde diese Version in einer Gruppe von Entwicklern intensiv weiterentwickelt. Diese produktive Arbeit brachte im Dezember 2005 die erste Produktversion. „Ruby on Rails" (weiter auch Rails) ist nicht nur das erste Web-Framework für Ruby, sondern ist eine Sammlung von neuen Konzepten, Prinzipien und Modellen. Dieses Gesamtpaket stellt für die Web-Entwicklung einen wesentlichen Fortschritt dar. Allerdings steht „Rails", wie Ruby, unter einer OpenSource-Lizenz und liegt aktuell in der Version **Rails 2.0** vor.

Da diese Arbeit in ihrem Umfang begrenzt ist, gibt dieses Kapitel einen groben Überblick über Rails. Zum genaueren Kennenlernen bietet sich das Buch „Agile web development with Rails" [ADWR] an.

3.2 Prinzipien

Ruby on Rails wird von mehreren Prinzipien geprägt, die im Weiterem vorgestellt werden.

3.2.1 MVC

Das Framework basiert auf einer sauberen Model-View-Controller-Architektur (MVC). Wie der Name schon sagt gibt es in dem MVC-Entwurfsmuster drei Komponenten: ein Model, einen View und einen Controller. Die Abbildung 3 auf der nächsten Seite veranschaulicht das Zusammenspiel dieser Komponenten. Rails stellt für diese Komponenten jeweils eine unterstützende Komponente bereit. Domain-Objekte, so genannte Modelle, werden mit Hilfe der Sub-Frameworks „ActiveRecords" erzeugt, dessen Funktionsweise im Weiteren genau erklärt wird. Für Ihre Controller sind „ActionController" und für die Views „ActionView" verantwortlich. Die Projektstruktur belegt diese Trennung der Komponenten spürbar, da in einem Rails-Projekt für jede Komponente gleichnamige Ordner angelegt werden.

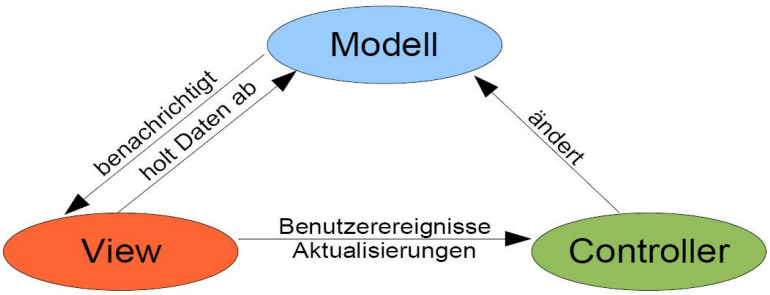


Abbildung 3: MVC-Entwurfsmuster [W12] - (22.03.08)

Die Ordnerstruktur einer Ruby on Rails Anwendung ist in der Abbildung 4 dargestellt.

Abbildung 4: Ordnerstruktur

Diese klare Trennung der Schichten verkörpert die Trennung der Verantwortlichkeiten und führt zu einer Verringerung der Abhängigkeiten im Code. Das kennzeichnet eine gut strukturierte und langfristig wartbare Anwendung.

3.2.2 DRY

Ein weiteres Prinzip, das eine große Rolle in Rails spielt, ist „Don't repeat yourself“ kurz DRY-Prinzip. Dieses besagt – Wissen hat in einem System nur eine einzige und eindeutige Repräsentation. Weder Daten noch Funktionalität sollten redundant vorkommen, da es sonst zum erhöhten Wartungsaufwand kommt. In Rails muss ein Sachverhalt einmal genau beschrieben werden. Dieser Sachverhalt ist damit festgeschrieben und für die gesamte Webanwendung bereits erklärt. Rails setzt das DRY-Prinzip in allen Bereichen konsequent um und weiß jede Art von Redundanz zu verhindern. Beispielsweise müssen für eine Datenbanktabelle keine korrespondierende Attribute und keine Getter-, Setter-Methoden in ihrem Domain-Objekt definiert werden. Das übernimmt für den Entwickler Rails und erzeugt sie automatisch.

3.2.3 Convention over Configuration

Das Prinzip verfolgt das Ziel, die Anzahl der notwendigen Konfigurationen mittels einer Verabredung - „Convention“ zu verringern. Durch einfache Namenskonvention kann eine Verbindung zwischen einer Klasse und einer Datenbanktabelle hergestellt werden. Dafür erhält die Datenbanktabelle den Pluralnamen des dazugehörigen Domain-Objekts. Ein weiteres Beispiel: der Name einer Methode, die einen Request verarbeitet, wird aus der URL des eingehenden HTTP-Requests ermittelt. Das funktioniert ohne Konfiguration solange der Entwickler die gewünschte „Convention“ einhält. Das Einhalten der Konventionen unterstützt produktive und effiziente Programmierung. Anderenfalls wird jedes Abweichen von der Konvention mit zusätzlichem Programmieraufwand bestraft.

Nicht desto trotzt ermöglicht Rails dem Entwickler auch eine eigene „Configuration“ vozurnehmen. Spätestens im Produktvertrieb gibt es den Wunsch, die Zugangsdaten zur Datenbank zu konfigurieren. Die Tatsache, dass Konventionen im Hintergrund völlig unsichtbar für den Entwickler arbeiten, kann als ein Kritikpunkt angesehen werden. Das kann beim Einstieg in Rails zu einigen Schwierigkeiten führen. Deshalb müssen diese Abläufe einmal verstanden und angewandt werden.

3.2.4 Agilität

*„**Agile Softwareentwicklung** ist der Oberbegriff für den Einsatz von Agilität (lat. agilis 'flink, beweglich') in der Softwareentwicklung. Je nach Kontext bezieht sich der Begriff auf Teilbereiche der Softwareentwicklung – wie im Fall von Agile Modeling – oder auf den gesamten Softwareentwicklungsprozess – exemplarisch ist Extreme Programming anzuführen. Agile Softwareentwicklung zeichnet sich durch den geringen bürokratischen Aufwand und wenige, flexible Regeln aus“*.[W04]-10.01.08

Nach Agiler Softwareentwicklung wird der Softwareentwicklungsprozess flexibler und schlanker gestaltet, als das bei den klassischen Vorgehensmodellen der Fall ist. Der Fokus wird auf die zu erreichenden Ziele und auf die technischen und sozialen Probleme bei der Softwareentwicklung gelegt. Die Agile Softwareentwicklung stellt eine Gegenbewegung zu den oft als schwergewichtig und bürokratisch angesehenen Softwareentwicklungsprozessen, wie dem „Rational Unified Process“ oder dem „V-Modell“ dar. Die agile Prinzipien bilden Handlungsgrundsätze bei der Agilen Softwareentwicklung. Es sind keine direkt umsetzbare Verfahren, sondern Grundsätze, die auf Methoden, die ein Prozess bilden sollen, angewendet werden können. Ein agiles Prinzip ist also ein Leitsatz für die agile Arbeit. Beispiele für agile Prinzipien:

- vorhandene Ressourcen mehrfach verwenden
- einfach (KISS-Prinzip)
- zweckmäßig

- kundennah
- gemeinsamer Code-Besitz (Collective Code Ownership)

Diese Prinzipien werden mittels agiler Methoden umgesetzt. Das sind konkrete Verfahren während der Softwareentwicklung, die sich auf die agilen Prinzipien stützen. Der Übergang zwischen Prinzipien und Methoden ist fließend. Die Methoden beabsichtigen die Aufwandskurve bei den Entwicklungsprozessen möglichst flach zu halten. Beispiele für agile Methoden:

- Paarprogrammierung
- testgetriebene Entwicklung
- ständige Refaktorisierungen
- Story-Cards
- schnelle Codereviews

In Rails spiegeln sich viele dieser Methoden und Prinzipien wieder. Einfache Testbarkeit wird durch Verwendung von Unit Tests erreicht, was die testgetriebene Softwareentwicklung ermöglicht. Kein redundanter Code entspricht dem KISS-Prinzip und erleichtert die Refaktorisierung. Wenige Konfigurationen führen zu den kurzen und schnellen Entwicklungszyklen, was das unmittelbare Feedback dem Entwickler und dem Kunden liefert. Daher ist Rails agil.

3.2.5 Scaffolding

Mit Hilfe von Scaffolding (engl.: Gerüstbau) wird eine einfache aber lauffähige Version der entwickelten Anwendung erstellt. Scaffolding erzeugt basierend auf dem Datenbankschema, entsprechende Views und Controller, die die Erzeugung, Anzeige, Verarbeitung und Speicherung von Modellen ermöglichen. Die Anwendung kann anschließend um individuelle Funktionalität erweitert werden, dabei bleibt sie jederzeit voll funktionsfähig und ausführbar.

3.2.6 Migrations

In Rails werden die Modellattribute direkt in der Datenbank definiert. Das macht jede Modelländerung problematisch. Abhilfe schaffen die so genannten „Migrations“. Das sind Ruby-Scripte, die Operationen in der Datenbank ausführen können und vollen Zugriff auf den Datenbestand haben. Migrationen können sowohl die Struktur als auch Daten selbst ändern. Somit kann ein Modell in Rails definiert werden und anschließend mit dem Befehl `rake db:migrate` auf die Datenbank abgebildet werden. Dabei werden sämtliche Datenbanktabellen, die für das Modell nötig sind, automatisch erzeugt. Vorab muss das entsprechende Datenbankschema existieren. Das kann sowohl

auf gewöhnliche Weise mit SQL wie auch mit dem Befehl `rake db:create` erstellt werden.

3.3 Komponenten

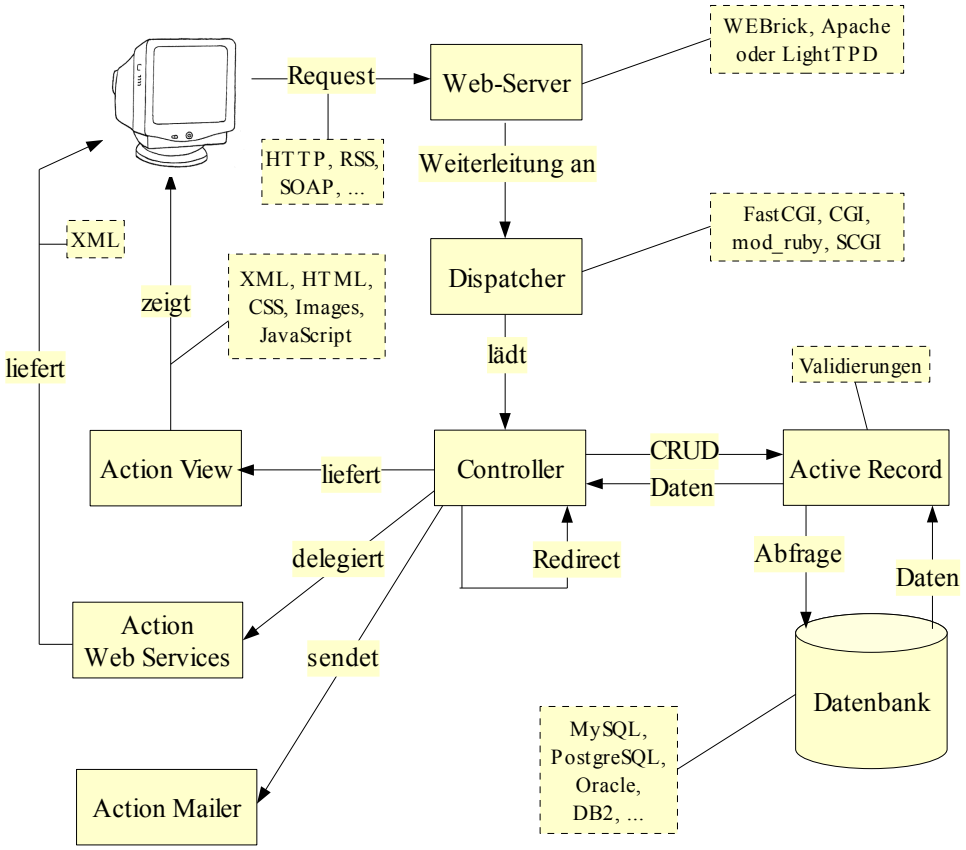


Abbildung 5: Bestandteile und Zusammenspiel (angelehnt an [RDRR]*)*

Die Abbildung 5 zeigt die Komponenten von Rails. In diesem Kapitel wird das Zusammenspiel einiger Komponenten beschrieben. Der Client sendet einen Request, der vom HTTP-Server entgegengenommen wird, wie das schon im Kapitel 1.4 beschrieben wurde. Der Server leitet den Aufruf an den „Dispatcher“ weiter. Der Dispatcher ist ein Ruby-Programm, welches in jedem Ruby-Projekt enthalten ist. Dieses Programm delegiert die Anfrage an den zuständigen „Controller“. Jeder Controller basiert auf dem Sub-Framework „ActionController“, welches typischerweise für die Aktionen wie Erzeugen oder Lesen auf einem Domain-Objekt zuständig ist. Dieses verwendet in seiner Arbeit ein weiteres Ruby Sub-Framework „ActiveRecord“, welches u.a die Verbindung zu den Datenbanktabellen bereitstellt. Dieser Mechanismus wird im folgenden Kapitel erläutert. Das Domain-Objekt validiert ggf. auch die übergebenen Parameter und liefert entsprechende Fehlermeldungen, wenn es zu diesen kommt.

In Anschluss führt der Controller entweder eine Weiterleitung auf einen anderen Controller aus oder beginnt mit der Auslieferung der Antwort. In der Regel besteht diese Antwort aus einem HTML View, der mittels „ActionView“ aus dem HTML-Template erzeugt wurde. Wenn es sich bei diesem Request um eine Web-Service-Anfrage handelt, dann wird diese von „Action Web Service“ durch ein XML-Dokument beantwortet.

3.3.1 Action Pack

„ActionPack“ stellt eine Kombination der Rails-Frameworks „ActionController“ und „ActionView“ dar. Dabei übernimmt der „ActionController“ den Teil des Controllers, der den Request entgegen nimmt und einen View als Response liefert. Somit wird die Logik der Verarbeitung kontrolliert und es steht für das C im MVC-Muster. Für die Repräsentation der Daten ist dann „ActionView“ zuständig und steht für das V im MVC. „ActionView“ erzeugt die Views, die über Template-Dateien definiert werden. Diese Template-Dateien enthalten neben HTML auch den eingebetteten Ruby-Code. Dem zur Folge hatten diese Dateien die Erweiterung „rhtml“. Ab Rails 2.0 lautet die Erweiterung „html.erb“. Komplexere Logik wird in Hilfsmodule ausgelagert, da sie nicht zu den Aufgaben des Views gehört. Rails bietet jede Menge von solchen Modulen bereits an.

Es ist möglich mittels „Partial Views“ die Seiten aus Teil-Views zusammenzustellen. Die Grundstruktur und das Aussehen der Seite werden durch die Verwendung von Layouts in den zentralen Views definiert, was sich als sehr praktisch erweist. Die Funktionalität von Controllern und Views kann über „Components“ in anderen Controllern wieder verwendet werden. An der Stelle kommt das agile Prinzip „Wiederverwendbarkeit“ zur Geltung.

Der Prozess der Verarbeitung eines Requests durch einen „ActionController“ und die Erzeugung eines Views wird in Rails als „Action“ bezeichnet. Eine Action wird als öffentliche Methode in einer Controller-Klasse implementiert. Beispielweise ein `UserController` stellt Methoden wie z.B. `list()`, `new()` oder `edit()` als Actions zur Verfügung. Üblicherweise werden die Domain-Objekte (z.B. ein Benutzer) durch Actions erstellt, gelesen, aktualisiert und gelöscht. Eine Action liefert als Ergebnis einen View oder führt eine Weiterleitung auf eine andere Action aus. Zum Beispiel beim Aufruf der URL: `http://localhost:3000/user/show/5` leitet Rails die Anfrage zu dem zugehörigen Controller namens `UserController` und ruft die Methode `show()` auf. Als Parameter wir die Zahl 5 übergeben.

```
Class UserController < ApplicationController
      def show
            @user = User.find(params[:id])
      end
end
```

Der gewünschte Benutzer mit der ID 5 wird gesucht und in der Variable `@user` gespeichert. Danach wird ein Action-View gerendert. Dieser wird anhand des Controllernamens und der aufgerufenen Aktion identifiziert. In diesem Fall wird aus dem Ordner namens views/user die Datei `show.html.erb` geladen.

Unter anderem wird mit Hilfe eines Controllers ein Zugriff auf die Request- und Session-Parametern auf einfache Weise möglich. Die Anforderung, ob der Anwender angemeldet ist, dass vor jeder Action geprüft werden muss, kann durch Filtern eines Request flexibel erfüllt werden. Es ist auch sehr hilfreich durch den Einsatz eines Zwischenspeichers die Meldungen zwischen zwei Requests elegant auszutauschen.

3.4 Active Record

In diesem Abschnitt wird die Rails-Komponente „ActiveRecord“ vorgestellt. Diese Komponente gehört zu dem eigentlichen Kern dieser Arbeit. Deshalb verdient sie eine detailliertere Beschreibung als die schon vorgestellten Komponenten. Beginnend mit der Motivation, wird die Funktionsweise von „ActiveRecord“ beschrieben und anhand der Beispiele genau erläutert. Abschließend wird in der Zusammenfassung das Fazit zu dieser Komponente gezogen.

3.4.1 Motivation

In der aktuellen Softwarewelt werden die Daten zum größten Teil durch Objekte repräsentiert. Um diese Daten persistent zu halten, kommen in meisten Fällen jedoch relationale Datenbanksysteme zum Einsatz. Die Verbindung dieser zwei unterschiedlichen Konzepte ist mit einigen Schwierigkeiten verbunden. Denn es gibt keinen trivialen Weg die Daten aus einer relationalen Datenbank, z.B ganze Zeile oder einzelne Felder in eigenständige Objekte zu laden und auf dem umgekehrten Wege zu speichern.

Als Erstes bietet sich SQL zum Lösen dieses Problems an. Für eine Anwendung mit einer geringen Komplexität, bei der sich die Tabellen- und Objektmenge noch in Rahmen hält, mag es sinnvoll sein. Bei komplexeren Datenmodellen muss ein anderer Lösungsweg gefunden werden. Ein Vorschlag ist der ORM-Ansatz.

3.4.2 ORM

ORM steht für „Object Relational Mapping“. Nach diesem Ansatz werden die Daten zwischen Applikation und Datenbank schlüssig und fehlerfrei transferiert. Die gespeicherten Informationen werden aus Datenbanken gelesen, um die Objekte daraus zu erzeugen, die in der Anwendung benutzt werden können. Das geschieht in der Regel über die „Object Relational Wrapper“ (weiter ORM). Falls neue Instanzen in der Anwendung erzeugt werden, müssen diese von ORM in entsprechende Datenbankeinträge umgewandelt werden.

Eine transparente Implementierung von ORM entscheidet über die Einfachheit ihrer Anwendbarkeit in der Applikation. Es wird angestrebt, am Wenigsten unterscheiden zu müssen zwischen Objekten und Datenbankeinträgen. Dadurch soll es dem Entwickler ermöglicht werden, sich von der Datenbankebene zu abstrahieren, sich nur in der Modellebene komfortabel zu bewegen und dabei komplett auf SQL zu verzichten. Um das anbieten zu können, benötigen viele ORM eine aufwändige Beschreibung der Beziehungen zwischen Datenbankeinträgen und Objekten. Durch diese Beschreibung findet sich für jedes Objektattribut ein Äquivalent in der Datenbanktabelle. Das ist, neben der Klassendefinition und der Tabellenstruktur, schon die dritte Beschreibung des Modells. Das führt dazu, dass Änderungen an mehreren Stellen durchgeführt werden müssen.

In Rails ist der ORM-Ansatz hinter dem Namen Active Records verborgen. Der einzige Unterschied zu den anderen ORM-Implementierungen zeigt sich in der Konfiguration. Durch die Namenskonvention ist fast keine Konfiguration notwendig.

3.4.3 Active Record – das Pattern

Das ActiveRecord ist ein Sub-Framework von Rails. Das Framework stellt die Verbindung zwischen Domain-Objekten und Datenbank her. Wichtig dabei ist, dass die Objekte **komfortabel** in der DB abgespeichert werden.

*„Dem ActiveRecord-Framework liegt das gleichnamige Pattern **Active Record** zugrunde, das von Martin Fowler beschrieben wurde. Zentrale Idee von ActiveRecord ist die Verwendung von Klassen zur Repräsentation einer Datenbanktabelle. Eine ActiveRecord Klasse korrespondiert dabei mit genau einer Datenbanktabelle. Eine Instanz einer ActiveRecord Klasse repräsentiert genau eine Datenzeile in dieser Tabelle. Die Felder einer ActiveRecord Klasse müssen 1:1 mit den Feldern der zu gehörigen Tabelle korrespondieren. Während in Java dafür entsprechende Attribute mit Gettern und Settern programmiert werden (DRY-Verletzung), werden die Attribute von Rails ActiveRecord ausschließlich in der Tabelle definiert (DRY-Einhaltung). Dank Ruby ist es möglich, ActiveRecord Klassen zur Laufzeit um Attribute und Zugriffsmethoden für die zugehörigen Tabellenfelder zu erweitern“.*[RDRR]

Mit anderen Worten eine ActiveRecord Klasse entspricht dabei einer Datenbanktabelle

und ein Objekt dieser Klasse einer Datenzeile in dieser Tabelle. Jedes Objekt beinhaltet dabei einige für sich passende Persistenzroutinen und die Geschäftslogik. Die Abbildung 6 zeigt vereinfacht die Zusammenhänge.

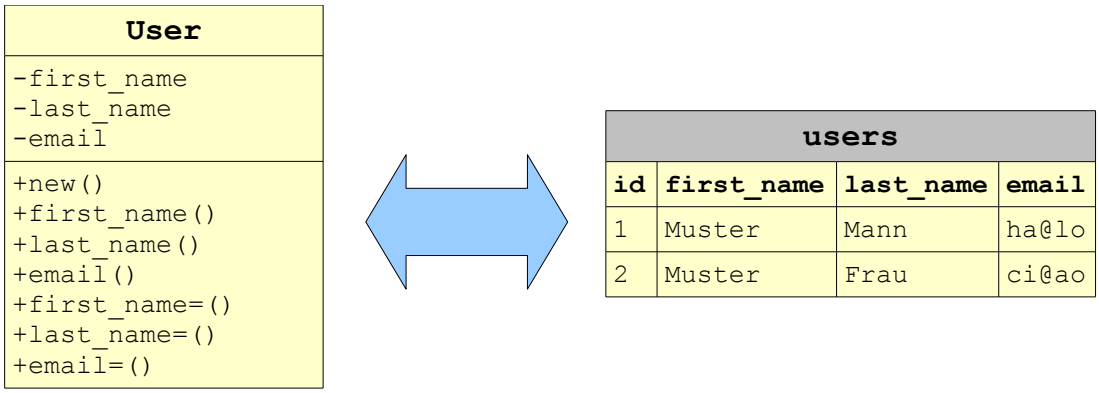


Abbildung 6: Vom Domain-Objekt zu Datenbanktabelle

3.4.4 Namenskonvention

Per Konvention erwartet ActiveRecord die Pluralform als Tabellennamen des zugehörigen Klassennamens. Das erleichtert die Zuordnung des Namens einer Klasse zu der Datenbanktabelle. Beispielsweise die Klasse „User“ in der Anwendung wird groß geschrieben, bekommt den Tabellennamen „users“ in der Datenbank. Zu beachten ist, dass der Name klein geschrieben wird. Momentan unterstützt Rails nur die englische Grammatik bei der Pluralisierung des Klassennamens. Sogar die unregelmäßige Formen werden abgeleitet, wie zum Beispiel „Person“ -> „people“, weitere Verben können hinzugefügt werden. Das Fehlen der deutschen Grammatik kann negativ angesehen werden, was aber zu keinen Einbußen in der Funktionalität führt. „Benutzer“ bekommt trotzt seinen deutschen Namen eine Datenbanktabelle „benutzers“. Dieses Verhalten kann in solchen Fällen unterbunden werden. Dazu wird die Datei `config\enviromment.rb` um folgende Option ergänzt:

```
ActiveRecord::Base.pluralize_table_names = false
```

Manchmal ist die Gleichheit der Namen zwischen Klasse und Tabelle eher irreführend als hilfreich. Dafür bietet Rails die Möglichkeit, den Tabellennamen einer ActiveRecord Klasse explizit anzugeben. Ein Beispiel dazu befindet sich auf der nächsten Seite.

```
class User < ActiveRecord::base
      set_table_name "mamamia" #beliebiger Tabellenname
end
```

3.4.5 CRUD

Jede ActiveRecord Klasse bieten von Anfang an die Kernfunktionalität eines ORM-basierten Frameworks, die durch den Akronym CRUD beschrieben wird. Das sind die Anfangsbuchstaben von ***c****reate,* ***r****ead,* ***u****pdate und* ***d****elete.* Die beschreiben das Erzeugen, Laden, Aktualisieren und Löschen von ActiveRecord Objekten. Diese Funktionalitäten werden von der Basisklasse geerbt:

```
ActiveRecord::Base
```

Um diese Kernfunktionalitäten zu testen wird zuerst eine Active Record-Klasse erzeugt:

```
class User < ActiveRecords::Base
end
```

Zu der Klasse wird eine Datenbanktabelle benötigt. Es gibt zwei Möglichkeiten so eine Tabelle zu erstellen. Die eine ist über eine Datenbankmanagement-Software oder mit Hilfe einer Migration. Ein Migrationsscript könnte dann so aussehen:

```
class CreateUsers < ActiveRecord::Migration
      def self.up
            create_table :users do | t |
                  t.string :vorname
                  t.string :nachname
            end
      end
      def self.down
            drop_table :users
      end
end
```

Create

Der Default-Konstruktor `new` erzeugt ein Active Record Objekt:

```
user = User.new
user.vorname = "Hans"
```

Zusätzlich verfügt jede Active Record Klasse über einen Konstruktor, der als Parameter eine Hash mit Attributwerten erwartet:

```
user = User.new(:vorname => "Hans")
```

Dieser Hash-basierte Konstruktor eignet sich für direkte Erzeugung der Modell-Objekten anhand von Request-Parametern:

```
user = User.new(params[:user])
```

Das erzeugte Objekt befindet sich erstmal im Speicher. Um die Objekt in die Datenbank zu speichern, muss die Methode `save()` aufgerufen werden:

```
user = User.new(:vorname => "Hans")
user.save
```

Nach diesem Aufruf wird eine neue Zeile in die Tabelle `users` geschrieben und `id` des Objektes bekommt außerdem seinen Wert automatisch aus der Datenbank. Active Record bietet auch die Möglichkeit, die Objekte in einem Schritt zu erzeugen und anschließend zu speichern. Dafür wird die Methode `create()`, die eine Hash mit Attributwerten erwaret, aufgerufen. Die `create()` - Methode hat eine weitere nützliche Eigenschaft – ihr können mehrere Datensätze übergeben werden:

```
user = User.create([
{     :vorname => „Hans"
      :nachname => „Meyer"},
{     :vorname => „Cruso"
      :nachname => „Robinson"},
{     :vorname => „Gustav"
      :nachname => „Olafsson"}])
```

Read

Das Laden der ActiveRecord – Instanzen wird von der statischen Methode `find()` übernommen. Diese Methode ist vielfältig parametrisierbar. Das Objekt kann am einfachsten über seine ID geladen werden:

```
user = User.find(1)
```

Wenn das gesuchte Objekt in der Datenbank nicht gefunden wird, wirft Rails eine `ActiveRecord::RecordNotFound` – Exception. Die ID-basierte Suche funktioniert auch mit Listen oder Arrays von IDs:

```
user_list = User.find(1,2)
user_array = User.find([1,2])
```

In beiden Fällen ist der Rückgabewert ein Array mit Objekten. Sollte es dazu kommen, dass mindestens eine ID nicht existiert, wird ebenfalls eine `ActiveRecord::RecordNotFound` – Exception geworfen. Üblicherweise wird nicht nur nach dem Primärschlüssel gesucht, sondern nach bestimmten Kriterien. Die Methode `find()` kann um die Anzahlt der zurückgelieferten Werte und um eine Bedingung (`:conditions` später mehr dazu) erweitert werden. Wenn alle Datensätze

aus der Tabelle `users` benötigt werden, kommt das Symbol `:all` als Parameter zum Einsatz:

```
all_users = User.find(:all)
```

In der Funktionsweise ist es nichts anderes als ein SQL-Statement:

```
select * from users.
```

Das Ergebnis des Methodenaufrufs ist ein Array aller `User`-Objekte. Hingegen liefert der Aufruf:

```
any_user = User.find(:first)
```

nur das erste User-Objekt, was mit reinem SQL unmöglich ist. Eine weitere Active Record-Methode ist `reload()`. Die lädt die Attribute eines Objektes frisch aus der Datenbank. Mit dieser Methode kann sichergestellt werden, dass die Modellattribute korrekt gespeichert werden:

```
usermanagement.users << User.new
usermanegement.reload
assert_equal 1, usermanagement.users.length
```

Update

In der Datenbank gespeicherte Objekte werden durch den Aufruf der schon bekannten Methode `save()` aktualisiert. Die Methode liefert einen Booleschen Wert zurück, der anzeigt, ob das Speichern geklappt hat. Es gibt eine zweite Variante dieser Methode `save!()`, die eine Exeption, falls das Speichern misslungen ist, wirft . Häufig kommt es zu diesem Fehler durch eine fehlgeschlagene Validierung. Zum Beispiel, das Attribut `name` darf nicht leer sein (`validates_presenve_of :name`) dann liefert die erste Variante `false`:

```
@user = User.new
puts user.save
$ false
```

Die Methode `save!()` liefert hingegen eine `ActiveRecord::RecordInvalid`-Exception:

```
@user = User.new
@user.save!
$ ActiveRecord::RecordInvalid
/lib/ruby/gems/1.8/gems/activerecord-
1.10.1/lib/active_record/validations.rb:132:in‘save!’
```

Weiter bietet Active Record zwei Methoden zum Speichern und Aktualisieren in einem Schritt. Die Methode `update_attribute()` benötigt den Namen des Attributs und den zugehörigen Wert. Ein Beispiel dazu ist auf der nächsten Seite.

```
@user.update_attribute("name","Schmidt")
person = User.find(@user.id)
assert_equal"Schmidt", person.name
```

Die `update_attributes()`- Methode erwartet eine Hash mit den zu aktualisierenden Attributwerten:

```
User.update_attributes (:name => "Schmidt",
                        :vorname => "Sven")
```

Auf Grund der Möglichkeit, eine Hash als Parameter zu übergeben, eignet sich diese Methode besonders für das Aktualisieren von Modellinstanzen auf Basis der Request-Parameter eines vorausgehenden HTTP-Requests:

```
User.update_attributes (params[:user])
```

Für das gleichzeitige Laden und Aktualisieren wird von ActiveRecord die Klassenmethode `update()` angeboten. Es werden die ID des zu aktualisierenden Objekts und eine Hash mit Attributwerten erwartet:

```
user = User.update (1, :name => "Frost")
```

`Update()` unterscheidet sich von `save()` darin, dass sie das aktualisierte Objekt zurück liefert und nicht den Booleschen Wert. Eine weitere Klassenmethode ist `udate_all()`. Die wird für die Aktualisierung mehrerer Objekte in einem Schritt benutzt. Als Parameter wird das zu aktualisierende Attribut mit dem Wert erwartet:

```
User.update_all ("name = Schmidt")
```

Die Menge der Objekte kann optional über Bedingungen eingeschränkt werden.

```
User.update_all ("status = moderator", "name = 'Schmidt'")
```

Dabei liegt folgendes SQL-Satement zu Grunde:

```
update set users status = moderator where name = 'Schmidt'
```

Delete

ActiveRecord verfügt über eine Reihe von Methoden, die fürs Löschen der Modellinstanzen verantwortlich sind, wobei sie ein unterschiedliches Verhalten vorweisen. Die erste Löschmethode aus der Palette ist `delete()`, anhand einer ID oder einer Liste von ID's löscht sie die Datensätze aus der Datenbank:

```
User.delete(1)

User.delete([1,3,5])
```

Analog zu der `update_all()` erwartet `delete_all()` eine Bedingung, die die zu löschenden Datensätze spezifiziert:

```
User.delete_all ("status = moderator" and "name = 'Schmidt'")
```

Zwei weitere Methoden sind `destroy()` und `destroy_all()`. Die arbeiten sehr

ähnlich wie die Delete-Methoden. Mit dem Unterschied, dass die `destroy()` auch als eine parameterlose Instanzmethode aufgerufen werden kann:

```
user = User.find(1)
user.destroy
```

Zu beachten dabei ist, dass neben den Löschoperationen noch die ActiveRecord-Callbacks ausgeführt werden. Das sind Methoden, die vor bzw. nach der eigentlichen Operation zwischengeschaltet werden. Zum Beispiel der Callback before_destroy() wird vor dem Löschen eines Objekts aufgerufen. Praktischerweise können die Callbacks für den Zweck benutzt werden eventuell noch vorhandene Abhängigkeiten zu prüfen und dementsprechend die Löschoperation abzubrechen. Dieser Mechanismus ist aus der Datenbankwelt als „Trigger“ bekannt.

3.4.6 Finder

Das Thema der Objektsuche ist von großer Bedeutung auf Grund seiner Wichtigkeit. Das Aufspüren von Objekten über ihre IDs wurde bereits kurz vorgestellt. Problematisch wird es, wenn die ID des gewünschten Objekts unbekannt ist. Für diesen Fall muss es möglich sein, das Objekt auf eine andere Weise zu finden. ActiveRecord bietet dazu eine Reihe von Optionen für die `find()` - Methode.

:conditions

Dieser Parameter dient zur Angabe einer Suchbedingung und akzeptiert als solches ein Array oder ein String. Die einfachste Variante für die Angabe einer Suchbedingung könnte so aussehen:

```
User.find(:first, :conditions => "name = 'Schmidt'")
```

Für anspruchsvollere Suchabfragen soll ein Array der Suchbedingungen verwendet werden. Beispielsweise kann eine Liste aller Benutzer, die sich vor dem 1.1.2008 registriert haben, wie folgt gefunden werden:

```
User.find(:all, :conditions => ["registred < ?", "2008-01-01"])
```

Da die Suchbedingungen beliebig kombiniert werden können, sieht eine Anfrage nach allen Benutzern im Alter zwischen 20 und 40 Jahren so aus:

```
User.find(:all, :conditions =>
      ["born => ? and born <= ?",
       "1968-01-01", "1988-01-01"])
```

Die Verwendung vieler Parameter in den Suchbedingungen wirkt sich negativ auf deren Übersichtlichkeit aus. Demzufolge bietet ActiveRecord die Verwendung von Symbolen an. Wie das geht, ist auf der nächsten Seite zu sehen.

```
User.find(:all, :conditions =>
      ["born >= :between and born <= :and",
       {:between >= "1968-01-01", :and => "1988-01-01"}])
```

:order

Wie der Name erahnen lässt, geht es bei diesem Parameter um sortierte Reihenfolge von Suchergebnissen. Mit diesem Parameter kann bestimmt werden, wonach es sortiert werden soll. Zusätzlich kann die Art der Sortierung (absteigend, aufsteigend) angegeben werden. Folgendes Beispiel veranschaulicht dieses Verhalten:

```
User.find(:all, :conditions =>["name = ?", "Schulz"],
                :order => "first_name asc, born desc")
```

Es werden alle Benutzer gefunden, die mit dem Nachnamen „Schulz“ heißen. Die werden nach Vornamen sortiert, wobei die Schlüsselwörter `asc` und `desc` die Sortierrichtung vorgeben.

:limit und :offset

Der `:limit`-Parameter beschränkt die Anzahl der zurückgelieferten Ergebnisse und der `:offset`-Parameter erlaubt in der Ergebnissmenge zu blättern bzw. die Ergebnisse seitenweise anzeigen zu lassen:

```
alle_messages = Message.find(
      :all,
      :limit => 5, # 5 Datensätze werden zurückgegeben
      :offset => 20, # Überspringe die ersten 20 Nachrichten
      :order => "id desc", # Datensätze werden nach der ID
                           # absteigend sortiert
      :conditions => ["author = ?", "Schulz"])
```

:joins

In den relationalen Datenbanken sind die Daten meist über mehrere Tabellen verstreut. Um spezielle Anfragen erfüllen zu können, müssen die Informationen aus verschiedenen Tabellen zusammengesucht werden. Für diese Fälle bietet ActiveRecord den `:joins`-Parameter. Als Beispiel dazu sollen alle Forum-Moderatoren gefunden werden, die eine Nachricht mit dem Betreff erstellt haben, in dem das Wort „Ruby“ vorkommt. Die Nachrichten werden in der Tabelle `messages` und die Moderatoren in der Tabelle `users` gespeichert. Das Beispiel dazu ist auf der nächsten Seite dargestellt.

```
ruby_moderators = User.find(
      :all,
      :conditions => "me.subject like '%Ruby%'",
      :joins => "as us inner join messages as me on" +
                "us.id = me.moderator_id")
```

Das Beispiel scheint konstruiert zu sein. Der Grund dafür ist, dass ActiveRecord die Relationen auf der Objekt-Ebene abbildet, sodass der `:joins`-Parameter selten benötigt wird.

3.4.7 Dynamische Finder

Das ist ein interessantes Konstrukt zum Auffinden von Objekten, welches ActiveRecord alternativ zum vorher behandelten Verfahren anbietet. Dynamische Finder sind Attributt-basiert, bei der Verwendung bedeutet das folgendes:

```
User.find_by_name("Schulz")
```

Der Aufruf ist selbsterklärend und liefert das gleiche Ergebnis wie:

```
User.find(:first, :conditions => ["name = ?,"Schulz"])
```

Wenn alle Objekte gesucht werden, fangen dynamische Finder mit `find_all_by` an. Gefolgt von einem oder mehreren Attributen, die mit Hilfe der logischen Operatoren beliebig verknüpft werden können:

```
User.find_all_by_name_and_city("Schulz", "Hamburg")
```

ActiveRecord erzeugt dynamische Finder bei ihrer ersten Benutzung. Nach dem Aufruf der Methode wird diese von ActiveRecord in einen gewöhnlichen `find()`-Aufruf konvertiert. Falls die Methode nicht generiert werden kann, wirft ActiveRecord eine `NoMethodError`-Exeption.

find_by_sql()

Eigentlich werden die SQL-Details in der find()-Methode von ActiveRecord nur gekapselt. Warum sollen die Finder überhaupt benutzt werden? In Anbetracht der Performance bringen sie auch keine Vorteile im Vergleich zum normalen SQL. Zumal kann der Umstieg sehr schwierig werden. Dennoch können sie bei einfachen Anfragen verwendet werden.

Zusätzlich bietet ActiveRecord die Möglichkeit an, SQL weiterhin direkt zu benutzen, was sich gerade bei komplexen oder performancekritischen Anweisungen auszahlt. Mit Hilfe der statischen Methode `find_by_sql` wird ein SQL-Befehl direkt an die Datenbank abgesetzt.

Folgender Aufruf gibt ein Array mit allen gespeicherten Benutzern zurück:

```
User.find_by_sql("select * from users")
```

Auf einer Seite bringt ActiveRecord interessante Suchmechanismen mit, die auf den ersten Blick durch ihre Einfachheit bestechen. Auf der anderen Seite wird ein visierter SQL-Experte wahrscheinlich bei der SQL-Variante bleiben.

3.4.8 Assoziationen

Die meisten Anwendungen basieren auf einem Datenmodell, welches aus mehreren Tabellen besteht, die außerdem miteinander in Beziehung stehen. Auf der Datenbankebene lassen sich solche Beziehungen durch Definition von Fremdschlüsseln abbilden. ActiveRecord ermöglicht die Modellierung dieser Beziehungen auf der Modellebene. Das Framework wandelt die Datenbank-Fremdschlüssel (*"low-level"*) in Objektbeziehungen (*"high-level"*) um. Mit Hilfe spezieller Deklarationen lassen sich mit ActiveRecord drei Assoziationstypen abbilden (Auszug aus [RRE]):

1. ***1:1 – Beziehung****: Ein Datensatz einer Tabelle steht mit genau einem Datensatz einer anderen Tabelle in Beziehung.*
2. ***1:N - Beziehung****: Ein Datensatz einer Tabelle steht mit mehreren Datensätzen einer anderen Tabelle in Beziehung, wobei die Datensätze aus der zweiten Tabelle jedoch nur mit einem Datensatz aus der ersten Tabelle in Beziehung stehen.*
3. ***N:M – Beziehung****: Eine beliebige Anzahl von Datensätzen aus einer Tabelle stehen mit einer beliebigen Anzahl an Datensätzen aus einer zweiten Tabelle in Beziehung.*

Grundlagen

Hier wird die grundlegende Funktionsweise der Rails - Assoziationen am Beispiel einer 1:N - Beziehung demonstriert: Flat -*Rooms (Eine Wohnung kann mehrere Zimmer haben). Ein Migrationsscript auf der nächsten Seite demonstriert die 1:N – Beziehung. Die Tabelle `rooms` hat den Fremdschlüssel `flat_id`, dadurch wurde die gewünschte Beziehung zwischen Flat und Room hergestellt. Dennoch kann diese Assoziation momentan nur auf der Datenbankebene bearbeitet werden. Um diese Assoziation auf der Modellebene benutzen zu können, kommen die ActiveRecord-Assoziationsmakros zum Einsatz. Das sind statische Methoden, die eine Domain-Klasse zur Laufzeit um zusätzliche Methoden erweitern.

```
class CreateFlatsAndRooms < ActiveRecord::Migration
      def self.up
            create_table :flats do |t|
                  t.integer :nummer
                  t.integer :size
            end
            create_table :rooms do |t|
                  t.integer :kind
                  t.integer :size
                  t.integer :flat_id
            end
      end
      def self.down
            drop_table :flats
            drop_table :rooms
      end
end
```

Für einen Zugriff aus der `Flat`-Klasse auf die assoziierten `Room`-Objekte, benötigt `Flat` den Aufruf der `has_many`-Methode:

```
class Flat < ActiveRecord::Base
      has_many :rooms
end
```

Die `Flat`-Klasse wird von ActiveRecord um Zugriffsmethoden auf die assoziierten Objekte erweitert. Die Methode `rooms()` ist eine davon. Sie liefert ein Array von `Room`-Objekten zurück:

```
wohnung = Flat.find(1)
wohnung.room().each ...
```

Wenn der Zugriff eines Room-Objekts auf sein Flat-Objekt gewünscht ist, dann muss die Room-Klasse eine `belongs_to` Direktive enthalten:

```
class Room < ActiveRecord::Base
      belongs_to :flat
end
```

Dadurch wird die Room-Klasse ebenfalls um weitere Methoden von ActiveRecord erweitert. Beispielweise die Methode `flat()` gibt das Flat-Objekt zurück, zu dem die Room-Klasse letztendlich gehört (assoziiert):

```
wohnung = Flat.find(1)
wohnung.rooms().each do |zimmer|
      puts zimmer.size
end
```

Wichtig ist, dass die Assoziationsmakros keine Pflicht sind. Der Entwickler soll vom

Fall zu Fall unterscheiden, ob die jeweiligen Zugriffe aus der Modellsicht gewährt sein sollen. Somit könnte gegebenenfalls die `belongs_to`-Direktive entfallen. Dabei taucht die Frage auf: Kann ActiveRecord nicht komplett auf die Assoziationsmakros verzichten, wenn die Beziehungen schon sowieso auf der Datenbankebene definiert sind? Leider ist das für Assoziationen nicht möglich, aus der Tabellenstruktur, die Art der Assoziation zu erkennen, da sich die 1:1 und 1:N Beziehungen in der Tabellenstruktur nicht unterscheiden. An der Stelle fehlt Rails die notwendige Information über den Assoziationstyp, um die ORM-Mapping-Konfiguration komplett automatisiert durchzuführen. So ist die zusätzliche Deklaration auf Modellebene erforderlich.

Im Weiteren werden drei Beziehungstypen einzeln vorgestellt.

1:1 – Beziehungen: has_one - belongs_to

Zur Demonstration dieser Beziehung kann eine User-Contact-Beziehung benutzt werden, wobei ein Benutzer exakt eine Kontaktinformation besitzt und eine Kontaktinformation genau zu einem Benutzer gehört. Dazu werden beide Klassen `User` (Parent-Klasse) und `Contactinfo` (Child-Klasse) benötigt und dazugehörigen Tabellen über Migrationen erstellt. Dabei hält die Child-Klasse, in diesem Fall `Contactinfo`, den Fremdschlüssel (user_id) der Relation. Das Migrationsscript zum Erzeugen der Tabelle `contactinfos` sieht wie folgt aus:

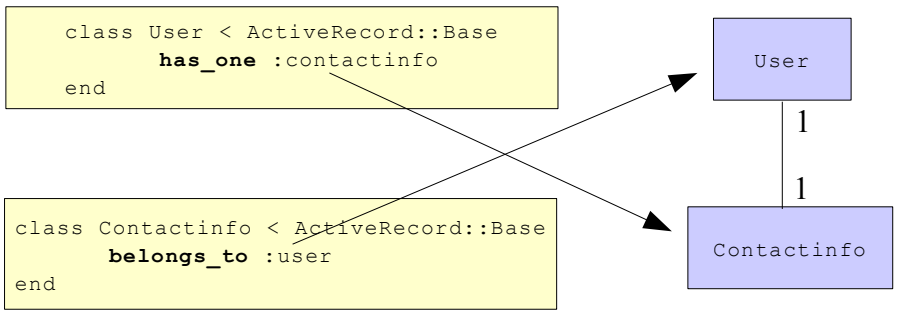
```
class CreateContactinfos < ActiveRecord::Migration
      def self.up
            create_table :contactinfos do |t|
                  t.string :tel
                  t.string :mob
                  t.string :email
                  t.integer :user_id
                  t.timestamps
            end
      end

      def self.down
            drop_table :contactinfos
      end
end
```

Das Migrationsscript zum Erzeugen der anderen Tabelle `Users` ist ähnlich aufgebaut und muss nicht extra dargestellt werden.

Auf der Modellebene wird die Assoziation zwischen den beiden Klassen durch die Assoziationsmakros `has_one` und `belongs_to` in den Klassen selbst definiert. Das Schlüsselwort `belongs_to` wird in der Klasse verwendet, die den Fremdschlüssel der anderen Tabelle enthält. Das heißt in der Child-Klasse `Contactinfo`.

Dementsprechend gehört das Schlüsselwort `has_one` in die Parent-Klasse `User`. Laut der ActiveRecord-Namenskonvention entspricht der Fremdschlüsselname dem Namen der entsprechenden Klasse mit dem Suffix `_id`. Der Entwickler muss sich beim Anlegen der Tabellen an die Namenskonvention halten, damit ActiveRecord die gewünschte Relation modellieren kann. Beim Abweichen von der Konvention kann der Fremdschlüssel mit Hilfe des Parameters `:foreign_key` explizit angegeben werden. Der gesamte Sachverhalt kann so dargestellt werden:



Dabei werden keine expliziten Fremdschlüsseln von ActiveRecord in der Datenbank erzeugt. Das entspricht dem Paradigma, dass keinerlei Logik innerhalb des Datenbanksystems abgebildet wird. Jedoch signalisiert das Schlüsselwort `belongs_to` ActiveRecord, in welcher der beiden beteiligten Datenbanktabellen der Fremdschlüssel abgelegt wird. Außerdem gibt dieses Schlüsselwort an, dass beim Speichern eines Parent-Objekts alle Child-Objekte automatisch validiert werden. Für die Objekte, die per `has_one` verbunden sind, gilt das nicht.

Beide Deklarationen tragen noch eine weitere Funktion. Sie veranlassen ActiveRecord zur dynamischen Erweiterung der beteiligten Klassen um eine Reihe zusätzlicher Methoden, die die Benutzung der Assoziation auf der Modellebene ermöglichen. Diese Methoden werden an konkreten Beispielen beschrieben.

Methoden, die in der `User`-Klasse erzeugt werden:

contactinfo=()

Die Methode `contactinfo=()` weist einem Benutzer eine Kontaktinformation zu:

```
user = User.new(:last_name => "Schmidt")
user.contactinfo = Contactinfo.new(:tel => "040/99999")
user.save
```

Der Aufruf von `save()` speichert neben dem `User`-Objekt auch das erzeugte `Contactinfo`-Objekt mit.

contactinfo()

Diese Methode liefert die Kontaktinformation von einem Benutzer:

```
person = User.find(user.id)
puts person.contactinfo.tel -> $  040/99999
```

create_contactinfo()

Diese Methode erzeugt eine `Contactinfo`-Instanz und weist sie dem Benutzer zu.

```
user = User.create(:last_name => "Schmidt")
user.create_contactinfo(:tel => "040/99999")
```

built_contactinfo()

Diese Methode unterscheidet sich von der create_contactinfo() darin, dass sie den expliziten Aufruf von `save()` zum Speichern der erzeugten Instanz benötigt.

Analog zu diesen Methoden werden folgende Methoden in der `Contactinfo`-Klasse erzeugt: ***user(), user=(), create_user(), built_user(), user.nil?()***. Aufgrund ihrer gleichen Funktionalität müssen sie nicht noch mal erklärt werden.

user.nil?()

Sie prüft, ob ein assoziiertes Objekt existiert:

```
contact = Contactinfo.create(:tel => "040/99999")
assert contact.user.nil?
```

Assoziationen können zusätzlich durch die Angabe von Parametern konfiguriert werden. Der einzige Pflichtparameter ist der Name der Assoziation. Weitere Parameter sind optional. Aus Platzgründen werden hier nicht alle Parametern vorgestellt:

:class_name

Dieser Parameter ist für den Namen der assoziierten Klasse zuständig. Er ist nur dann notwendig, wenn der Klassenname von dem Namen der Assoziation abweicht und kann eingesetzt werden, um einige Modelle zu spezialisieren. Zum Beispiel steht die Benutzerverwaltung in der Beziehung mit allen Benutzern. Dabei soll ein Benutzer die Rolle des Administrators übernehmen. Diese Beziehung kann dann `:admin` genannt werden. Da der Administrator, genau wie die anderen Benutzers in der Tabelle `users` gespeichert wird, ist für diese Assoziation die Angabe des zugehörigen Klassennamen `User` sinnvoll:

```
class Usermanagement < ActiveRecord::Base
      has_one :admin, :class_name => "User"
end
```

Der class_name Parameter erzeugt neue `admin()`-Methode, die User-Instanzen akzeptiert und zurückliefert:

```
usermanagement = Usermanagement.new
usermanagement.admin = User.new(:last_name => "Schulz")
```

:conditions

Dieser Parameter dient der Angabe von bestimmten Bedingungen, die vom assoziierten Objekt erfüllt werden müssen. Es kann gewünscht werden, dass ein Benutzer die Rolle

des Administrators nur dann annehmen kann, wenn er vor einem Jahr registriert wurde:

```
class Usermanagement < ActiveRecord::Base
      has_one :admin,
              :class_name => "User",
              :conditions => "years_from_registred => 1"
end
```

Diese Formulierung bewirkt, dass ein assoziierter Administrator nur dann geladen wird, wenn er vor mindestens einem Jahr registriert wurde.

:dependent

Dieser Parameter bestimmt, was mit dem Child-Objekt passieren soll, nachdem der Parent-Objekt gelöscht wird oder ein anderes Child-Objekt zugewiesen bekommt. Eine Kontaktinformation macht ohne ihren Besitzer oder wenn dieser eine neue Kontaktinformation bekommen hat, keinen Sinn:

```
class User < ActiveRecord::Base
      has_one :contactinfo, dependent => :destroy
end
```

Das bewirkt, dass nach dem Löschen eines Benutzers der zugehörige Kontakt in der Datenbank nicht mehr existiert. Das heißt ActiveRecord löscht das abhängige Objekt automatisch.

Die `belongs_to` - Deklaration unterstützt alle diese Parametern auch, außer des `:dependend` - Parameters. Zusätzlich kann ein weiterer Parameter angegeben werden, der von der `has_one`-Deklaration nicht unterstützt wird.

:counter_cache => true

Er liefert die Anzahl der assoziierten Child-Objekte. Für die 1:1 Beziehung kann dieser nur die werte 0 oder 1 annehmen.

In der Tabelle 3 sind alle erzeugten Methoden noch mal aufgelistet:

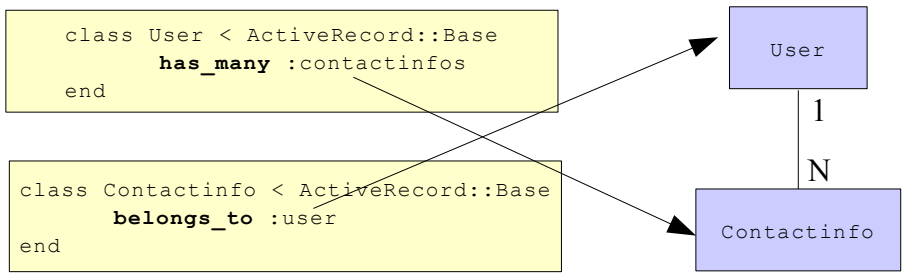
1:1-Assoziation	**Parent: User**	**Child: Contactinfo**
Deklaration	has_one()	belongs-to()
erzeugte Methoden	contactinfo() contactinfo=() create_contactinfo() build_contactinfo()	user() user=() user.nil() create_user() built_user()

Tabelle 3: 1:1 Assoziation: Methodenübersicht

1:N – Beziehungen: has_many – belongs_to

Dieser Beziehungstyp unterscheidet sich von dem 1:1 – Typ darin, dass die Parent-Klasse anstatt einer Objektinstanz mehrere Objektinstanzen verwalten muss. Diese Anzahl an Instanzen kann als Array angesprochen und weiterverwendet werden. Ferner ist es möglich, einzelne Child-Objekte hinzuzufügen und nach bestimmten Child-Objekten zu suchen. Die Child-Klasse wird mit der `belongs_to` - Direktive ausgezeichnet. Dabei ist die Sicht des Child-Objektes völlig identisch mit der Sicht des Child-Objektes aus der 1:1-Relation. Es werden gleiche Methoden erzeugt und gleiche Parametern zur Anpassung der Assoziation benutzt. Deshalb wird an dieser Stelle nicht nochmal darauf eingegangen.

Hingegen unterscheidet sich die Parent-Klasse von der 1:1 Beziehung in einigen Aspekten. Zur Demonstration kann das zuvor besprochene Beispiel wiederverwendet werden. Mit der Annahme, dass ein Benutzer aus der Modellsicht mehrere Kontaktinformationen hat (Hauptwohnsitz und Ferienwohnung), um dieses Verhalten modellieren zu können. Dabei werden die selben Migrationsscripte verwendet, da keine Änderungen auf der Datenbankebene nötig sind. Auf der Modellebene bekommt die User-Klasse die `has_many` - Direktive mit dem Plural von Contactinfo als Parameter. Das Modell Contactinfos gibt es als solches nicht, die Entwickler von ActiveRecord haben dabei den Wert mehr auf die Lesbarkeit des Codes gelegt, als auf die technische Korrektheit. Das Bild auf der nächsten Seite veranschaulicht das gewünschte Verhalten.



Obwohl die vorgenommenen Änderungen nicht wirklich spektakulär sind, ist die 1:N somit modelliert. Vom Interesse sind die Methoden, um die die User-Klasse auf Grund der `has_many`- Deklaration, von ActiveRecord automatisch erweitert wird.

contactinfos=()

Die Methode weist einem Benutzer nicht nur ein Contacinfo-Objekt, sondern ein Array mit Kontaktinformationen zu:

```
user = User.new(:last_name => "Schmidt")
user.contactinfos = [Contactinfo.new(:tel => "040/99999")]
user.save
```

contactinfos()

Liefert eine Liste der Kontaktinfos eines Benutzers.

contactinfos<<()

Der Methodenaufruf fügt eine Kontaktinformation der Kontaktliste eines Benutzers an:

```
user.contactinfos << Contactinfo.new(:tel => "040/66666")
```

Neue Kontaktinformation wird unmittelbar nach ihrer Erzeugung und dem Aufruf „<<“ in die Datenbank geschrieben. Das bedeutet, der Aufruf `user.save` kann in diesem Fall entfallen.

contactinfos.delete()

Damit wird die Verbindung zwischen den Kontaktinformationen und ihrem Benutzer gelöscht, wobei die Daten aus der Datenbank nicht gelöscht werden, sondern der Fremdschlüssel `user_id` auf `nil` gesetzt wird .

```
user.contactinfos.delete(040/66666)
```

contactinfos.empty?()

Prüft, ob Benutzer assoziierte Kontaktinfos besitzt.

contactinfos.clear()

Leert die Kontaktinfos-Liste eines Benutzers. Die Kontakte bleiben in der Datenbank jedoch enthalten.

contactinfos.size()

Liefert die Anzahl mit dem Benutzer verbundenen Kontaktinformationen.

contactinfos.size()

Liefert die Anzahl mit dem Benutzer verbundenen Kontaktinformationen.

contactinfos.find()

Durchsucht die Kontaktinfos-Liste nach bestimmten Suchkriterien. Die Parameter entsprechen denen der Methode `find()` (sihe Abschnitt 3.4.6).

contactinfos.build()

Die Methode erzeugt ein neues Kontakt-Objekt und fügt es dem Ende der Kontaktinfo-Liste an, ohne es zu speichern.

```
user = User.new(:last_name => "Schmidt")
user.contactinfos.build("tel" => "040/99999")
```

contactinfos.create()

Erzeugt ein neues Kontakt-Objekt und fügt es dem Ende der Kontaktinfo-Liste an. Die Kontakinfos werden ohne expliziten Aufruf von `save()` gespeichert. Der Methodenaufruf ist identisch zu `build()`.

Genauso wie `has_one` kann die `has_many`-Assoziation durch Angabe von optionalen Parametern angepasst werden. Folgende Parameter `:class_name`, `:conditions`, `:order` und `:foreign_key` werden analog zu `has_one`-Assoziation gehandhabt. Deshalb werden nur nicht beschriebene Parameter

`:finder_sql` und `:counter_sql` erklärt.

:finder_sql

Dieser Parameter erlaubt dem Entwickler für das Laden der assoziierten Kontaktinfo-Objekten ein SQL-Statement vorzugeben. Dieser Parameter ist nur bei Verwendung komplexer Assoziationen, die über mehrere Tabellen gehen, relevant.

:counter_sql

Gibt das SQL-Statement zum Ermitteln der Anzahl der assoziierten Objekte vor.

Die Parameter der `belongs_to`- Assoziation wurden bereits erklärt. Nur ein Parameter soll an dieser Stelle im Zusammenhang mit 1:N-Relation genauer erläutert werden.

:counter_cache

Dieser Parameter lässt den unnötigen Datenbankzugriff bei wiederholtem Methodenaufruf `contactinfos.size()` vermeiden:

```
class Contactinfo < ActiveRecord::Base
      belongs_to :user, counter_cache => true  end
```

Somit wird die Anzahl der assoziierten Child-Objekte gecashed. Obwohl das Ein- und Ausschalten des Cashing in der Child-Klasse stattfindet, befindet sich der Cache in der Datenbanktabelle der Parent-Klasse:

```
class AddContactinfosCountToUsers < ActiveRecord::Migration
      def self.up
          add_column  :users, :contactinfos_count, :integer,
                      :null => false, default => 0
      end
      def self.down
          remove_column :users, :contactinfos_count
      end
end
```

Bei der Definition des Caches greift die Namenskonvention ein. Der Cache Name besteht aus dem Namen der Assoziation gefolg von `_count`. So ergibt sich der Name `contactinfos_count`. Der Default-Wert des Felds muss 0 sein. Die Funktionsweise des Caches wird im Folgenden erklärt. Beim Hinzufügen oder Entfernen des Child-Objekts wird der Cache automatisch aktualisiert. Die `size()`-Methode liefert den Wert des Counter-Cache. Somit kann die Anzahl der Child-Objekte ohne Datenbankzugriffen ermittelt werden. Das wirkt sich positiv auf Performance aus, da die Datenbankzugriffe länger dauern als lokale Methodenaufrufe. In der Tabelle 4

sind erzeugte Methoden für jeweilige Deklaration im Überblick aufgelistet.

1:N-Assoziation	**Parent: User**	**Child: Contactinfo**
Deklaration	has_many()	belongs-to()
erzeugte Methoden	contactinfos() contactinfos=() contactinfos<<() contactinfos.delete() contactinfos.empty?() contactinfos.clear() contactinfos.size() contactinfos.find() contactinfos.build() contactinfos.create()	user() user=() user.nil() create_user() built_user()

Tabelle 4: 1:N-Assoziation: Methodenübersicht

N:M – Beziehungen: has_and_belongs_to_many

Bei dieser Beziehung kann ein Child-Objekt zu mehreren Parent-Objekten in Relation stehen. Gleichzeitig steht ein Parent-Objekt in Relation zu 0 bis N Child-Objekten. Ein Beispiel für so eine Assoziation ist die Beziehung zwischen einem Verein und dessen Mitgliedern. Eine Person kann in mehreren Vereinen Mitglied sein, dabei hat ein Verein in der Regel viele Mitglieder.

Dieser Sachverhalt bereitet ein Problem bei der Modellierung. In den relationalen Datenbanken werden drei Tabellen gebraucht, um dieses Modell abbilden zu können. In einer Tabelle werden Vereine gepflegt, in der zweiten die Mitglieder und in der dritten werden die eigentlichen Beziehungen zwischen den beiden Entitäten, in Form eines zusammengesetzten Fremdschlüssels gehalten.

ActiveRecords stößt ebenso an dieses Problem in seiner Verfolgung des Paradigmas, dass für eine Domain-Klasse schlicht eine Datenbank-Tabelle benutzt wird. Die Domain-Klassen selbst werden wie bekannt auf jeweils einer Datenbank-Tabelle abgebildet. Die Beziehung bekommt aber auch eine eigene Tabelle, die oft als Join-Tabelle benannt wird. Somit werden für zwei Domain-Klassen insgesamt drei Tabellen gebraucht. Dabei muss der Entwickler für die Existenz dieser Tabelle sorgen. Beim Anlegen der Join-Tabelle muss die Namenskonvention von ActiveRecords beachtet werden. ActiveRecord möchte als Tabellennamen den verketteten Namen der assoziierten Tabellen in der alphabetischen Reihenfolge haben. Die Tabelle kann mit einem Migrationsscript angelegt werden, der auf der nächsten Seite dargestellt ist.

```
class CreateClubsMembers < ActiveRecord::Migration
      def self.up
            create_table :clubs_members do |t|
                  t.integer :club_id
                  t.integer :member_id
            end
      end
      def self.down
            drop_table :clubs_members
      end
end
```

Die Join-Tabelle beinhaltet zwei Spalten, beide haben in ihren Namen den Suffix _id, was darauf hindeutet, dass es sich dabei um die Primärschlusseln aus den beiden beteiligten Tabellen handelt. Da ActiveRecord auf der Datenbankebene keine Schlüsseln (Constraints) anlegt, wird die N:M-Beziehung auf Modellebene durch die Assoziationsmethode `has_and_belongs_to_many` modeliert. Die beiden assoziierten Klassen müssen um diese erweitert werden. Ab dem Zeitpunkt weiß ActiveRecord, dass beide Klassen in einer N:M-Relation zu einander stehen und wie diese Beziehung zu pflegen ist:

```
class Club < ActiveRecord::Base
      has_andbelongs_to_many :members
end


class Member < ActiveRecord::Base
      has_andbelongs_to_many :clubs
end
```

Diese Deklaration hat zur Folge, dass die Beziehung um einige Methoden von ActiveRecord erweitert wird. Diese werden im Weiteren bezogen auf das Club-Member Beispiel vorgestellt.

members()

Liefert ein Array der assoziierten Memberobjekte.

members<<(Member)

Methode fügt einen Mitglied ans Ende der Mitgliederliste und speichert die Assoziation in der Join-Tabelle. Sodass der Entwickler sich darum nicht kümmern muss:

```
hsv = Club.create(:name => "HSV")
spieler = Member.create(:firstname => "Rafael")
hsv.members << spieler
```

Die Gesamtanzahl der Spieler beim HSV-Verein kann mit dem Aufruf

`hsv.members.size` geprüft werden.

members.push_with_attributes(Member, join_attributes)

Diese Methode macht das Gleiche wie die `members<<()`-Methode. Über die Hash `join_attributes` werden zusätzliche Werte mitgeben, die in der Join-Tabelle gespeichert werden. Diese Methode soll dann verwendet werden, wenn es assoziationübergreifende Attribute gibt.

members.delete(Member,...)

Löscht einen oder mehrere Mitglieder aus dem Verein.

members=(members)

Fügt eine Liste der Mitglieder hinzu und speichert die entsprechende Assoziationen in der Join-Tabelle. Schon vorhandene Mitglieder werden nicht eingefügt.

members_ids(ids)

Ersetzt Mitglieder, deren IDs mit den IDs der übergebenen Liste übereinstimmen.

members.clear()

Alle Mitglieder werden aus der Liste gelöscht, die bleiben aber in der Datenbank erhalten.

members.empty?()

Liefert einen booleschen Wert. Für eine leere Liste den `true`-Wert.

members.size()

Liefert die Anzahl der Mitglieder im Verein.

members.find(id)

Sucht nach dem Objekt in den assoziierten Mitgliedern mit der gegebenen ID:

```
hsv = Club.create(:name => "HSV")
spieler = Member.create(:firstname => "Rafael")
hsv.members << spieler
assert_equal spieler, hsv.members.find(spieler.id)
-> true
```

Die `has_and_belongs_to_many` – Deklaration verfügt über mehrere optionale Parameter. Die Parameter `:class_name`, `:foreign_key`, `:conditions`, `:order` unterscheiden sich nicht von den vorher besprochenen Parametern. Die restlichen Parametern werden kurz vorgestellt.

:join_table

Gibt den Namen der Join-Tabelle an, sofern er sich von der Namenskonvention abweicht.

:association_foreign_key

Dient der expliziten Angabe des Fremdschlüssels der assoziierten Klassen.

:unique

Damit kann angegeben werden, ob Duplikate in der Liste der assoziierten Objekte ignoriert werden sollen. Es sind boolesche Werte erlaubt `true` oder `false`.

:finder_sql

Gibt das alternative SQL-Statement zum Laden der Assoziation an.

:delete_sql

Gibt das alternative SQL-Statement zum Löschen der Assoziation an.

:insert_sql

Gibt das alternative SQL-Statement zum Einfügen der Assoziation in die Join-Tabelle an.

Die Tabelle 5 fasst die automatisch erzeugte Methoden zusammen.

N:M-Assoziation	**N-Klasse: Club**	**M-Klasse: Member**
Deklaration	has_and_belongs_to_many()	
erzeugte Methoden	members() members=() members<<() members.delete() members.empty?() members.clear() members.size() members.find() members_ids() members.push_with_attributes()	clubs() clubs=() clubs<<() clubs.delete() clubs.empty?() clubs.clear() clubs.size() clubs.find() clubs_ids() clubs.push_with_attributes()

Tabelle 5: N:M-Assoziation: Methodenübersicht

Das waren die grundsätzlichen Assoziation, die von ActiveRecord unterstützt werden. Aufgrund der zeitlichen Begrenzung werden weitere Assoziationen im Rahmen dieser Arbeit nicht behandelt. Es bleibt zu erwähnen, dass ActiveRecord polymorphe Assoziationen: has_many – belongs_to und seit Rails 1.1 eine “has_many:through“ Assoziation unterstützt. Die “has_many:through“-Beziehung ist eine Variante der N:M Relation. Sie benötigt neben einer Join-Tabelle noch ein eigenes ActiveRecord-Modell. Auch die Selbstreferenzierung von Modellen (Aufgabe besteht aus Unteraufgaben) ist mit ActiveRecord möglich.

3.5 Zusammenfassung

Insgesamt fällt auf, dass Rails bzw. ActiveRecord dem Entwickler viel Arbeit abnimmt, wenn er sich an die vorgegebene Namenskonvention hält. Erfreulich ist die Tatsache, dass die CRUD-Operationen schon vom Framework zur Verfügung gestellt werden. Der Entwickler kann sofort, nach einem geringen Konfigurationsaufwand für die Datenbankverbindung mit der Programmierung der Anwendung loslegen, ohne sich um die Persistenz der Objekte kümmern zu müssen. Diese Anforderung an ein ORM - Framework erfüllt ActiveRecord zu 100%. Der Umgang mit dem Joins-Parameter bei der Objektsuche hingegen ähnelt sich stark dem aus der relationalen Datenbankwelt. Für die Anforderung, bei der ein ORM-Ansatz ganz auf SQL verzichtet soll, ist es daher nicht zufriedenstellend. Das alt bekannte Problem des ORM bei einer N:M-Beziehung löst ActiveRecord zwar nicht, automatisiert aber die Pflege der Join-Tabelle, sodass der Entwickler die Objekte weiterhin datenbankunabhängig manipulieren kann.

Wenig positiv für den Neueinsteiger wird es, sobald er von der Konvention abweicht und einen fehlerhaften Code produziert hat. Vieles läuft im Hintergrund ab, völlig unbemerkt für ihn. Dadurch wird zwangsläufig die Fehlerursache verborgen und die Fehlersuche erschwert.

Im Ganzen verwischt ActiveRecord die Grenze zwischen der Modellebene und der Datenbankebene, indem es auf die gewöhnlichen Constraints auf der Datenbankebene komplett verzichtet. Dadurch wird der Wechsel zu einer anderen Datenbank vereinfacht.

Im nächsten Kapitel wird das JavaServer Faces Framework vorgestellt.

4 Java

„Ein alter Bekannte neu gekleidet."

4.1 Einleitung

Java ist eine objektorientierte Programmiersprache von der Firma Sun Microsystems. Die Sprache ist heutzutage sehr verbreitet und populär. Sie wird von den großen Unternehmen und kleinen Firmen bei der Softwareentwicklung erfolgreich eingesetzt. Zudem wird sie an vielen Fachhochschulen und Universitäten in Deutschland und weltweit unterrichtet. Aus diesem Grund wird es auf die Programmiersprache selbst in dieser Arbeit nicht weiter eingegangen. Java schafft aber die Basis für viele Frameworks (sieh Kapitel 3) die sich mit der Web-Entwicklung beschäftigen. Um ein Framework von diesen javabasierten Frameworks „JavaServer Faces" geht es im diesem Kapitel.

Das Kapitel stellt keinerlei vollständiges Nachschlagewerk zur JavaServer Faces dar. Um diesen Zweck erfüllen zu können, müsste die Arbeit um die 400 Seiten umfassen. Abhilfe zu dem Thema schaffen Bücher, welche dem Literaturverzeichnis entnommen werden können. Die Intention dieses Kapitels liegt eher daran: die Technologie überblicksartig vorzustellen, damit die Zusammenhänge, bei der Realisierung der Beispielanwendung im weiteren Kapitel, nachvollziehbar werden. Auch die Auswahl zwischen anderen Web-Technologien sollte nach diesem Kapitel klar sein.

4.2 Hintergrund

Im Jahre 1995 wurde Java von Sun vorgestellt. Laut Sun beabsichtigte Java zwei Hauptziele: das erste Ziel war, die portable Programmierung der Hardware-naher Gerätesteuerung (z. B. Wasch- und Kaffeemaschinen), das zweite Ziel war, das damalige Web interaktiver zu gestalten. Dank erstem Ziel gibt es die Plattformunabhängigkeit von Java durch die JVM (Java Virtual Machine). Dem zweiten verdanken wir die Applets [JSFFP]. Heute ist festzustellen, dass beide Ziele nicht, beziehungsweise nicht in dem prognostiziertem Ausmaß, erreicht wurden. Obwohl Java auf vielen Handys verbreitet ist, wird sie nicht zur Steuerung des Geräts benutzt, sondern für die Programmierung von Spielen und Applikationen. Die Verwendung der Applets in der heutigen Webanwendungen ist, aufgrund der notwendigen Java-Plug-In Installation, sehr begrenzt. Das ist keinesfalls als Niederlage von Java zu betrachten.

Die Programmiersprache Java, oder genauer gesagt die Plattform Java, hat noch nie da gewesenen Erfolg unter den Programmiersprachen. Dieser Erfolg ist im Bereich der server-seitigen Sprache zur Entwicklung unternehmenskritischer Anwendungen anzusiedeln. Dazu gehört sehr beliebtes Modell der Thin-Client-Anwendungen, bei denen der Benutzer durch einen Web-Browser die Anwendung bedient. Dieses Modell wird von vielen Java-Spezifikationen unterstützt.

4.3 Motivation

Es soll eine Webanwendung basierend auf Java entwickelt werden. Dazu ist ein entsprechendes Framework gesucht, weil die Frameworks in der Regel eine Architektur vorgeben und was viel wichtiger ist, dem Entwickler viel Routinearbeit abnehmen.

Eine Google-Suche nach „Framework“ am 20.02.08 hat 99.200.000 Treffer ergeben. Im Bereich der Web-Entwicklung mit Java sind JavaServer Pages (JSP), Struts und Spring die Bekanntesten. JavaServer Faces (JSF) ist das neuste Framework in diesem Bereich. Als Nachfolger sollte es besser als die Vorgänger sein und etwas Neues muss unbedingt ausprobiert werden. Somit ist die Wahl auf JSF gefallen.

4.4 Zukunftssicherheit

Im November 2003 wurde die Java-2-Enterprise-Edition (Java-EE) in der 1.4 Version, die Servlets, JavaServer Pages und Enterprise-JavaBeans umfasst, verabschiedet. Im März 2004 wurde das final Release der JSF-Spezifikation 1.0 veröffentlicht. Die Version 1.5 der Java oder neu 5.0 wurde im Mai 2006 freigegeben. Nun umfasst die Java-EE die JavaServer Faces. Das hat für alle Hersteller von Java-EE-Application-

Server zu bedeuten, wenn sie spezifikationskonform sein wollen, müssen ihre Server auch JavaServer Faces unterstützen. Somit ist die Verbreitung von JSF unabdingbar. Die aktuellste Version 1.2 wurde am 11.Mai 2006 veröffentlicht [JSFFP].

4.5 JavaServer Faces

JavaServer Faces sind ein Framework für die Entwicklung von Webapplikationen. Dieser Standard zur Entwicklung von Benutzeroberflächen gehört zu einer Webtechnologie der Java Plattform, Enterprise Edition kurz Java EE und basiert auf Servlets und JSP-Technologie. Zudem ist es ein komponentenbasiertes Framework. Das ermöglicht dem Entwickler, auf einfache Weise die JSF-Komponenten zum Beispiel für Benutzerschnittstellen in die Webseiten einzubinden oder die Navigation zu definieren. Die Ausgabeseiten werden nicht in HTML geschrieben, sondern werden aus Komponenten zusammengesetzt. Grafische Tools bieten hierzu Drag&Drop Funktionalität, sodass die Web-Oberfläche zusammen geklickt und sofort im Vorschau angezeigt werden kann. Ein Vertreter dieser Sparte ist ein OpenSource Projekt, WTP - Web Tools Plattform, welches als Plug-In in Eclipse installiert werden kann. Die Anzahl der Komponenten wird lediglich durch eingebundene Bibliotheken begrenzt. Alleine die Standardbibliothek JSTL (Java Standard Tag Library) bringt eine große Auswahl der Komponenten mit. Dazu gibt es weitere wie RichFaces von Jboss und MyFaces von Apache. Diese werden im Rahmen dieser Arbeit nicht verwendet, da die Testanwendung mit JSTL auskommt. Zusätzlich können JSF durch eigene Komponenten erweitert werden. Auf den Bau der eigenen Komponenten wird nicht weiter eingegangen.

Bei der GUI-Entwicklung hat sich das MVC-Entwurfsmuster (sieh Kapitel 3) durchgesetzt. Dieses Pattern wird durch JavaServer Faces durchgängig realisiert. Die View-Schicht wird von der Logik getrennt gehalten, sodass die beiden Schichten getrennt voneinander entwickelt werden können. Dabei kann die Logik durch ein beliebiges Modell implementiert werden. POJOs (Plain Old Java Objects), EJBs (Enterprise JavaBeans) oder andere beliebige Java-Objekte stehen dabei zur Auswahl. Um das MVC zu vervollständigen, fehlt nur der Controller. Der wird zum einen durch die JSF-Komponenten (es sind nicht nur GUI-Komponenten), zum anderen durch so genannte Handler, die in Java vom Entwickler realisiert werden müssen. Die Frage, wie das Ganze mit einander zusammenzuarbeiten hat, wird durch verständliche Konfiguration, basierend auf XML, beantwortet. Bei der Konfiguration wird der Benutzer von der IDE (in dieser Arbeit wird dafür Eclipse eingesetzt) grafisch unterstützt, daraus wird dann XML generiert. Zeitgemäß kann eine JSF-Webanwendung durch verschiedene Frameworks um Ajax-Funktionalitäten erweitert werden. Dennoch wird Ajax im Rahmen dieser Arbeit nicht behandelt.

Bei jeder großen Webanwendung kommt fast immer eine Datenbank zum Einsatz, um Daten persistent zu halten. JSF kann mit allen gängigen Datenbanken umgehen. Im Rahmen dieser Arbeit wird analog zu Rails eine relationale MySQL-Datenbank

verwendet. Zur Kommunikation zwischen der objektorientierten Logik der Applikation und der relationalen DB gibt es eine eigene ORM-Lösung für Java, nämlich Hibernate von Apache Fundation. Das Hibernate wird aus mehreren Gründen nicht eingesetzt. Zum einen würde die Einarbeitungszeit den Zeitrahmen der Arbeit sprengen. Zum anderen ist die Testanwendung nicht so umfangreich und nicht so komplex. Deshalb ist der Einsatz von JDBC gerechtfertigt. Zum dritten schafft es auch ohnehin die beabsichtigte Testbasis.

4.5.1 RPL

RPL steht für „Request Processing Lifecycle“ und stellt den Lebenszyklus einer JSF-basierten Webanwendung von der Anfrage bis zum Rendern der Antwort dar und ist sehr von der Bedeutung für das Verständnis von JSF. Da die JavaServer Faces mit Servlet-API realisiert werden und Servlets auf dem Request-Response-Modell des zugrunde liegenden HTTP-Protokolls basieren, erben sie damit die Eigenschaften einer HTTP-Anfrage und einer HTTP-Antwort. Das Protokoll hat im Bezug auf die Webanwendungen seine Nachteile, wie z.B die Zustandslosigkeit. Deren Umgehung wird durch ein vollständiges MVC-Konzept erreicht. Der Controller kann mittels Change-Event die Änderungen zwischen zwei HTTP-Anfragen erkennen. Um das zu ermöglichen, muss der Zustand vor den Benutzereingaben gespeichert und bei der neuen Anfrage mit dem aktuellen Zustand verglichen werden. Dabei müssen Validierungen und Konvertierungen durchgeführt, Events verarbeitet und eventuell neue erzeugt werden. Dieses komplexe Verfahren wird durch „Request Processing Lifecycle“ beschrieben und ist in der Abbildung 7 dargestellt:

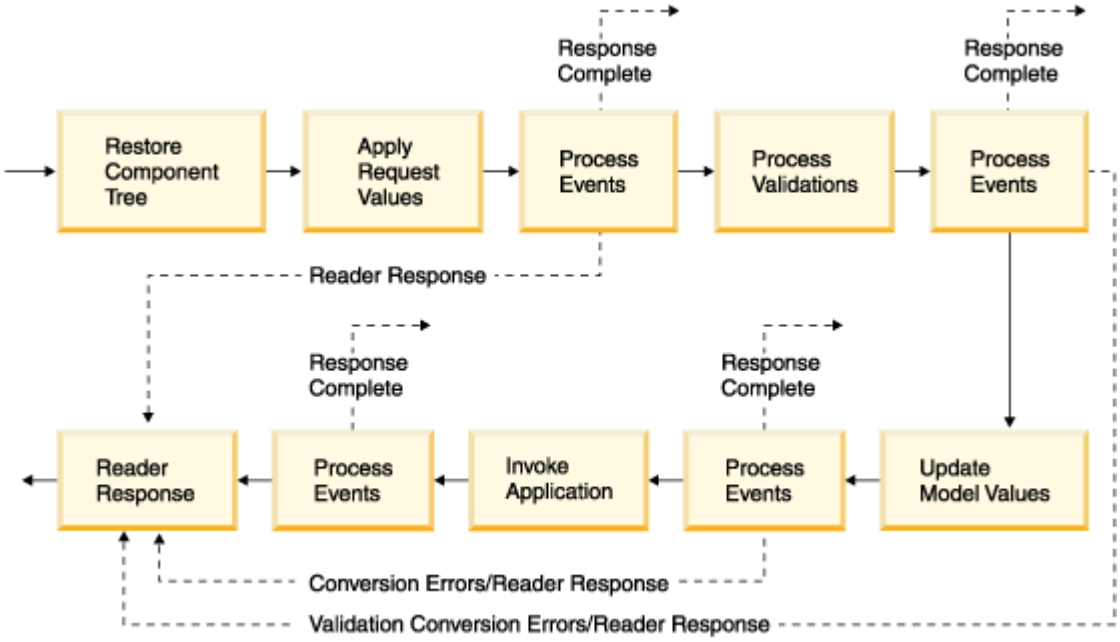


Abbildung 7: Request Processing Lifecycle [RPL]

Die Bearbeitung einer JSF-Anfrage beginnt, nachdem der Server bzw. JSF-Servlet den HTTP-Request erhalten hat. Zwischen sechs Bearbeitungsphasen sind Event-Verarbeitungsphasen “Process Event“ vorgesehen. Bei diesen werden die Events an interessierende Event-Listener übergeben. Bei der Abarbeitung der Events können die JSF-Komponenten verändert oder die Anwendungsdaten verarbeitet werden. Gegebenenfalls kann es auf das Rendern der Antwort verzichtet und direkt ans Ende der Bearbeitung gesprungen werden. Dies ist durch die gestrichelten Linien mit „Response Complete“ dargestellt. Der Fall tritt bei den binären Daten, wie JPG-Grafik oder PDF-Dokument, auf. Bei den Validierungs- und Konvertierungsfehlern werden die nachfolgenden Phasen übersprungen und nur eine Antwort generiert. In der Regel handelt es sich dabei um Fehlermeldungen.

4.5.2 Expression-Language

JSF-Expression-Language (JSF-EL, kurz EL) wird überwiegend in der GUI-Schicht verwendet um die Anwendungsdaten aus dem Modell zu lesen bzw. ins Modell zu schreiben. Sie basiert auf der JSP-EL mit dem Hauptunterschied, dass die Wertebindungen in beide Richtungen, lesend und schreibend, also „bijektiv“, ausgeführt werden. Die Grundlegende Syntax sieht einen String mit einem führenden Rautezeichen gefolgt von den geschweiften Klammern vor:

```
"#{ EL }"
```

Der EL-Ausdruck steht in der Klammern und kann eine Wertebindung, eine Methodenbindung, einen einfachen arithmetischen oder logischen Ausdruck enthalten. Auch die Kombinationen aus diesen Varianten sind möglich außer Methodenbindung.

4.5.3 Bean-Properties

Als Bean-Property werden Attribute einer Java-Klasse bezeichnet unter Voraussetzung, dass die Attribute über Getter- oder Setter-Methoden verfügen und die Klasse selbst als Manage-Bean in JSF eingetragen ist. Über eine Wertebindung (engl. Value Binding) kann der Wert einer UI-Komponente (z.B ein Tetxausgabefeld) an eine Bean-Property, an die UI-Komponente selbst gebunden oder die Properties einer UI-Komponente initialisiert werden. Ein Beispiel für eine Wertebindung sieht so aus:

```
<h:outputText value="#{user.firstname}" />
```

Die UI-Komponente „Outputtext“ soll den Benutzervornamen mittels Value-Binding auf der Webseite anzeigen. Dabei ist die User-Klasse ein Bean mit der Property firstname. Dementsprechend gibt es in der User-Klasse eine `getFirstname()`-Methode. Laut JSF-EL wird der Beanname klein geschrieben. Beim Zugriff auf die Bean-Property wird der Präfix „get“ weggelassen und der Rest klein geschrieben. Um

den Benutzervornamen in das Modell einzulesen, muss eine andere UI-Komponente verwendet werden:

```
<h:inputText value="#{user.firstname}" />
```

Anhand der Komponente weiß JSF, ob es sich um einen schreibenden oder lesenden Zugriff handelt.

Bei einer Methodenbindung (engl. Value-Binding) wird eine Bean-Methode referenziert. Dies wird beispielsweise bei den Event-Handlern und Validierungsmethoden verwendet. Als Beispiel dazu soll ein Benutzer beim Klicken auf einen save-Button in die Datenbank gespeichert werden. Für das Speicher sorgt die save()-Methode der User-Klasse. Der entsprechende Code dazu sieht dann so aus:

```
<h:commandButton value="Save" action="#{user.save}" />
```

Beim Button-Klick wird eine Action ausgelöst, diese ruft die save()-Methoder der User-Klasse auf. Dabei wird der Code dieser Methode ausgeführt und der Benutzer in die DB geschrieben.

4.6 Zusammenfassung

Bei der Vorstellung der JSF-Technologie mussten viele Aspekte und Hintergründe ausfallen. Das hat zur Folge, dass die einzelne Bearbeitungsphasen des RPL nicht detailliert beschrieben werden konnten. Auch solche Aspekte, wie Validierung, Konvertierung, Fehlerbehandlung, Internationalisierung, Event-Verarbeitung, Initialisierung und Lebensdauer der Beans und Navigationrules konnten nicht beschrieben werden. Das ist noch nicht alles was JFS zu bieten hat.

Zur Auswahl der Technologie haben folgende Punkte beigetragen. Das Framework basiert auf Java, was gute Zukunftsaussichten für diese Technologie verspricht. Das Framework ist ein OpenSource und kann kostenlos benutzt werden. Es eignet sich besonders gut zum direkten Vergleich mit Rails, da es auch auf dem MVC-Konzept basiert und es ist möglich, die gleiche Datenbank einzusetzen.

Im nächsten Kapitel wird eine Vergleichsmethode zweier Webanwendungen vorgestellt.

5 Vergleichsmethode

„Software zu vergleichen ist nicht trivial!"

In dieser Arbeit sollen zwei Webanwendungen miteinander verglichen werden. Dafür müssen in erster Linie die Messkriterien festgelegt werden. Mit anderen Worten stellt sich die Frage: „Was soll denn verglichen werden?" Dann muss eine Antwort auf die Frage gefunden werden: „Wie kann das verglichen werden?" Dazu wird eine passende Vergleichsmethode ausgewählt. Um den Vergleich möglichst objektiv zu gestalten, müssen beide Anwendungen gleiche Voraussetzungen erfüllen. Dieses Kapitel gibt die Antworten auf all diese Fragen.

5.1 Allgemein

Im Allgemeinen ist es unmöglich zu sagen, dass eine Software besser ist als die andere. Dafür beinhaltet eine Software zu viele verschiedene Kriterien, wie zum Beispiel: Entwicklungsaufwand, Wartbarkeit, Wiederverwendbarkeit, Testbarkeit, Codequalität, Performance, Benutzbarkeit, Robustheit, Korrektheit und die Antwortzeit. In diesen einzelnen Kriterien kann eine Software sehr wohl verglichen werden. Dabei lässt sich der Vergleich zweier Anwendungen auf analytische Qualitätssicherungsverfahren, die schon in der Software-Qualitätssicherung bekannt sind, zurückführen. Diese Verfahren

lassen sich je nach Zielsetzung in drei Klassen unterscheiden.

Die Klassifikation nach [LST] sieht so aus:

- Testende Verfahren
  - Dynamische Testverfahren
  - Statische Testverfahren
- Verifizierende Verfahren
  - Verifikation
  - Symbolische Ausführung
- Analysierende Verfahren
  - Analyse der Bindungsart
  - Metriken
  - Grafiken und Tabellen
  - Anomalienanalyse

Testende Verfahren haben das Ziel, Fehler zu erkennen. Verifizierende Verfahren wollen die Korrektheit einer Systemkomponente beweisen. Analysierende Verfahren vermessen und/oder stellen bestimmte Eigenschaften von Systemkomponenten dar. An dieser Stelle sei eine Systemkomponente einem System gleichgesetzt, da ein System aus einzelnen Komponenten zusammengesetzt wird oder gar aus einer Komponente bestehen kann.

Da diese Arbeit einen Vergleich zum Gegenstand hat, eignet sich das analysierende Verfahren am Besten. Das Verfahren, Metriken, belegt diese Auswahl.

„Metriken erlauben es, Eigenschaften, wie die strukturelle Komplexität, die Programmlänge oder den Grad der Kommentierung quantitativ zu ermitteln. Die Ergebnisse erlauben einen Quervergleich mit bisherigen Maßzahlen.“ [LST]

Andere Verfahren werden in diese Arbeit nicht einfließen. Um den Vergleich durchführen zu können, müssten zunächst vergleichbare Werte gegeben sein, die explizit für jede Anwendung gemessen werden können. In der Softwareentwicklung gibt es hierfür spezielle Messmetriken, die unter dem Namen Softwaremetrik zusammengefasst sind.

*„Eine **Softwaremetrik** ist eine Funktion, die eine Software-Einheit in einen Zahlenwert abbildet. Dieser berechnete Wert ist interpretierbar als der Erfüllungsgrad einer Qualitätseigenschaft der Software-Einheit“.* [IEEE Standard 1061, 1992]

Mit einer Software-Einheit ist in der Regel der zugrunde liegende Quellcode gemeint. In Abhängigkeit von der Sicht auf ein System können unterschiedliche Informationen interessant sein, was zur Folge eine Definition von vielen unterschiedlichen Metriken hat. Dabei misst jede Metrik nur einen bestimmten Bereich und nicht das gesamte Software-System. Je nach Bereichen werden die Metriken für Systemkomponenten, wie in der Abbildung 8 auf der Nächsten Seite zu sehen ist, gegliedert:

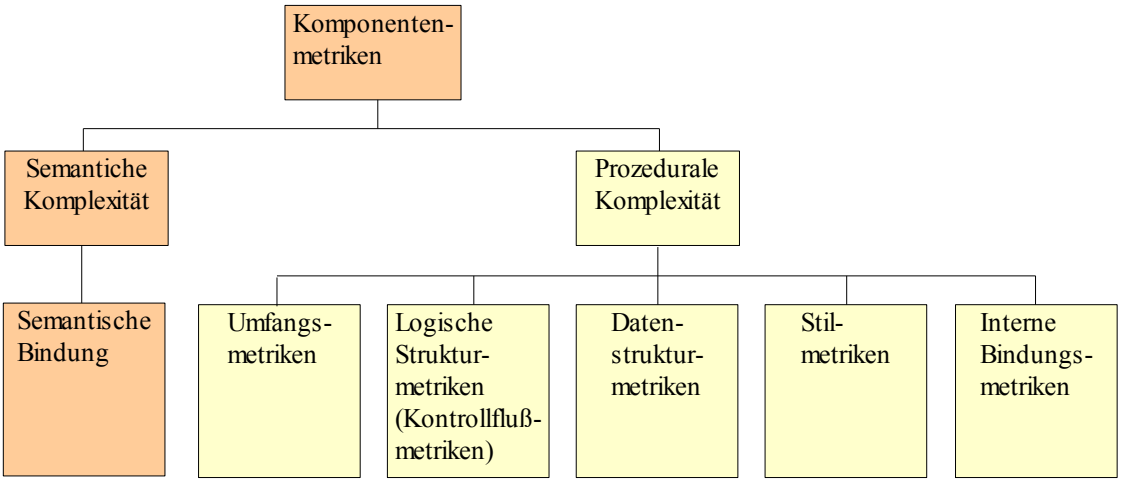


Abbildung 8: Klassifikation von Komponentenmetriken [LST]

Die semantische Komplexität einer Systemkomponente lässt sich mit Hilfe von Metriken nicht messen. Daher ist der linke Zweig etwas dunkler dargestellt. Zur Messung der prozeduralen Komplexität gibt es fünf Gruppen von Metriken. Dennoch können alle Metriken im Rahmen der Arbeit nicht behandelt werden. Die Abbildung dient eher der Übersicht. Nur einige Metriken werden beispielhaft vorgestellt. Detailliertere Information zu den Metriken können aus dem [LST] entnommen werden.

Als erste Metriken wurden die Umfangsmetriken eingesetzt, da sie einfache Informationen, wie Größe der Datei, Anzahl der Funktionen, Anzahl der Programmzeilen verwendet haben. Die bekanntesten Metriken dafür sind:

- LOC – Line of code, bezeichnet die Anzahl der Codezeilen im Programm.
- ELOC – Executable line of code, Anzahl der ausführbaren Codezeilen.

Diese Metriken sagen etwas über die Größe des Programms und gar nichts über die Qualität des Quellcodes aus. Bei den logischen Strukturmetriken ist die zyklomatische Zahl von McCabe am Bekanntesten. Als Basis für die Metrik wird der Kontrollflussgraph benutzt. Die Datenstrukturmetriken messen die Anzahl der Variablen, ihre Gültigkeit und Lebensdauer. Stilmetriken messen, ob der Quellcode richtig eingerückt ist oder ob die Namenskonvention eingehalten wurde. Diese Metriken sind schwer objektiv zu messen. Interne Bindungsmetriken messen die syntaktische Bindung durch Prüfungdes Codes jeder Systemkomponente.

Eine andere Art von Metriken sind die Metriken für objektorientierte Systemkomponente. Diese werden benötigt, um objektorientierte Konzepte zu berücksichtigen. Dafür wurden meist die „klassischen“ Metriken modifiziert und neue Metriken entwickelt.

Beispiele dazu:

- WMC - Weighted methods per class, bezeichnet die Anzahl der Methoden in einer Klasse. Dieses Maß ist ein Indikator für Wartungs- und

Weiterentwicklungsaufwand.

- DIT - Depth in inheritance tree, bezeichnet die Anzahl der Oberklassen einer Klasse und ist somit ein Indikator für hohe Wiederverwendbarkeit und gute Strukturierung.
- NOC - Number of children, bezeicnet die Anzahl der direkten Kinder einer Klasse. Es ist ein Indikator für die Wichtigkeit der Klasse, für die Auswirkungen von Änderungen und für den Testaufwand nach den Änderungen.

Es gibt zudem Metriken, wie etwa die Function-Point-Analyse, die für Aufwandsabschätzung in der Analysephase genutzt wird. Es können auch weitere Metriken definiert, von den anderen abgeleitet oder kombiniert werden, mit dem Vorhaben, die Metrik dem Messkriterium genauer anzupassen. Im Allgemeinem sollen Metriken, nach [LST], folgende Güterkriterien erfüllen:

- Objektivität (Intersubjektivität) – Ein Maß ist objektiv, wenn keine subjektiven Einflüsse des Messenden auf die Messung möglich sind.
- Zuverlässigkeit (Messgenauigkeit) – Bei der Wiederholung der Messung unter denselben Messbedingungen werden dieselben Ergebnisse erzielt, d.h. das Maß ist stabil und präzise (zuverlässig).
- Validität (Gültigkeit, Messtauglichkeit) – Die Messergebnisse erlauben einen eindeutigen und unmittelbaren Rückschluss auf die Ausprägung der Kenngröße.
- Normierung – Es gibt eine Skala, auf der die Messergebnisse eindeutig abgebildet werden. Gibt es eine Vergleichbarkeitsskala, dann ist ein Maß normiert.
- Vergleichbarkeit – Kann ein Maß mit anderen in Relation gesetzt werden, dann heißt es vergleichbar.
- Ökonomie – Die Messung muss mit geringen Kosten durchgeführt werden können. Die Ökonomie hängt vom Automatisierungsgrad, der Anzahl der Messgrößen und der Anzahl der Berechnungsschritte ab.
- Nützlichkeit – Werden mit einer Messung praktische Bedürfnisse erfüllt, dann ist ein Maß nützlich.

Bei dieser Arbeit handelt es sich um die Entwicklung zweier Webanwendungen mit dem Einsatz von neuen Technologien. Aufgrund dessen wird der Wert auf den Entwicklungsaufwand und auf die Performance der beiden Anwendungen gelegt. Dafür werden entsprechende Metriken im Weiteren vorgestellt.

5.2 Entwicklungsaufwand

Der Entwicklungsaufwand kann in der Analysephase mit der Function-Point-Analyse

geschätzt werden.

„Die Function-Point-Analyse (engl. Function Point Analysis) ist ein Verfahren zur Bestimmung des fachlich-funktionalen Umfangs einer EDV-Anwendung bzw. eines EDV-Projektes.“ [W05]-20.01.08.

Function-Points werden in der Softwareentwicklung als Basis für Aufwandsschätzung und Benchmarking herangezogen. Dieses Verfahren eignet sich dann, wenn es schon einige Erfahrungswerte aus den vorherigen Projekten vorliegen. Da es zu den in dieser Arbeit entwickelten Webanwendungen keine Erfahrungswerte gibt, kann dieses Verfahren nicht angewendet werden. Zudem haben beide Anwendungen gleiche funktionale Anforderungen. Das heißt, es würden gleiche Function-Points für beide Anwendungen herauskommen müssen. Deswegen wird der Aufwand nicht geschätzt, sondern gemessen. Für die Messung werden die investierten Arbeitsstunden bei der Entwicklung zusammengezählt und in Arbeitstage umgerechnet. Somit wird sich eine Zahl für jede Anwendung ergeben, die später ausgewertet wird.

5.3 Performance

Der Begriff Performance ist nicht eindeutig definiert. Damit wird meistens die Reaktionszeit des Systems auf ein Ereignis bezeichnet. Auch die Begriffe System und Ereignis können unterschiedlich definiert werden. In unserem Fall ist mit dem System die Webanwendung gemeint. Als Ereignis wird eine Benutzeraktivität definiert. Die Reaktionszeit wird im wesentlichen von zwei Faktoren bestimmt vom Ressourcenverbrauch und von der Blockierzeit. Die Ressourcen sind durch die CPU, Datenspeicher, Netzwerk und den Hauptspeicher bestimmt. Jede Ressource verbraucht Zeit, was sich auf die Antwortzeit auswirkt. Da beide Anwendungen auf dem selben Rechner getestet werden, werden sie auch über gleiche Ressourcen verfügen. Es ist zu vermuten, dass die Performancemessung gleiche Ergebnisse bringt. Aber, wenn die Anwendungen diese Ressourcen unterschiedlich verwenden, wird sich das in den Messergebnissen erkennbar machen.

Zur Blockierzeit kommt es durch Verzögerungen bei der Bereitstellung der erforderlichen Ressourcen. Zu den möglichen Gründen hierfür gehören: Konkurrierender Zugriff (viele Benutzer greifen gleichzeitig auf die Webseite zu), Fehlende Verfügbarkeit (Netzwerkverbindung oder Datenbankverbindung ist unterbrochen), Abhängigkeit von anderen Berechnungen.

Bei einer Webanwendung spielen folgende Aspekte eine wichtige Rolle:

- Ladezeit der Seite
- Paralleler Zugriff
- Belastbarkeit

Diese Aspekte können mit Hilfe von speziellen Werkzeugen (sieh Kapitel 8) getestet und gemessen werden. Der parallele Zugriff erfordert keine echten Benutzer, da diese

durch die Tools simuliert werden können. Die Belastbarkeit der Anwendungen kann dann durch spezielle Tests, die im Kapitel 8 detailliert erklärt werden, geprüft werden.

5.4 Zusammenfassung

Aus diesem Kapitel lässt sich schließen, dass als Vergleichsmethode für die beiden Webanwendungen die Metriken am Besten passen. Es musste auch deutlich gemacht werden, dass es viel mehr Metriken gibt, als in diesem Kapitel vorgestellt werden konnten und diese sich auf bestimmte Kriterien einer Software beziehen. Des Weiteren wurden für diese Arbeit ausgewählten Metriken ausführlich vorgestellt.

Wie die Messungen aufgebaut, ausgeführt und ausgewertet werden, wird im Kapitel 8 erklärt. Das nächste Kapitel widmet sich dem Design der Anwendungen.

6 Design der Anwendungen

„Softwarearchitektur wird oft vernachlässigt!"

In diesem Kapitel wird der fachliche Rahmen der beiden Webanwendungen vorgestellt. Bei der Vorstellung der Architektur können die fachlichen Aspekte für beide Anwendungen zusammengefasst präsentiert werden. Die technische Architektur wird dagegen explizit für jede Anwendung dargestellt.

6.1 Einleitung

Als Thema für die Anwendungen wurde die Benutzerverwaltung ausgewählt. Dafür sprechen mehrere Gründe. Erstens durfte die Anwendung nicht zu umfangreich und zu komplex ausfallen, weil es sich bei der Entwicklung um neue Technologien handelt. Zweitens musste die Anwendung trotz der geringen Komplexität die wesentlichen Komponenten einer Webanwendung beinhalten. Drittens sollte die Weiterentwicklung der Anwendungen nicht ausgeschlossen bleiben.

Die zu entwickelnde Software ist keinesfalls vollständig, enthält jedoch folgende Komponenten: Login, Datenbank, Validierung und Navigation. Das sind schon die wesentlichen Komponenten einer echten Webanwendung mit dem Unterschied, dass sie

in ihrer Funktionalität und Komplexität schlanker ausfallen. Zum Beispiel wurde das Login-Verfahren vereinfacht und enthält keine wirkliche Authentifikazion. Beim Login werden die eingegebene Benutzername und Passwort mit denen in der Datenbank verglichen. Dabei liegen die geheimzuhaltende Daten im Klartext in der Datenbank vor. Für die Übertragung der Daten wurde kein HTTPS-Protokoll benutzt, weil das für den eigentlichen Zweck der Arbeit nicht relevant ist. Das HTTPS-Protokoll unterstützt die SSL-Verschlüsselung. Es wurde auch auf die Rechteverwaltung verzichtet, im gesamten System gibt es nur eine Rolle. Der eingeloggte User darf alles, er ist gleichzeitig der Administrator. Jeder Benutzer kann somit neu Benutzer anlegen, Benutzer editieren und löschen. Für eine reale Anwendung macht es wenig Sinn, für eine Testanwendung ist aber durchaus nutzbar.

In der Praxis ist es nicht unüblich, wenn die fachlichen Anforderungen an eine Anwendung in einer Textform vorliegen. Daraus können die zu realisierende Anwendungsfälle abgeleitet werden und mit Hilfe von Use-Cases festgehalten werden.

6.2 Fachliche Architektur

Fachliche Architektur soll dem zukünftigen Anwender auf eine verständliche Weise erklären, was die Anwendung macht.

6.2.1 Überblick

Eine textuelle Beschreibung des Software-Systems könnte dann wie folgt aussehen. Um das System zu betreten, müsste sich der Benutzer vorher anmelden. Wenn die Anmeldung aufgrund eines falschen Passworts oder Benutzernamens fehlschlägt, wird der Eingang verwehrt. Ist der Anmeldevorgang erfolgreich, wird der Benutzer auf die Begrüßungsseite weitergeleitet, wo er dann persönlich mit dem Vornamen begrüßt wird. Der Benutzer hat die Möglichkeit, seine persönlichen Daten zu bearbeiten oder einen neuen Benutzer anzulegen. Diese Änderungen können dauerhaft abgespeichert oder rückgängig gemacht werden. Zudem sieht er die Liste aller registrierten Benutzern in Form einer Tabelle. Dabei kann jeder Benutzer aus der Liste angezeigt, bearbeitet oder gelöscht werden. Zusätzlich muss es die Möglichkeit geben, zu jedem Benutzer ein Kommentar abzugeben. Außerdem muss der Benutzer sich vom System abmelden können.

6.2.2 Use-Cases

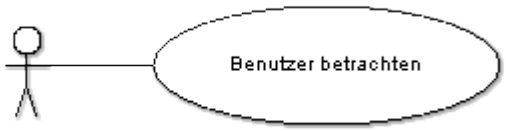
Use-Cases lassen sich auf Deutsch als Anwendungsfall übersetzen und sind ein Teil der Unified Modeling Language (UML). UML ist ein Standard für die Objektorientierte Modellierung. Dieser Standard beinhaltet mehrere Arten an Diagrammen, die ein System auf verschiedene Weisen beschreiben. Das Use-Cases-Diagramm ist ein Beispiel dafür. Mit Hilfe von Use-Cases können die Anwendungsfälle, die später zu realisieren sind, recht anschaulich und wenig techniklastig definiert werden. Sodass sogar die in UML-Notation unerfahrene Anwender damit umgehen können. Es ist daher wichtig, weil die Use-Cases meist in der Zusammenarbeit von Kunden und Entwicklern erarbeitet werden. Use-Cases sind somit ein ideales Werkzeug um die Anforderungen an ein System auf verständliche Weise darzustellen.

Aus der textuellen Beschreibung lassen sich folgende Anwendungsfälle ableiten:

Use-Case-Name	**Anmelden am System**
Vorbedingung	Der Benutzer ist am System nicht angemeldet.
Nachbedingung Erfolg	Benutzer ist mit Namen und Passwort im System angemeldet und befindet sich auf der Welcome-Seite.
Nachbedingung Misserfolg	Benutzer ist im System nicht angemeldet und erhält entsprechende Fehlermeldung. Benutzer befindet sich weiterhin auf der Login-Seite.
Akteure	Benutzer
Beschreibung	Der Benutzer meldet sich durch Eingabe von Name und Passwort an und kann das System benutzen. Bei falscher Anmeldung wird er darüber Informiert.

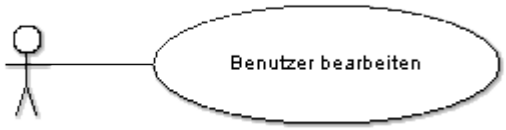
Tabelle 6: Use-Cases: Anmelden am System

Weitere Anwendungsfälle kommen auf der nächsten Seite.



Use-Case-Name	**Benutzerdaten betrachten**
Vorbedingung	Der Benutzer ist am System angemeldet.
Nachbedingung Erfolg	Benutzer kann sich die Benutzerdaten eines anderen Benutzers ansehen. Er kann auch die Kommentare zu dem Benutzer lesen und ein Eigenes abgeben. Benutzer befindet sich auf der Show-Seite.
Nachbedingung Misserfolg	-
Akteure	Benutzer
Beschreibung	Der Benutzer hat die Möglichkeit, die Benutzerdaten und die Kommentare eines anderen Benutzers anzusehen und sein eigenes Kommentar zu diesem Benutzer abzugeben.

Tabelle 7: Use-Cases: Benutzerdaten betrachten

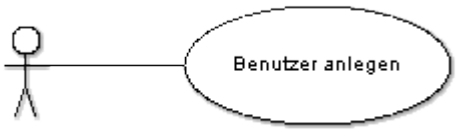


Use-Case-Name	**Benutzerdaten bearbeiten**
Vorbedingung	Der Benutzer ist am System angemeldet.
Nachbedingung Erfolg	Benutzer kann die Benutzerdaten eines anderen Benutzers bearbeiten. Er kann auch seine Kommentare bearbeiten und löschen. Benutzer befindet sich auf der Edit-Seite.
Nachbedingung Misserfolg	-
Akteure	Benutzer
Beschreibung	Der Benutzer hat die Möglichkeit, die Benutzerdaten und die Kommentare eines anderen Benutzers zu bearbeiten.

Tabelle 8: Use-Cases: Benutzerdaten bearbeiten

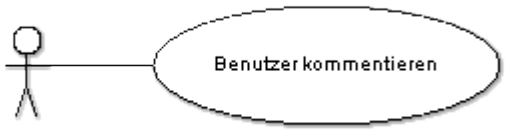
Use-Case-Name	**Benutzer löschen**
Vorbedingung	Der Benutzer ist am System angemeldet.
Nachbedingung Erfolg	Benutzer kann einen beliebigen Benutzer löschen. Der gelöschte Benutzer ist nicht mehr in der Liste aller Benutzer vorhanden.
Nachbedingung Misserfolg	-
Akteure	Benutzer
Beschreibung	Der Benutzer hat die Möglichkeit einen Benutzer zu löschen.

Tabelle 9: Use-Cases: Benutzer löschen



Use-Case-Name	**Benutzer anlegen**
Vorbedingung	Der Benutzer ist am System angemeldet.
Nachbedingung Erfolg	Benutzer kann einen neuen Benutzer anlegen. Der neu Benutzer ist in die Liste aller Benutzer neu hinzugekommen.
Nachbedingung Misserfolg	-
Akteure	Benutzer
Beschreibung	Der Benutzer hat die Möglichkeit einen Benutzer zu erstellen.

Tabelle 10: Use-Cases: Benutzer anlegen



Use-Case-Name	**Benutzer kommentieren**
Vorbedingung	Der Benutzer ist am System angemeldet.
Nachbedingung Erfolg	Benutzer kann zu einem beliebigen Benutzer sein Kommentar abgeben. Dieser Kommentar kann gelesen werden.
Nachbedingung Misserfolg	-
Akteure	Benutzer
Beschreibung	Der Benutzer hat die Möglichkeit zu einem Benutzer ein Kommentar zu schreiben.

Tabelle 11: Use-Cases: Benutzer kommentieren

Use-Case-Name	**Abmelden vom System**
Vorbedingung	Der Benutzer ist am System angemeldet.
Nachbedingung Erfolg	Benutzer kann sich jederzeit vom System abmelden. Er wird automatisch auf die Login-Seite geleitet.
Nachbedingung Misserfolg	-
Akteure	Benutzer
Beschreibung	Der Benutzer hat die Möglichkeit sich vom System abzumelden.

Tabelle 12: Use-Cases: Abmelden vom System

Insgesamt sind sieben Anwendungsfällen entstanden, die im nächsten Kapitel implementiert werden.

6.3 Technische Architektur

Eine Softwarearchitektur ist „*eine strukturierte oder hierarchische Anordnung der Systemkomponenten sowie Beschreibung ihrer Beziehungen“* - [Balzert]. Bei der technischen Architektur ist diese Beschreibung aus der Sicht des Entwicklers zu betrachten. In diesem Abschnitt wird die technische Architektur der beiden Anwendungen beschrieben.

6.3.1 Usim on Rails

„Usim on Rails“ - ist der Name für die erste Webanwendung. Der Name beginnt mit dem, vom Autor erfundenen Wort „Usim“, das aus dem englischen Wort für die Benutzerverwaltung – Usermanagement abgeleitet wurde. Der zweite Teil des Namens ist „on Rails“. Dieser Teil hat den Bezug auf das Framework „Ruby on Rails“, mit dem die Anwendung gebaut wurde. Die zweite Anwendung heißt dementsprechend „Usim on JSF“. So können beide Anwendungen bei der Beschreibung leicht unterschieden werden.

Die technische Architektur von „Usim on Rails“ stellt eine klassische Drei-Schichten-Architektur dar.

„Die dreischichtige Architektur (englisch three tier architecture) ist eine Client-Server-Architektur, die softwareseitig drei Schichten hat.“

Dabei werden drei Schichten unterschieden:

- *„Präsentationsschicht (client tier) − Diese, auch Front-End bezeichnet, ist für die Repräsentation der Daten, Benutzereingaben und die Benutzerschnittstelle verantwortlich“.*
- *„Logikschicht (application-server tier, Businessschicht, Middle Tier oder Enterprise Tier) − Sie beinhaltet alle Verarbeitungsmechanismen. Hier ist sozusagen die Anwendungslogik vereint“.*
- *„Datenhaltungsschicht (data-server tier, back end) − Sie enthält die Datenbank und ist verantwortlich für das Speichern und Laden von Daten“.* [W07]-5.03.08.

Auf die Einzelheiten der Softwarearchitekturen im Allgemeinen wird im Rahmen dieser Arbeit nicht eingegangen. In der Abbildung 9 auf der nächsten Seite ist der Gesamtüberblick dieser Architektur, bezogen auf „Usim on Rails“, dargestellt. Anschließend wird das Zusammenspiel der einzelnen Schichten mit den Rails-Komponenten vorgestellt.

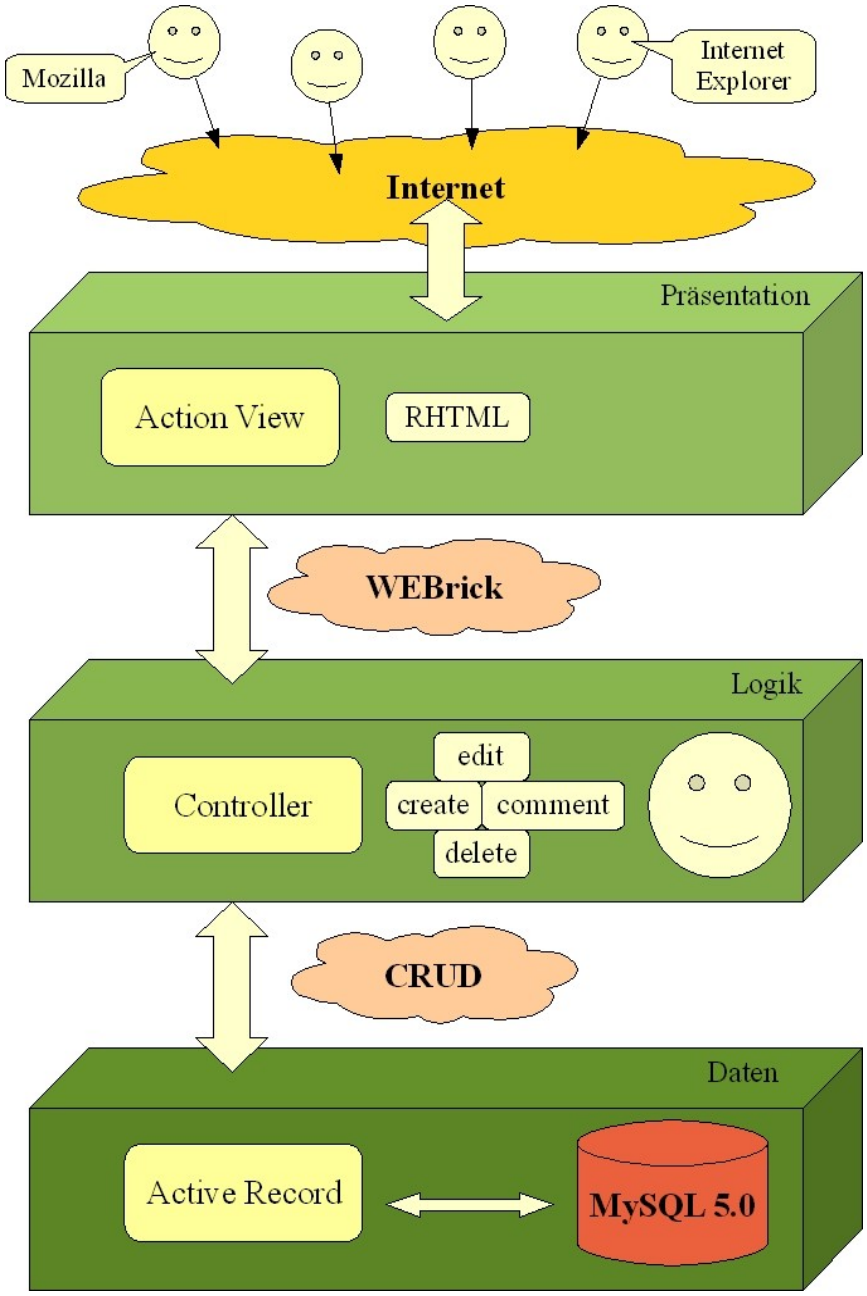
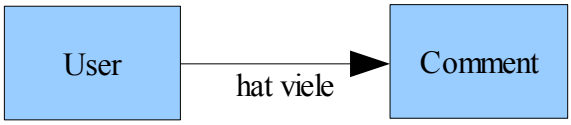


Abbildung 9: Technische Architektur : Usim on Rails

Die Anwender greifen mittels unterschiedlicher Web-Browsers (Internet Explorer, Mozilla, Netscape u.s.w) parallel übers Internet auf die Anwendung zu. Dabei kommunizieren sie nur mit der Präsentationsschicht der Anwendung. Diese kümmert sich um die Bereitstellung der Daten für die Benutzer und nimmt deren Eingabedaten entgegen. Bei Rails ist das Sub-Framework „ActionView“ für die Repräsentation der Daten zuständig. Views werden dabei mit Hilfe von RHTML dargestellt. RHTML

unterscheidet sich vom normalen HTML Format darin, dass zusätzlich der Ruby-Code in die Seite eingebettet werden kann. Die Präsentationsschicht tauscht ihre Daten mit der Logikschicht aus. Dort wird die gesamte Anwendungslogik, die bereits bei der fachlichen Architektur beschrieben wurde, abgewickelt. Die Anwendungslogik wird vom „ActionController“ geregelt. Der Controller kontrolliert das Domain-Modell der gesamten Anwendung. Bei beiden Anwendungen ist das Domain-Modell recht übersichtlich und besteht nur aus zwei Klassen:



Die Klasse „User“ modelliert die Benutzer des Systems. Die Kommentare werden durch die Klasse „Comment“ modelliert. Sie stehen in einer N:1-Beziehung zu den Benutzern, d.h. ein Benutzer kann beliebig viele Kommentare haben und ein Kommentar gehört genau zu einem Benutzer.

Die Wolke zwischen der Präsentationsschicht und der Logikschicht repräsentiert den Web-Server auf dem die gesamte Anwendung läuft. Rails bietet einen eigenen Web-Server namens „WEBrick“ an. Dieser kommt zum Einsatz, um den Konfigurationsaufwand zu minimieren. Einerseits manipuliert die Logikschicht die Daten, andererseits müssen die Daten persistent gehalten werden. Dafür erfolgt die Kommunikation mit der Datenhaltungsschicht mittels „CRUD“ - Operationen, die von Rails zur Verfügung gestellt werden. Die Daten werden in der „MySQL 5.0“ - Datenbank abgespeichert. Wie die Daten abzuspeichern sind, regelt „ActiveRecord“ selbst.

6.3.2 Usim on JSF

Eine Java basierte Webanwendung besteht in der Regel aus einer Kombination aus Servlets und JSP-Seiten. Für die Umsetzung so einer Anwendung kann es jedoch unterschiedliche Architekturen geben. Die Firma Sun hat zwei Architekturmodelle definiert, nach deren Vorgaben ein Entwickler sich bei der Erstellung von Webanwendungen richten kann oder sollte. [JSFAB]

Architekturmodell 1

In diesem Modell gibt es lediglich JSP-Seiten sowie JavaBeans. Jeder Request wird auf eine JSP-Seite weitergeleitet. Dabei gibt es keine zentrale Komponente, die eine Steuerfunktion übernimmt. Jede einzelne Seite ist für die korrekte Navigation zuständig. Die gesamte Anwendungslogik ist entweder in den JavaBeans oder in der JSP-Seite selbst hinterlegt. Dieser Ansatz eignet sich primär für kleinere, überschaubare Anwendungen.

Beim Ablauf eines Requests gelangt dieser an eine JSP-Seite, die bei Bedarf die JavaBeans instanzieren und verwenden kann. Die JavaBeans greifen ggf. auf eine Datenbank zu und liefern die Werte über Beaneigenschaften zur JSP-Seite zurück. Diese wird abgearbeitet und das Ergebnis an den Browser zurückgeliefert.

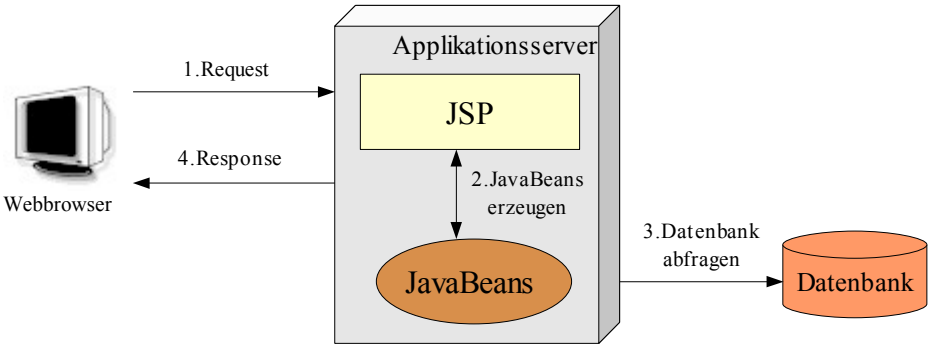


Abbildung 10: Modell-1-Architektur (angelehnt an [JSFAB])

Die Modell-1-Architektur weist einige gravierende Nachteile auf:

- Keine zentrale Stelle für die Navigation. Die Navigation ist auf alle JSP-Seiten verteilt. Bei Änderungen muss jede einzelne Seite angefasst werden.
- Extrem viel Logik ist in der JSP-Seite selbst enthalten.
- Durch zu viel Programmlogik in der JSP-Seite wird es für einen Webdesigner extrem schwierig, selbst Änderungen an der Seite vornehmen zu können.

Viele Prototypen oder kleinere Webanwendungen basieren auf dieser Architektur. Trotz ihrer Mängel kann es im Einzelfall Sinn machen, Anwendungen gemäß Modell 1 aufzubauen.

Architekturmodell 2

Das Architekturmodell 2 eliminiert viele Nachteile des Modells 1. Die Architektur entspricht dem MVC Designpattern. Wenn das klassische MVC auf eine Webanwendung übertragen wird, wird auch von einer Variante des MVC-Musters Model 2 gesprochen.

Model 2 *ist eine für Webanwendungen spezialisierte Variante des Architekturmusters Model-View-Controller (MVC, siehe Kapitel 3). Es ist eine serverseitige Implementierung des MVC-Musters. Sie beschreibt insbesondere die Aufteilung der Anfragebearbeitung in eine Front-Komponente (Controller) und eine Präsentationskomponente (View). HTTP-Anfragen werden von einem Controller entgegengenommen und verarbeitet. Der Controller stellt aufgrund der Anfrage Objekte im Session- oder Request-Kontext bereit und entscheidet, zu welcher Präsentationskomponente er intern weiterleitet. Die Präsentation bezieht ihre Ausgabedaten aus dem Aufruf des Controllers.* [W09]-19.03.08.

Als zentrales Element des Architekturmodells 2 dient ein Servlet, das bei jedem Request angesprochen wird. Es fungiert somit in der Rolle eines Controllers. Im Controller-Servlet erfolgt auch die Steuerung der Navigation. Das Servlet ist auch dafür zuständig, dass die JavaBeans, die in der JSP-Seite benötigt werden, korrekt und vor allem rechtzeitig befüllt werden. Gegebenenfalls finden die Datenbankzugriffe statt. Die JSP-Seite muss sich darum nicht kümmern, sie greift nur auf die Daten der Beans zu.

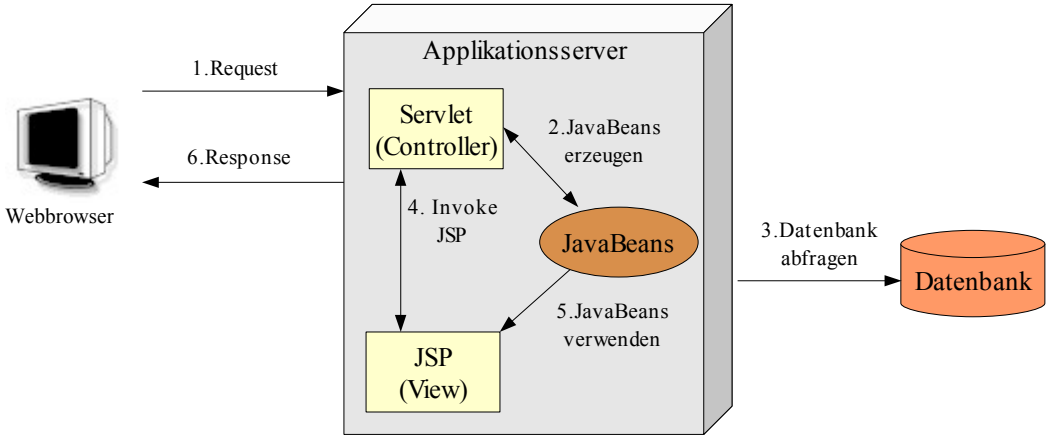


Abbildung 11: Modell-2-Architektur (angelehnt an [JSFAB])

Der Vorteil dieser Architektur liegt hauptsächlich in einer weiteren Modularisierung und Aufteilung der Zuständigkeiten (***separation of concerns***). Viele Frameworks im Java-Umfeld (auch JavaServer Faces) basieren genau auf diesem Ansatz.

Diese Architektur dient als Grundlage für „Usim on JSF“. Zusammen mit den technischen Aspekten ergibt sich folgende Architektur, die in der Abbildung 12 auf der nächsten Seite dargestellt wird.

Zur Darstellung der Daten erzeugt JSF die JSP-View Seiten. In diesen Views können JSF-Komponenten aus verschiedenen Bibliotheken und die standard HTML verwendet werden. In dieser Arbeit werden JSF-Komponenten aus JSTL 1.1.2 und JSF 1.2 verwendet. Die standard HTML wird nur wenig verwendet.

Die Domain-Objekte einer Java-EE-Anwendung werden in der Regel mit Enterprise-JavaBeans implementiert. Für die Anforderungen an „Usim on JSF“ sind keine EJBs notwendig, es reichen die so genannte POJOs (Plain Old Java Objects) aus. Deshalb kann die Anwendung mit POJOs entwickelt werden. Die Rolle des Controllers übernehmen die so genannten Handler. Das sind Java Klassen, die extra implementiert werden müssen. Damit die Kommunikation zwischen den Views und den Handlern statt finden kann, müssen diese im Framework als „Managed Bean“ registriert werden. Die Handler haben die Kontrolle über die POJOs.

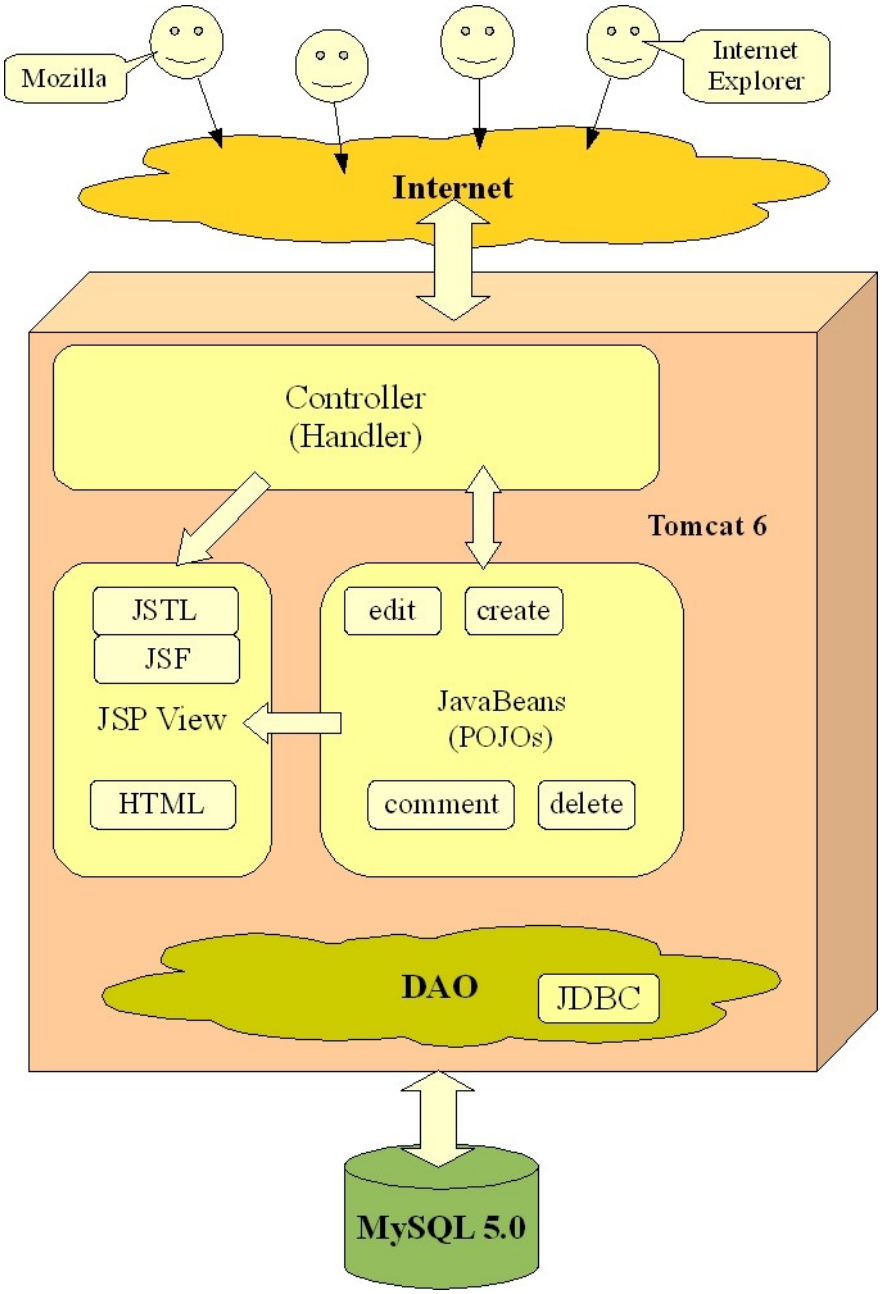


Abbildung 12: Technische Architektur : Usim on JSF

Um die persistenten Anwendungsdaten aus der Datenbank in die POJOs zu laden und auf dem umgekehrtem Wege die Daten in die Datenbank abzuspeichern, wird kein Object-Relational-Mapper eingesetzt. Das geschieht mit Hilfe von JDBC. Um die JDBC-Anfragen von der Anwendungslogik abzukapseln, wird ein „Data Access Object“- Entwurfsmuster angewendet. Wie das genau implementiert ist, wird im nächsten Kapitel erklärt. Analog zu „Usim on Rails“ wird eine „MySQL 5.0“ Datenbank eingesetzt. Da JSF keinen eigenen Web-Server anbietet, wird ein TomCat 6 der Apache Fundation eingesetzt.

6.4 Zusammenfassung

Das Kapitel hat gezeigt, dass beide Anwendungen über eine identische fachliche Architektur verfügen.

Die technische Architektur der beiden Anwendungen ist ähnlich ausfallen, da beide Frameworks auf dem MVC basieren. Wobei Rails diese Architektur vorgibt. Das heißt es würde dem Entwickler schwer fallen nach einer anderen Architektur eine Rails-Anwendung zu bauen. Bei den JSF handelt es sich um eine individuelle Entscheidung des Autors laut Vorgaben von Sun. JSF sind an sich an keine Architekturvorgaben gebunden.

Der wesentliche Unterschied besteht darin, dass „Ruby on Rails“ fast alle benötigten Komponenten außer der Datenbank mit sich bringt. Bei JSF müssen dagegen einige Bestandteile der Architektur explizit implementiert werden, wie Controller und CRUD-Operationen, was mit dem zusätzlichen Aufwand verbunden ist. Wie sich das auf die Implementierung der beiden Anwendungen auswirkt, wird im nächsten Kapitel verdeutlicht.

7 Implementierung

„Just do it!"

Im Rahmen dieses Kapitels werden die wesentlichen Implementierungsschritte der beiden Anwendungen vorgestellt. Gelegentlich wird es Quellcode-Auszüge geben.

Es wird zunächst auf die Entwicklungsumgebung und auf die rechtliche Seite der bei dieser Arbeit eingesetzten Softwarepakete eingegangen, bevor die Realisierung der beiden Anwendungen beschrieben wird.

Der gesamte Quellcode ist auf der beiliegenden CD vorhanden, sowie die Installationsanleitungen zu finden sind.

7.1 Einleitung

Bevor es mit der Implementierung begonnen werden kann, müssen einige Entscheidungen getroffen werden. Die Entscheidungen über die Wahl der Programmiersprachen, der Technologien und über die Softwarearchitektur sind im Verlaufe dieser Arbeit schon gefallen. Offen geblieben ist jedoch die Frage der Entwicklungsumgebung. Auch die rechtlichen Fragen zu der in dieser Arbeit eingesetzten Software, müssen geklärt werden.

7.2 Entwicklungsumgebung

Als Entwicklungsumgebung wird „Eclipse IDE for Java EE Developers“ in der 3.3.2-Version verwendet, die kostenlos von der http://www.eclipse.org/downloads/ heruntergeladen werden kann. Der Hauptgrund für diese Wahl ist folgender. Beide Anwendungen können unter „Eclipse“ entwickelt werden. Für „Ruby on Rails“ gibt es ein Eclipse-Plugin. Für den JSF-Einsatz kann Eclipse um JSF-Funktionalitäten erweitert werden. Die MySQL Datenbank in der 5.0-Version kann kostenfrei von der http://dev.mysql.com/downloads/mysql/5.0.html heruntergeladen werden. Für die Entwicklung der „Usim on JSF“ Anwendung wird ein Web-Server benötigt, der Java unterstützt. In dieser Arbeit wird der TomCat 6.0.13 eingesetzt. Der kann von der http://tomcat.apache.org/download-60.cgi heruntergeladen werden kann. Die Abbildung 13 gibt den Gesamteindruck über die eingesetzte Software.

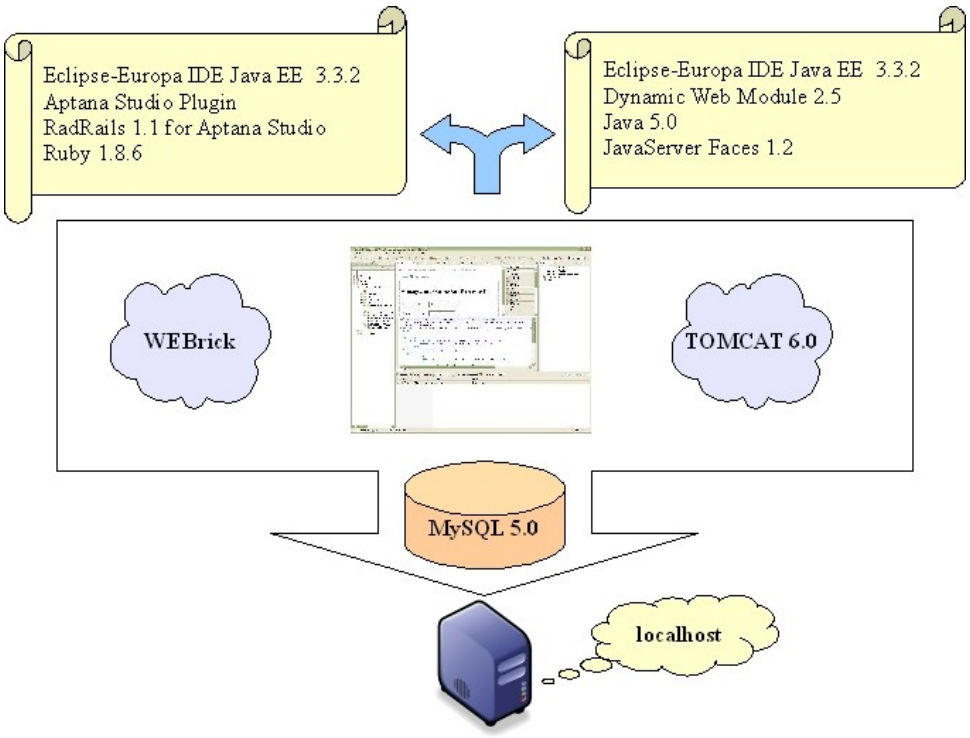


Abbildung 13: Entwicklungsumgebung

7.3 Lizenzen

Alle Softwarepakete, die in dieser Arbeit eingesetzt wurden, sind OpenSource. Diese Software kann kostenfrei installiert werden, unterliegt aber unterschiedlichen Lizenzen. An dieser Stelle wird nicht erläutert, was die Lizenzen erlauben und was sie einschränken, sondern es wird einen Überblick der Softwarelizenzen in Form der Tabelle 13 auf der nächsten Seite geben.

Software	Lizenz
Eclipse	EPL (früher CPL)
Java	BCL
Aptana	Duales Lizenzmodell:GPL v3.0 und Aptana Public Licence v1.0
Rails	MIT-Lizenz
Ruby	GPL und eigene (siehe Lizenz)
Tomcat	Apache-Lizenz (GPL v3.0)
WEBrick	Ruby-Lizenz und GPL
Jmeter	Apache License Version 2.0
Canoo Web Test	Eigene Lizenz auf http://webtest.canoo.com/webtest/manual/license.html
Argo UML	BSD-Lizenz

Tabelle 13: Softwarelizenzen [W]-13.03.08

7.4 Usim on Rails

Bei der Realisierung der Anwendung konnten alle im Kapitel 6.2.2 beschriebenen Anwendungsfälle erfolgreich implementiert werden. Als erstes wurden die Domain-Objekte mittels des Scaffold-Generator (sieh Kapitel 3.2.5) erzeugt. Das erzeugte Modell ist auf der ist in der Abbildung 14 dargestellt:

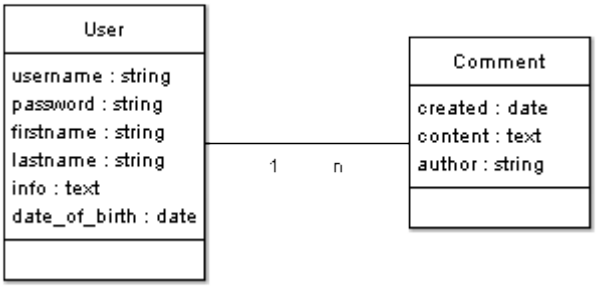


Abbildung 14: Klassendiagramm : Usim on Rails

Auf der Datenbankebene mussten, wie von Rails versprochen wird, keine Restriktionen definiert werden.

Login

Bei der Implementierung der Login-Funktion wurde ein so genannter Filter verwendet. Dieser überprüft vor jeder Action, ob der Benutzer am System angemeldet ist. Die einfachste Möglichkeit, dies zu realisieren, ist die Installation eines Before-Filters in der zentralen Controllerklasse `ApplicationController`, von der alle Controller der Anwendung erben. Der Filter sieht dann so aus:

```
before_filter :authenticate,:except => [:login, :sign_on]
```

Dieser Filter ruft die `authenticate()`-Methode auf, um zu prüfen, ob ein Benutzer in der Session gespeichert bzw. am System angemeldet ist. Wenn kein Benutzer vorhanden ist, wird der Admin-Controller aufgerufen. Dieser Controller ist nur für das Login und Logout zuständig. Dieser Controller hat insgesamt drei Methoden: `login()`, `logout()` und `sign_on()`. Am interessanten ist die `sign_on()`- Methode:

```
def sign_on
  user = User.find(:first, :conditions =>
  ["username = BINARY ? AND password = BINARY ?",
  params[:user][:username], params[:user][:password]])
    if user
      session[:user] = user
      redirect_to(:controller => "users",
      :action => "index")
    else
      render :action => 'login'
    end
end
```

Dabei wird es nach einem Benutzer gesucht, bei dem der Benutzername und das Passwort mit den Eingegebenen übereinstimmt. Wird so ein Benutzer gefunden, dann darf er das System betreten. Er wird in die Session gepackt und die Seite mit dem Überblick aller Benutzer wird angezeigt. Im anderen Fall wird wieder die Login-Seite angezeigt. Da es bei der Anwendung keine Funktion zum Registrieren neuer Benutzer gibt, muss dafür gesorgt werden, dass in der Datenbank mindestens ein Benutzer mit einem „Username“ und „Password“ vorhanden ist. Ansonsten kann das System nicht betreten und nicht benutzt werden.

Logout

Dieser Anwendungsfall der Anwendung war sehr einfach zu implementieren. Zum Ausloggen mussten lediglich die Session geleert `reset_session()` und der Benutzer auf die Startseite weitergeleitet werden.

Benutzer/Kommentare erstellen, anzeigen, bearbeiten und löschen.

Diese Funktionen wurden von Rails automatisch zur Verfügung gestellt und konnten zum größten Teil unverändert übernommen werden. Die Änderungen haben in den automatisch erstellten Views stattgefunden. Das gleiche Vorgehen galt für die Implementierung der Kommentaren.

Benutzer kommentieren

Für das Einfügen von Kommentaren wurde das Modell um die 1:N-Beziehung zwischen einem Benutzer und seinen Kommentaren ergänzt. Dazu hat die Comment-Klasse ein zusätzliches `user_id` Attribut und eine `belongs_to :user` - Deklaration definiert bekommen. Die User-Klasse wurde um die `has_many :comments`,

`:dependent => :destroy` – Deklaration erweitert. Das Einfügen eines Kommentars wurde im `UsersController` programmiert. Dort wurde die `add_comments()` – Methode implementiert. Ihr Code sieht dann so aus:

```
class UsersController < ApplicationController
      def add_comment
            user = User.find(params[:id])
            @comment = Comment.new(:user => user)
            render :template => 'comments/edit'
      end
      …
end
```

Dabei wird der zu kommentierende Benutzer durch die `find()`-Methode anhand der ID in die Variable `user` geladen. Danach wird der Comment-Konstruktor zum Erzeugen eines neuen Comment-Objektes aufgerufen. Diesem Comment-Objekt wird gleich im Konstruktor der geladene Benutzer übergeben. Dadurch weiß das Kommentar-Objekt zu welchem Benutzer es gehört. Schließlich wird der Edit-View zum Eingeben des Kommentartextes gerendert.

7.5 Usim on JSF

Bei der Realisierung der Anwendung wurden nicht alle im Kapitel 6.2.2 beschriebene Anwendungsfälle im vollem Umfang implementiert. Auf Grund eines höheren Konfigurationsaufwands bei der Einrichtung der Entwicklungsumgebung, mussten die Anwendungsfälle, die sich in ihrer Fachlichkeit ähneln, zusammengeführt werden. Dazu gehören das Anzeigen und Bearbeiten der Benutzerdaten. Beim Anzeigen der Stammdaten können diese gleichzeitig geändert werden. Trotzdem wird die Anwendung durch diese Änderung keinesfalls in ihrer Funktionalität eingeschränkt.

Im Gegensatz zu „Usim on Rails“ musste die gewünschte 1:N-Beziehung zwischen einem Benutzer und seinem Kommentar auch auf der Datenbank-Ebene definiert werden. Dazu wurden in der Tabelle „Comments“ benötigten Fremdschlüsseln definiert. Auf der Modellebene wurde die Beziehung durch die Aggregation realisiert. Das dazugehörige Klassendiagramm ist in der Abbildung 15 auf der nächsten Seite dargestellt.

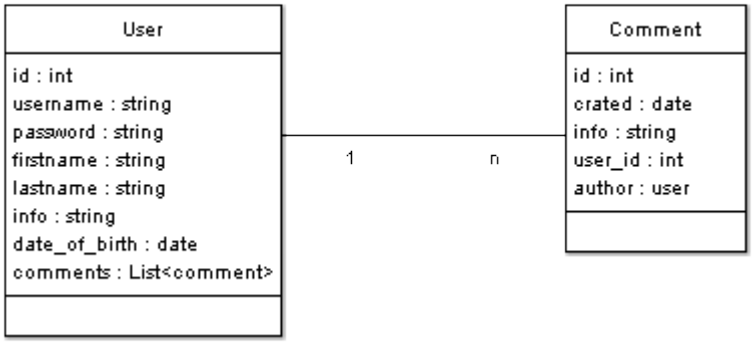


Abbildung 15: Klassendiagramm: Usim on JSF

Ein weiterer Unterschied zu „Usim on Rails“ ist die Verwendung eines einzigen Controllers für die gesamte Anwendung. Dieser verwaltet beide Domain-Objekte, kommuniziert mit mehreren JSP-Views und ist für das Login und Logout zuständig. Der Controller ist als „UserHandler“ in der Abbildung 16 im Zusammenhang mit anderen Objekten dargestellt:

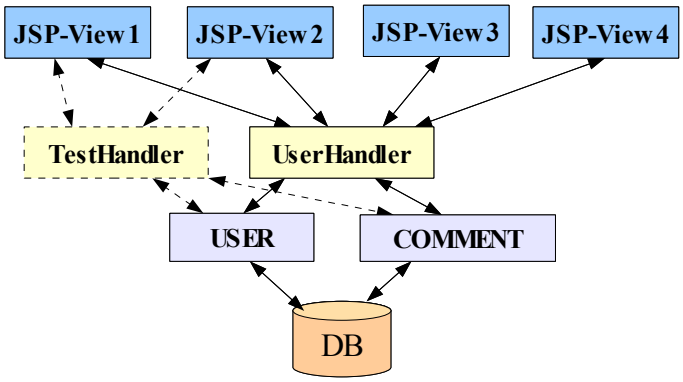


Abbildung 16: Einordnung der Objekte

Auf Grund der geringen Komplexität der Anwendung wurde die Entscheidung getroffen einen Controller zu verwenden. Für größere Projekte wäre das Konzept zu unflexibel. Als Beispiel für eine Architektur mit mehreren Controllern, ist der „TestController“ mit gestrichelten Linien dargestellt. Dieser wird nur im Testteil der Anwendung verwendet.

Login

Für die Login-Funktion wurde keine Session verwendet, sondern nur ein boolesches Flag. Dieser wird auf True gesetzt, wenn die Kombination des Passworts und Benutzernamens in der Datenbank gefunden wurde. Auch hier muss zum Betreten des Systems mindestens ein Benutzer in der Datenbank existieren.

Logout

Beim Ausloggen wird das Flag auf False gesetzt und der Benutzer auf die Startseite weitergeleitet.

Benutzer/Kommentare erstellen, anzeigen, bearbeiten und löschen.

An dieser Stelle musste ein ordentlicher Programmieraufwand geleistet werden. Im Vergleich zu „Usim on Rails“, wobei keine einzige Codezeile geschrieben wurde, mussten die CRUD-Operationen für beide Domain-Klassen, sowie einige Hilfsmethoden sorgfältig programmiert werden. Diese Angelegenheit erwies sich als fehlerträchtig. Um etwas Ordnung in den Code einzubringen, Fehlerquellen zu lokalisieren und die Trennung zwischen der Logikschicht und der Datenhaltungsschicht deutlich zu halten, wurde versucht das DAO-Pattern zu realisieren.

Data Access Object *(DAO, deutsch: „Datenzugriffsobjekt“) ist ein Entwurfsmuster, das den Zugriff auf unterschiedliche Arten von Datenquellen (z. B. Datenbanken, Dateisystem, etc.) so kapselt, dass die angesprochene Datenquelle ausgetauscht werden kann, ohne den aufrufenden Code zu ändern. Dadurch soll die eigentliche Programmlogik von technischen Details der Datenspeicherung befreit werden und flexibler einsetzbar sein.* [W08]-14.03.08.

Dabei werden eigentlichen SQL-Anfragen in einem DAO-Objekt gekapselt. So, dass der Controller mit SQL nichts mehr zu tun hat. In der Anwendung wurden insgesamt zwei DAO-Klassen implementiert UserDAO und CommentDAO. Mit dem folgenden Code-Beispiel sollte die Verwendung eines DAO verdeutlicht werden.

```
publik  String save() throws SQLException {
        new UserDAO().saveOrUpdate(user);
        ...
        return "welcome";
        }
```

Die `save()`-Methode des UserHandlers speichert einen Benutzer in der Datenbank ab. Sie ruft intern die `saveOrUpdate()`-Methode vom UserDAO auf. Dabei bleiben SQL-Anfragen für den Controller verborgen.

Benutzer kommentieren

Beim Einfügen von Kommentaren zu dem Benutzer mussten folgende Fälle berücksichtigt werden. Zuerst mussten im Kommentar die Informationen über seinen Autor und über den Benutzer, zu dem er geschrieben wurde, gesetzt werden. Dann musste der Kommentar in die Kommentarliste des Benutzers eingefügt und anschließend in die Datenbank gespeichert werden.

7.6 Qualitätssicherung

Sicherlich ist die Qualitätssicherung in einem Kundenprojekt sehr wichtig. Es wird oft ein großer Testaufwand getrieben, um den gewissen Grad an Softwarequalität für die

entwickelte Software zu erreichen. Bei einem iterativen Entwicklungsprozess werden oft die Regressionstest in Form von Unit Tests eingesetzt. Bei den größeren Projekten können auch Dienstleister im Bereich der Softwarequalität dazu geholt werden. Große Firmen haben eine eigene Abteilung für die Qualitätssicherung.

Ziel dieser Arbeit konnte nicht sein, eine hoch qualitative Software zu entwickeln. Es wurde beabsichtigt eine Testsoftware zu erstellen. Daher musste in der Entwicklungsphase kein wirklicher Testaufwand getrieben werden. Dennoch wurden beide Anwendungen vom Autor während der Entwicklung gegen allen Anwendungsfällen getestet.

Ohne der Sicherheit, dass die Anwendungen den nötigen Grad der Fehlerfreiheit erfüllen, könnte der Vergleich der Anwendungen gar nicht statt finden. Denn die Messergebnisse wären dadurch verfälscht.

7.7 Zusammenfassung

Beide Webanwendungen wurden gemäß der entworfenen Anwendungsfällen mit wenigen Abweichungen implementiert. Es wurde eine webbasierte Benutzerveraltung mit dem Einsatz unterschiedlicher Technologien fertiggestellt. Zum Teil wurden unterschiedliche Programmiertechniken für die gleiche Aufgaben verwendet, wie das Login und das DAO-Pattern.

Es wurde die Erkenntnis gewonnen, dass eine Software mit unterschiedlichen Technologien erfolgreich erstellt werden kann. Welche der beiden Technologien die bessere Wahl für diese Art der Anwendungen ist, wird sich anhand der Messergebnissen aus dem nächsten Kapitel zeigen.

Letztendlich hat der Erfolg bei der Entwicklung die vorausgesetzte Grundlage für den geplanten Vergleich der beiden Anwendungen geschaffen. Diesem Vergleich widmet sich das folgende Kapitel 8.

8 Versuch

„Wo zwei wetten, muss einer verlieren." [W13] - (22.03.08)

Im diesem Kapitels wird beschrieben, wie die beiden ausgewählten Metriken auf die entwickelte Software angewendet wurden. Zuerst wird der Entwicklungsaufwand aufgezeichnet. Dabei wird der gesamte Ablauf der Aufwandsermittlung vorgestellt und mit einem Fazit abgeschlossen. Danach wird über die Performance der Anwendungen berichtet. Dabei werden der Vergleichsaufbau, die Vergleichsdurchführung und die Vergleichsauswertung beschrieben. Bei dem Vergleichsaufbau wird die dafür geschaffene Testumgebung mit den eingesetzten Werkzeugen vorgestellt.

8.1 Entwicklungsaufwand

Um den Entwicklungsaufwand messen zu können, muss zuerst eine Messeinheit definiert werden. Wie es im Kapitel 5.2 erwähnt wurde, wird dafür als kleinste Einheit eine Arbeitsstunde gewählt. Die Arbeitsstunden werden in die Arbeitstage umgerechnet. Bei der Umrechnung wird es von einem Arbeitstag ausgegangen, der 8 Arbeitsstunden beträgt. Die Anzahl der Arbeitstage wird als Vergleichsgröße für den Entwicklungsaufwand in die Ergebnisse eingehen.

Zur Kontrolle der Angemessenheit der Ergebnisse wird eine obere Schranke festgelegt.

Das heißt die Summen beider ermittelten Aufwänden darf diese Schranke nicht übersteigen. Zum Festlegen der oberen Schranke kann die Bearbeitungszeit für diese Arbeit als Basis betrachtet werden. Daraus wird der tatsächlich investierte Aufwand für die Arbeit entnommen. Dieser Gesamtaufwand kann zunächst als die obere Schranke gesetzt werden. Sie kann wie folgt ausgerechnet werden.

Die Bearbeitungszeit für die Arbeit betrug insgesamt 6 Monate. Die investierte Zeit während der Bearbeitungszeit betrug 20 Stunde in der Woche. Das ergibt 80 Stunden im Monat. 80 Stunden mal 6 Monate ergeben 480 Stunden. Das ist der Aufwand, der insgesamt für diese Arbeit geleistet wurde. Das ist gleichzeitig die gesuchte obere Schranke.

Folglich werden die Ergebnisse der Ermittlung aller Aufwände vorgestellt. Dabei wird der Entwicklungsaufwand in zwei Entwicklungsphasen unterteilt. Eine Phase ist die Einarbeitung in die Technologie und die zweite Phase ist die Implementierung. Anschließend wird die Ermittlung selbst beschrieben.

8.1.1 Ergebnisse

Die Ermittlung der Aufwände hat folgende Ergebnisse erbracht, die in der Abbildung 17 grafisch dargestellt sind:

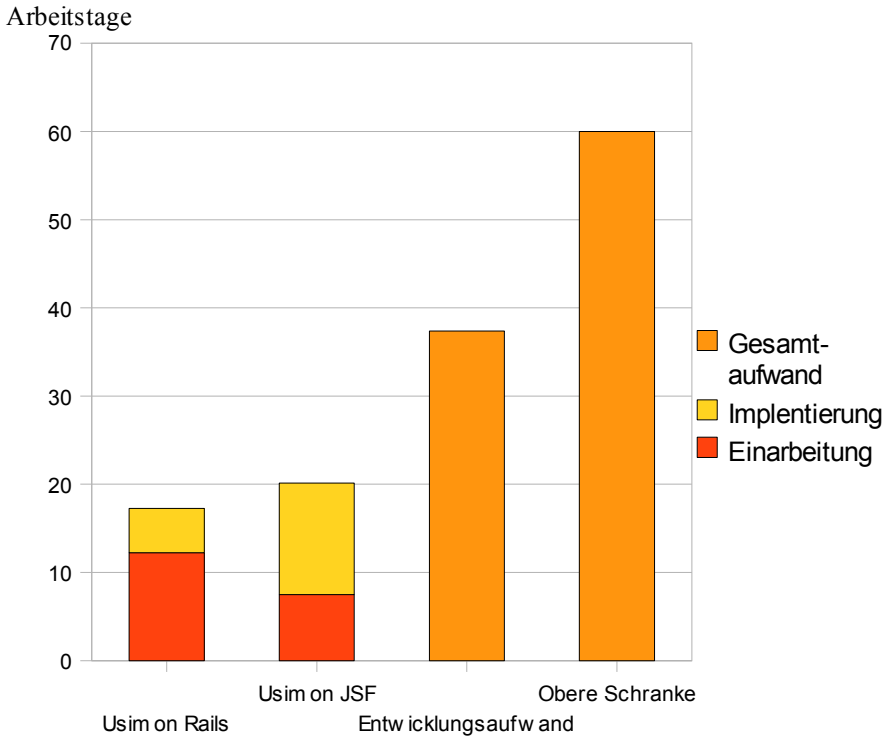


Abbildung 17: Aufwände

Zur Übersichtlichkeit sind die Ergebnisse in der Tabelle 14 zusammengefasst:

	Usim on Rails	**Usim on JSF**
Einarbeitung	98 / 12,25	60 / 7,5
Implementierung	40 / 5,00	101 / 12,625
Entwiklungsaufwand	138 / 17,25	161 / 20,125
Entwicklungsaufwand gesamt	299 / 37,375	
Ober Schranke	480 / 60	
Einheiten	Stunden / Arbeitstage	

Tabelle 14: Entwicklungsaufwand – Ergebnisse

8.1.2 Ermittlung

Usim on Rails

Da „Ruby on Rails“ für den Autor völlig neu war, musste relativ viel Zeit in die Einarbeitung investiert werden. Im Prinzip waren es drei neue Sachen: eine neue Art der Anwendung (Webanwendung), eine neue Programmiersprache (Ruby) und ein neues Framework (Ruby on Rails). Das hat dazu geführt, dass die Einarbeitung 98 Stunden in Anspruch genommen hat. Dafür ging es mit der Implementierung schnell voran. Insgesamt wurden dafür 40 Stunden benötigt.

Usim on JSF

Bei der Implementierung von „Usim on JSF“ konnte ein Teil der gesammelten Erfahrung aus „Usim on Rails“ mitgenommen werden. Die Erfahrung im Bereich der Webanwendung hatte einen positiven Einfluss auf den Aufwand für die Einarbeitung in die „JavaServer Faces“ Technologie bewirkt. Die Programmiersprache Java benötigte keine Einarbeitung. Diese Tatsachen haben den Einarbeitungsaufwand auf 60 Stunden verringert. Für die Implementierung waren 101 Stunden notwendig.

Gesamtaufwand

Zur Ermittlung des Gesamtaufwands wurden die Aufwände aus den beiden Entwicklungsphasen zusammenaddiert. Für „Usim on Rails“ hat sich der Gesamtaufwand von 138 Stunden ergeben, was in Arbeitstagen einen Wert von 138 / 8 = 17,25 ergibt. Für „Usim on JSF“ beträgt der Gesamtaufwand 161 Stunden, was umgerechnet 161 / 8 = 20,125 Arbeitstage ergibt.

Obere Schranke

Die obere Schranke in den Arbeitstagen ergibt 480 / 8 = 60.

8.1.3 Fazit

Aus der Abbildung 17 lässt sich ablesen, dass die Entwicklung der „Usim on Rails“ insgesamt weniger Aufwand benötigt hat. In Anbetracht der beiden Entwicklungsphasen zusammen, ist die Differenz nicht allzu groß. Von der besonderen Bedeutung ist die Tatsache, dass die Implementierungphase der „Usim on Rails“ deutlich kürzer als bei „Usim on JSF“ ausgefallen ist. Für die Real-Projekte bedeutet das, dass mit „Ruby on Rails“ eine Webanwendung sehr schnell und effizient erstellt werden kann. Vor allem die Erstellung von Prototypen geht sehr schnell. Das Ergebnis unterstreicht die Agilität von „Ruby on Rails“.

Obwohl die Einarbeitungsphase in JSF deutlich kürzer war als bei Rails, ist der Entwicklungsaufwand von „Usim on JSF“ größer ausgefallen. Das liegt an der längeren Implementierungsphase, was nicht bedeuten muss, dass die Entwicklung mit „JSF“ umständlicher ist. Der größere Aufwand ist darauf zurückzuführen, dass bei der Implementierung von „Usim on JSF“ zum Einen auf das ORM verzichtet wurde, zum Anderen ein größerer Konfigurationsaufwand zu leisten war.

Die ermittelten Ergebnisse können anhand der Tatsache, dass die obere Schranke nicht überschritten wurde, als angemessen betrachtet werden.

8.2 Performance

Wie es im Kapitel 5 schon beschrieben wurde, werden beide Webanwendung in den folgenden Aspekten getestet:

- Ladezeit der Seite
- Paralleler Zugriff
- Belastbarkeit

Alle diese Aspekte sind einem Leistungstest zuzuordnen.

„Unter dem Oberbegriff Leistungstest kann man den Massentest, den Zeittest, den Lasttest und den Stresstest unterordnen.“[LST]

„Beim Massentest werden die verarbeitbaren Datenmengen, beim Zeittest die Einhaltung von Zeitrestriktionen getestet.“[LST]

„Der Lasttest hat das Ziel, das System im erlaubten Grenzbereich auf Zuverlässigkeit zu Testen.“[LST]

„Beim Stresstest werden die definierten Grenzen des Systems bewusst überschritten.“[LST]

Da für die entwickelten Anwendungen keine Grenzbereiche definiert wurden, kann es beim Testen nicht unterschieden werden, ob die Messergebnisse im erlaubten Bereich

liegen oder diesen überschreiten. Deshalb können der Lasttest und der Stresstest zusammengefasst werden.

Wie jeder Bereich zu testen ist, wird im Versuchsaufbau beschrieben.

8.2.1 Aufbau

Testumgebung

Bei dem Aufbau der Testumgebung musste aus technischen Gründen auf die Onlinestellung der Anwendungen verzichtet werden. Die technischen Kapazitäten haben auch nicht ausgereicht ein Netzwerk aufzubauen. Deswegen wurden alle Tests auf dem localhost durchgeführt.

In der Abbildung 18 ist die tatsächlich aufgebaute Testumgebung mit den eingesetzten Testwerkzeugen („JMeter“ und „Web Test“ mehr dazu später) dargestellt. Die gewünschte Testumgebung ist oberhalb der gestrichelten Linie zu sehen.

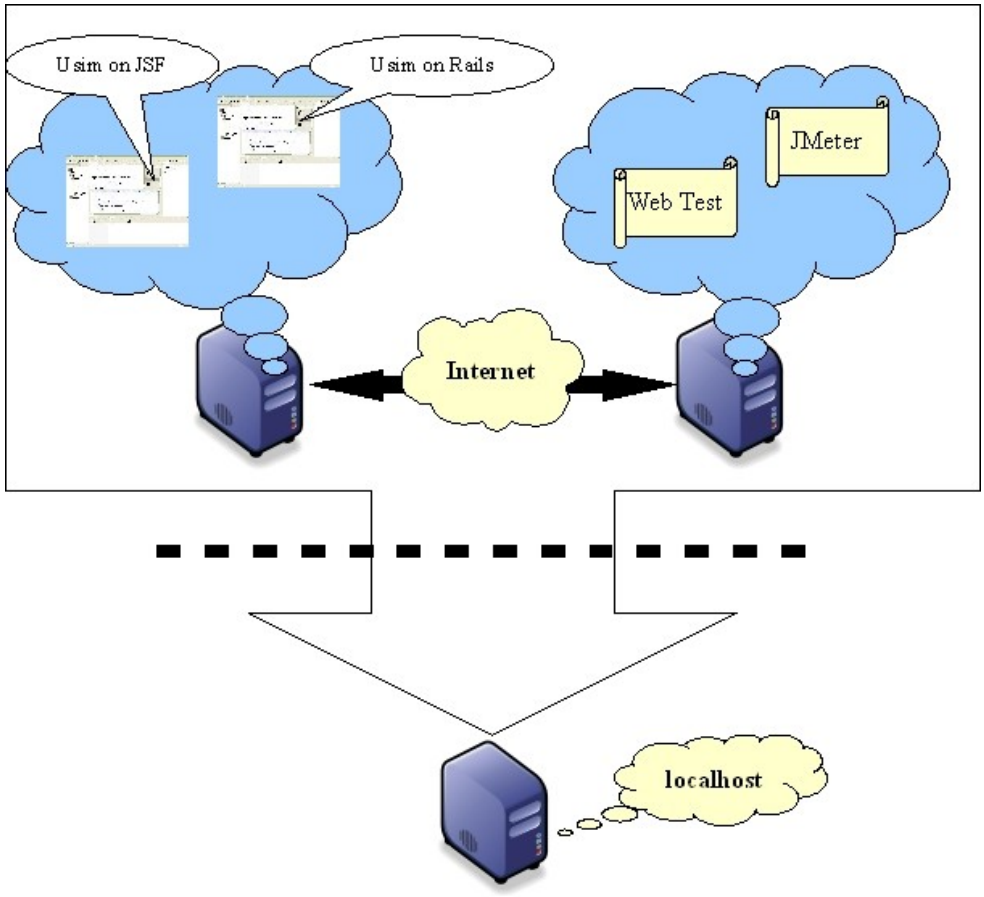


Abbildung 18: Testumgebung

Für einen Leistungstest hat es seine Nachteile. Wenn die Testwerkzeuge und die zu testende Software sich auf einem Rechner befinden, kann nicht mehr unterschieden werden, welche dieser Komponenten, zu wie viel Prozent ausgelastet sind. Beispielsweise, eine Software „A“ wird mit einem Testwerkzeug „B“ getestet. Die Tests zeigen, dass kein Datenfluss statt findet. Im diesem Fall ist es unklar ob „A“ oder „B“ an seine Grenzen gestoßen ist. Um diese Problematik zu beheben, müssten die Testkomponenten auf verschieden Rechner verteilt werden.

Ladezeit der Seite

Die Ladezeit einer Webseite kann ermittelt werden, indem die Zeit zwischen der an den Server gestellten Anfrage und der vom Server geschickten Antwort gemessen wird. Eine Möglichkeit der Messung wäre so ein Test selbst zu programmieren. Andere Möglichkeit, ist dafür ein Tool zu verwenden. In dieser Arbeit wurde das Tool „JMeter“ verwendet.

JMeter ist ein in Java geschriebenes OpenSource Werkzeug der Apache Fundation. Das Tool kann kostenfrei von der http://jakarta.apache.org/jmeter/ heruntergeladen werden. Auf der Webseite sind ausführliche Tutorials zur Benutzung des Tools vorhanden. Deshalb wird auf die Benutzung des Tool nicht weiter eingegangen.

Für diesen Test werden die HTTP-Requests mittels „JMeter“ erzeugt, die die Startseite und die Seite mit der Übersicht aller Benutzer aufrufen.

Paralleler Zugriff

Für diesen Test werden die parallele Zugriffe auf die Startseite und die Benutzerseite mit 10, 50, 100 und 200 Benutzern ausgeführt. Diese Benutzer werden mit Hilfe von „JMeter“ in Form der Threds simuliert.

Dieser Test ist ähnlich, wie der Test zur Ermittlung der Ladezeiten der Seite aufgebaut. Im Tool wird eine bestimmte URL der Webanwendung angegeben. Diese URL wird entsprechend der Benutzeranzahl oft aufgerufen.

Belastbarkeit

Die Belastbarkeit eines System kann mit einem Lasttest getestet werden. Entsprechend dem Massentest wird getestet, welche Mengen an Daten vom System verarbeitet werden können. Dafür muss eine Menge an Testdaten erzeugt werden. Kleinere Mengen an Testdaten könnten per Hand aufgenommen werden. Bei den großen Datenmengen macht es mehr Sinn die Testdaten automatisiert zu erzeugen.

Für diesen Zweck wurden in beide Anwendungen Scripte eingebaut, die jeweils 10, 100 und 1000 Benutzer mit Testdaten füllen und in die Datenbank speichern. Im ersten Testteil werden die Benutzer ohne Kommentare eingefügt. Im zweiten Testteil werden die Benutzer mit jeweils 10 und 100 Kommentaren eingefügt. Die Testscripte werden durch in die Seite eingebaute Buttons gestartet. Die Klicks auf die Button werden mit Hilfe eines OpenSource Tool „Web Test“ von der Firma „Canoo“ ausgeführt. Das Tool kann von der http://webtest.canoo.com/webtest/manual/Downloads.html heruntergeladen werden. Mit dem Tool kann eine beliebige Abfolge des Klicks auf einer Webseite mittels XML definiert werden. So werden gewünschte Kombinationen

an Testbenutzern erzeugt. Die Zeiten zwischen den Klicks werden vom Tool gemessen und die Ergebnisse im HTML Format ausgegeben. Da im Test auch die Zeiten gemessen werden, kann dieser zu dem Zeittest zugeordnet werden.

8.2.2 Durchführung

Für die Durchführung des Versuchs mussten noch die Restriktionen, die für den eindeutigen Benutzernamen sorgen, aus den Anwendungen herausgenommen werden. Der Grund dafür waren die Testdaten bei denen die Testbenutzer alle den gleichen Benutzernamen haben.

Ladezeit der Seite

Im ersten Teil des Tests wurde die Startseite der beiden Anwendungen aufgerufen. Im zweiten Teil wurde die Benutzerseite aufgerufen. Auf dieser Seite sind alle Benutzer der Anwendung aufgelistet. Für den Testlauf wurden 100 Benutzern mit jeweils 10 Kommentaren erzeugt. Die Ergebnisse der Testläufe sind in den Tabellen 15 und 16 dargestellt. Für alle Tests sind drei Werte angegeben. An der ersten Stelle steht der durchschnittliche Wert (avg), dann der minimale Wert (min) und der maximale Wert (max). Bei der Auswertung werden nur die Durchschnittswerte berücksichtigt.

Anwendung	**Usim on Rails**			**Usim on JSF**		
Anzahl der Wiederholungen	1000			1000		
Fehler %	0 %			0 %		
Ergebnisse (avg/min/max) msec	**95**	62	234	**9**	0	47

Tabelle 15: Ladezeit der Startseite

Anwendung	**Usim on Rails**			**Usim on JSF**		
Anzahl der Wiederholungen	100			100		
Fehler %	0 %			0 %		
Ergebnisse (avg/min/max) msec	**1187**	984	1469	**168**	141	2109

Tabelle 16: Ladezeit der Benutzerseite

Paralleler Zugriff

Bei dem Test wurden die Zugriffe auf die Startseite und auf die Benutzerseite ausgeführt. Dabei wurden die Testläufe mit einer unterschiedlichen Anzahl der simulierten Benutzer durchgeführt. Die Testläufe sind mit einer Nummer gekennzeichnet und in der Tabellen 17 und 18 dargestellt.

Anwendung		**Usim on Rails**			**Usim on JSF**		
1	Anzahl der simulierten Zugriffe	10			10		
	Anzahl der Wiederholungen	100			100		
	Fehler %	0 %			0 %		
	Ergebnisse (avg/min/max) msec	**1278**	359	1500	**118**	0	875
2	Anzahl der simulierten Zugriffe	50			50		
	Anzahl der Wiederholungen	10			10		
	Fehler %	0 %			0 %		
	Ergebnisse (avg/min/max) msec	**6451**	766	7235	**460**	0	890
3	Anzahl der simulierten Zugriffe	100			100		
	Anzahl der Wiederholungen	1			1		
	Fehler %	18 %			0 %		
	Ergebnisse (avg/min/max) msec	**6000**	859	12031	**1644**	62	2219
4	Anzahl der simulierten Zugriffe	200			200		
	Anzahl der Wiederholungen	1			1		
	Fehler %	66,5 %			29,5 %		
	Ergebnisse (avg/min/max) msec	**2640**	844	9407	**4475**	961	2368

Tabelle 17: Paralleler Zugriff auf die Startseite

Anwendung		**Usim on Rails**			**Usim on JSF**		
5	Anzahl der simulierten Zugriffe	10			10		
	Anzahl der Wiederholungen	10			10		
	Fehler %	0 %			0 %		
	Ergebnisse (avg/min/max) msec	**11305**	1594	12141	**1530**	484	1781
6	Anzahl der simulierten Zugriffe	50			50		
	Anzahl der Wiederholungen	10			10		
	Fehler %	0 %			0 %		
	Ergebnisse (avg/min/max) msec	**33529**	2204	66047	**8549**	1078	20860
7	Anzahl der simulierten Zugriffe	100			100		
	Anzahl der Wiederholungen	1			1		
	Fehler %	12 %			0 %		
	Ergebnisse (avg/min/max) msec	**54954**	891	124719	**11498**	2641	18516
8	Anzahl der simulierten Zugriffe	200			200		
	Anzahl der Wiederholungen	1			1		
	Fehler %	68 %			30 %		
	Ergebnisse (avg/min/max) msec	**17765**	906	96547	**11597**	954	26547

Tabelle 18: Paralleler Zugriff auf die Benutzerseite

Belastbarkeit

Alle Tests wurden 10 Mal wiederholt. Zur Unterscheidung sind einzelne Tests mit einer Testnummer gekennzeichnet. Die Ergebnisse der Messungen sind in der Tabelle 19 dargestellt.

Anwendung		**Usim on Rails**			**Usim on JSF**		
1	Anzahl der simulierten Benutzer in DB	10			10		
	Anzahl der Kommentare pro Benutzer	0			0		
	Ergebnisse (avg/min/max) msec	**808**	656	907	**298**	250	328
2	Anzahl der simulierten Benutzer in DB	100			100		
	Anzahl der Kommentare pro Benutzer	0			0		
	Ergebnisse (avg/min/max) msec	**4017**	3797	4344	**2070,2**	1750	2422
3	Anzahl der simulierten Benutzer in DB	1000			1000		
	Anzahl der Kommentare pro Benutzer	0			0		
	Ergebnisse (avg/min/max) sec	**35,7**	35	37	**19,8**	18	22
4	Anzahl der simulierten Benutzer in DB	10			10		
	Anzahl der Kommentare pro Benutzer	10			10		
	Ergebnisse (avg/min/max) msec	**3617,9**	3469	3844	**3390,6**	3109	3734
5	Anzahl der simulierten Benutzer in DB	100			100		
	Anzahl der Kommentare pro Benutzer	10			10		
	Ergebnisse (avg/min/max) sec	**34,9**	33	37	**31,07**	30	34
6	Anzahl der simulierten Benutzer in DB	1000			1000		
	Anzahl der Kommentare pro Benutzer	10			10		
	Ergebnisse (avg/min/max) min	**> 5**	> 5	> 5	**> 5**	> 5	> 5
7	Anzahl der simulierten Benutzer in DB	10			10		
	Anzahl der Kommentare pro Benutzer	100			100		
	Ergebnisse (avg/min/max) sec	**30,9**	29	33	**60**	32	97
8	Anzahl der simulierten Benutzer in DB	100			100		
	Anzahl der Kommentare pro Benutzer	100			100		
	Ergebnisse (avg/min/max) min	**> 5**	> 5	> 5	**> 5**	> 5	> 5

Tabelle 19: Belastbarkeit

8.2.3 Auswertung

Die Testergebnisse werden jeweils für einen Testbereich in einem Diagramm dargestellt und ausgewertet.

Ladezeit

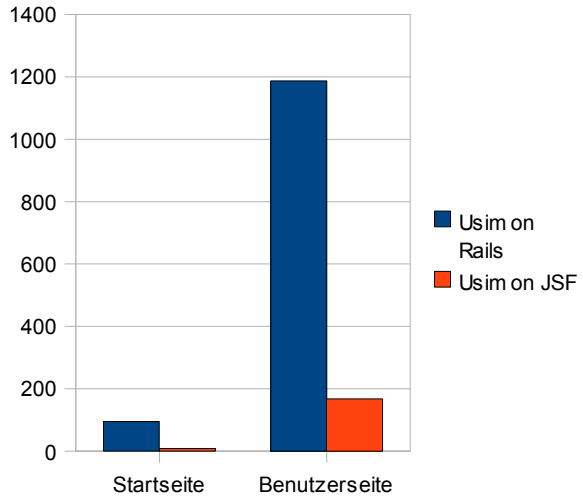


Abbildung 19: Ladezeit : Auswertung

Die Tests haben gezeigt, dass „Usim on JSF“ deutlich kürzere Ladezeiten, wie für die Startseite sowie auch für die Benutzerseite, erzielt.

Paralleler Zugriff

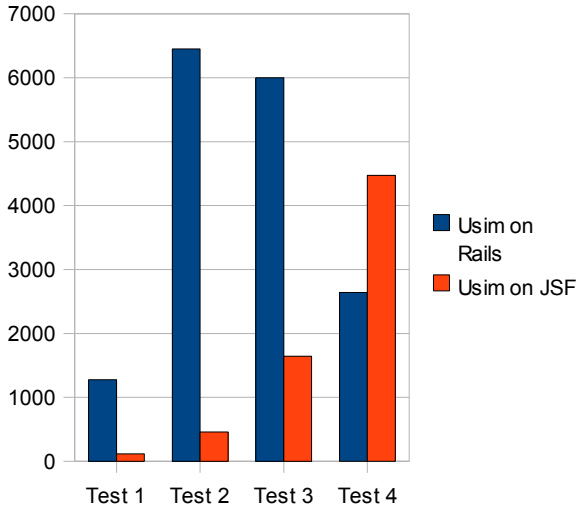


Abbildung 20: Paralleler Zugriff auf die Startseite : Auswertung

In den ersten drei Tests erzielte „Usim on JFS“ deutlich bessere Performance. Im Test 3 mit 100 Benutzer weiste „Usim on Rails“ eine Fehlerquote von 18 % auf, wobei die Fehlerquote von „Usim on JSF“ bei 0 % geblieben war. Im Test 4 mit 200 Benutzern hat „Usim on JSF“ schlechter als „Usim on Rails“ abgeschnitten. Die Betrachtung der Fehlerquote der beiden Anwendungen gibt den Aufschluss für dieses Verhalten. Die Fehlerquote für Usim on Rails lag bei 66,5%, für Usim on Rails lag sie bei 29%. Ein

fehlgeschlagener Request ergibt eine kürzere Antwortzeit. Dadurch kann der Durchschnittswert beeinflusst werden.

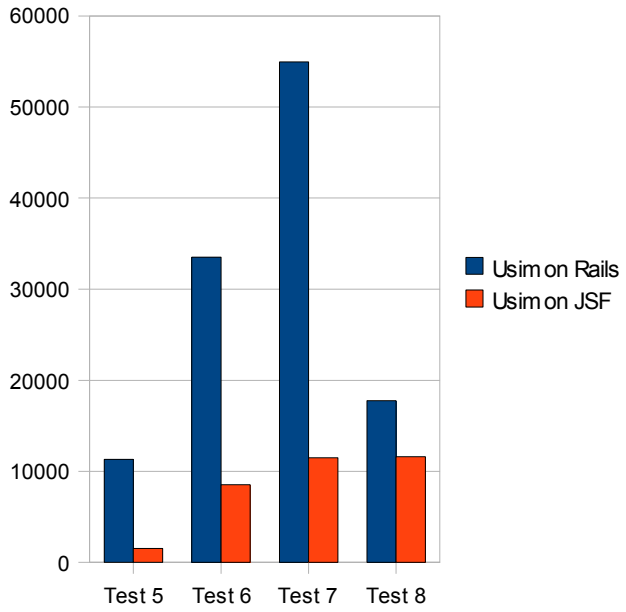


Abbildung 21: Paralleler Zugriff auf die Benutzerseite : Auswertung

Diesmal lag „Usim on JSF“ in allen Tests vorne. In den Tests 7 und 8 lag die Fehlerquote von „Usim on Rails“ bei 12% und 68%. Für „Usim on JSF“ lag sie bei 0% und 30%.

Belastbarkeit

Die Messergebnisse, die 5 Minuten übersteigen, wurden nicht berücksichtigt, da so eine Antwortzeit für eine Webanwendung unzulässig ist. Für die Darstellung wurden die Messergebnisse in Sekunden umgerechnet.

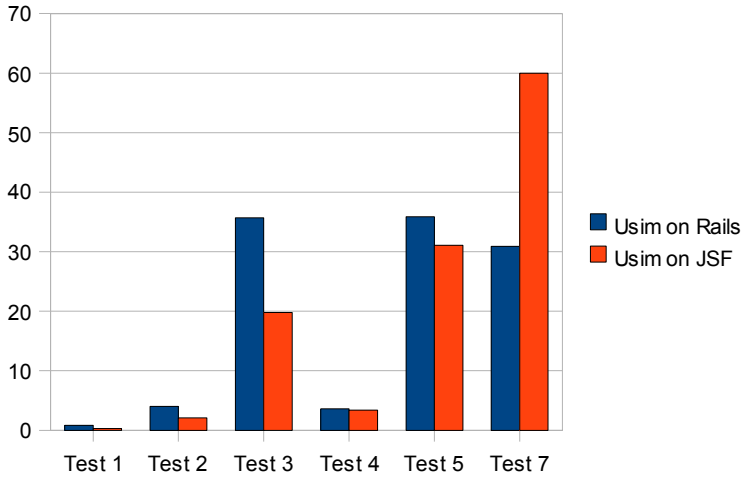


Abbildung 22: Belastbarkeit : Auswertung

Bei der Messung der Belastbarkeit konnte in fünf Tests eine bessere Performance von „Usim on JSF“ gemessen werden. Zwei Tests konnten nicht ausgewertet werden, da die Antwortzeit über 5 Minuten betrug. Nur im Test 7 erzielte „Usim on Rails“ kürzere Antwortzeiten.

8.2.4 Fazit

Beim Testen der Ladezeiten war „Usim on JSF“ deutlich schneller als „Usim on Rails“. Die Ladezeit der Startseite war 10 Mal und der Benutzerseite 7 Mal kürzer. Im parallelem Zugriff war „Usim on JSF“ im Schnitt fast 6 Mal schneller. In der Belastbarkeit war „Usim on JSF“ bei den Tests 1, 2, 3 jeweils 3, 2, 2 Mal schneller als „Usim on Rails“. Bei den Tests 4 und 5 lagen beide Anwendungen fast gleich, wobei „Usim on Rails“ leicht unterlegen war. Im Test Nummer 7 war „Usim on Rails“ 2 Mal schneller als „Usim on JSF“.

Insgesamt wurden 18 Tests durchgeführt. Davon konnten 2 Tests nicht ausgewertet werden. Bei den 14 von den 16 erfolgreichen Tests konnte eine bessere Performance für „Usim on JSF“ gemessen werden. Somit steht es 14 zu 2 für „Usim on JSF“. Auf Grund eines klaren Ergebnisses kann „Usim on JSF“ als Sieger des Abschnitts Performance gekrönt werden.

8.3 Zusammenfassung

In diesem Kapitel wurde beschrieben, wie der Entwicklungsaufwand für beide Anwendungen ermittelt wurde. Die ermittelten Ergebnisse wurden verglichen und vorgestellt.

Die Performancemessung wurde in drei Phasen aufgeteilt: Aufbau, Durchführung und Auswertung. In Rahmen dieser Phasen wurden dafür aufgebaute Testumgebung und dabei eingesetzte Testwerkzeuge und die angewandten Testmethoden vorgestellt. Für die Darstellung der Messergebnisse wurden Tabellen verwendet. Bei der Auswertung der Ergebnisses wurden zur Verdeutlichung Diagramme erstellt.

9 Fazit und Ausblick

„Der Endspurt!"

Zum Abschluss dieser Arbeit wird zunächst die Erreichung der Ziele aus der Aufgabenstellung für diese Arbeit anhand der wichtigsten Ergebnissen dokumentiert. Danach erfolgt ein Ausblick auf die Entwicklung der Webanwendungen mit den beiden Technologien, die in dieser Arbeit eingesetzt wurden.

9.1 Ergebnisse

Am Anfang dieser Arbeit wurden folgende Ziele festgelegt:

- Entwicklung zweier Webanwendungen unter Verwendung verschiedener Technologien.
- Vergleich dieser Anwendungen miteinander anhand der Ermittlung des Entwicklungsaufwands und der Messungen der Performance.

In den folgenden Abschnitten werden die Ergebnisse zusammengefasst.

9.1.1 Entwicklung der Webanwendungen

Der Einsatz der verschieden Technologien hat in beiden Fällen zum Erfolg geführt. „Ruby on Rails“ überzeugte mit einem durchdachten Konzept „Convention over Configuration“, sodass fast keine Konfiguration des Frameworks notwendig war. Vorteilhaft für eine schnelle und erfolgreiche Anwendungsentwicklung mit Rails ist Tatsache, dass Rails bereits einen großen Umfang an Web-Funktionalitäten mitbringt. Einerseits konnte der ActiveRecord Ansatz durch einen leicht verständlichen Umgang mit der Datenbank überzeugen. Andererseits werden die SQL-Befehle dabei zur Laufzeit generiert, was sich auf die Performance der erstellten Anwendung auswirkte.

„JavaServer Faces“ benötigten deutlich mehr Konfigurationsaufwand. Der Web-Server und die Datenbankanbindung mussten explizit konfiguriert werden. Das Framework bietet insgesamt viele Komponenten an, die in Rahmen dieser Arbeit nicht alle eingesetzt werden konnten. Die hohe Anzahl der Komponenten und die Möglichkeit eigene Komponenten zu erstellen, deuten auf eine hohe Flexibilität des Frameworks hin. Durch die gesammelte Erfahrung steht fest, dass ein Einsatz eines ORM-Frameworks wie Hibernet eine echte alternative zum JDBC darstellt.

Der Einsatz von Eclipse als IDE erwies sich als positiv. Vor allem für einen Entwickler, der schon Erfahrungen mit dieser Entwicklungsumgebung hat, ist diese Wahl zu empfehlen.

9.1.2 Vergleich

Die Ermittlung des Entwicklungsaufwands hat ergeben, dass die Entwicklung mit „Ruby on Rails“ insgesamt weniger Zeit beansprucht hat. Davon wurde ein größerer Teil in die Einarbeitungsphase investiert. Bei „JavaServer Faces“ war mehr Zeit für die Implementierungphase notwendig.

Die Messung der Performance hat gezeigt, dass „JavaServer Faces“ fast in allen Tests deutlich bessere Ergebnisse liefert. Dennoch bedeutet das nicht, dass der Einsatz von „Ruby on Rails“ für eine Webanwendung auf Grund der schlechteren Performance nicht akzeptabel wäre. Viele erfolgreiche Projekte der Gegenwart beweisen diese Aussage.

9.2 Ausblick

Beide Technologien unterstützen moderne AJAX-Technologie, sowie SOA- und REST-Architekturen. Dadurch steht dem praktischem Einsatz im Prinzip nichts im Wege. Zur Zeit finden sie ihren Einsatz in der Praxis bei der Weganwendungsentwicklung.

JSF werden bereits in vielen Projekten eingesetzt. Zudem sind JSF ein offizieller

Standard von SUN und werden ständig weiter entwickelt. Große IT-Unternehmen vermarkten standardkonforme Oberflächenkomponenten wie Apache MyFaces, Oracle ADF Faces und JBoss RichFaces.

Anzahl der Ruby-Projekte, sowie die Ruby-Community wachsen permanent. Die Liste der Webanwendungen, die mit Rails gebaut sind, ist groß. Die Anwendungen aus dieser Liste können auf der http://www.rubyonrails.org/applications - (23.03.08) angeschaut werden.

Mittlerweile bieten viele Anbieter für „Ruby on Rails“ Web-Hosting an. Sie sind auf der http://wiki.rubyonrails.com/rails/pages/RailsWebHosts -(23.03.08) aufgelistet.

Ab dem 1.03.08 wird Ruby an der HAW als eine Grundprogrammiersprache im ersten Semester des Informatik Studiengangs unterrichtet.

Inhalt der beiliegenden CD

Die beiliegende CD hat folgenden Inhalt:

- /content.txt: Inhalt der CD.
- /bachelorthesis.pdf: Dieses Dokument.
- /Development: SourceCode
  - /usimOnRails.zip: Eclipse-Projekt für den Ruby on Rails Teil.
  - /usimOnJSF.zip: Eclipse-Projekt für den JavaServer Faces Teil.
- /Results: Messergebnisse aus dem Versuchsteil
  - /test_results.zip: Ergebnisse aller durchgeführten Tests.
- /Documentation: Installationsanleitung
  - /install.odt: Installationsanweisungen für die Einrichtung der Entwicklungsumgebung.
  - /usimOnJSF-backup.sql: Backup-script zum Wiederherstellen der Datenbank für „Usim on JSF“.

Literaturverzeichnis

[ADWR] - Dave Thomas; David Heinemeier Hansson; Leon Breedt: ***Agile web development with Rails: a pragmatic guide.*** Pragmatic Bookshelf 2005.

[Balzert] - Helmut Balzert: ***Lehrbuch der Software-Technik.*** Spektrum 2. Auflage 2001.

[BRPC] - Petert Cooper: ***Beginning Ruby : From Novice to Professional.*** Apress 2007.

[CJSF] - David Geary: ***Core JavaServer Faces***. 1. print. Upper Saddle River, NJ: Prentice Hall, 2004.

[DRRW] - Bruce A. Tate; Curt Hibbs: ***Durchstarten mit Ruby on Rails: Ultra-schnelle Web-Programmierung.*** O'Reilly 2007.

[JSFAB] - Andy Bosch: ***JavaServer Faces***. Addison-Wesley 2004.

[JSFFP] - Bernd Müller: ***Java Server Faces - ein Arbeitsbuchfür die Praxis***. Hanser 2006.

[LPP] - URL – http://home.vrweb.de/~juergen.katins/ruby/buch/index.html – (5.11.07)

[LST] - Helmut Balzert: ***Lehrbuch der Software-Technik***. Spektrum 1998.

[RDRR] - Ralf Wirdemann; Thomas Baustert: ***Rapid web development mit Ruby on Rails.*** Hanser 2006.

[RLFU] - URL - http://www.ruby-lang.org/de/documentation/ruby-from-otherlanguages (30.11.07)

[RPL] - URL – http://publib.boulder.ibm.com/infocenter/iadthelp/v7r0/index.jsptopic=/com.ibm.etools.jsf.doc/topics/cjsf.html. -(22.03.08)

[RRE] - Martin Marinschek; Wolfgang Radinger: ***Ruby on Rails: Einstieg in die effiziente Webentwicklung***. dpunkt.verlag 1. Auflage 2006.

[W08] - URL – http://de.wikipedia.org/wiki/Hauptseite. (13.03.08)

[W01] - URL – http://de.wikipedia.org/wiki/Internet. - (30.09.07)

[W02] - URL - http://ruby-doc.org/ - (22.03.08)

[W03] - URL - http://www.rubyonrails.de/- (16.10.07)

[W04] - URL – http://de.wikipedia.org/wiki/Agile_Softwareentwicklung. - (22.03.08)

[W05] - URL – http://de.wikipedia.org/wiki/Function-Point-Verfahren. - (22.03.08)

[W06] - URL – http://de.wikipedia.org/wiki/Framework. - (22.03.08)

[W07] - URL – http://de.wikipedia.org/wiki/Schichtenarchitektur. - (22.03.08)

[W08] - URL – http://de.wikipedia.org/wiki/Data_Access_Object. - (22.03.08)

[W09] - URL – http://de.wikipedia.org/wiki/Model_View_Controller#Model_2.- (22.03.08)

[W10] - URL – http://de.wikipedia.org/wiki/Webanwendung. - (22.03.08)

[W11] - URL – www.umlatte.com/PLUG/intro_to_rails/matz.jpg.- (22.03.08)

[W12] - URL - http://www.fh-wedel.de/~si/seminare/ws06/Ausarbeitung/ 10.JavaServerFaces/image/mvc.jpg.-(22.03.08)

[W13] - URL – http://www.zitate-online.de/thema/wetten/. - (22.03.08)

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §24(5) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 25. März 2008 _____

Ort, Datum Unterschrift