

Die approbierte Originalversion dieser Diplom-/Masterarbeit ist an der Hauptbibliothek der Technischen Universität Wien aufgestellt (<http://www.ub.tuwien.ac.at>).

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology (<http://www.ub.tuwien.ac.at/englweb/>).



TU Wien

Business Informatics Group

Institut für Softwaretechnik und Interaktive Systeme

Codegeneration with Ruby on Rails

Bridging the Gap between Design and Implementation

Diplomarbeit zur Erlangung des akademischen Grades eines

Magister der Sozial- und Wirtschaftswissenschaften
Magister rerum socialium oeconomicarumque
(Mag. rer. soc. oec.)

eingereicht bei o. Univ.-Prof. Mag. Dipl.-Ing. Dr. Gerti Kappel
mitbetreuender Assistent: Mag. Manuel Wimmer

Alexander Dick

Wien, 15. November 2006

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benützt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Wien, 15. November 2006

Alexander Dick

Danksagung

Zu allererst möchte ich von ganzem Herzen meinen Eltern Ida und Leo Dick für ihre besondere Unterstützung meine Dankbarkeit ausdrücken. Zum einen ließen sie mir freie Hand bei der Wahl meines Studiums und zum anderen waren sie während meiner Studienzeit die finanzielle und moralische Stütze, ohne die ein positiver Abschluss des Diplomstudiums Wirtschaftsinformatik meinerseits nicht möglich gewesen wäre.

Ein ganz besonderer Dank gilt meiner Freundin Birgit König und meinen Brüdern Peter und Christoph Dick, die besonders durch ihre Motivation und Unterstützung zu einer erfolgreichen Beendigung meines Studiums und Verfassung dieser Arbeit beigetragen haben.

Natürlich möchte ich mich auch herzlich bei meinem Diplomarbeitsbetreuer Mag. Manuel Wimmer bedanken, der mir mit seiner fachlichen Kompetenz zur Seite stand und mir durch seine Betreuung den Weg zum positiven Abschluss dieser Arbeit und damit des Studiums gewiesen hat.

Bedanken möchte ich mich auch bei Familie Landrock, die mich kurz vor Beendigung meines Studiums in ihr Unternehmen aufgenommen hat, und besonders bei meinem Chef, Herrn Dr. Georg Landrock, der mir durch eine flexible Zeiteinteilung die rasche Ausarbeitung dieser Diplomarbeit ermöglicht hat.

Last but not least möchte ich noch all jenen ein herzliches Dankeschön aussprechen, die mich während meines Studiums als Kollegen begleitet und mir dadurch eine schöne Zeit bereitet haben, an die ich mich sehr gerne zurück erinnere.

Kurzfassung

Durch die Veränderung des World Wide Web von einem statischen Informationsmedium hin zu einem dynamischen Anwendungsmedium ist der Bedarf an Webanwendungen, die für die Bereitstellung von Diensten über das Web verantwortlich sind, so hoch wie nie. Es haben sich bereits sehr viele Technologien etabliert, mit Hilfe derer es möglich ist, Webanwendungen zu entwickeln. Die Anforderungen an solche Webanwendungen sind sehr vielfältig und mit einer oft kürzeren Entwicklungszeit, die für die Erzeugung der Anwendung zur Verfügung steht, geht eine meist höhere Komplexität der benötigten Funktionalität einher.

Um es den Entwicklern zu ermöglichen, sich mit dem eigentlichen Problem auseinander zu setzen und nicht für jede Webanwendung das Rad neu erfinden zu müssen, werden für die verschiedenen Technologien Frameworks geschaffen, die den Programmierer mit Hilfe integrierter Funktionalitäten bei der Implementierung unterstützen sollen.

Eines dieser Frameworks, das sich erst seit kurzer Zeit im Web etabliert, ist Ruby on Rails. Es bietet Funktionalitäten, die es dem Entwickler ermöglichen rasch einen ersten, funktionstüchtigen Prototyp einer Webanwendung zu generieren, mit Hilfe dessen sofort Feedback vom Benutzer eingeholt werden kann. Wie es bei vielen Anwendungen im Web der Fall ist, basieren auch die Applikationen, die mit Ruby on Rails erstellt werden auf einem relationalen Datenbanksystem. Der Prototyp, der mit Hilfe eines in Rails integrierten Codegenerators erzeugt wird, bietet jedoch nur eingeschränkte Funktionalität und es wäre wünschenswert einerseits diese zu erweitern und andererseits die Anwendung ausgehend von einem Modell, das die Datenbank beschreibt, generieren zu können.

Diese Arbeit versucht in einem ersten Schritt den von Ruby on Rails bereitgestellten Scaffold-Generator so zu erweitern, dass der Prototyp auch Validierungsfunktionalität beherrscht. Dies geschieht mit Hilfe zweier zusätzlicher Datenbanktabellen, die Regeln für die Validierung bereitstellen sollen. Weiters soll die Abbildung von Beziehungen zwischen Datenbanktabellen automatisch in die Modellklassen der Webanwendung eingebunden werden.

Im zweiten Schritt wird ein grafischer Editor erstellt, der für die Erstellung von logischen Datenbankmodellen herangezogen werden kann.

Schließlich soll am Ende ein Framework entstehen, das zuerst zur Modellierung von ER-Diagrammen dient, danach aus dem Modell über einen Zwischenschritt der Codegeneration die Struktur der Datenbank erstellt und zum Abschluss mit Hilfe des erweiterten Ruby-Generators den Prototyp einer datenintensiven Webanwendung erstellt. Aufgrund der Kombination von Modellierung und dem Einsatz von Ruby on Rails wird dieses Framework *Models on Rails* genannt.

Abstract

Because the World Wide Web is changing from a static medium for information interchange into a dynamic application medium, the need for these applications, which are responsible for providing services over the Web, is higher than ever. In the last few years a very high number of technologies has emerged, which make it possible to develop such web-applications. There is a large variety of requirements these applications have to fulfill and often a shorter time developers have for developing them goes hand in hand with a higher complexity of their necessary features.

To make it possible for the developers to concentrate on the main problem and not to invent the wheel again and again a large amount of frameworks has been created according to the different technologies, which tend to support the developers by their implementation work with integrated functionalities.

Only a short time ago one of these frameworks started its presence on the web, namely Ruby on Rails. It offers functionality that makes it possible for developers to easily and quickly generate a first prototype of a web-application. With this prototype the developer is able to get feedback from the user after a short development time. Like many other web-applications the applications built with Ruby on Rails are based on a relational database-system. Unfortunately the functionality the generated prototype offers is limited and it would be desirable on the one hand to extend the functionality and on the other hand to automatically generate the web-application out of shaped models which describe the structure of the underlying database.

This thesis tries in a first step to extend the Scaffold-Generator, which comes with Ruby on Rails, so that the generated prototype integrates validation-functionality. To achieve this goal, two additional tables are inserted into the database, which are responsible for saving the validation rules and mapping the table-attributes on these rules. Additionally the relationships between the database-tables should automatically be added to the classes of the models.

In the second step a graphical editor is built which supports the creation of logical database models. These models should be the starting point for generating a web-application.

The final goal of this thesis is to establish a framework for generating web-applications. At the beginning it should give the developer the ability to create ER-Models. After that comes an intermediate code-generation step for building the database. And finally the extended Rails-Generator constructs the prototype of a data intensive web-application with enhanced functionality. Because of the combination of modeling and using Ruby on Rails the framework is named *Models on Rails*.

Inhaltsverzeichnis

1	EINLEITUNG.....	1
1.1	MOTIVATION, PROBLEMSTELLUNG UND ZIELSETZUNG	1
1.2	VERWANDTE ARBEITEN	3
1.3	AUFBAU DER ARBEIT	4
2	ALLGEMEINES ZU WEBAPPLIKATIONEN	6
2.1	SOFTWARE ENGINEERING	6
2.2	WEB ENGINEERING	8
2.3	AGILE SOFTWAREENTWICKLUNG	9
2.4	KATEGORISIERUNG VON WEBAPPLIKATIONEN	11
2.4.1	<i>Dokumentenzentrierte Websites</i>	<i>11</i>
2.4.2	<i>Interaktive Webanwendungen</i>	<i>11</i>
2.4.3	<i>Transaktionale Webanwendungen</i>	<i>11</i>
2.4.4	<i>Workflow-basierte Webanwendungen</i>	<i>12</i>
2.4.5	<i>Kollaborative Webanwendungen</i>	<i>12</i>
2.4.6	<i>Portalorientierte Webanwendungen</i>	<i>12</i>
2.4.7	<i>Ubiquitäre Webanwendungen</i>	<i>12</i>
2.5	DATENINTENSIVE WEBANWENDUNGEN	13
2.5.1	<i>Das logische Datenbankmodell und ER</i>	<i>13</i>
2.5.2	<i>Das Implementierungsdatenmodell und SQL</i>	<i>14</i>
2.6	ARCHITEKTUREN	15
2.6.1	<i>2-Schichten-Architektur</i>	<i>15</i>
2.6.2	<i>(n > 2)-Schichten-Architekturen</i>	<i>16</i>
2.7	WEB SERVER	17
2.8	APPLICATION SERVER	18
3	DESIGN PATTERNS - ENTWURFSMUSTER.....	20
3.1	DEFINITION	20
3.2	MODEL VIEW CONTROLLER (MVC)	22
3.3	MODEL 1	23
3.4	MODEL 2	24
3.5	FRONT CONTROLLER	25

3.6	APPLICATION CONTROLLER	26
3.7	ACTIVE RECORD	27
3.8	DATA ACCESS OBJECT (DAO).....	28
4	WEBAPPLIKATIONS-FRAMEWORKS – EIN VERGLEICH	29
4.1	FRAMEWORKS ALLGEMEIN.....	30
4.1.1	<i>Anpassung</i>	30
4.1.2	<i>Anwendungsbereich</i>	31
4.1.3	<i>Interaktion</i>	31
4.2	APACHE STRUTS	32
4.2.1	<i>Model</i>	32
4.2.2	<i>View</i>	34
4.2.3	<i>Controller</i>	38
4.3	EXKURS: RUBY	40
4.4	RUBY ON RAILS	41
4.4.1	<i>Active Record (Model)</i>	41
4.4.2	<i>Action Pack (View und Controller)</i>	42
4.4.3	<i>Weitere Komponenten</i>	44
4.5.	VERGLEICH DER BEIDEN FRAMEWORKS	45
4.5.1	<i>Model View Controller-Entwurfsmuster</i>	45
4.5.2	<i>Request-Bearbeitung</i>	46
4.5.3	<i>Datenaustausch zwischen View und Controller</i>	46
4.5.4	<i>Konfiguration und Konvention</i>	46
4.5.5	<i>Prototyping</i>	47
4.5.6	<i>Entwicklungsgeschwindigkeit</i>	47
4.5.7	<i>Validierung</i>	48
4.5.8	<i>Internationalisierung</i>	48
4.5.9	<i>Zusammenfassung</i>	49
5	CODEGENERIERUNG MIT RUBY ON RAILS	50
5.1	CODEGENERATION ALLGEMEIN	50
5.1.1	<i>Begriffsbestimmung und Definition</i>	50
5.1.2	<i>Einteilung von Codegeneratoren</i>	51
5.1.3	<i>Vor- und Nachteile des Einsatzes von Codegeneratoren</i>	52
5.2	DIE CODEGENERATION IN RUBY ON RAILS	54
5.2.1	<i>Allgemeines</i>	54
5.2.2	<i>Vorhandene Codegeneratoren</i>	55
5.2.3	<i>Arbeitsweise der Generators</i>	57

5.3	ERSTELLUNG EINER RAILS-BEISPIELANWENDUNG	58
6	DER ERWEITERTE RAILS-GENERATOR	66
6.1	AUSGANGSBASIS: SCAFFOLDING.....	66
6.1.1	<i>Controller</i>	67
6.1.2	<i>Model</i>	68
6.1.3	<i>View</i>	69
6.2	ANFORDERUNGEN AN DEN ERWEITERTEN RAILS-GENERATOR.....	69
6.2.1	<i>Validierung der Modellobjekte</i>	70
6.2.2	<i>Assoziationen zwischen den Modellklassen</i>	74
6.3	ERSTELLEN EINES EIGENEN RAILS-GENERATORS.....	76
6.3.1	<i>Vorgehensweise bei der Erstellung des erweiterten Rails-Generators</i>	78
6.3.2	<i>Funktionsweise des erweiterten Rails-Generators</i>	79
7	MODELS ON RAILS.....	86
7.1	ÜBERBLICK	86
7.2	ERSTELLEN DES ER-MODELLS.....	88
7.3	MODELLTRANSFORMATION	90
7.4	GENERIEREN DER WEBANWENDUNG	96
8	ZUSAMMENFASSUNG UND AUSBLICK	99
	ANHANG.....	101
	ABKÜRZUNGSVERZEICHNIS.....	103
	ABBILDUNGSVERZEICHNIS	104
	TABELLENVERZEICHNIS	106
	LITERATURVERZEICHNIS	107

Kapitel 1

Einleitung

1.1 Motivation, Problemstellung und Zielsetzung

Das World Wide Web verwandelt sich immer stärker von einem statischen Informationsmedium, für das es ursprünglich entwickelt wurde zu einem dynamischen Anwendungsmedium. Die Aufgaben, die diese Webanwendungen erfüllen müssen, werden immer komplexer und auch die Technologien, die für deren Entwicklung eingesetzt werden, werden immer vielfältiger. Um dieser ständigen Veränderung Herr zu werden und dem Entwickler bei der Schaffung von Applikationen für das Web zu unterstützen, wurden - und werden noch laufend - Frameworks geschaffen, die, meist auf eine spezielle Technologie bezogen, integrierte Funktionalitäten bieten. Sie helfen dem Programmierer sich um das eigentliche Problem zu kümmern und sich nicht bei der Erstellung einer neuen Anwendung mit der Implementierung von Funktionalitäten auseinandersetzen zu müssen, die sich als Grundfunktionalitäten einer Vielzahl von Anwendungen herausstellen. Zur Erfüllung dieser Aufgaben werden Frameworks herangezogen.

Diese Arbeit nimmt zuerst zwei Vertreter solcher Frameworks genauer unter die Lupe, die auf dem Model View Controller-Entwurfsmuster basieren. Das erste der beiden ist Apache Struts, das in diesem unter ständigem Wandel befindlichen Bereich der Webprogrammierung, eine sehr lange Geschichte hat. Anwendungen, die auf Grundlage dieses Frameworks erstellt werden, basieren auf der bekannten und sehr weit verbreiteten Programmiersprache Java. Obwohl sich das Framework als „quasi Standard“ für Webapplikations-Frameworks etabliert hat, wird es in der Lite-

ratur häufig als schwergewichtiges Framework gehandelt, da es eine lange Einarbeitungszeit und einen hohen Konfigurationsaufwand benötigt, bis der erste verwendbare Entwurf einer Webanwendung vorliegt.

Der zweite Vertreter solcher Frameworks, mit dem sich diese Arbeit beschäftigt, ist das noch sehr junge Ruby on Rails. Es wurde im Zuge der Erstellung einer Webanwendung heraus entwickelt und stellt damit einen sehr starken Bezug zur Praxis her. Wie der Name Ruby on Rails – zu Deutsch: „Ruby auf Schienen“ – schon verrät, basiert das Framework auf der Programmiersprache Ruby und die erstellten Anwendungen benötigen als Datenbasis ein relationales Datenbanksystem. Es handelt sich hierbei um eine vollständig objektorientierte Sprache, die in Japan entwickelt wurde und dort auch schon sehr verbreitet ist, nun aber auch immer mehr Anhänger in den USA und Europa findet. Ruby on Rails kann im Vergleich zu Struts als leichtgewichtiges Framework angesehen werden. Es ist mit dem Framework möglich mit Hilfe seiner integrierten Codegeneratoren innerhalb weniger Minuten einen funktionsfähigen Prototyp einer Webanwendung zu erzeugen, auf dessen Basis der Entwickler seine Programmierfähigkeiten fortsetzen kann, ohne den generierten Code verwerfen zu müssen. Natürlich bietet der erhaltene Prototyp nur ein geringes Maß an Funktionalität.

Um die Vorteile von Ruby on Rails noch weiter auszubauen, wird in dieser Arbeit die Funktionalität des in dem Framework integrierten Codegenerators so erweitert, dass er auch eine automatische Validierung in die erzeugten Modellklassen integriert. Dies soll einerseits durch die Erstellung eines erweiterten Codegenerators der innerhalb des Frameworks verwendet werden kann, geschehen, andererseits wird die zugrunde liegende Datenbasis so erweitert, dass Validierungsregeln, die auf die gespeicherten Datensätze angewendet werden sollen, bereits vorhanden sind und nur noch vom erweiterten Generator integriert werden müssen.

Wie erwähnt basieren die Rails-Webanwendungen auf einer relationalen Datenbank, dessen Struktur vor der Erstellung der Anwendung vom Entwickler festgelegt werden muss. Um dem Programmierer eine benutzerfreundliche Möglichkeit zu bieten ein anschauliches logisches Datenbankmodell zu erstellen und sich nicht mit dem Implementierungsdatenmodell in Form von SQL-Anweisungen herumschlagen zu müssen, ist ein weiteres Ziel dieser Arbeit einen Editor für dieses Modell zu entwickeln, aus dem wiederum mit Hilfe von automatischer Codegenerierung die Struktur der Datenbank für die Webanwendung erstellt und darauf aufbauend der erweiterte Prototyp einer Webanwendung in Ruby on Rails generiert werden kann.

1.2 Verwandte Arbeiten

Natürlich ist diese Arbeit nicht die erste, die sich mit der Modellierung von Datenbanken und der automatischen Codegenerierung auseinandersetzt. Der folgende Abschnitt zeigt kurz drei weitere Arbeiten auf, deren inhaltliche Schwerpunkte sich mit denen dieser Diplomarbeit einerseits gut vergleichen andererseits jedoch auch klar davon abgrenzen lassen:

Mit der Modellierung und der automatischen Generierung von datenintensiven Webanwendungen beschäftigt sich beispielsweise WebML [Ceri03]. Es handelt sich bei WebML um eine konzeptionelle Sprache, die durch eine Vielzahl von Modellen Unterstützung für den Entwurf von Webanwendungen bietet. Es behandelt nicht nur die Modellierung der Datenebene, wie es in dieser Arbeit der Fall ist, sondern berücksichtigt auch insbesondere die Hypertextebene einer Webanwendung.

Ein Framework, das sich selbst als „Transform Engine“ bezeichnet ist AndroMDA [Andr06]. Es bietet die Möglichkeit UML-Modelle in Code für verschiedenste Plattformen wie J2EE oder .NET zu transformieren, in dem es sich strikt an die Einhaltung des Model Driven Architecture (MDA) Paradigmas hält. Um die Codegenerationsfähigkeit dieses Frameworks verwenden zu können, müssen jedoch eigene sogenannte „Cardridges“ geschrieben werden, mit Hilfe derer AndroMDA erweitert werden kann. Das Framework ist zwar in der Lage, UML-Modelle zu transformieren, es bietet jedoch selbst keine Möglichkeit zur Erstellung solcher Modelle. Es kann jedoch eine Vielzahl von Modellen, die in gängigen UML-Modellierungswerkzeugen erstellt werden, für die Weiterverarbeitung in AndroMDA verwendet werden.

Qcodo [Qcod06] ist ein objektorientiertes Webapplikationsframework, das auf der Script-Sprache PHP basiert. Es stellt eine Plattform für die schnelle Entwicklung von Webanwendungen bereit. Ähnlich wie Ruby on Rails versucht es mit Hilfe von Codegenerierung die agile Methodik des Prototyping einzusetzen und dadurch eine schnelle Erstellung von weiterverwendbaren Anwendungsentwürfen zu ermöglichen. Dieses Framework befindet sich allerdings noch in der Entstehungsphase. Bei Verfassung dieser Arbeit gab es jedoch nur Beta-Versionen und keine fertige Release-Version.

1.3 Aufbau der Arbeit

Die Arbeit wird in insgesamt 8 Kapitel untergliedert, deren Inhalt nun kurz zusammengefasst wird:

Kapitel 2 geht auf allgemeine Aspekte in Verbindung mit der Entwicklung von Webanwendungen ein. Zu Beginn werden kurz die Disziplinen Software Engineering Web Engineering erläutert und danach wird auf den neuen immer bedeutender werdenden Ansatz der Agilen Software Entwicklung eingegangen. Den Abschluss macht eine mögliche Kategorisierung von Webanwendungen gefolgt von einer genaueren Betrachtung der datenintensiven Webanwendungen. Danach wird auf den Aufbau von Webanwendungen mit der Beschreibung der Schichtenarchitekturen näher eingegangen. Den Abschluss des Kapitels machen die Erläuterungen der Begriffe Web Server und Application Server.

In Kapitel 3 findet sich eine Sammlung von Erklärungen verschiedener Entwurfsmuster, die bei der Erstellung von Webanwendungen eingesetzt werden. Diese Design Patterns treten insbesondere in Verbindung mit den beiden in Kapitel 4 beschriebenen Webapplikations-Frameworks auf.

Kapitel 4 gibt eine kurze Einführung in den Begriff Framework und erläutert danach die Bestandteile zweier freiverfügbarer Vertreter von Frameworks, die für die Entwicklung von Webanwendungen in zwei unterschiedlichen Programmiersprachen verwendet werden. Zum einen Apache Struts, das in mancher Literatur als quasi Standard Webapplikations-Framework gehandelt wird, schon sehr lange auf dem Markt ist und die Programmiersprache Java verwendet. Zum anderen das noch junge Framework Ruby on Rails, das sich trotz seines geringen Alters schon großer Beliebtheit erfreut, und um das sich ein regelrechter Hype entwickelt hat. Es verwendet die Programmiersprache Ruby. Die beiden Frameworks werden abschließend anhand ausgewählter Kriterien gegenübergestellt und die Ergebnisse in einer Tabelle zusammengefasst.

Kapitel 5 bespricht Grundlagen zum Thema Codegeneration und beschreibt die Codegenerationsfunktionalität, die das Framework Ruby on Rails bietet.

Kapitel 6 beschreibt die Entwicklung des erweiterten Codegenerators für die Erstellung des Prototyps einer datenintensiven Webanwendung. Es werden die Anforderungen an den Generator spezifiziert und die Vorgangsweise bei der Erstellung und die Funktionsweise des Generators näher erläutert.

Kapitel 7 rundet die Arbeit mit der Beschreibung des Gesamt-Frameworks „*Models on Rails*“ ab und erläutert die Vorgehensweise bei der Erstellung des grafischen Modell-Editors sowie den Codegenerierungsschritt, der für den Übergang vom Modell zum erweiterten Rails-Codegenerator notwendig ist.

Abschließend werden in Kapitel 8 die in der Arbeit gewonnenen Erkenntnisse zusammengefasst und Anregungen für weitere Arbeiten in diesem Bereich gegeben.

Kapitel 2

Allgemeines zu Webapplikationen

Dieses Kapitel soll einen allgemeinen Überblick über die wichtigsten Bereiche der Entwicklung von Webapplikationen geben, die für die weiteren Abschnitte dieser Arbeit von Bedeutung sind. Die Kapitel 2.1 bis 2.3 beschäftigen sich grob mit der Entwicklung von Software und geben kurze Einführungen in die Themen Software Engineering, Web Engineering und Agile Software Entwicklung. Kapitel 2.4 zeigt eine mögliche Kategorisierung von Webapplikationen auf, worauf anschließend in Kapitel 2.5 speziell auf datenintensive Webanwendungen näher eingegangen wird. Kapitel 2.6 beschreibt kurz Architekturen, die den möglichen Aufbau einer Webanwendung beschreiben. Den Abschluss machen Kapitel 2.7 und 2.8, die die Begriffe Web Server und Application Server näher erläutern.

2.1 Software Engineering

Software Engineering ist eine junge Ingenieurdisziplin. Von Beginn der Nutzung von Computern bis in die frühen 80er Jahre wurde unter dem Begriff nicht viel mehr verstanden als die Programmierung der zur Verfügung stehenden Computersysteme. Im wissenschaftlichen Bereich stand bis zu dieser Zeit die Untersuchung von Algorithmen und Datenstrukturen im Vordergrund, die in den damals bekannten Programmiersprachen umgesetzt wurden [Zuse01].

Der Begriff Software Engineering wurde 1967 von einer Forschungsgruppe der NATO geformt. Auf den Software Engineering Konferenzen der NATO von 1968 in Garmisch und 1969 in Rom wurden Softwareprogramme zum ersten Mal als Industrieprodukt bezeichnet [Zuse01].

Ab Mitte der 60er und in den 70er Jahren kam der Begriff der „Software-Krise“ auf und war Anlass für ein großes Umdenken in der Softwareentwicklung. Durch die oft geringe Qualität der erzeugten Software sowie das Scheitern vieler Projekte in dieser Branche wurde es notwendig Methoden und Prozesse zuschaffen, die eine effiziente und produktive Entwicklung von Softwaresystemen ermöglichen, sowie eine Standardisierung in diesem Bereich bewirken sollten. Es ist bei der Entwicklung von Software ein systematisches Vorgehen notwendig. Sie soll in einem festgelegten organisatorischen Rahmen erfolgen. [Balz96] definiert die Softwaretechnik als

“...zielorientierte Bereitstellung und systematische Verwendung von Prinzipien, Methoden und Werkzeugen für die arbeitsteilige, ingenieurmäßige Entwicklung und Anwendung von umfangreichen Softwaresystemen.”

Und so wurde im Laufe der Zeit eine Vielzahl von Prozessmodellen entwickelt, die einen solchen Rahmen beschreiben und ständig reihen sich neue in die Liste der Modelle ein. In [Balz98] werden folgende sieben Vorgehensmodelle beschrieben, die man als die bekanntesten Vertreter von Softwareentwicklungsprozessen nennen kann:

- Das Wasserfallmodell
- Das V-Modell
- Das Prototypen-Modell
- Das evolutionäre/inkrementelle Modell
- Das objektorientierte Modell
- Das nebenläufige Modell
- Das Spiralmodell

Diese Modelle müssen jedoch nicht nur für sich allein in einem Softwareprojekt angewendet werden. Es ist auch möglich in einem Unternehmen je nach Projekt mehrere verschiedene Prozessmodelle einzusetzen. Die Vorgehensmodelle bauen in weiterer Folge auf durchzuführenden Arbeitsschritten auf, die je nach Modell nur einmal oder mehrmals, sequentiell oder parallel durchlaufen werden. Im Software Engineering sind folgende Arbeitsschritte zu nennen, die im Großteil der Prozessmodelle durchgeführt werden: Analyse, Entwurf, Implementierung, Test und Wartung, die

von Projekt-, Qualitäts-, und Konfigurationsmanagement sowie der Arbeitsorganisation begleitet werden [Zuse01].

Nach [Kapp04] kann man die bekanten Softwareentwicklungsprozesse auch in leichtgewichtige und schwergewichtige Prozesse einteilen, je nach Grad der Formulierung des Prozesses. Werden bei der Durchführung eines Prozesses viele Dokumente, und Modelle erstellt, so spricht man von einem schwergewichtigen Prozess, im anderen Fall von einem leichtgewichtigen. Schwergewichtige Prozessmodelle kommen eher dann zum Einsatz, wenn große Entwicklerteams Anwendungen mit hohen Qualitätsansprüchen erstellen, wo hingegen leichtgewichtige Prozesse dort angewendet werden, wo kleinere Teams weniger umfangreiche Anwendungen erstellen. Ein Beispiel für einen schwergewichtigen Softwareentwicklungsprozess ist der Rational Unified Process (RUP) [JaBR99], ein Beispiel für einen leichtgewichtigen ist Extreme Programming (XP) [Becc04]. Den letztgenannten zählt man auch zu den agilen Softwareentwicklungsprozessen, die immer mehr an Bedeutung gewinnen und in Kapitel 2.3 näher erläutert werden.

2.2 Web Engineering

In der heutigen Zeit steigt die Bedeutung des World Wide Web, kurz Web, ständig an. Es gibt kaum einen Bereich des täglichen Lebens, in den das Web noch nicht Einzug gefunden hätte [Gini01]. Die Entwicklung des Web von einem Medium zum Austausch von statischen Informationen hin zu einem dynamischen Anwendungsmedium, in dem immer mehr Leute von einer Fülle von Webanwendungen abhängig sind, ist es notwendig, dass es auch für diese Art von Softwaresystemen strukturierte und gut organisierte Methoden für deren Erstellung und Betrieb gibt. Nach [Kapp04] unterscheidet der Einsatz des Web, also seiner Technologien und Standards, eine Webanwendung von traditionellen Softwaresystemen, und sie kann folgendermaßen definiert werden:

Eine Webanwendung ist ein Softwaresystem, das auf Spezifikationen des World Wide Web Consortiums (W3C) beruht und Web-spezifische Ressourcen wie Inhalte und Dienste bereitstellt, die über eine Benutzerschnittstelle, den Web-Browser, verwendet werden.

Obwohl die Anzahl der Webanwendungen, die zur Zeit genutzt werden, sehr hoch ist, die Anforderungen und der Komplexitätsgrad der Webanwendungen ständig steigt, erinnert die Art, in der sie entwickelt, verwaltet und gewartet werden eher an

die Zeit der Software-Krise in den 60er Jahren. Nach [Kapp04] erfolgt die Entwicklung von Webanwendungen sehr spontan und sie basiert meist auf dem Wissen und den Erfahrungen einzelner Programmierer, wodurch solche schnellen und unsauberen Entwicklungsmethoden meist zu einer geringen Qualität des Endprodukts und in weiterer Folge zu großen Problemen bei Betrieb und Wartung führen. In [Gini01] heißt es sogar, dass man als Konsequenz einer Beibehaltung dieser Art der Entwicklung von Webanwendungen von einer Neuauflage der Software-Krise, der „Web-Krise“, sprechen kann. Um dem entgegen zu wirken etablierte sich eine neue Disziplin, das Web Engineering. Es bezeichnet kein einmaliges Ereignis, sondern erstreckt sich ähnlich wie Software Engineering über den gesamten Lebenszyklus einer Webanwendung. In [Kapp04] wird Web Engineering folgendermaßen definiert:

Web Engineering ist die Anwendung systematischer und quantifizierbarer Ansätze (Konzepte, Methoden, Techniken, Werkzeuge), um Anforderungsbeschreibung, Entwurf, Implementierung Test, Betrieb und Wartung qualitativ hochwertiger Webanwendungen kosteneffektiv durchführen zu können.

Web Engineering bedeutet auch die wissenschaftliche Disziplin, die sich mit der Erforschung dieser Ansätze befasst.

Interessant ist in diesem Zusammenhang die Frage, in wie weit sich die oben genannten Softwareentwicklungsprozesse auf das Web Engineering übertragen lassen. Auf Grund der speziellen Charakteristika von Webanwendungen, ist es notwendig, die vorhandenen Vorgehensmodelle und Methoden aus dem Software Engineering anzupassen, oder neue Ansätze zu entwickeln. In [Kapp04] findet sich eine ausgedehnte Behandlung dieser Frage, wo eine mögliche Verwendung der beiden Prozesse RUP und XP im Web Engineering eingehend diskutiert wird.

2.3 Agile Softwareentwicklung

Agile Softwareentwicklung bezeichnet den Oberbegriff für Agilität im Software Engineering. Es kann sich einerseits auf die eingesetzten Methoden, andererseits auf den gesamten Softwareentwicklungsprozess beziehen. Sie entstand als Gegenbewegung zu den schwergewichtigen, bürokratielastigen Softwareentwicklungsprozessen. Viele Entwickler hegten den Wunsch nach Prozessen, die technische und soziale Probleme besser berücksichtigen sowie leichter umzusetzen und anzupassen sind [Wiki06a].

Im Jahr 2001 begründete eine Gruppe von siebzehn Softwareentwicklern das Manifest der agilen Softwareentwicklung, dessen Inhalt aus folgenden vier Punkten besteht [Agil01] (Übersetzung nach [Thom06]):

- *Individuals and interactions over processes and tools. - Individuen und Interaktionen kommen vor Prozessen und Werkzeugen.*
- *Working software over comprehensive documentation. - Funktionierende Software kommt vor umfassender Dokumentation.*
- *Customer collaboration over contract negotiation. - Zusammenarbeit mit dem Kunden vor der Aushandlung von Verträgen.*
- *Responding to change over following a plan. - Reaktion auf Veränderungen kommt vor dem Befolgen eines Plans.*

Die Autoren des agilen Manifests bezeichnen sich selbst als „Agile Alliance“ und versuchen das erfolgreiche Gelingen von Projekten, die nach den Prinzipien der agilen Softwareentwicklung durchgeführt werden, zu unterstützen und andere zu motivieren, ebenfalls nach agilen Methoden zu arbeiten [Agil06].

Immer häufiger findet die Agilität auch Einklang die die Entwicklung von Webanwendungen. So bezeichnet etwa der Entwickler des Web-Frameworks Ruby on Rails, dem in dieser Arbeit noch ein großes Maß an Aufmerksamkeit geschenkt wird (Kapitel 4, 5 und 6), seine Erfindung als agil [Thom06]. Seiner Meinung nach geht es bei der Entwicklung mit dem Framework um Individuen und Interaktionen. Es gibt darin keine schwergewichtigen Werkzeugsammlungen, keine komplizierten Konfigurationen und keine ausufernden Prozesse. Schon zu einem frühen Zeitpunkt im Entwicklungszyklus findet man eine laufende Software vor, die man dem Kunden präsentieren und damit rasch auf Änderungswünsche reagieren kann. Somit kann der Entwickler dem Kunden nicht das liefern, was geplant wurde, sondern das, was er wirklich braucht [Thom06]. Das Vorhandensein von lauffähiger Software bereits zu einem frühen Zeitpunkt bezeichnet eine der vielen agilen Methoden, von denen einige Beispiele etwa in [Wiki06a] zu finden sind, das Rapid Prototyping.

Rapid Prototyping

Mit Hilfe von Rapid Prototyping ist es möglich, schon in einem frühen Stadium der Softwareentwicklung ein Programm herzustellen, das zwar nur ansatzweise Funktionalität besitzt aber dafür rasch dem Kunden präsentiert werden kann. Dadurch erhält

der Entwickler ein sehr schnelles Feedback über die Eignung der von ihm angesteuerten Lösungsansätze. Oft ist es in der Softwareentwicklung so, dass die erstellten Prototypen beispielsweise nur zur Veranschaulichung der Benutzeroberfläche dienen. Das Webapplikations-Framework Ruby on Rails etwa gibt sich damit nicht zu frieden. Es bietet eine sehr gute Möglichkeit am Beginn der Entwicklung einer Webanwendung einen Prototyp zu erstellen, der aber im Laufe des Entwicklungszyklus nicht vollständig verworfen wird, sondern als Basis für die Weiterentwicklung dient. Auch dieses Thema spielt in späteren Abschnitten dieser Arbeit noch eine große Rolle.

2.4 Kategorisierung von Webapplikationen

Um einen Überblick zu bekommen, welche Arten von Webanwendungen zurzeit im Web vertreten sind, werden nun kurz die in [Kapp04] vorgestellten Kategorien erläutert. Es gibt natürlich auch Sonderformen, die sich entweder nicht eindeutig in eine der Kategorien einteilen lassen oder sich über mehrere Kategorien erstrecken.

2.4.1 Dokumentzentrierte Websites

Eigentlich stellen sie die Vorläufer von Webanwendungen dar. HTML-Seiten mit statischen Inhalten werden auf einem Webserver abgelegt und im Falle einer Anfrage als Antwort an den Client übermittelt. Die Aktualisierung der Websites erfolgt manuell. Die Vorteile liegen in der Einfachheit, der Stabilität und den kurzen Antwortzeiten, Nachteile sind die hohen Kosten und die schnelle Veralterung von Informationen durch das manuelle Nachziehen von Änderungen.

2.4.2 Interaktive Webanwendungen

Diese Art der Webapplikationen bieten dem Benutzer durch die Einführung von CGI (Common Gateway Interface) und der HTML-Formulartechnik eine erste einfache Form der Interaktivität. Sowohl Webseiten als auch Verweise auf andere Websites können dadurch abhängig von Benutzereingaben dynamisch generiert werden.

2.4.3 Transaktionale Webanwendungen

Bei dieser Art von Webapplikationen kann der Benutzer nicht nur Informationen von einer Website abrufen, sondern es wird ihm auch die Möglichkeit geboten Modifikationen vorzunehmen. Dadurch können Datenbestände dezentral aktualisiert werden.

Die Voraussetzungen dafür schaffen (meist relationale) Datenbankmanagementsysteme, die eine effiziente, konsistente Verwaltung der ständig zunehmenden Datenmenge von Webanwendungen erlauben.

2.4.4 Workflow-basierte Webanwendungen

Sie erlauben die Abwicklung von Geschäftsprozessen innerhalb oder zwischen unterschiedlichen Unternehmen, Behörden und privaten Benutzern. Um die Funktionalität solcher Anwendungen zu ermöglichen ist die Verfügbarkeit verschiedener Schnittstellen notwendig, um die Zusammenarbeit unterschiedlicher Systeme zu gewährleisten.

2.4.5 Kollaborative Webanwendungen

Diese Art von Webanwendungen wird insbesondere zur Kooperation bei unstrukturierten Vorgängen eingesetzt, wobei der Bedarf an Kommunikation zwischen den beteiligten Benutzern besonders hoch ist. Kollaborative Webanwendungen bieten vorrangig Unterstützung um gemeinsame Informationen erstellen, bearbeiten und verwalten zu können.

2.4.6 Portalorientierte Webanwendungen.

Sie vereinen den Zugriff auf verteilte, potentiell heterogene Informationsquellen und Dienste im Sinne eines Single Point of Access. Viele Internetdienstleistungsunternehmen bieten sie als zentrale Anlaufseiten für den Einstieg ins Web an. Neben diesen allgemeinen Portalen findet man jedoch auch spezialisierte Portale wie Unternehmens-, Marktplatz- oder Communityportale.

2.4.7 Ubiquitäre Webanwendungen

Diese Kategorie von Webanwendungen stellt personalisierte Dienste zu jeder Zeit an jedem Ort und für eine Vielzahl von Endgeräten zur Verfügung, womit ein allgegenwärtiger Zugriff möglich wird. Eine wesentliche Voraussetzung dafür ist die Kenntnis des Kontexts, in dem die Webanwendung gerade verwendet wird, um bei einer Änderung des Kontexts dynamisch, also zur Laufzeit, Anpassungen an der Webanwendung vornehmen zu können.

2.5 Datenintensive Webanwendungen

Das Hauptaugenmerk dieser Arbeit in Bezug auf die verschiedenen Arten von Webanwendungen wird auf die Entwicklung von datenintensiven Webanwendungen gelegt. Auf den ersten Blick fällt auf, dass diese Kategorie nicht in den in Kapitel 2.4 beschriebenen aufscheint. Das kommt daher, dass sie sich nicht eindeutig einer dieser Kategorien zuordnen lassen. Sowohl transaktionale, als auch kollaborative und portalorientierte Webanwendungen können zugleich auch datenintensive Webanwendungen sein. Sie bezeichnen die Art von Applikationen, deren Hauptaufgabe es ist eine große Menge an strukturierten Daten zu verwalten und einer breiten Masse von Benutzern den Zugang zu diesen Daten zu ermöglichen [Ceri03]. In den häufigsten Fällen werden diese Daten als Datensätze in einer Datenbank gespeichert, die von so genannten Datenbankmanagementsystemen, abgekürzt DBMS – im speziellen von relationalen DBMS (RDBMS) – verwaltet werden. Mit dem Heranwachsen des Webs bekommen diese Arten von Webapplikationen immer mehr Bedeutung, da etwa E-Commerceanwendungen oder digitale Bibliotheken in diese Kategorie einzureihen sind. Die starke Ausbreitung datenintensiver Webanwendungen führt dazu, dass sich das Web in ein umfassendes und weltweit zugängliches Informationsmedium verwandelt hat, das von sehr vielen Firmen und Organisationen genutzt wird [Ceri03].

Die Komponente der Daten stellt in Bezug auf datenintensive Webanwendungen eine der Hauptbestandteile dar. Nach [Ceri03] ist der Einsatz von relationalen Datenbanken führend was die Erstellung von Informationssystemen betrifft und etwa 80% aller im Web verfügbaren Anwendungen basieren auf dem Inhalt von Datenbanken. In ihnen werden die Daten als so genannte Relationen gespeichert, die als Tabellen aus Zeilen und Spalten dargestellt werden. Jeder Datenbankstruktur liegt ein logisches (auch konzeptuelles) Datenmodell, das Konzepte zur Beschreibung der realen Welt, wie sie der Benutzer wahrnimmt, bietet. Dieses logische Datenmodell wird beim Datenbankentwurf in ein Implementierungsdatenmodell transformiert, das vom DBMS verwendet wird [Elma02].

2.5.1 Das logische Datenbankmodell und ER

Das Ziel des konzeptuellen Datenbankmodells ist es, die Daten, die von der Anwendung verwendet werden, auf eine formale und intuitive Art und Weise darzustellen. Dadurch wird das vorhandene Wissen über die Daten der Anwendung auf eine einfache, lesbare Weise zur Darstellung gebracht [Ceri03].

Für die Beschreibung des konzeptuellen Datenbankmodells hat sich als erfolgreichste und populärste Art der Darstellung das *Entity-Relationship-Model* (ER-Modell) herauskristallisiert, das 1976 von Peter Chen eingeführt wurde [Chen76]. Basis dieser Art der Datenmodellierung ist die Entität, die ein Objekt aus der realen Welt mit einer unabhängigen Existenz darstellt [Elma02]. Sie kann ein Objekt bezeichnen, das physisch existiert, z.B. ein Haus, ein Auto, oder etwas, das konzeptuell existiert, z.B. eine Firma, eine Lehrveranstaltung. Jede Entität besitzt Attribute, also Eigenschaften, die sie beschreiben, und jedes Attribut hat einen bestimmten Wert. Besteht eine Beziehung zwischen Attributen einer Entität und einem anderen Entitätstyp, so wird dieser Zusammenhang im ER-Modell nicht als Attribut, sondern als Beziehung dargestellt [Elma02]. Die Beziehungen werden durch Kardinalitätsverhältnisse (1 zu 1, 1 zu n, m zu n Beziehungen) charakterisiert, die eine Einschränkung auf die Anzahl der an der Beziehung beteiligten Instanzen (konkreter Ausprägungen) von Entitäten einführen [Ceri03].

2.5.2 Das Implementierungsdatenmodell und SQL

Ausgehend von einem vorhandenen konzeptuellen Datenbankmodell erfolgt bei der Erstellung einer Datenbank eine Transformation in ein Implementierungsdatenmodell. Dieses Modell, das in Bezug auf RDBMS häufig relationales Datenbankschema bezeichnet wird, beschreibt die Struktur der Datenbank (nicht jedoch den Inhalt). Es wird mit Hilfe einer so genannten Datendefinitionssprache (Data Definition Language DDL) definiert und in weiterer Folge vom DBMS in die physische Struktur der Datenbank übersetzt. Um dem Benutzer den Zugriff auf die Datenbank zum Einfügen, Löschen oder Ändern von Datensätzen zu ermöglichen, bieten die DBMS eine Datenmanipulationssprache (Data Manipulation Language DML). Eine umfassende Datenbanksprache, die beide der oben genannten Sprachen vereint ist die *Structured Query Language* (SQL) [Elma02].

SQL ist die am weitesten verbreitete Sprache um Daten aus Datenbanken abzufragen oder sie zu manipulieren. Einerseits bietet sie in Bezug auf die Struktur der Datenbank Anweisungen zum Erstellen (CREATE TABLE), Ändern (ALTER TABLE) und Löschen (DROP TABLE). Andererseits werden in Bezug auf die Manipulation der Daten umfangreiche Anweisungen wie Auswählen (SELECT), Einfügen (INSERT), Aktualisieren (UPDATE) oder Löschen (DELETE) bereitgestellt. SQL ist heute die Standardsprache für kommerzielle RDBMS. Im Jahre 1986 wurde der erste Standard durch eine Zusammenarbeit von ANSI und ISO verabschiedet, der in den Jahren 1992 und 2000 weiterentwickelt wurde.

Mit der Erstellung von logischen Datenbankmodellen sowie Implementierungsdatenmodellen und der weiteren Verwendung dieser Modelle wird sich Kapitel 7 noch weiter beschäftigen.

2.6 Architekturen

Die Qualität einer Webanwendung ist sehr häufig von ihrer zugrunde liegenden Architektur abhängig. Im Laufe der Zeit haben sich verschiedene Ansätze herauskristallisiert, welche Art von Architektur für welche Art von Webanwendung in geeigneter Weise eingesetzt werden kann, um den hohen Qualitätsansprüchen, die an die Anwendungen gestellt werden, gerecht zu werden. Der Begriff Architektur kann im Bereich der Softwareentwicklung mehrere Bedeutungen haben. Dieses Kapitel geht auf die strukturelle Bedeutung einer Architektur ein und beschreibt insbesondere die Schichtarchitekturen, die in Verbindung mit der Entwicklung von Webanwendungen sehr häufig eingesetzt werden.

2.6.1 2-Schichten-Architektur

Der Einsatz dieser Schichtarchitektur begann mit den Anfängen der Client-Server Applikationen. In diesen beinhaltete der Client die Benutzerschnittstelle und die Applikationslogik und der Server stellte im Allgemeinen eine relationale Datenbank dar, an die der Client direkt SQL-Anfragen senden konnte. Die Ergebnisse wurden von der Clientanwendung verarbeitet und für den Benutzer zur Darstellung gebracht.

Überträgt man die 2-Schichten-Architektur, auch Client/Server-Architektur genannt, auf Webanwendungen so stellt ein Web Server allein die Dienste für einen Client bereit [Anas01]. Der Client, der in den meisten Fällen aus einem Web-Browser besteht, stellt eine Anfrage einer bestimmten Ressource an den Server. Es kann sich hierbei um die Anforderung einer statischen HTML-Seite handeln, die keine Verarbeitung durch den Server benötigt, da die Seite so wie sie ist an den Client zurückgesendet werden kann, oder um eine Anfrage von Daten aus einem Datenbestand (etwa einer Datenbank). In diesem Fall übernimmt der Server die gesamte Verarbeitung der Anfrage (etwa in Form von CGI-Skripten) und generiert dynamische HTML-Seiten, die dann als Antwort an den Client zurückgegeben werden. [Kapp03]

Dieser Architekturansatz eignet sich vor allem für einfache Webanwendungen, die vorwiegend statische und nur wenige dynamische Inhalte an den Client weitergeben müssen. Bei komplexeren Webanwendungen, die eine Vielzahl von Benutzeranfra-

gen verarbeiten, große Datenbestände verwalten und die auch eine viel komplexere Geschäftslogik implementieren müssen, führt bei der Entwicklung kein Weg an einer mehrschichtigen Architektur vorbei.

2.6.2 ($n > 2$)-Schichten-Architekturen

Bei der 3-Schichten-Architektur kann man von einer Aufteilung in Präsentationsschicht, Anwendungslogik und Datenhaltungsschicht (Persistenzschicht) sprechen.

- Die *Präsentationsschicht* bezeichnet die Interaktion des Benutzers mit der Applikation mit Hilfe des Web-Browsers (Client) und die Verarbeitung und Weiterleitung der Benutzereingaben an den Application Server sowie die Übermittlung der Ergebnisse an den Client durch den Web Server.
- Die *Schicht der Anwendungslogik* kann man als die komplexeste der genannten Schichten bezeichnen. Hier befinden sich die Anwendungen und Dienste, die von der Applikation benötigt werden. Der Client beinhaltet bei dieser Art der Architektur keine Anwendungslogik. Diese Aufgabe hat der Application Server inne, der in Kapitel 2.8 noch näher erläutert wird.
- Die *Datenhaltungsschicht* bezeichnet die zu bearbeitenden Daten und wird sehr häufig von einem RDBMS dargestellt.

Um die Schicht der Anwendungslogik besser von den beiden anderen abzukoppeln, werden in vielen Fällen noch zwei weitere Schichten eingefügt, die für die Verbindung der drei Schichten verantwortlich sind, womit man eine 5-Schichten-Architektur erhält:

- Die *Steuerungsschicht* ist für die Trennung von Präsentation und Geschäftslogik verantwortlich. Sie leitet die Anfragen des Benutzers an die richtige Stelle der Anwendungslogik weiter und bereitet die Ergebnisse für die Präsentationsschicht auf.
- Die *Datenzugriffsschicht* trennt die Anwendungslogik von der Datenhaltung. Sie sorgt dafür, dass Zugriffsoperationen auf die Datenbasis aus den Objekten der Geschäftslogik ausgelagert werden.

Abbildung 2.1 zeigt noch einmal die genannten Architekturen im Überblick:

Schichtenarchitekturen

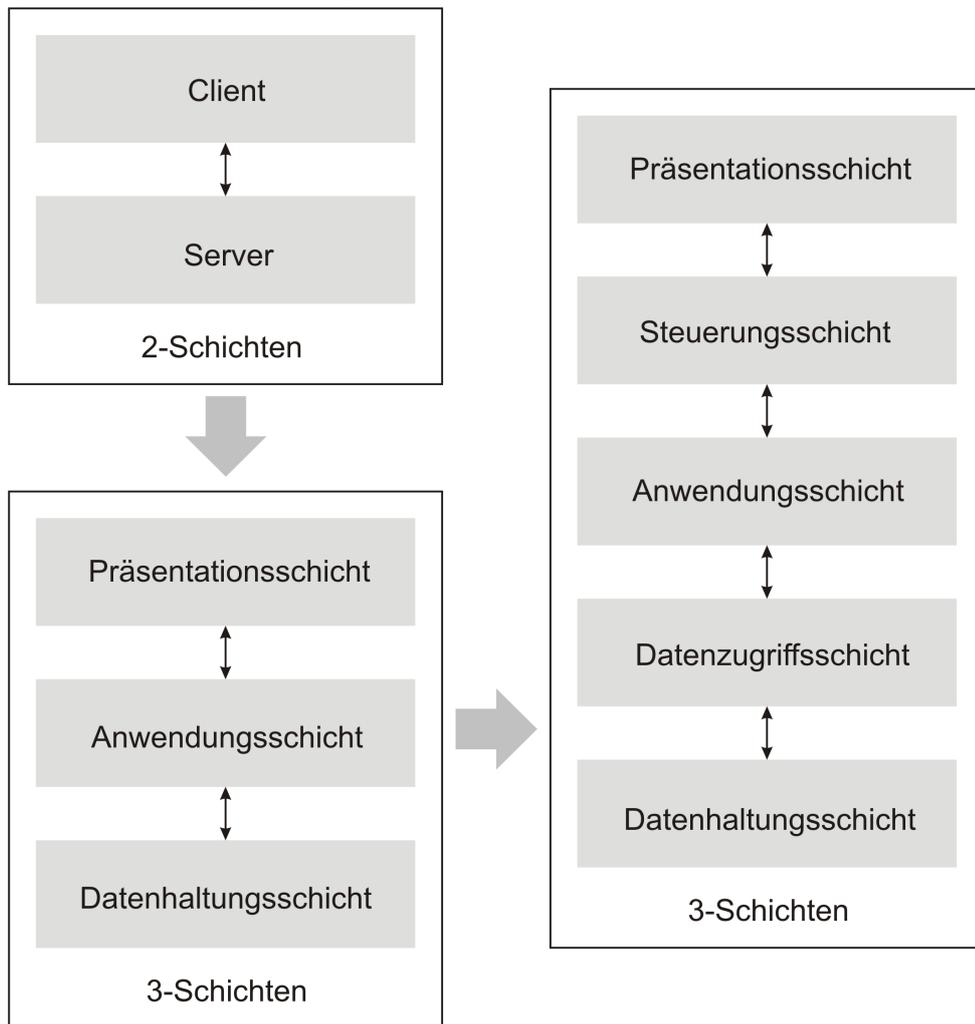


Abbildung 2.1 Schichtenarchitekturen

2.7 Web Server

Die Bezeichnung Web Server steht für eine Sammlung von Software-Modulen, die auf einem Computer installiert werden müssen, der mit dem Internet verbunden ist, um im Web zur Verfügung zu stehen. Der Web Server bearbeitet HTTP-Anfragen (Requests) und sendet HTML-Dokumente als Antwort (Response) zurück. Der Begriff Web Server ist eigentlich mehrdeutig. Einerseits bezeichnet er die Software, die auf dem Computer installiert ist, andererseits ist auch von dem Computer selbst als Web Server die Rede. Oft spricht man auch von einer Kombination von beiden. Am häufigsten ist jedoch die Software gemeint, die im Web bestimmte Dienste bereitstellen soll. Deshalb wird auch in der folgenden Beschreibung immer auf die erstgenannte Begriffsbestimmung des Web Servers Bezug genommen.

Ein Web Server beinhaltet typischerweise einen HTTP-Server, eine CGI-Umgebung und möglicherweise ein FTP-Modul. Manche Web Server beinhalten auch APIs um serverseitige Module wie Plug-Ins oder Extensions zu erstellen. Diese gehen aber schon eher in die Richtung eines Application Servers. [Jabl04] sprechen von einem Standard Application Server, der für die serverseitige Geschäftslogik zuständig ist. Dieser Standard Application Server muss nicht als physikalisches Modul vorhanden sein, sondern stellt ein logisches Modul dar, das für diverse Aufgaben im Hintergrund zuständig ist.

Jeder Internet-Host, auf dem ein Web Server läuft, beinhaltet genau einen HTTP-Server, der über den Port 80 auf Anfragen wartet. Es kann natürlich auch ein anderer Port sein, dieser wird jedoch standardmäßig als WWW-Port konfiguriert. Erhält der HTTP-Server eine Anfrage, so arbeitet er diese ab und erteilt dem Standard Application Server den Auftrag, entweder eine statische Seite abzurufen oder ein CGI-Programm auszuführen. In jedem Fall gibt der Standard Application Server eine HTML-Seite zurück, aus der der HTTP-Server eine HTTP-Antwort formt und an den Client zurücksendet [Jabl04].

In der frühen Entwicklung des WWW war CGI beinahe die einzige Möglichkeit, serverseitige Programme laufen zu lassen, die auf einen HTTP-Request reagieren sollten. Es gab im Großen und Ganzen keine alternativen Technologien. Das Vorhandensein von Programmen zur Durchführung von Geschäftslogik wurde immer wichtiger und alles geschah durch die Aufrufe von CGI, wodurch der Standard Application Server seiner Aufgabe bald nicht mehr gewachsen war. Aufgrund von schlechtem Ressourcenmanagement war die Performanz des Standard Application Servers bei der Ausführung von CGI-Programmen nicht mehr ausreichend. Eine alternative Technologie wurde notwendig, die den Standard Application Server erweitern soll was zur Verwendung einer weiterentwickelten Application Server Technologie führte [Jabl04].

2.8 Application Server

Der Begriff Application Server wird am häufigsten in Zusammenhang mit komponentenorientierter Programmierung verwendet. Der Application Server fungiert als Container, in dem verschiedene Komponenten der Geschäftslogik aufgesetzt werden und die Dienste, die der Container bereitstellt, verwenden können.

Application Servers stellen die Verbindung zwischen HTTP-Servern und den Back-End-Systemen (wie DBMS) dar und ersetzen die oben beschriebenen traditionellen Standard Application Server. Weiters stellen sie Funktionalitäten bezüglich Sicherheit, Transaktionskontrolle, hoher Performanz und Skalierbarkeit zur Verfügung [Jabl04].

Das in der Industrie verwendete Konzept des Application Servers unterscheidet sich etwas von dem oben beschriebenen. Diese so genannten Enterprise Application Servers weisen eine etwas unterschiedliche Zusammensetzung auf und beinhalten eine Vielzahl an Produkten wie etwa folgende [Jabl04]:

- Einen HTTP-Server
- Ein Web Service System
- Einen oder mehrere Container für verschiedene Arten von Geschäftslogik (wie etwa EJB Container)
- Manchmal einen Content Management Server
- Diverse Schnittstellen zur Verbindungsherstellung zu Back-End-Systemen
- Manche beinhalten sogar ein Datenbanksystem

Das ursprüngliche Problem in Besitz einer Webpräsenz zu sein hat sich dahingehend verlagert, die Webanwendungen besser und robuster zu machen. Der ursprünglich ausreichende Standard Application Server musste durch den starken Anstieg der Komplexität der Geschäftslogik erweitert werden und dem den Anforderungen besser gewachsenen Application Server weichen.

Kapitel 3

Design Patterns - Entwurfsmuster

In den Siebziger Jahren schrieb Christopher Alexander einige Bücher über Muster im Bereich der Ziviltechnik und der Architektur [Alex77] [Alex79]. Allmählich griff die Softwaregemeinschaft diese Idee auf und passte sie für die Softwareentwicklung entsprechend an. Entwurfsmuster in der Software wurden mit dem Buch [Gamm95] populär, in dem die Autoren eine Sammlung von dreiundzwanzig Entwurfsmustern zusammengefasst haben.

Kapitel 3.1 gibt einen kurzen Einblick darüber, was ein Design Pattern ist und wie einige bekannte Definitionen dafür lauten. Die darauf folgenden Kapitel 3.2 bis 3.8 stellen kurz eine Auswahl von Entwurfsmustern vor, die häufig bei der Erstellung von Webanwendungen eingesetzt und im Zusammenhang mit den in der Arbeit behandelten Webapplikations-Frameworks verwendet werden.

3.1 Definition

Es gibt in der Literatur zahlreiche Definitionen in Bezug auf Entwurfsmuster. Christopher Alexander definiert in [Alex79] ein Muster folgendermaßen:

Each Pattern is a three-part rule, which expresses a relation between a certain context, a problem and a solution. - Jedes Muster ist eine Regel, die aus drei Teilen besteht. Sie drückt eine Relation zwischen einem bestimmten Kontext, einem Problem und einer Lösung aus.

Eine weitere eher abstrakte Definition von Entwurfsmustern gibt Martin Fowler in [Fow197]:

A Pattern is an idea that has been useful in one practical context and will probably be useful in others. - Ein Muster ist eine Idee, die in einem speziellen Zusammenhang nützlich war und möglicherweise in einem anderen Zusammenhang nützlich sein wird.

Es gibt eine Vielzahl an Definitionen, die beschreiben sollen, was ein Entwurfsmuster ist. Im Allgemeinen dreht es sich jedoch immer um ein Problem und deren Lösung in einem bestimmten Kontext. In diesem Satz kommen drei Begriffe vor, die einer genaueren Erläuterung bedürfen [Alur03]:

Ein *Kontext* ist die Umgebung, die Situation oder miteinander verbundene Bedingungen innerhalb derer irgendetwas existiert.

Ein *Problem* ist eine unbeantwortete Frage, irgendetwas, das untersucht und gelöst werden muss.

Normalerweise ist das Problem durch den Kontext, in dem es existiert eingeschränkt. Hat man nun eine Lösung zu einem Problem innerhalb eines bestimmten Kontexts, so muss es sich hierbei nicht notwendigerweise um ein Muster handeln. Es ist notwendig, dass ein Muster auch erneut eingesetzt werden kann, ansonsten macht es keinen Sinn. Deshalb muss bei der Definition eines Entwurfsmusters auch die Wiederverwendung enthalten sein.

Einige der wichtigsten Charakteristika, die Entwurfsmuster ausmachen, werden in [Alur03] folgendermaßen zusammengefasst:

- Muster werden durch Erfahrung beobachtet.
- Muster werden typischerweise in einem strukturierten Format formuliert.
- Muster verhindern die Neuerfindung des Rades.
- Muster existieren auf verschiedenen Abstraktionsebenen.
- Muster unterliegen einer ständigen Verbesserung.
- Muster sind wieder verwendbare Artefakte.
- Muster kommunizieren Entwürfe.

- Muster können gemeinsam verwendet werden, um ein größeres Problem zu lösen.

Eine genauere Beschäftigung mit der Frage, was ein Entwurfsmuster ist, sowie eine Beschreibung aller der dreiundzwanzig oben genannten Entwurfsmuster würde den Rahmen dieser Arbeit jedoch bei weitem sprengen weshalb im folgenden Kapitel nur auf einige wenige Muster eingegangen wird, die im Zusammenhang mit der Entwicklung von Webanwendungen und insbesondere innerhalb der in der Arbeit beschriebenen Webapplikations-Frameworks häufig zum Einsatz kommen.

3.2 Model View Controller (MVC)

Das Model View Controller-Entwurfsmuster wurde in den späten 70er Jahren für Smalltalk entwickelt, um eine strikte Trennung von Daten (Model), Darstellung (View) und Kontrollfluss (Controller) zu erreichen und damit die Austauschbarkeit der einzelnen Komponenten zu fördern [Weße05]. Das MVC-Muster besteht somit im Wesentlichen aus den folgenden drei Komponenten [Fowl03]:

- Das *Model* stellt ein Objekt aus der Anwendungsschicht dar, das Daten repräsentiert.
- Die *View* repräsentiert die Darstellung des Modells in der Benutzerschnittstelle. Sie ist nur für die Anzeige verantwortlich und führt keine Aktionen aus, die zu einer Veränderung der Informationen des Modells führen.
- Der *Controller* nimmt die Benutzereingaben entgegen, manipuliert das Modell und weist die View darauf an, die Darstellung korrekt zu aktualisieren.

Abbildung 3.1 zeigt das Zusammenspiel der drei Komponenten und die Abhängigkeiten der MVC-Bestandteile untereinander. Es fällt auf, dass die View zwar abhängig ist vom Model, aber das Model unabhängig ist von der View. Dadurch können einerseits Veränderungen an der Programmierung des Modells ohne Rücksicht auf die View vorgenommen werden und es ist auch möglich, die Logik der View zu verändern, ohne dass das Model geändert werden muss.

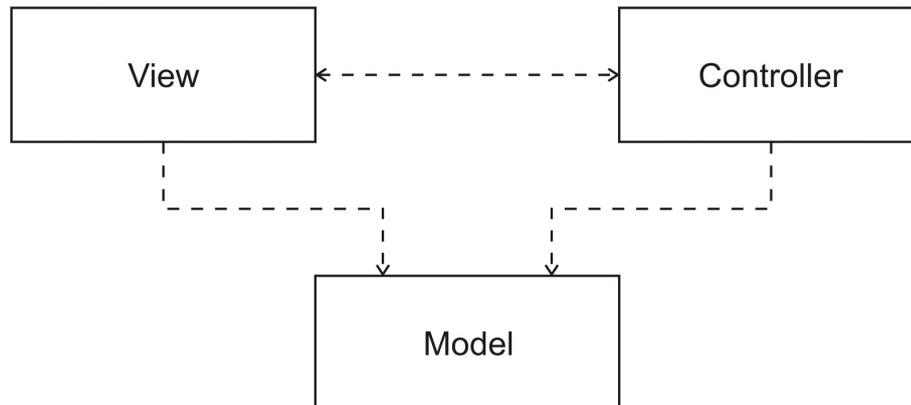


Abbildung 3.1 Das Model View Controller-Entwurfsmuster [Fowl03]

3.3 Model 1

In der Literatur zu Java-Webanwendungen werden häufig die Begriffe Model 1 und Model 2 verwendet. Ursprünglich stammt diese Terminologie aus der Spezifikation der JavaServer Pages Technologie [Sun06c], die zwei Ansätze für Java-Webanwendungen beschrieben hat. Die Begriffe Model 1 und Model 2 sind mittlerweile aus der Spezifikation entfernt worden, sie werden jedoch immer noch verwendet [Weße05].

Die Architektur des Model 1 beschreibt die alleinige Verwendung von JavaServer Pages [Sun06c] bei der Web-Programmierung. Ein Web-Browser hat direkten Zugriff auf eine JSP. Die Modellschicht wird durch eine JavaBean repräsentiert, die ihren Zustand mit einer Datenbank abgleicht. Die JSP steuert den Ablauf und stellt die Daten der JavaBean dar [Weße05]. Abbildung 3.1 veranschaulicht diesen Ansatz. Einfachere Architekturen verzichten außerdem auf die Verwendung von JavaBeans und sprechen innerhalb der JSP die Datenbank per JDBC [Sun06d] an wodurch es zu einer völligen Vernachlässigung einer Aufteilung in Schichten kommt, was einzig und allein – wenn überhaupt – bei sehr kleinen Webanwendungen zu empfehlen ist.

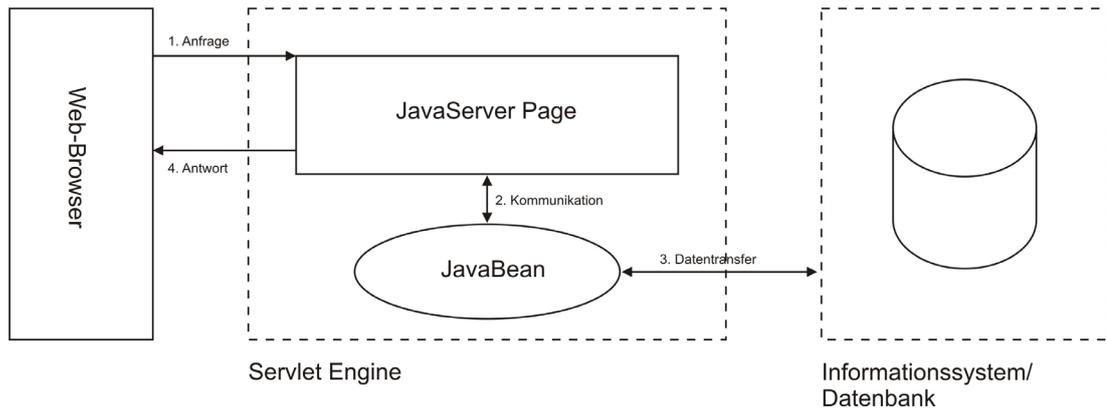


Abbildung 3.2 Model 1-Ansatz für Java-Webanwendungen [Weße05]

3.4 Model 2

Beim Model 2-Ansatz wird zusätzlich zu den JSPs ein Servlet verwendet. Dieses Servlet übernimmt die Aufgabe des Controllers aus dem MVC-Entwurfsmuster. Alle Anfragen eines Web-Browsers werden durch das Controller-Servlet der Webanwendung bearbeitet. Der Controller entscheidet aufgrund der HTTP-Anfrage, welche JavaBeans (Model) erzeugt werden müssen und welche JSP (View) für die Darstellung der Daten verantwortlich ist. Die JavaBeans beziehen ihre Daten aus einer Datenbank oder einem Informationssystem [Weße05]. Abbildung 3.2 stellt den beschriebenen Sachverhalt grafisch dar.

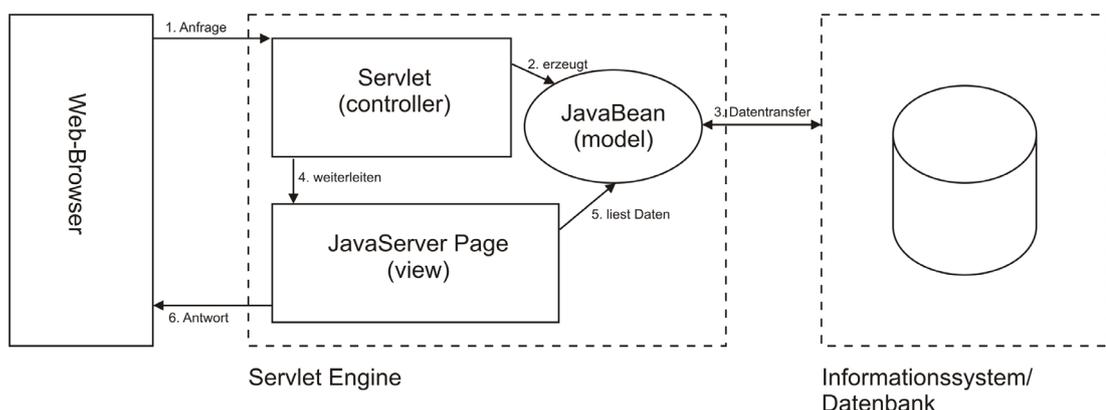


Abbildung 3.3 Das Model 2 für Java Webanwendungen [Weße05]

Der Unterschied zwischen dem Model 1 und dem Model 2 Ansatz besteht darin, dass bei Model 2-Anwendungen alle HTTP-Anfragen direkt an ein zentrales Controller-Servlet gesendet werden. Durch die Bereitstellung des Servlets als zentrales Steuerungselement wird eine klare Separation von Daten (Model), Darstellung (View) und Kontrollfluss (Controller) vorgenommen, wie es auch bei MVC der Fall ist.

Trotz dieser Gemeinsamkeit unterscheiden sich Model 2 und MVC dahingehend, dass Model 2 ein Ansatz für Webanwendungen ist und das klassische MVC-Entwurfsmuster bei Desktop-Anwendungen eingesetzt wird, wo sich die View-Objekte bei einem Model-Objekt registrieren und bei einer Zustandsänderung des Modells automatisch informiert werden. Dies ist bei Webanwendungen durch die Verwendung des zustandslosen HTTP-Protokolls nicht möglich und die Views können deshalb nicht automatisch über Zustandsänderungen des Modells benachrichtigt werden [Weße05].

Viele Frameworks, die für die Erstellung von Webanwendungen eingesetzt werden, basieren auf dem Model 2-Ansatz, wobei in den jeweiligen Dokumentationen der Frameworks meist von MVC die Rede ist. Der Vorteil der Verwendung dieser Ansätze ist, dass sich die damit erstellten Webanwendungen leichter pflegen und erweitern lassen, da die Views nicht direkt mit einander Verknüpft sind. Ihre Abhängigkeiten werden durch den Controller definiert, der etwa in einer XML-Datei konfiguriert wird, um ihn für andere Anwendungen wieder verwenden zu können [Weße05].

3.5 Front Controller

Ein Problem, das häufig bei der Erstellung von Webanwendungen auftritt, ist die Behandlung von eingehenden Anfragen an die Applikation. Gibt es keine zentrale Stelle, die für die Bearbeitung von Requests zuständig ist, so wird Code, der eben diese Aufgabe erledigen soll, an mehrere Stellen im Programm kopiert, so auch in die Views der Anwendung. Es ist jedoch nicht zielführend auf diese Weise Code der Applikationslogik mit der Präsentationslogik zu vermischen, da dies die Modularität der Anwendung erheblich beeinträchtigt. Außerdem stellt sich die Wartbarkeit des Systems äußerst schwierig dar, da eine einzelne Codeänderung an mehreren unterschiedlichen Stellen im Programm vorgenommen werden muss [Alur03].

Eine Lösung dieses Problems kann durch den Einsatz des Front Controller-Entwurfsmusters erzielt werden. Der Front Controller stellt dabei die zentrale Stelle für die Entgegennahme und Verwaltung von eingehenden Benutzeranfragen dar. Der Code wird an einer zentralen Position konzentriert und muss nicht in mehrere Teile des Programms kopiert werden, was einerseits die Fehleranfälligkeit reduziert und andererseits ist eine weit bessere Möglichkeit zur Wiederverwendung des Codes gegeben. Häufig verwendet ein Front Controller einen Application Controller, der für das Management von Actions und Views verantwortlich ist.

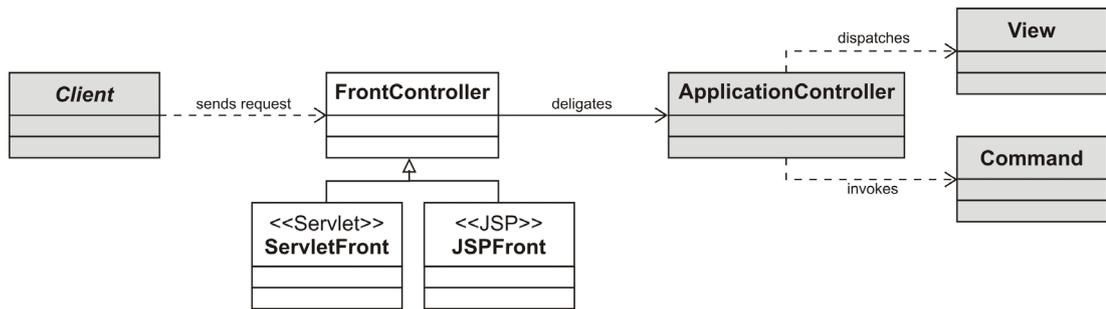


Abbildung 3.4 Front Controller-Entwurfsmuster [Alur03]

3.6 Application Controller

In der Präsentationsschicht müssen bei eingehenden Anfragen zwei wesentliche Bearbeitungsschritte durchgeführt werden [Alur03]:

- Das Auffinden und Aufrufen der Aktion, die für die Bearbeitung der Anfrage zuständig ist, das so genannte *Action Management*.
- Das Auffinden und das Weitergeben an die geeignete View, das so genannte *View Management*

Das Action- und das View-Management könnten zwar auch im Front Controller untergebracht werden, was eine Zentralisierung der Funktionalität zur Folge hätte. Bei steigender Komplexität der Anwendung ist es jedoch von Vorteil, die beiden Aufgaben an vom Front Controller getrennte Teile der Applikation zu delegieren, wodurch die Modularität, die Wartbarkeit und die Wiederverwendbarkeit erhöht werden. Dieses Ziel kann durch den Einsatz eines Application Controllers erreicht werden, dessen Aufgabe die Bereitstellung und Ausführung von anfrageverarbeitenden Komponenten, wie Commands oder Views, in der Anwendung ist [Alur03].

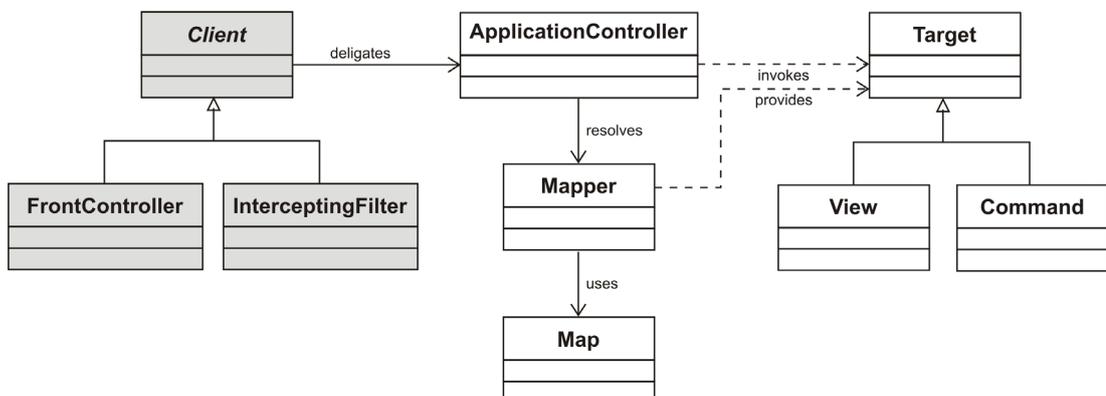


Abbildung 3.5 Application Controller-Entwurfsmuster [Alur03]

Dieses Entwurfsmuster wird als Teil des Webapplikations-Frameworks Apache Struts eingesetzt, wo deklarative Mappings verwendet werden um sowohl Action- als auch View-Management bereitzustellen.

3.7 Active Record

Ein Active Record bezeichnet ein Objekt, das eine Zeile einer Tabelle in der Datenbank darstellt, den Datenbankzugriff kapselt, sowie Geschäftslogik für die Daten enthält [Fowl03].

Die Verwendung von Active Records eignet sich besonders dann, wenn die eingesetzten Klassen sehr stark mit den Tabellen der verwendeten Datenbank übereinstimmen. Die Struktur eines Active Records sollte genau jener der Datenbanktabelle entsprechen: Ein Attribut der Klasse stimmt mit genau einer Spalte der zugehörigen Tabelle überein. Die Active Records sind für das Speichern und Laden in und aus der Datenbank verantwortlich und stellen weiters Methoden für den Datenbankzugriff bereit.

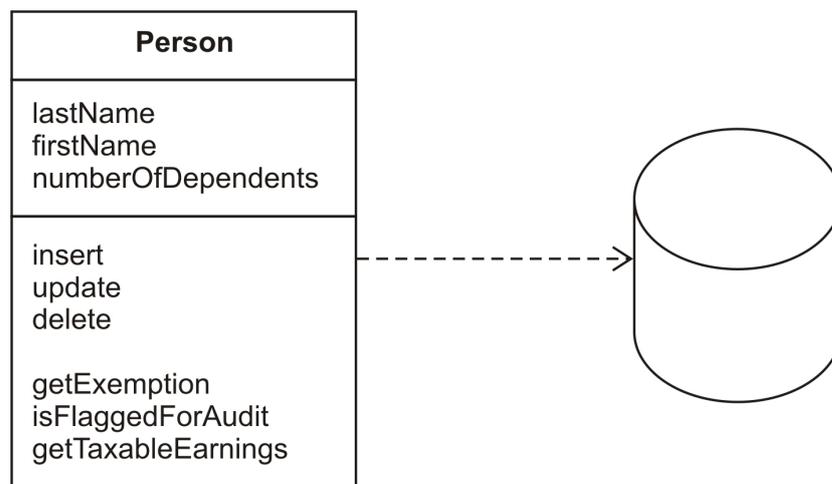


Abbildung 3.6 Active Record-Entwurfsmuster [Fowl03]

Das Active Record Pattern wird vor allem dann eingesetzt, wenn die Geschäftslogik nicht zu komplex ist, wie etwa bei CRUD-Operationen, und sich die Datenbanktabellen einfach auf die Objekte abbilden lassen. Ein wichtiger Bestandteil des Webapplikations-Frameworks Ruby on Rails, auf das in den folgenden Abschnitten noch genauer eingegangen wird, ist das Active Record-Modul, das das gleichnamige Entwurfsmuster implementiert und damit ebendiese Funktionsweise den mit dem Framework entwickelten Anwendungen bereitstellt. Eine genauere Beschreibung des Active Record-Moduls findet sich in Kapitel 4.4.1.

3.8 Data Access Object (DAO)

Data Access Objects werden verwendet, um die Anwendungsschicht von der Persistenzschicht zu trennen und somit den Zugriff auf die Daten zu kapseln. DAOs implementieren den Zugriff auf die Datenquelle und stellen die notwendigen Mechanismen für den Client bereit, egal um welchen Typ es sich bei der Persistenzschicht handelt. Das bedeutet, dass die gesamte Implementierung der Datenschicht vor der Anwendungsschicht verborgen bleibt und diese nur über die Schnittstellen des DAO mit der Datenbank kommuniziert. Somit können sich ohne Änderung der Schnittstellen sowohl die Datenschicht, als auch die DAOs ändern, ohne dadurch die Anwendungslogik korrigieren zu müssen [Alur03].

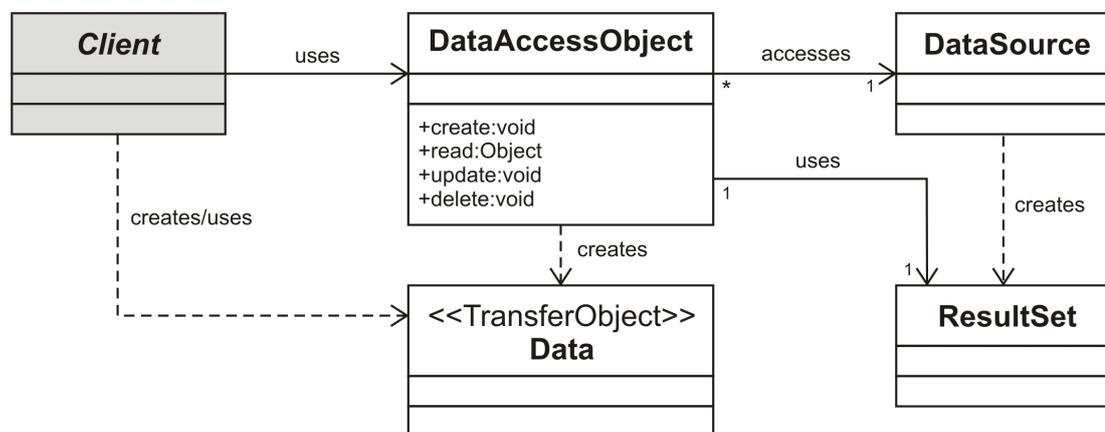


Abbildung 3.7 Data Access Object-Entwurfsmuster [Alur03]

Kapitel 4

Webapplikations-Frameworks – ein Vergleich

Dieses Kapitel betrachtet die Konzepte zweier Frameworks zur Erstellung von Webapplikationen genauer. Kapitel 4.1 gibt eine allgemeine Einführung in diese Technologie, was ist ein Framework und in welche Kategorien können Frameworks eingeteilt werden. Kapitel 4.2 nimmt das sehr weit verbreitete und bekannte Framework Apache Struts als Vertreter der Frameworks, die unter Verwendung der Programmiersprache Java [Sun06b] eingesetzt werden, genauer unter die Lupe. Kapitel 4.4 gibt eine ausführliche Beschreibung des immer beliebter werdenden, noch sehr jungen Frameworks Ruby on Rails [Hein06a], das die Programmiersprache Ruby [Ruby06] verwendet, über die der Exkurs in Kapitel 4.3 ein paar Worte verliert.. Kapitel 4.5 versucht anhand ausgewählter Kriterien die beiden angesprochenen Frameworks miteinander zu vergleichen und deren Gemeinsamkeiten und Unterschiede bzw. Vor- und Nachteile aufzuzeigen.

Die Hauptgründe dafür, warum gerade diese beiden Frameworks für den Vergleich ausgewählt wurden, sind einerseits die Verwendung zweier unterschiedlicher Programmiersprachen, wodurch interessante Unterschiede herausgearbeitet werden können. Andererseits basieren beide Frameworks auf dem in Kapitel 3.2 beschriebenen Model View Controller Entwurfsmuster, was die Ausgangsbasis dafür darstellt, die Frameworks anhand der Funktionalitäten, die sie in Bezug auf dieses Entwurfsmuster bieten, genauer zu untersuchen und aufzuzeigen.

4.1 Frameworks allgemein

Framework ist ein Begriff aus der Softwaretechnik und wird insbesondere im Rahmen der objektorientierten Softwareentwicklung sowie bei komponentenbasierten Entwicklungsansätzen verwendet. Es entspricht einer Sammlung verschiedener, individueller Komponenten mit definiertem Kooperationsverhalten und soll eine bestimmte Aufgabe erfüllen. Wörtlich übersetzt bedeutet Framework Gerippe, Gerüst, oder Rahmenwerk. Darin drückt sich aus, dass ein Framework in der Regel eine Anwendungsarchitektur, also einen Rahmen, vorgibt – im Gegensatz zu einer reinen Klassenbibliothek. Beispielsweise legt ein objektorientiertes Framework die Struktur wesentlicher Klassen und Objekte sowie ein Modell des Kontrollflusses in der Applikation fest. In diesem Sinn werden Frameworks im Wesentlichen mit dem Ziel einer Wiederverwendung architektonischer Muster entwickelt und genutzt [Wiki06b].

Da solche Muster meist nicht ohne die Berücksichtigung einer konkreten Anwendungsdomäne entworfen werden können, sind Frameworks oft domänenspezifisch oder doch auf einen bestimmten Anwendungstyp beschränkt. Nach [Froe98] können Frameworks in Bezug auf ihren Anwendungsbereich, ihre Anpassung, und der Art der Interaktion zwischen der Anwendung und dem Framework folgendermaßen eingeteilt werden:

4.1.1 Anpassung

Je nach der Art und Weise, wie Frameworks vom Entwickler verwendet werden müssen, unterscheidet [Froe98] zwischen White-Box- und Black-Box-Frameworks:

- *White-Box-Frameworks* bestehen unter anderem aus einer Reihe von abstrakten Klassen. Die Aufgabe des Entwicklers besteht bei der Verwendung solcher Frameworks darin, die abstrakten Methoden dieser Klassen in den abgeleiteten Unterklassen der konkreten Applikation zu überschreiben. Um dies tun zu können, ist eine gute Kenntnis der Framework-Klassen, also des Quellcodes des Frameworks, erforderlich.
- *Black-Box-Frameworks* sind Frameworks, deren Anpassungskonzept nicht auf unfertigen Klassen, sondern auf einer Anzahl fertiger Komponenten basiert. Die gewünschten Änderungen werden nicht durch die Vervollständigung von Unterklassen, sondern durch die Komposition dieser Komponenten

erreicht. Die Frameworks bieten bereits fertige Klassen an, die der Entwickler instanzieren und so zu einer Applikation zusammensetzen kann.

Verfügbare Frameworks sind zumeist Mischformen zwischen White-Box- und Black-Box-Frameworks. Mit steigendem Reifegrad tendieren Frameworks zum Black-Box-Ansatz.

4.1.2 Anwendungsbereich

Nach ihrem Anwendungsbereich lassen sich Frameworks wie folgt unterteilen [Froe98]:

- *Application Frameworks* bilden den Programmrahmen für eine bestimmte Klasse von Anwendungen in einer horizontalen Ebene über verschiedene Problembereiche, das heißt von Funktionen und Programmstrukturen, die bei allen Anwendungen dieser Klasse von Bedeutung sind.
- *Domain Frameworks* hingegen bilden den Programmrahmen für einen bestimmten Problembereich, also die vertikale Ebene innerhalb einer bestimmten Domäne. Sie stellen Funktionen und Strukturen zur Verfügung, die im Allgemeinen zur Lösung dieses Problembereichs benötigt werden.
- *Support Frameworks* bieten Basisfunktionalitäten, auf denen weitere Frameworks oder Anwendungen aufgebaut werden können. Sie fassen Klassen und Methoden zusammen, die Unterstützung auf einer bestimmten Abstraktionsebene für ein breites Anwendungsfeld bieten.

4.1.3 Interaktion

Eine weitere Möglichkeit Frameworks untereinander zu unterscheiden findet sich ebenfalls in [Froe98] und sie ist abhängig davon, wie das Framework mit der Applikation interagiert:

- *Aufgerufene Frameworks* (pull-based-interaction) sind in etwa gleichzusetzen mit Codebibliotheken. Die Applikation verwendet das Framework, indem sie Funktionen und Methoden aufruft, die von der Bibliothek bereitgestellt werden.
- *Aufrufende Frameworks* (push-based-interaction) rufen die in der Applikation erstellten Methoden und Komponenten auf.

4.2 Apache Struts

Das *Apache Struts Framework*, kurz Struts, ist ein populäres Open Source Webapplikations-Framework, das von der Apache Software Foundation geschaffen wurde, um Java-Webanwendungen zu erstellen [Apac06]. Das Framework bietet eine stabile Applikationsarchitektur und ermöglicht es so, eine Web-Applikation zu entwickeln, die auf bewährten Standards wie dem MVC-Paradigma basiert. Somit ist es möglich, qualitativ hochwertigen Code zu erzeugen, der überschaubar und wartbar bleibt. Wenn man von der notwendigen Einarbeitungszeit absieht, kann sich ein Entwickler so auf die eigentliche Applikation konzentrieren und muss sich nicht um die Mechanik kümmern [Cava04].

Struts ist ein Framework, das sehr stark auf der Konfiguration der Applikation mit Hilfe von XML-Dateien basiert, die beim Starten der Anwendung benötigt werden um notwendige Komponenten zu erstellen und zu laden. Die beiden wichtigsten Konfigurationsdateien sind die Dateien „web.xml“ und „struts-config.xml“. Die Datei „web.xml“ auch *Deployment Descriptor* genannt dient wie bei jeder Java-Webapplikation der allgemeinen Konfiguration der Webanwendung und ist für die Initialisierung allgemeiner Ressourcen zuständig [Apac06]. Hier wird auch das ActionServlet einer Struts-Anwendung eingetragen. Die Datei „struts-config.xml“ oder auch Struts Konfigurationsdatei wird vom ActionServlet verwendet und dient der Initialisierung anwendungsspezifischer Ressourcen wie ActionForms, ActionMappings oder ActionForwards [Apac06].

Wie eingangs erwähnt basiert die Architektur sehr vieler Frameworks zur Erstellung von Webanwendungen auf dem MVC-Entwurfsmuster (siehe Kapitel 3.2), so ist es auch bei Struts. Die folgenden Unterkapitel sollen darauf eingehen, wie das Framework aufgebaut ist und welche Funktionalität es in Bezug auf dieses Design Pattern bietet. Deshalb findet sich hier auch eine Einteilung der Kapitel in Model, View und Controller.

4.2.1 Model

Struts bietet nach [Cava04] selbst keine eigene Funktionalität, was das Model und damit die Datenzugriffsschicht betrifft. In Struts werden vorhandene Modell-Architekturen integriert. Es beschränkt sich jedoch nicht auf eine einzelne, sondern meist werden folgende, auf Java basierende Technologien verwendet: Enterprise JavaBeans (EJB) [Sun06a], Java Data Objects (JDO) [Sun03] oder sogar Frame-

works, die eine Implementierung der Datenzugriffsschicht darstellen, wie Hibernate [Hibe06].

Manche Quellen in der Literatur sprechen jedoch davon, dass es doch einen Teil von Struts gibt, der der Model-Komponente anzurechnen ist nämlich die ActionForms. Bei [Bond03] etwa wird der Aufbau einer Struts-Applikation folgendermaßen dargestellt:

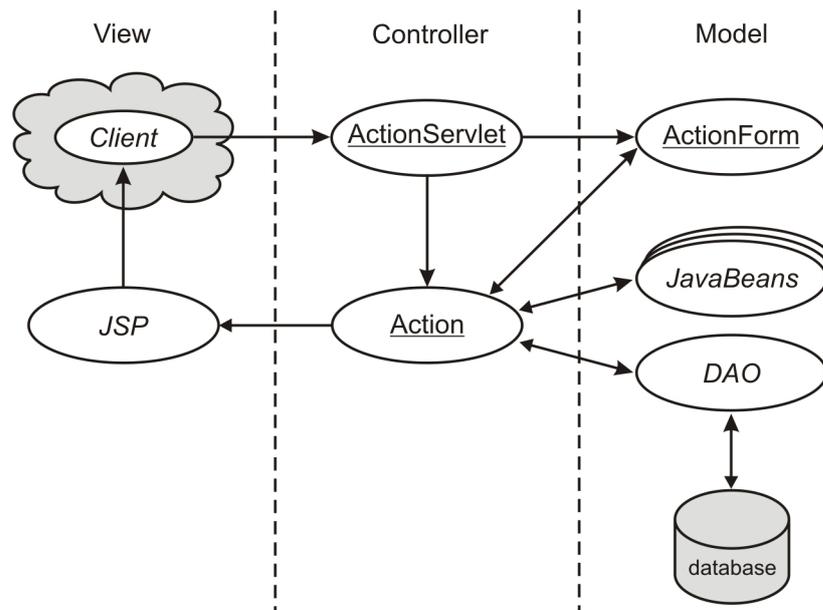


Abbildung 4.1 Hauptkomponenten einer Struts-Applikation [Bond03]

ActionForm

Diese Klassen werden verwendet, um Benutzereingaben aus einem HTML-Formular zu speichern und sie an die Action-Klasse weiterzugeben. Da Benutzer oft ungültige Daten eingeben, ist es für eine Webapplikation notwendig, diese Daten temporär zu speichern, um sie bei Fehlern wieder anzeigen zu können. Da kommt die Action-Form-Klasse ins Spiel. Sie dient als Zwischenspeicher und nimmt die Benutzerdaten auf, während sie den Validierungsprozess durchlaufen. Man kann auch sagen, die ActionForm dient als „Firewall“ für die Applikation, da sie verdächtige oder ungültige Daten von der Geschäftslogik fernhält, bis sie als korrekt validiert worden sind [Cava04].

Weiters können ActionForms von JSPs verwendet werden um die Daten, die aus der Geschäftslogik zurückkommen, darzustellen. Dadurch werden die HTML-Formulare konsistent, da sie die Daten immer von der ActionForm beziehen und nicht von verschiedenen Modellobjekten.

Werden die Benutzerdaten als gültig validiert, so wird die ActionForm an die execute()-Methode der Action-Klasse übergeben. Von dort können die Daten aus der ActionForm gelesen und an die Geschäftsschicht weitergegeben werden. Die ActionForm selbst sollte jedoch nicht an die Geschäftslogik übergeben werden. Ein besserer Ansatz ist es nur die Daten der ActionForm an ein Objekt der Modellschicht zu übergeben, um eine strikte Trennung der beiden Schichten zu erreichen.

Weiters muss nicht für jedes HTML-Formular eine eigene ActionForm erstellt werden. Eine ActionForm kann mit mehreren ActionMappings verknüpft werden, was bedeutet, dass eine ActionForm unter mehreren HTML-Formularen geteilt werden kann.

validate()-Methode

Diese Methode ist verantwortlich für die Durchführung der Validierung innerhalb der ActionForm-Klasse. Die Konfiguration der ActionForms wird in der Struts Konfigurationsdatei vorgenommen. Es müssen die Elemente <name>, <scope> und <validate> in jedem ActionMapping, das die ActionForm verwendet, spezifiziert werden.

ActionErrors

Diese Klasse kapselt ein oder mehrere Fehler, die in der Applikation auftreten. Die validate()-Methode führt die Validierungen durch und fügt dem ActionErrors-Objekt ActionMessage-Objekte hinzu, wenn Fehler auftreten.

In der validate()-Methode können jegliche Validierungen der Eingaben durchgeführt werden. Es empfiehlt sich jedoch die Verwendung des Validator-Frameworks, in dem die meisten Standard-Validierungsregeln bereits definiert sind (siehe nächster Abschnitt).

4.2.2 View

Die Komponenten, die in Struts für die View zuständig sind, basieren primär auf der Verwendung der JavaServer Pages Technologie (JSP) [Sun06c]. Die Funktionen, die die Komponenten des Frameworks bieten und worum sich der Programmierer selbst nicht zu kümmern braucht sind:

- Unterstützung für Internationalisierung

- Akzeptanz von Benutzereingaben
- Validierung
- Fehlerbehandlung

Zusätzliche Komponenten, die von oder in Verbindung mit JSP für die Darstellung von Views verwendet werden können, sind:

- HTML-Dokumente
- JSP Custom Tag Libraries
- JavaScript
- Stylesheets
- Multimedia Dateien
- Message Resource Bundles sowie
- ActionForm-Klassen

Internationalisierung

Internationalisierung bezeichnet den Prozess die Software so zu entwerfen, dass sie mehrere Sprachen und Gebiete unterstützt und man bei hinzufügen einer neuen Sprache kein Reengineering also keine Neuentwicklung der Applikation durchführen muss [Cava04]. Eine Applikation, die Internationalisierung unterstützt, besitzt folgende Eigenschaften:

- Zusätzliche Sprachen können unterstützt werden, ohne den Code verändern zu müssen.
- Textelemente, Nachrichten und Bilder werden außerhalb des Codes gespeichert.
- Kulturabhängige Daten wie Datum und Uhrzeit, Dezimalwerte und Währungen werden an die Sprache und den Aufenthaltsort des Benutzers angepasst (und damit richtig formatiert).
- Nicht-Standard Zeichensätze werden unterstützt.

- Die Applikation kann rasch an neue Sprachen und / oder Regionen angepasst werden.

Will man eine Applikation internationalisieren, so müssen alle diese Eigenschaften umgesetzt werden. Es macht keinen Sinn, etwa den Text in der korrekten Sprache anzuzeigen, aber die Zahlen und Währungen nicht richtig darzustellen. Der Benutzer würde ein derartiges Verhalten der Applikation als nicht angenehm empfinden. Java [Sun06b] selbst bietet durch die Klasse `java.util.Locale` eine umfangreiche Funktionalität was die Internationalisierung von Applikationen betrifft.

Die Unterstützung, die Struts in Bezug auf Internationalisierung bietet, beschränkt sich auf die Präsentation von Text und Bildern innerhalb der Webapplikation. Eine Funktionalität, die Eingaben in nichttraditionellen Sprachen unterstützen würde, ist in dem Framework nicht vorhanden. Abhängig davon, wie die Konfiguration der Applikation mit Hilfe von Struts aussieht, kann das Framework das Gebietschema des Benutzers herausfinden und es in dessen Session speichern. Wurde das Gebietschema erkannt, so kann Struts mit Hilfe von Resource Bundles die darin enthaltenen Texte und Ressourcen für die jeweilige Sprache bzw. das jeweilige Gebiet herausuchen und darstellen [Cava04].

Tiles-Framework

Um die Trennung von Inhalt und Darstellung besser zu implementieren bietet Struts einen eigenen Ansatz, der auch eine Reduzierung von Redundanzen mit sich bringt: das Tiles-Framework.

Es basiert auf einem Template-Mechanismus, der es erlaubt ein Layout zu erstellen und den Inhalt der Seiten dynamisch in das Layout einzufügen. Das Tiles-Framework bietet folgende Funktionalitäten [Cava04]:

- Template Unterstützung
- Dynamische Seitengenerierung
- Bildschirm Definition
- Unterstützung für Tile- und Layout-Wiederverwendung
- Unterstützung für Internationalisierung

Der Inhalt wird weiterhin mit Hilfe von JSP und JavaBeans dargestellt. Das Layout wird in einer eigenen JSP oder in XML-Dateien spezifiziert.

Was ist ein Tile?

Ein Tile ist ein Bereich innerhalb einer Webseite. Eine Webseite kann aus nur einem Bereich bestehen oder in mehrere Bereiche aufgeteilt sein. Eine JSP besteht im Normalfall aus mehreren Bereichen oder Tiles. Der wichtigste Punkt an einem Tile ist, dass es wiederverwendbar ist. Dies gilt sowohl für Layouts als auch für Inhalte. Es ist möglich Tile-Komponenten innerhalb einer Applikation – ja sogar innerhalb mehrerer, unterschiedlicher Applikationen – zu verwenden [Cava04].

Ein Layout im Tiles-Framework entspricht einem Template. Ein Layout wird dafür verwendet, eine Menge von Tiles zusammenzufügen um das Format einer Webseite zu spezifizieren. Layouts werden ebenfalls als Tiles angesehen, die wiederverwendet werden können, und es ist üblich eine Bibliothek an Layouts anzulegen um diese in verschiedenen Projekten wieder verwenden zu können. Das Tiles-Framework selbst beinhaltet einige vorgefertigte Layouts, die unverändert verwendet oder an die eigenen Bedürfnisse angepasst werden können.

Validator-Framework

Wie zuvor erwähnt ist es zwar möglich die Validierung in jede ActionForm-Klasse zu integrieren, diese Vorgehensweise bringt jedoch einige Nachteile mit sich:

- Es kommt zu Redundanzen innerhalb der Applikation, wenn die Validierungslogik in jeder ActionForm implementiert wird. Auch wenn man nur eine ActionForm verwendet, kann es vorkommen, dass man die Validierungsregeln kopieren muss.
- Das zweite Problem stellt die Wartung der Applikation dar. Muss die Validierung in einer ActionForm geändert werden, so muss diese neu kompiliert werden.

Das Validator-Framework erlaubt es, die gesamte Validierungslogik aus der ActionForm zu entfernen und sie für die Webapplikation deklarativ in externen XML-Files festzulegen. Damit wird die Anwendung in dieser Hinsicht viel einfacher zu warten und konfigurieren. Ein weiterer großer Vorteil des Frameworks ist, dass es leicht erweiterbar ist. Es beinhaltet zwar einerseits sehr viele Standard-Validierungsregeln andererseits ist es auch sehr leicht möglich eigene Regeln hinzuzufügen.

4.2.3 Controller

Das Front Controller Design Pattern (siehe Kapitel 3.5) verwendet einen einzelnen Controller um alle Benutzeranfragen durch eine zentrale Stelle zu leiten. Dieses Entwurfsmuster bringt einige Vorteile für die Funktionalität einer Webapplikation wie etwa die Zentralisierung von wichtigen Diensten wie Sicherheit, Internationalisierung oder Berichterstattung an einer Stelle nämlich dem Controller. Dies erlaubt eine konsistente Anwendung dieser Dienste für alle Anfragen. Ist es nötig Änderungen am Verhalten dieser Dienste vorzunehmen, so können die Veränderungen, die möglicherweise die gesamte Applikation betreffen an einem relativ kleinen und abgeschlossenen Bereich der Applikation durchgeführt werden [Cava04].

Der Struts-Controller hat unter anderem die folgenden, wesentlichen Aufgaben:

- Abfangen von Client-Requests.
- Mapping eines jeden Requests an eine spezifische Geschäftsoperation.
- Sammeln von Ergebnissen der Geschäftsoperation und Bereitstellung der Ergebnisse für den Client.
- Anweisung der View, die für die Darstellung – basierend auf dem momentanen Zustand und den Ergebnissen der Geschäftsoperation – zuständig ist.

Im Struts Framework sind folgende Komponenten für die Aufgaben des Controllers verantwortlich:

ActionServlet

Das ActionServlet dient dazu alle Anfragen an die Applikation abzufangen. Jeder Request des Clients muss zuerst das ActionServlet passieren, bevor er an irgendeine andere Stelle fortfahren kann. Erhält das ActionServlet einen Request über die doGet()- oder doPost()-Methode, so wird die process()-Methode aufgerufen um die Anfrage zu bearbeiten.

Wird eine erweiterte ActionServlet-Klasse verwendet, so muss diese als <servlet>-Element im Deployment Descriptor der Webapplikation eingefügt und konfiguriert werden. Bei Verwendung der Standard-Implementierung, die in Struts enthalten ist, ist das nicht notwendig.

RequestProcessor

Als nächster Schritt ruft das `ActionServlet` die Methode `process()` des `RequestProcessor` auf und übergibt dabei die aktuellen `Request`- und `Response`-Objekte als Parameter.

Die `RequestProcessor`-Klasse wurde eingeführt um die Anfragenbearbeitung einer Applikation anpassen zu können. Obwohl diese Anpassung auch durch eine Erweiterung von `ActionServlet` möglich wäre, wurde die Einführung dieser Klasse notwendig, damit jedes Modul der Webapplikation einen eigenen angepassten `RequestProcessor` besitzt. Außerdem konnte so das `ActionServlet` entlastet werden. Die `RequestProcessor`-Klasse beinhaltet einige Methoden, die überschrieben werden können, um die notwendigen Funktionalitäten zu erreichen. Wie beim `ActionServlet` ist auch beim `RequestProcessor` bei Verwendung einer angepassten Klasse eine Konfiguration notwendig. Diesmal muss jedoch ein `<controller>`-Element in die Struts-Konfigurationsdatei eingefügt werden.

Action

Die `Action`-Klasse stellt die Brücke zwischen Client-Anfragen und Geschäftsoperationen dar. Eine `Action` wird normalerweise entworfen um eine einzelne Geschäftsoperation im Auftrag des Clients durchzuführen (was nicht bedeutet, dass eine `Action` nur eine Aufgabe ausführen kann) [Cava04]. Es sollte bei der Implementierung darauf geachtet werden, dass für jede funktionale Einheit eine `Action` erstellt wird und die Funktionalitäten nicht vermischt werden. Die wichtigste Methode der `Action`-Klasse stellt die `execute()`-Methode dar, die vom `RequestProcessor` aufgerufen wird und ein `ActionForward`-Objekt zurückliefert.

ActionForward

Die `execute()`-Methode der `Action`-Klasse besitzt als Rückgabewert ein `ActionForward`. Ein `ActionForward` stellt nach [Cava04] die Abstraktion einer Webressource dar (im Normalfall eine `JSP` oder ein `Servlet`). Mithilfe der Methode `findForward()` des `ActionMappings` wird das `ActionForward` herausgefunden, das in der Struts-Konfigurationsdatei eingetragen ist. Das Argument der `findForward()`-Methode muss mit einem der Namen übereinstimmen der entweder in den `Global-Forwards` oder in der `Action`, von der aus die Methode aufgerufen wurde, spezifiziert ist. Danach kommt es entweder zum `Forward` oder zum `Redirect`.

Innerhalb des Frameworks gibt es einige vorgefertigte `Action`-Klassen, die leicht in die Applikation integriert werden können und dem Entwickler eine Menge Zeit ersparen.

4.3 Exkurs: Ruby

Die Programmiersprache Ruby wurde 1995 von Yukihiro Matsumoto [Ruby06] mit dem Ziel entwickelt, eine vollständig objektorientierte, leicht zu erlernende Programmiersprache zu schaffen, die die Vorteile vieler vorhandener Sprachen (wie etwa Smalltalk, Perl, Python,...) miteinander vereint und nicht nur eine Script-Sprache darstellen soll. Ruby ist elegant aufgebaut, besitzt eine saubere Syntax und enthält mächtige eingebaute Funktionen. Klassen und Objekte sind in Ruby sehr natürlich zu bearbeiten und die Programme bleiben übersichtlich und gut lesbar [Röhr02].

Der Entwickler von Ruby war bemüht folgende Eigenschaften der Sprache herauszuarbeiten:

- *Konsequent*: Eine kleine Menge von Regeln sollte die ganze Sprache definieren. Ruby vertritt das Prinzip der kleinsten Überraschung und der geringsten Anstrengung.
- *Kompakt*: Es gibt keine überflüssigen Sprachelemente.
- *Flexibel*: Eine Sprache sollte genug Spielraum bieten, sodass man alles mit ihr machen kann. Dabei sollten einfache Aufgaben auch einfach zu lösen und komplexe Probleme zumindest überhaupt lösbar sein.
- *Anpassungsfähig*: Computersprachen sollen Menschen helfen, sich auf das Problem zu konzentrieren und nicht mit der Sprache zu kämpfen, nur weil sich ein Algorithmus in eben dieser nicht adäquat formulieren lässt.

Ruby ist weiters sehr portabel und dadurch auf den unterschiedlichsten Plattformen einsetzbar. Die dynamische Typisierung trägt dazu bei, dass Entwürfe mit einem minimalen Aufwand erstellt werden, und bei Bedarf auch zu größeren Programmen ausgebaut werden können, ohne dass sprachbedingte Hürden zu überwinden sind. Die Sprache ist weiters sehr anpassungsfähig; sie ist einerseits innerhalb der Sprache selbst erweiterbar, andererseits kann sie sehr leicht mit Bibliotheken dritter kombiniert werden. Ihre Leichtgewichtigkeit und der geringe Anteil an Systemressourcen, den die Sprache benötigt, zählen zu weiteren Vorteilen [Röhr02].

Für eine genauere Betrachtung des Aufbaus und der Arbeitsweise sowie von Beispielen der Programmiersprache sei auf [Ruby06], [Röhr02] oder [Thom04] verwiesen.

4.4 Ruby on Rails

Ruby on Rails, kurz Rails, wurde 2004 von David Heinemeier Hansson [Hein06a] als quelloffenes Framework zur Erstellung von Webanwendungen entwickelt. Es ist in der Programmiersprache Ruby geschrieben und die Architektur der Webanwendungen, die man mit dem Framework erzeugen kann, basiert ebenso wie bei Apache Struts auf dem MVC-Entwurfsmuster. Die Philosophie, die hinter der Idee von Rails steckt, wird von den beiden Konzepten *DRY* [Hunt03] und *Konvention vor Konfiguration* dominiert [Thom06]:

- *DRY* steht für *Don't Repeat Yourself*, also „Wiederhole dich nicht“, was bedeutet, dass jedes Stückchen Wissen in einem System nur an einer einzigen Stelle zum Ausdruck gebracht werden soll. Nach [Thom06] findet man in einer Rails-Anwendung nur sehr wenige Wiederholungen. Was man zu sagen hat, sagt man an einer Stelle.
- *Konvention vor Konfiguration*: Dieses Prinzip besagt, dass Rails für beinahe jeden Aspekt bei der Zusammenstellung der Webanwendung über sinnvolle Vorgabewerte verfügt. Befolgt man die Konventionen, so kommt man laut [Thom06] bei der Erstellung einer Rails-Anwendung mit weniger Quellcode aus, als bei einer typischen Java-Webanwendung für die XML-Konfiguration benötigt werden. Ist es weiters notwendig, sich über die Konfiguration hinwegzusetzen, so macht es Rails einem leicht dies zu tun.

4.4.1 Active Record (Model)

Was die Model-Komponente des Frameworks in Bezug auf MVC betrifft, unterscheidet sich Rails wesentlich von anderen Frameworks. Mit Rails wird ein vollständiges Framework ausgeliefert, das einerseits vom Entwickler einer Rails-Anwendung verwendet werden kann (aber nicht muss) und andererseits auch außerhalb von Rails eingesetzt werden kann, da es sich um ein unabhängiges Framework handelt: *Active Record*. Es ist kein Zufall, dass es bei dieser Komponente, die für die Kapselung des Datenzugriffs in einer Rails-Anwendung zuständig ist, zu einer Namensgleichheit mit dem in Kapitel 3.8 beschriebenen Entwurfsmuster kommt, da es eine Implementierung eben dieses Design Patterns darstellt. Bei dem Active Record-Modul handelt es sich um eine Datenzugriffsschicht, die im Wesentlichen folgende Abbildung der relationalen Daten auf das objektorientierte Modell vornimmt:

Tabellen werden auf Klassen abgebildet, Zeilen auf Objekte, Spalten auf Attribute.

Dadurch werden keine Datenbankabfragen direkt in den Code eingebunden, wie es bei den serverseitigen Scriptsprachen häufig der Fall ist. Es existieren eine Menge an Bibliotheken, die ebendiese Abbildung – auch genannt *objekt-relationales Mapping* (ORM) – umsetzen, jedoch unterscheidet sich Active Record von ihnen wesentlich in Hinblick auf ihre Konfiguration [Thom06]. Active Record minimiert den Konfigurationsaufwand für Entwickler durch die Einführung von Konventionen. In diesem Fall wird eine Namenskonvention verwendet um die Beziehung zwischen einer Datenbanktabelle und der zugehörigen Modellklasse herzustellen. Ist der Name des Modells die Einzahl des Tabellennamens, so ordnet das Framework die Tabelle dem Modell zu. Rails beherrscht auch irreguläre Mehrzahlbildungen wie *company - companies*. Das Mapping ist jedoch auf die englische Form von Singular und Plural beschränkt [Mari06].

4.4.2 Action Pack (View und Controller)

Die View nimmt Anfragen des Benutzers entgegen und erzeugt Ereignisse für den Controller und der Controller stellt die Daten für die View bereit. Aufgrund der Tatsache, dass innerhalb des MVC-Musters die beiden Komponenten View und Controller sehr stark miteinander in Verbindung stehen, wurden die Funktionalitäten der beiden innerhalb von Rails zu einem Modul zusammengefasst: dem *Action Pack*. Das bedeutet jedoch nicht, dass der Code der beiden Komponenten in einer Rails-Anwendung vermischt wird. Rails stellt eine klare Trennung für den Code der Darstellungs- und der Anwendungsschicht bereit [Thom06]:

View

Die Komponente, die in Rails für die Darstellung der Ausgabe verantwortlich ist, ist das *ActionView*-Modul und ist Bestandteil von Action Pack. Rails stellt eine Vielzahl von Möglichkeiten der Darstellung von Websites auf dem Browser bereit. Die einfachste Methode ist statischer Text in Form von HTML. Da jedoch bei der Entwicklung von Webanwendungen weniger statische Inhalte als dynamisch generierter Kontext in den Vordergrund treten, liegt die Verwendung von RHTML-Templates bei der Entwicklung mit Rails als View nahe. In RHTML-Template ist in seiner einfachsten Form eine HTML-Datei. Um den dynamischen Charakter zu erhalten wird sie mit Hilfe von ERb (Embedded Ruby)-Code ergänzt. Man kann diese Templates mit JSP-Dateien in Java-Webanwendungen vergleichen, wo Javacode in den HTML-Text eingebettet wird. Bei der Erstellung dieser Templates, sollte man darauf achten, sie nicht mit Code zu überladen und vor allem keine Programmlogik zu integrieren.

In Rails ist auch die Ausgabe im XML-Format mit Hilfe von Rubycode über so genannte Builder-Templates möglich [Thom06].

Controller

Der *ActionController* aus dem Action Pack-Modul stellt in Rails das logische Zentrum der Webanwendung dar. Er koordiniert die Interaktionen zwischen dem Benutzer, den Views und dem Modell, wobei das Framework den Großteil der Interaktionen übernimmt und der Entwickler sich auf die Implementierung der Anwendungslogik konzentrieren kann. Wird eine Anfrage an die Webanwendung abgesetzt, so wird der Controller aufgerufen. Er extrahiert die Daten der Anfrage und aktualisiert das Modell. Danach wählt er die jeweilige View, übergibt die Daten und die Ausgabe wird aktualisiert. Weiters ist der Controller nach [Thom06] auch für folgende Hilfsdienste zuständig:

- Die Weiterleitung von externen Anfragen an interne Aktionen.
- Die Verwaltung der Zwischenspeicherung, die einen Performanzschub der Anwendungen mit sich bringt.
- Die Verwaltung der Hilfsmodule, mit denen die Fähigkeiten der View-Templates erweitert werden.
- Die Verwaltung der Sessions, die den Benutzern den Eindruck einer fortwährenden Interaktion mit den Anwendungen vermittelt.

Routing

Wird eine Anfrage an eine Webanwendung gestellt, so wird diese im einfachsten Fall verarbeitet und eine Antwort an den Browser zurückgesendet. Woher die Anwendung weiß, was zu tun ist, ist bei den jeweiligen Implementierungen unterschiedlich. Bei Struts beispielsweise wird das Mapping der URL auf die auszuführende Action in der Struts-Konfigurationsdatei festgelegt. Rails verzichtet nach seinem grundlegenden Prinzip auf eine solche Konfiguration. Die Information der Anfrage wird in die URL kodiert und der Routing-Mechanismus von Rails, der sich die Funktionalität der Reflection von Ruby zu Nutze macht, bestimmt, was damit gemacht werden soll. Dabei hält sich Rails an das DRY-Prinzip, da die Methode die den eingehenden Request behandelt nur einmal auftaucht, in der Controller-Klasse, die für die Verarbeitung der Anfrage zuständig ist. Die Anfrage

`http://my.shop.com/store/show_product/123`

aus [Thom06] beispielsweise sagt, dass auf dem Server `my.shop.com` die `show-product()`-Methode der Klasse `StoreController` aufgerufen werden soll, um das Produkt mit der ID 123 anzuzeigen. Die Form der URL `controller/action/id` muss aber nicht so eingehalten werden. Sie ist nur als ein sinnvoller Vorgabewert einer weiteren Konvention in Rails zusehen. Wie die Muster verändert werden können, anhand derer der Routing-Mechanismus die URLs auflöst, wird in [Thom06] näher erläutert.

4.4.3 Weitere Komponenten

Interne Module

Das Webapplikations-Framework Ruby on Rails beinhaltet noch zwei weitere Module, deren Funktionsweise in [Thom06] genauer beschrieben wird und die deshalb hier nur kurz erwähnt werden:

- *Action Mailer*: Dieses Modul stellt Funktionalitäten zum Senden und Empfangen von E-Mails bereit.
- *Action Web Service*: Dieses Modul bietet Funktionalitäten für die Programmierung von Web Services. Es sorgt für die serverseitige Unterstützung der Protokolle SOAP [W3C03] und XML-RPC [XML99].

Externe Komponenten

Für die Entwicklung von Webanwendungen mit Rails wird natürlich auch ein *Application Server* benötigt, auf dem die erstellten Anwendungen laufen. Zum Entwickeln und Testen bietet sich der WeBrick Web Server, der bei der Installation von Ruby mitgeliefert wird, als Application Server an. Für den produktiven Einsatz wird Apache oder Lighttpd mit FastCGI empfohlen, aber auch jeder andere Web Server mit CGI- oder FastCGI-Unterstützung kann verwendet werden. Unter Apache kann `mod_ruby` die Performanz verbessern.

Für die Datenhaltung mit Active Record wird ein relationales Datenbankmanagementsystem benötigt. Verschiedene RDBMS werden unterstützt, einschließlich MySQL, PostgreSQL, SQLite, DB2, Oracle and SQL Server. Es können jedoch die Modelle auch ohne Active Record und damit auch ohne relationale Datenbank geschrieben werden.

Abbildung 4.2 zeigt zusammenfassend wie sich das Zusammenspiel der Komponenten von Rails darstellt – speziell auch in Bezug auf das MVC-Entwurfsmuster:

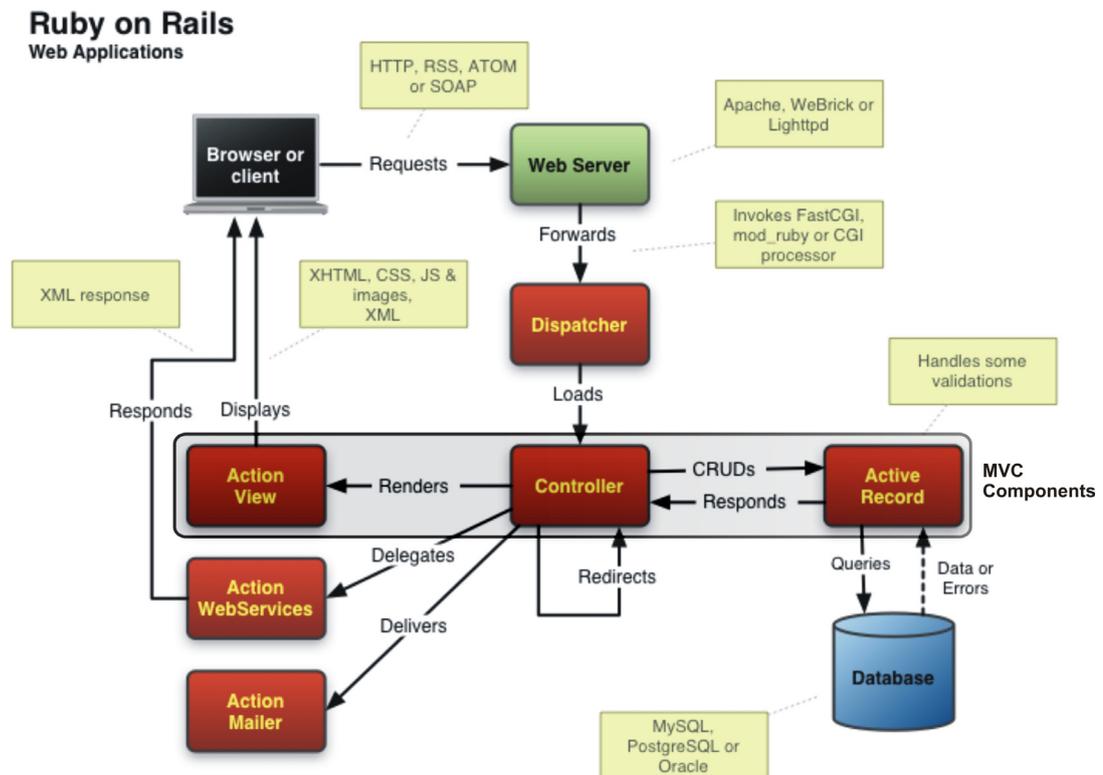


Abbildung 4.2 Ruby on Rails – Web Applications (geändert) [Rail06d]

4.5. Vergleich der beiden Frameworks

Der folgende Abschnitt gibt einen kurzen Überblick über die Gemeinsamkeiten und Unterschiede der beiden beschriebenen Webapplikations-Frameworks Apache Struts sowie Ruby on Rails. Anhand ausgesuchter Kriterien, die einer Auswahl von Anforderungen an eine Vielzahl von Webanwendungen entsprechen, werden Vor- und Nachteile der gegenübergestellten Frameworks aufgezeigt und in einer abschließenden Tabelle zusammengefasst.

4.5.1 Model View Controller-Entwurfsmuster

In Bezug auf MVC rühmen sich beide Frameworks damit das notwendige Rüstzeug bereitzustellen, anhand dessen Webanwendungen auf Basis dieses Entwurfsmusters entwickelt werden können. In Bezug auf die Architektur, die eine Anwendung durch Verwendung eines der beiden Frameworks erhält, mag dies stimmen. Jedoch kann

man nur bei Ruby on Rails von einer Bereitstellung von Komponenten für alle drei Teile von MVC sprechen, da Struts für das Model keine eigene Implementierung besitzt. Man muss man hier auf die Verwendung eines anderen Ansatzes zurückgreifen, was bei Rails durch das Vorhandensein des Active Record-Moduls nicht der Fall ist. Active Record bietet durch eine konsequente Einhaltung des DRY-Prinzps eine effiziente Implementierung der Datenzugriffsschicht: Die Modellattribute werden ausschließlich einmal definiert, nämlich in der zum Modellobjekt gehörenden Datenbanktabelle.

4.5.2 Request-Bearbeitung

In Struts werden Anfragen von Klassen bearbeitet. Rails hingegen nimmt die Bearbeitung von Requests in Methoden vor. Nach [Wird06] ist der Ansatz von Rails leichter zu handhaben und führt auch zu weniger Quellcode. Ein weiterer Punkt ist der Aufrufmechanismus der Actions. Während in Struts jede Action in der Struts-Konfigurationsdatei eingetragen werden muss, werden in Rails die Actions über Reflection aufgerufen wodurch sich der Konfigurationsaufwand verringern lässt.

4.5.3 Datenaustausch zwischen View und Controller

Für die Übergabe von Anfrageparametern werden in Struts ActionForms verwendet, die explizit programmiert und in der Struts-Konfigurationsdatei eingetragen werden müssen. Der Vorteil an den ActionForms liegt jedoch darin, dass sie typsicher sind.

Rails verwendet in diesem Zusammenhang die Hash (ein Schlüssel-Wert-Paar in Ruby) *params*, die dem Entwickler in jeder Action automatisch als Instanzvariable des Controllers zur Verfügung steht. Sie muss nicht programmiert oder konfiguriert werden. Da es in Ruby das Konzept der Typisierung nicht gibt ist daher natürlich die Hash nicht typsicher.

4.5.4 Konfiguration und Konvention

Rails-Anwendungen kommen, sofern man sich an die Konventionen hält, beinahe ohne Konfiguration aus. Struts hingegen muss sehr häufig konfiguriert werden wie folgende Beispiele aus [Wird06] zeigen:

- URLs müssen auf Actions abgebildet werden.
- Formulare müssen konfiguriert werden.

- FormBeans müssen konfiguriert und mit Formularen verbunden werden.
- ActionForwards müssen konfiguriert werden.

Durch den minimalen Konfigurationsaufwand werden bei Rails-Anwendungen eine hohe Entwicklungsgeschwindigkeit sowie eine gute Wartbarkeit ermöglicht. Konventionen wie sie in Rails häufig eingesetzt werden, gibt es bei Struts in dieser Form nicht.

4.5.5 Prototyping

Aufgrund der in Rails integrierten Code-Generatoren, vor allem des Scaffold-Generators, ist es mit diesem Framework sehr schnell möglich eine kleine, benutzbare Version einer Webanwendung zu implementieren, die rasches Feedback an den Benutzer liefert. Es müssen nun nur nach und nach die Actions erweitert werden, aber es bleibt immer eine funktionsfähige Version bestehen, die sofort getestet werden kann. Auch die Verwendung des integrierten Web Servers WeBRick trägt dazu bei, die erstellte Anwendung ohne viel Aufwand und Hinzufügen notwendiger, externer Komponenten testen zu können.

4.5.6 Entwicklungsgeschwindigkeit

Da Ruby eine interpretierte Sprache ist, müssen in Rails entwickelte Anwendungen nicht kompiliert werden. Weiters entfällt im Unterschied zu Java-Webanwendungen das Deployment, was somit zu einer höheren Entwicklungsgeschwindigkeit für Rails-Anwendungen führt. Ein weiterer Punkt, der sich positiv auf die Entwicklungsgeschwindigkeit von Rails-Anwendungen auswirkt, ist die dynamische Natur der verwendeten Programmiersprache Ruby: Nach Änderung der Datenbank stehen den Modellobjekten sofort die zugehörigen Attribute sowie Getter- und Setter-Methoden zur Verfügung. Werden neue Actions im Controller angelegt, so können diese sofort und ohne Konfiguration aufgerufen werden. Jede Änderung am Code sorgt für unmittelbares Feedback.

Die Zeitspanne zwischen einer Codeänderung und der Ausführung der Anwendung ist bei Rails kaum messbar und kann sich bei Struts-Anwendungen aus den genannten Gründen auch über mehrere Minuten erstrecken.

4.5.7 Validierung

Die Validierung von Benutzereingaben erfolgt in Rails innerhalb der Modellobjekte durch die von Active Record zur Verfügung gestellten Validierungsmethoden. Es gibt einerseits vorgefertigte Methoden für häufig notwendige Validierungen. Es ist aber andererseits auch möglich eigene Regeln durch Implementierung der Methode `validate()` zu erstellen.

In Struts wird die Validierung entweder direkt in den ActionForms implementiert, oder es wird das Validator-Framework eingesetzt, wobei die zweite Variante als die flexiblere vorzuziehen ist. Auch das Validator-Framework enthält Standard-Validierungsregeln, die jedoch ebenfalls leicht erweitert werden können.

4.5.8 Internationalisierung

Durch die Verwendung von Ressource Bundles bietet Struts eine – wenn auch eingeschränkte – Möglichkeit für die Internationalisierung von Webanwendungen.

In der Literatur von Rails finden sich in Bezug auf dieses Kriterium noch keinerlei Hinweise, weshalb hier davon ausgegangen wird, dass Internationalisierung von diesem Framework (noch) nicht unterstützt wird.

4.5.9 Zusammenfassung

	Ruby on Rails	Apache Struts
MVC-Entwurfsmuster	M: Active Record	M: -
	V: Action View	V: JSP, Tiles
	C: Action Controller	C: ActionServlet, Action, ActionForward
Request-Bearbeitung	Action-Methoden	Action-Klassen
Datenaustausch zw. V-C	Param-Hash	ActionForms
Konfiguration	Wenig	Viel
Konventionen	Viele	Keine
Prototyping	Sehr gut möglich	Kaum möglich
Entwicklungsgeschwindigkeit	Hoch, weil Ruby dynamisch und interpretiert	Niedriger durch Kompilieren und Deployment
Validierung	Validierungs-Methoden	Validator-Framework, ActionForm
Internationalisierung	-	RessourceBundles

Tabelle 4.1 Gegenüberstellung Ruby on Rails – Struts

Kapitel 5

Codegenerierung mit Ruby on Rails

In diesem Kapitel soll die Funktionsweise der Codegenerierung des Webapplikationsframeworks Ruby on Rails näher beleuchtet und erweitert werden.

Das Unterkapitel 5.1 gibt eine allgemeine Einführung in die automatische Generierung von Quellcode anhand der Begriffsbestimmung und Definition, einer möglichen Einteilung von Codegeneratoren und Vor- und Nachteilen des Einsatzes von Codegeneratoren.

Kapitel 5.2 geht genauer auf die Codegenerierung innerhalb von Ruby on Rails ein. Nach der allgemeinen Beschreibung werden die wichtigsten Codegeneratoren, die in Rails integriert und am Beginn der Erstellung einer Webanwendung eingesetzt werden beschrieben und danach deren Arbeitsweise erklärt.

Abschließend wird in Kapitel 5.3 die Vorgehensweise bei der Erstellung einer Beispielanwendung mit Hilfe des Frameworks erläutern und dabei vor allem auf die hohe Geschwindigkeit bei der Generierung des Prototyps aufmerksam gemacht werden.

5.1 Codegeneration allgemein

5.1.1 Begriffsbestimmung und Definition

Codegenerierung kann definiert werden als die Erstellung von Programmcode anhand vorgegebener, abstrakter Spezifikationen. Es bedeutet in weiterer Folge die

Verwendung von Programmen, die dem Entwickler helfen Programme schneller und effizienter zu erstellen und dabei eine höhere Qualität des Endprodukts zu erzielen. Damit sollen Aufgaben, die häufig wiederkehrend auftreten nicht mehr manuell vom Programmierer, sondern automatisch von einem Codegenerator erledigt werden können. Somit kann sich der Entwickler auf die Erstellung jener Codefragmente spezialisieren, für die eine automatische Generierung nicht oder nur mit hohem Aufwand möglich wäre. Ein Codegenerator als solcher stellt im engsten Sinne ein Computerprogramm dar, das aus bestimmten, vorgegebenen Eingaben (Spezifikationen, Modellen,...) oft mit Hilfe von ihm zur Verfügung stehenden Vorlagen eine bestimmte Ausgabe – in seiner häufigsten Anwendung Programmcode – erzeugt (generiert). Davon leitet sich auch der Name „Codegenerator“ ab.

5.1.2 Einteilung von Codegeneratoren

Es gibt verschiedene Möglichkeiten Codegeneratoren in mehrere Kategorien zu unterteilen. Im Rahmen dieser Arbeit viel die Entscheidung auf die folgende in [Frau06] verwendete Einteilung in drei Gruppen:

- Codegenerator zur *vollständigen Erzeugung*: Diese Generatoren erzeugen aus einer Spezifikation vollständigen und korrekten Quellcode für den Compiler. Der erstellte Code wird vom Programmierer nicht mehr geändert, angepasst oder erweitert.
- Codegenerator zur *unvollständigen Erzeugung*: Ausgehend von einer Spezifikation erstellen diese Generatoren unvollständigen Quellcode, der nach der Erzeugung vom Programmierer erweitert und vervollständigt werden muss.
- Codegeneratoren für die *Rahmenerzeugung*: Diese Generatoren erstellen auf Basis einer/mehrerer Vorlage(n) (so genannten Templates) einen Quelltextrahmen, innerhalb dessen der Entwickler seine Programmierung vornimmt.

Die erste Kategorie der Codegeneratoren findet besonders in der immer aktueller werdenden Thematik der *Model Driven Architecture* (MDA) ihre Anwendung. Hierbei wird ausgehend von UML-Modellen zuerst die gesamte Architektur der zu erstellenden Software entworfen und danach mit Hilfe eines oder mehrerer Codegeneratoren die vollständige Anwendung implementiert. MDA bezeichnet jedoch nicht bloß einen Ansatz für die Codegenerierung. Sie geht weit darüber hinaus, weshalb für eine genauere Betrachtung hier nur ein Verweis auf weiterführende Literatur gegeben werden kann, z.B. [Wimm05].

Die zweite Gruppe der Codegeneratoren findet sich bereits sehr häufig in den verschiedensten Entwicklungsumgebungen. Hierbei handelt es sich meist um Generatoren, die zwar auch ausgehend von spezifizierten Modellen Quellcode erzeugen, jedoch handelt es sich bei dem hier erstellten Code meist entweder um Methodentrümpfe, die in weiterer Folge vom Programmierer ergänzt werden müssen, Klassen, die sogar Getter- und Setter-Methoden für deren Attribute besitzen, oder auch Klassen mit Attributen und Operationen, die in UML-Diagrammen vordefiniert werden können. Alle erzeugten Quelldateien lassen sich im Normalfall kompilieren, sie bieten jedoch keinerlei oder nur sehr eingeschränkte Funktionalität.

Die dritte Kategorie der Codegeneratoren erfreut sich im Bereich der Frameworks besonderer Beliebtheit. Die Codegeneratoren, die im Webapplikations-Framework Ruby on Rails zum Einsatz kommen, sind zum größten Teil in diese Gruppe einzureihen. Beispielsweise verwendet der App-Generator für jede neu erstellte Anwendung eine vorhandene Verzeichnisstruktur und Templates und versieht diese nur mit dem Namen der neuen Anwendung. Somit muss sich der Entwickler nicht um die manuelle Erstellung der Verzeichnisse und Dateien kümmern. Kapitel 5.2 beschäftigt sich eingehend mit der Codegenerierung innerhalb des Frameworks Ruby on Rails.

5.1.3 Vor- und Nachteile des Einsatzes von Codegeneratoren

Wie bei jeder Technologie stellen sich im Laufe der Zeit Vor- und Nachteile ihres Einsatzes heraus, so auch bei der Anwendung von Codegeneratoren:

Vorteile

Als die größten Vorteile, die den Einsatz von Codegeneratoren sinnvoll machen, sind folgende zu nennen [Dalg06]:

- *Qualität*: Codegeneratoren verwenden Spezifikationen oder Vorlagen um ihre Endergebnisse zu produzieren. Je besser die Spezifikation umso besser der generierte Code. Wenn man daran arbeitet, die Qualität der Spezifikation und der verwendeten Templates zu erhöhen, so erhöht man damit auch die Qualität des Ausgabeprodukts. Fehler können für die generierten Codeteile an zentralen Stellen (in den Templates) korrigiert werden und müssen nicht mühsam in allen Quelldateien gesucht und gefunden werden.

- *Konsistenz*: Mit dem Einsatz von Codegeneratoren bleiben festgelegte Programmierkonventionen durchgehend bestehen, wodurch der Code strukturiert und leichter lesbar wird. Auch wenn Programmierer vorgegebene Konventionen einhalten, wird die Strukturierung des Codes verschiedener Entwickler immer von einander abweichen. Weiters wird die Verwendung erstellter Schnittstellen vereinfacht und es kann leichter weiterer, generierter Code darauf aufgesetzt werden.
- *Produktivität*: Die Vorteile in der Produktivität eines Codegenerators liegen vor allem in der Agilität die Quellcodebasis neu zu erstellen, um geänderte Anforderungen, die sich während eines Projektes ergeben, erfüllen zu können. Mit einem Durchlauf des Codegenerators können große Mengen an Code hinzugefügt oder aus der Basis herausgenommen werden.
- *Abstraktion*: Besonders Codegeneratoren, die sprachneutral arbeiten bieten in diesem Punkt einen wesentlichen Vorteil, da sie die Abstraktion, von der bei der Definition der Spezifikation ausgegangen wird, übernehmen und in den Generator integrieren. So kann die Lösung für ein und dasselbe Problem auf mehrere Zielplattformen übertragen werden, wenn es in unterschiedlichen Sprachen erzeugt werden kann.

Nachteile

Selbstverständlich gibt es auch Nachteile in Bezug auf die Verwendung von Codegeneratoren oder Voraussetzungen, unter denen sich der Einsatz eines Codegenerators nicht als sinnvoll erweist:

- *Manuelle Erweiterung*: In den meisten Fällen ist es notwendig, den automatisch generierten Code durch manuell erstellten Code zu ergänzen. Wie groß diese Ergänzungen sind, ist von Projekt zu Projekt unterschiedlich und hängt mit den Anforderungen an das fertige Endprodukt zusammen.
- *Fehlerhafte Spezifikation*: Ein Nachteil, der sich in Verbindung mit der Codegenerierung ergibt, der aber genau genommen nicht vom Codegenerator aus geht, ist die Erzeugung fehlerhaften Codes aufgrund schlechten Eingabespezifikationen oder Templates. In diesem Fall reagiert der Codegenerator natürlich nur auf die ihm vorgegebenen Eingaben für deren Korrektheit er jedoch nicht verantwortlich ist.

- *Fehlende Notwendigkeit*: Es stellt sich natürlich auch immer die Frage, wie groß die Notwendigkeit für den Einsatz eines Codegenerators ist. Einen Generator neu zu erstellen, für ein Problem, das es nur ein einziges Mal zu lösen gilt, ist wahrscheinlich weit aufwändiger, als die Lösung manuell und ohne Verwendung eines Codegenerators zu programmieren. Deshalb ist es immer notwendig, bei der Überlegung einen Codegenerator einzusetzen, vorerst den Aufwand des Einsatzes oder einer Neuerstellung abzuschätzen.
- *Wartung und Pflege*: Häufig kommt es beim Einsatz eines Codegenerators vor, dass sich die Anforderungen ändern, aber die Funktionsweise des Generators nicht entsprechend angepasst wird und somit der generierte Code manuell geändert werden muss. Es ist wichtig auch auf die Wartung und Pflege eines Codegenerators zu achten und seine Funktionalität bei sich ändernden Anforderungen zu korrigieren oder zu erweitern.

5.2 Die Codegeneration in Ruby on Rails

5.2.1 Allgemeines

Bei der Verwendung des Webapplikations-Frameworks Ruby on Rails stößt man sehr häufig auf die Verwendung von Codegeneratoren, die dem Programmierer bei der Erledigung seiner Aufgaben unterstützen und ihm Arbeit abnehmen. Da sich der Großteil des recherchierten Materials zu diesem Thema auf Seiten aus dem Web bezieht, wird in weiterer Folge der englische Begriff *Generators* für die Codegeneratoren in Ruby on Rails verwendet.

Generators werden dazu verwendet „out of the box“-Quellcode zu erzeugen, der in einer Rails-Applikation verwendet werden kann. Wird ein Generator (im Normalfall mit Hilfe des Befehls `ruby script/generate namedesgenerators`) gestartet, werden neue Dateien (Models, Views, Controllers, Helpers) erstellt und der Applikation hinzugefügt und angepasst. Dies geschieht mit Hilfe von vorhandenen Templates, die dem Generator zur Verfügung stehen. Generators erzeugen normalerweise nur ein Minimum an Code. Meist gerade soviel, dass sich der Entwickler nicht um das lästige Anlegen von Verzeichnisstruktur, Klassen-Dateien und Methoden-Rümpfen innerhalb dieser Dateien kümmern muss. Im Anschluss daran ist der Programmierer angehalten, diesen Code zu erweitern um die Funktionalität, die er von der zu implementierenden Webanwendung benötigt, zu erhalten. Generators sind nützliche Programme, mit Hilfe derer Ruby-Codesegmente erstellt werden kön-

nen, die normalerweise wenig Funktionalität bieten und nach ihrer Erstellung vom Entwickler auf die speziellen Bedürfnisse angepasst und erweitert werden. Sie sind im Normalfall für eine einzige Aufgabe zuständig und erfüllen diese sehr gut [Rail06b].

Die Entwicklung eines eigenen Generators im Rahmen von Ruby on Rails stellt sich als äußerst sinnvoll dar, da damit Arbeiten, die sehr häufig und immer nach demselben Schema erledigt werden müssen, vom Generator übernommen werden können. Besonders dann, wenn es um die Entwicklung vieler Projekte geht, in denen ähnliche Funktionalitäten gefordert sind, bringt der Einsatz eines Codegenerators durch eine Reduktion immer wiederkehrender Arbeitsschritte einen immensen Vorteil mit sich. Ruby on Rails stellt einerseits selbst eine Menge an Codegeneratoren bereit, die bei der Entwicklung von Webanwendungen verwendet werden können, andererseits steigt die Zahl der Generatoren, die über das Web frei verfügbar sind ständig an.

5.2.2 Vorhandene Codegeneratoren

Die in Ruby on Rails integrierten Codegeneratoren werden hauptsächlich am Beginn der Erstellung einer Webapplikation eingesetzt und nehmen dem Entwickler lästige Arbeitsschritte, die bei jeder neuen Anwendungsentwicklung notwendig wären, weitestgehend ab. Sie wurden bereits im allgemeinen Einführungskapitel über Ruby on Rails kurz erwähnt und werden hier nun näher erläutert. Die wichtigsten dieser Generatoren sind:

5.2.2.1 *App-Generator*

Zu Beginn der Entwicklung einer Webapplikation mit Rails steht man vor der Frage „Wie soll die Applikation heißen?“ und der Entwickler muss sich in weiterer Folge für einen beliebigen Namen entscheiden, denn der App-Generator wird mit dem Aufruf `rails` gestartet und bekommt den Namen der Applikation als Argument übergeben. Mit Hilfe dieses Applikationsnamens erstellt der Generator die Verzeichnisstruktur für die neue Webanwendung in Ruby on Rails. Damit kann der Entwickler beginnen, die Datenbank zu konfigurieren (das Konfigurationsfile steht bereits zu Verfügung) und seine Arbeit, die Implementierung der Webapplikation, beginnen. Es ist jedoch sinnvoll, mit der Ausführung eines der folgenden Generators fortzusetzen.

5.2.2.2 *Model-Generator*

Dieser Generator ist zuständig für die Erstellung von rudimentären Modell-Klassen. Sie bilden die Daten der Anwendung in Form von Ruby-Klassen ab, während die echten Daten in der Datenbank gespeichert sind (nähere Erläuterung der Modelle siehe Kapitel 4.4.1). Um die Modelle zu erzeugen wird der Befehl

```
ruby script/generate model mymodel
```

aufgerufen. Die so erstellten Klassen werden immer im bereits vorhandenen Verzeichnis `app/models` abgelegt, wodurch stets das MVC-Muster zu erkennen ist, auf dem Rails-Anwendungen streng aufbauen.

5.2.2.3 Controller-Generator

Dieser Generator ist zuständig für die Erstellung von Controller-Klassen, die das Bindeglied zwischen der Benutzerschnittstelle, den Views, und den Modellklassen sind. Ihnen obliegt die Verantwortung über die Bearbeitung von Benutzeranfragen und dem Kontrollfluss innerhalb der Webapplikation. Mit der Erstellung eines Controllers werden auch eine oder mehrere Views erzeugt. Der Aufruf des Befehls

```
ruby script/generate Controller mycontroller myview1 myview2
```

führt dazu, dass ein Controller im Verzeichnis `app/controllers`, zwei Views im Verzeichnis `app/views`, eine Helper-Klasse im Verzeichnis `app/helpers` und eine Testsuite im Verzeichnis `test/functional` angelegt werden.

5.2.2.4 Scaffold-Generator

Bei der Betrachtung von Ruby on Rails stellt sich der Scaffold-Generator als besonders interessant dar. Mit Hilfe dieses Generators ist es möglich innerhalb weniger Minuten eine erste lauffähige Version einer Webapplikation zu erzeugen, die dem Benutzer Funktionalitäten wie das Erstellen, Anzeigen, Bearbeiten und Löschen von Daten in der Datenbank anhand von Webformularen zur Verfügung stellt. Der Generator verwendet die Funktionalität der ersten beiden Generators und erweitert einerseits den Controller um eine Vielzahl an Actions, um die erwähnte Funktionalität bereitzustellen andererseits die Views, um die Interaktion mit dem Benutzer zu ermöglichen. Für ein angegebenes Model wird ein Controller erzeugt. Ist das Model noch nicht vorhanden, so wird auch dieses neu angelegt. Es ist auch möglich Views beim Aufruf des Generators als Argumente anzugeben, die dann von ihm generiert werden. Die erzeugten Dateien werden wie bei den beiden zuvor genannten Generators in den jeweiligen Verzeichnissen der Webanwendung abgelegt.

5.2.3 Arbeitsweise der Generators

Die Generators sind kleine Programme, die Ruby-Code erzeugen und dem Programmierer die Arbeit leichter machen sollen. Jeder Generator, der in Ruby on Rails verwendet oder dafür erstellt wird, besitzt folgende Verzeichnisstruktur:

`.rails\generators\[generator_name]` ist das Stammverzeichnis eines jeden Generators. Innerhalb dieses Verzeichnisses befinden sich die beiden Dateien `USAGE` und `[generator_name].rb`.

Die `USAGE`-Datei ist eine Textdatei, die für die Kommandozeile beschreibt, wie der Generator verwendet wird. Der Text wird entweder bei Aufruf des Generators ohne anschließende Parameter, mit dem Parameter `-h` oder durch einen Aufruf innerhalb der Generator-Klasse (z.B. aufgrund eines fehlerhaften Aufrufs des Generators durch den Benutzer) ausgegeben.

Die Datei `[generator_name].rb` ist eine Ruby-Quellcode-Datei und stellt die Hauptklasse des Generators dar. Meistens besitzen die Generators nur eine Klassendatei, da sich die Funktionalität in dieser einen Klasse unterbringen lässt. Es ist auch nicht die Aufgabe eines Generators sehr viel an Funktionalität bereitzustellen.

Alle Generators basieren auf der Verwendung von Vorlagen, die im Unterverzeichnis `\templates`, das alle Template-Dateien, die vom jeweiligen Generator bei der Erzeugung verwendet werden, enthält. Diese Templates tragen auch schon den Namen, der ihrer späteren Verwendung entspricht. So etwa gibt es im `Templates`-Ordner des `Controller-Generators` ein Template mit dem Namen `controller.rb` das nach seiner Verwendung `[controller_name]_controller.rb` heißen wird. Bei diesen Vorlagen handelt es sich um `ERb`-Dateien, deren Inhalt eine Mischung aus Klar-Text und eingebettetem Ruby-Code darstellt. Sie stellen eine Zwischenstation bei der Erstellung von Nur-Text-Dateien dar, die vom Generator erzeugt werden sollen. Sie besitzen keine spezielle Endung sondern nur die, die für die erstellte Datei Sinn macht, z.B. die Endung `.rb` für eine Datei, die eine Ruby-Klasse enthält, oder `.rhtml` für View-Dateien. Die erstellten Dateien erhalten vom Generator den benötigten Namen.

Die Entwicklung von Generators erweist sich als eine gute und einfache Möglichkeit das Framework Ruby on Rails zu erweitern und sich die Arbeit mit dem Framework noch weiter zu erleichtern. Es können einerseits bestehende Generators weiterentwickelt werden, andererseits völlig neue und eigenständige Generators implementiert und in jedem beliebigen Projekt verwendet werden. Die Möglichkeit bestehende

Generators durch neue Generators zu ersetzen besteht ebenfalls: Der neue Generator muss nur den Namen des bereits vorhandenen annehmen und schon wird der „alte“ überschrieben. Es hängt nur davon ab, in welchem Verzeichnis der Generator abgelegt wird. Das Script „generate“, das für den Aufruf der Generators verantwortlich ist, sucht in folgenden Verzeichnissen auf der Festplatte des verwendeten Rechners nach installierten Generators:

1. RAILS_ROOT/lib/generators
2. RAILS_ROOT/vendor/generators
3. RAILS_ROOT/vendor/plugins/plugin_name/generators
4. USER_HOME/.rails/generators
5. gems, die mit `_generator` enden
6. die in Rails integrierten Generators (innerhalb von rails gem)

5.3 Erstellung einer Rails-Beispielanwendung

Um mit Ruby on Rails eine funktionierende Beispielanwendung zu implementieren sind nicht sehr viele Schritte notwendig. Rails ist nicht nur ein Framework, sondern auch eine Sammlung von Helper-Scripts, die viele Vorgänge beim Erstellen einer Webapplikation automatisieren und damit die Entwicklung um ein Vielfaches vereinfachen. Um eine leere Webanwendung zu erstellen, muss nur in dem Verzeichnis, in dem die Applikation erzeugt werden soll, von der Kommandozeile aus der Befehl

```
rails applicationname
```

(was einem der genannten Helper-Scripts entspricht) ausgeführt werden (wobei `applicationname` durch den Namen der Anwendung ersetzt wird). Rails erstellt die gesamte Verzeichnisstruktur und die Standard-Initialisierungsdateien der Webapplikation. Um zu testen, ob das Anlegen der Applikation erfolgreich war, führt man den Befehl

```
ruby script/server
```

im Verzeichnis der Applikation aus, das den in der Distribution von Ruby mitgelieferten Web Server WeBRick startet. Danach öffnet man den Browser und navigiert auf die Adresse:

```
http://localhost:3000
```

Nun sollte die Standard-Startseite von Ruby on Rails erscheinen, die in etwa wie folgt aussieht:

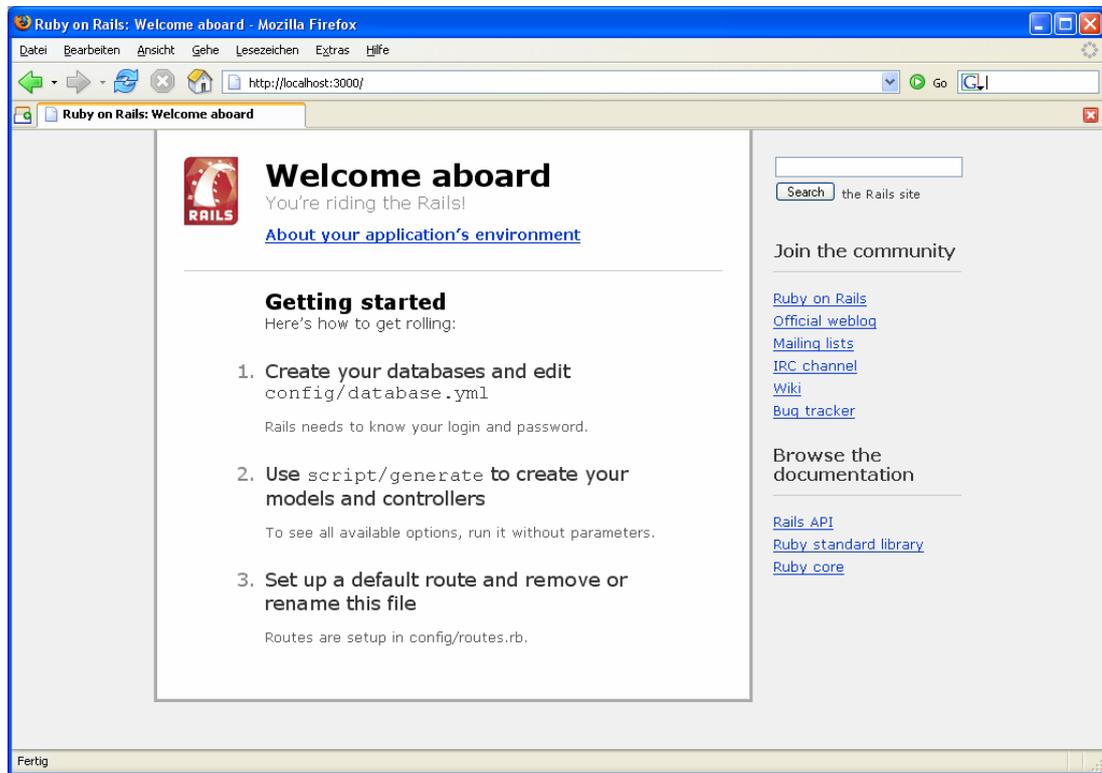


Abbildung 5.1 Standard-Startseite von Ruby on Rails

Wie erwähnt wird die Verzeichnisstruktur von Rails selbst erstellt. Das Verzeichnis, mit dem man bei der Erstellung einer Webapplikation am meisten beschäftigt ist, ist das Verzeichnis `applicationname/app`. Es enthält folgende Unterverzeichnisse mit bestimmten Aufgaben:

- `controllers` – Hier sucht Rails nach Controller-Klassen. Die Controller sind zuständig dafür, Benutzeranfragen zu verarbeiten.
- `models` – Dieser Ordner enthält die Model-Klassen, die die Daten aus der Datenbank abbilden. Die Klassen werden in Rails sehr einfach gehalten.
- `views` – Dieses Verzeichnis beinhaltet die Anzeigeschablonen, die mit Daten aus der Applikation gefüllt werden, in HTML konvertiert und an den Benutzer zurückgegeben werden.
- `helpers` – Hier finden sich Klassen, die die anderen (`model`, `view` und `controller`) unterstützen, damit diese nicht zu groß werden und sich auf ihre speziellen Aufgaben konzentrieren können.

Im vorangegangenen Abschnitt wurde das Routing von Rails beschrieben dessen Funktionsweise nun anhand eines Beispiels genauer erläutert wird: Die Controller verarbeiten Anfragen des Benutzers. Die URL des Requests wird auf eine Controller-Klasse und die darin enthaltenen Methoden abgebildet. Nun wird mit Hilfe eines weiteren Generator-Scripts im Verzeichnis der Webapplikation der Controller Test erstellt und zwar mit Hilfe des Befehls

```
ruby script/generate controller Test
```

Rails erstellt den Rumpf der Klasse TestController in der Datei test_controller.rb. Navigiert man nun die URL

```
http://localhost:3000/Test
```

an, so bekommt man die Meldung, dass Rails die Action *index* sucht, diese aber nicht findet. Es wird nun die Controller-Klasse so angepasst, dass die Action gefunden wird, was etwa zu folgendem Resultat führt:

```
class TestController < ApplicationController
  def index
    render_text "Hello World!"
  end
end
```

Abbildung 5.2 Klasse TestController

Nun sollte bei nochmaligem Aufruf der URL

```
http://localhost:3000/Test
```

der Text „Hello World!“ zu lesen sein. Durch die Definition der Methode *index* wird die Action gefunden. Das Mapping funktioniert (auch der Aufruf

```
http://localhost:3000/Test/index
```

sollte dasselbe Ergebnis liefern).

Für die folgenden Schritte wird als Beispiel eine Anwendung für die Stammdatenverwaltung eines Vereins erstellt.

Schritt1: Konfiguration der Datenbank

Mit Hilfe des Befehls `rails example` wird das Grundgerüst der Anwendung generiert. Für die Erstellung der Beispielanwendung in Rails ist es natürlich unerlässlich, eine Datenbank zu verwenden um die benötigten Daten speichern zu können. Es bietet sich an für die Beispielanwendung das freiverfügbare DBMS MySQL zu verwenden, das auf [Mysq06] kostenlos erworben werden kann. Für die Applika-

tion wird eine Datenbank verwendet, die den Namen „mydb“ trägt. Diese wird angelegt und danach Rails so konfiguriert, dass es die Datenbank verwenden kann (das ist einer der seltenen Schritte, bei denen Rails konfiguriert werden muss). Die Konfiguration findet in der Datei `example/config/database.yml` statt. Hier unterscheidet Rails zwischen drei Datenbanken für Entwicklung, Produktion und Test. Für die Erstellung der Beispielanwendung ist diese Unterscheidung nicht wichtig und es wird für alle drei Umgebungen derselbe Datenbankname eingetragen. Natürlich müssen auch Benutzername und Passwort für den Zugriff auf die Datenbank eingegeben werden. Damit erhält die Konfigurationsdatei etwa folgendes Aussehen:

```
development:
  adapter: mysql
  database: mydb
  username: root
  password:
  host: localhost

# Warning: The database defined as 'test' will be erased and
# re-generated from your development database when you run
# 'rake'.
# Do not set this db to the same as development or production.

test:
  adapter: mysql
  database: mydb
  username: root
  password:
  host: localhost

production:
  adapter: mysql
  database: mydb
  username: root
  password:
  host: localhost
```

Abbildung 5.3 Rails: Datenbank-Konfigurationsdatei `database.yml`

Bei Veränderungen in der Konfiguration von Rails muss der Server neu gestartet werden!

Schritt 2: Erstellen der Datenbanktabellen

Da die Beispielanwendung die Implementierung einer Mitgliederverwaltung darstellen soll, ist es nahe liegend eine Mitgliedertabelle anzulegen (Tabelle *members*) und einige passende Felder zu erstellen. Es ist darauf zu achten, die Felder in den Tabellen mit Kleinbuchstaben zu benennen (auch die Spalte `id`), da die Modell-Methoden von Active Record ansonsten nicht korrekt mit der Datenbank kommunizieren kön-

nen. Die Tabellen in der Datenbank sollen um der Namenskonvention zu folgen im Plural angegeben werden (members, locations, etc.).

Schritt 3: Erstellen des Modells

Als nächstes wird eine Modell-Klasse erstellt, die die Daten aus der Datenbank abbildet. Folgender Befehl erzeugt die Modell-Klasse:

```
ruby script/generate model Member
```

Es wird im Verzeichnis `example/app/model` die Datei `member.rb` erstellt, die den Rumpf der Modell-Klasse enthält. Diese scheinbar leere Klasse ist das Geschäftsobjekt, das Rails verwendet um die *members*-Tabelle aus der Datenbank abzubilden. Modelle folgen dem Active Record Pattern, was bedeutet, dass ein Modell mit genau einer Datenbanktabelle assoziiert ist. Es fällt auf, dass für die Erstellung eines Modells nur wenige Schritte notwendig sind, was daran liegt, dass eine Namenskonvention benutzt wurde, die Rails verwendet. Es werden Singular Modell-Klassen (Member) auf Plural-Tabellen (members) in der Datenbank abgebildet. Rails kennt einige Regeln zur Bildung des Plurals in der englischen Sprache z.B. *company* – *companies*, *person* – *people*, etc.

Eine Besonderheit von Rails ist, dass weder die Attribute aus der Datenbanktabelle, noch Getter- und Setter-Methoden explizit im Modell definiert werden. Rails nutzt hier ein Feature von Ruby, nämlich die Erweiterbarkeit von Klassen zur Laufzeit. Für jedes Attribut aus der assoziierten Tabelle steht auf dem Modell automatisch eine Getter- und Setter-Methode zur Verfügung – ein Beispiel für die Einhaltung des DRY-Prinzips in Rails.

Die Schritte 2 und 3 werden nun für alle Tabellen der Datenbank wiederholt.

Schritt 4: Modellassoziationen

Um auch wirklich mit den Domain-Modellen arbeiten zu können, ist es natürlich notwendig, Verbindungen zu anderen Modellen herstellen zu können. Die Beziehungen zwischen den Modellen werden in Rails als Klassenmethoden innerhalb der Modellklassen definiert. Rails bietet folgende Modellassoziationen:

Methode im Modell	Modellasoziation
belongs_to	1:1
has_one	1:1
has_many	1:n
has_many_and_belongs_to_many	m:n

Tabelle 5.1 Rails: Modellasoziationen

Die Relationen *belongs_to* und *has_one* beschreiben beiden eine 1:1-Beziehung. Der Unterschied zwischen den beiden bezieht sich hauptsächlich auf die Positionierung des Fremdschlüssels innerhalb der Tabelle. Was bedeutet, dass das Modell mit der *belongs_to*-Assoziation den Fremdschlüssel aus der Beziehung aufnimmt.

In der Beispielanwendung wird der Fremdschlüssel der Beziehung Mitglied – Ort in die Tabelle *members* aufgenommen, also wird in der Klassendefinition des *Member*-Modells die Klassenmethode

```
belongs_to :location
```

eingetragen. Damit aus der Datenbank der zugehörige Ort gelesen wird, muss in der Tabelle *members* das Attribut *location_id* (gegebenenfalls als Fremdschlüssel) definiert werden. Der Name beschreibt per Rails-Konvention durch sein Suffix *_id* die Verbindung zu der Tabelle *locations* und zu dem dortigen Primärschlüsselattribut *id*.

Schritt 5: Erstellen des Controllers

Der nächste Schritt besteht in der Erzeugung des Controllers, der für die Anfragen auf die *members*-Tabelle zuständig ist. Wie bereits in der Erklärung des URL-Mappings erwähnt, wird der Controller mit Hilfe von

```
ruby script/generate controller Members
```

generiert. Es wird die Datei *members_controller.rb*, die den Rumpf der Controller-Klasse enthält, im Verzeichnis *example/app/controller* erstellt.

Schritt 6: Scaffolding

Nun ist es möglich zu zeigen, wie einfach es mit Rails ist eine CRUD-Anwendung (create, retrieve, update, delete) zu erstellen. Innerhalb der Klassendefinition des soeben erzeugten Controllers muss nur die Codezeile

```
scaffold :member
```

eingegeben werden und schon ist die erste kleine Beispielapplikation fertig. Die Scaffolding-Technik von Rails macht es möglich, dass der Controller zu einem CRUD-Controller avanciert. `scaffold :member` erzeugt zur Laufzeit dynamisch die Controller-Actions *list*, *show*, *destroy*, *create*, *edit* und *update*. Die erzeugten Methoden delegieren im Anschluss an ihre Ausführung automatisch an einen gleichnamigen View, der genau wie die Action-Methode dynamisch generiert wird. So erhält etwa der Edit-View Felder für jedes Attribut des Member-Modells.

Nun ist eine erste lauffähige Version der Webapplikation vorhanden. Über die Eingabe der URL

```
http://localhost:3000/members/list
```

wird die Controller-Action *list* aktiviert die alle in der Datenbank vorhandenen Mitglieder (*members*) lädt und im gleichnamigen View anzeigt. Der View enthält außerdem Links auf weitere Actions zum Anzeigen, Bearbeiten und Löschen vorhandener bzw. zum Anlegen neuer Mitglieder. Beim ersten Start der Anwendung ist die Datenbank leer, weshalb vom List-View noch keine Mitglieder angezeigt werden. Durch einen Klick auf den Link „New member“ und der damit verbundenen Aktivierung der Edit-Action wird jedoch die Erfassung neuer Daten möglich.

Die Erweiterung von Modellen durch Anpassung der zugehörigen Tabellenstruktur hat unmittelbare Auswirkungen auf die Applikation. Wird etwa die Tabelle *members* in der Datenbank um ein Feld erweitert, so erhalten alle mit dem Member-Modell assoziierten Views beim nächsten Laden das zusätzliche Feld. Scaffolding ist vor allem für Rapid Prototyping gedacht und wird in produktiven Anwendungen im Normalfall mit eigenem Code ergänzt.

Schritt 7: Erweiterung der Applikation

Scaffolding kann nur ein erster Schritt auf dem Weg zu einer vollständigen Web-Anwendung sein, ist aber auf jeden Fall eine effiziente Methode, um schnell mit dem Programm arbeiten zu können und initial Daten in die Datenbank zu bekommen. Der Vorteil einer durch Scaffolding erzeugten Rails-Applikation ist neben der unmittelbaren Benutzbarkeit die Möglichkeit zur sukzessiven und iterativen Erweiterung der Applikation, ohne die zuvor entwickelten Teile wegwerfen zu müssen. Da es sich bei der Anwendung nicht um einen Prototyp, sondern um eine sauber auf Basis von MVC entworfene Anwendung handelt, kann man diese Schritt für Schritt weiterentwickeln.

Dies erfolgt durch Überschreiben einzelner Controller-Actions. Gemäß den Rails-Konventionen überschreibt eine selbst definierte Controller-Action die gleichnamige Scaffold-Action, lässt jedoch die anderen Scaffold-Actions weiterhin bestehen. Da nur einzelne Actions überschrieben werden, bleibt die Anwendung zu jedem Zeitpunkt benutzbar. Die List-Action könnte etwa folgendermaßen im Members-Controller überschrieben werden:

```
class MembersController < ApplicationController
  scaffold :member
  def list
    @members = Member.find_all
  end
end
```

Abbildung 5.4 Klasse MembersController

Sobald eine Action überschrieben ist, generiert Rails keinen zugehörigen View mehr. Der View muss selbst implementiert werden, was in Rails im RHTML-Format erfolgt, einem HTML-Derivat, mit eingebettetem Ruby-Code. Wie in JSP ist der Ruby-Code über die Tags `<% %>` und `<%= %>` in HTML-Seiten einzubetten.

Rails stellt auch Methoden für die Validierung zur Verfügung, die in den Klassen der Modell-Objekte eingefügt werden müssen. Auf diesen Schritt wird im folgenden Kapitel 6 näher eingegangen, weshalb hier nur darauf verwiesen wird.

Kapitel 6

Der erweiterte Rails-Generator

In diesem Kapitel geht es um die Erstellung eines eigenen Codegenerators für Ruby on Rails und damit um die Erweiterung des Frameworks um geforderte Funktionalitäten. Ausgangspunkt ist eine Beschreibung der Ergebnisse, die man beim Einsatz des Scaffold-Generators erhält. Darauf folgt die Spezifikation der Anforderungen an den neuen Generator, an die die Beschreibung, wie man einen eigenen Codegenerator erstellt, anschließt. Den Abschluss des Kapitels bildet eine genaue Beschreibung der Vorgehensweise bei der Erstellung sowie der Funktionsweise des eigentlichen Generators selbst.

6.1 Ausgangsbasis: Scaffolding

Die Verwendung des Scaffold-Generators und die Tatsache, dass man damit in kurzer Zeit einen funktionsfähigen Prototyp einer Webanwendung erstellen kann, versetzen viele Menschen ins Staunen. Natürlich ist das Endergebnis nicht perfekt und der Funktionsumfang, den der Prototyp besitzt, nicht übermäßig groß, jedoch hat es für die Weiterentwicklung der Anwendung einen großen Vorteil, denn der vorhandene Code, der vom Generator erstellt worden ist, muss nicht vollständig neu erstellt oder sogar weggeworfen werden. Er kann Schritt für Schritt erweitert und das Ergebnis in Handumdrehen getestet werden.

6.1.1 Controller

Wirft man einen Blick auf den Inhalt der Dateien, die vom Scaffold-Generator erzeugt werden, so fällt auf, dass die Controller-Klasse schon relativ viel Code enthält. Es wurden Methoden für die Aktionen `index`, `list`, `show`, `new`, `create`, `edit`, `update` und `destroy` erzeugt, die für die aktuelle Funktionalität des Prototyps im Großen und Ganzen verantwortlich sind:

Index: Ist die Standard-Aktion und wird aufgerufen, wenn keine der anderen Aktionen also nur der Controller selbst aufgerufen wird. Sie beinhaltet standardmäßig den Aufruf der List-Aktion.

List: Gibt den Inhalt der Tabelle des Modells aus, für das die Aktion aufgerufen wird. Standardmäßig werden alle Zeilen und alle Spalten der angeforderten Tabelle angezeigt.

Show: Ruft die Detailansicht für den ausgewählten Datensatz auf. Wird diese Aktion alleine verwendet also durch den Aufruf von `controller/show`, so tritt ein Fehler auf, da sie eine ID des Datensatzes benötigt um diesen richtig anzeigen zu können. `Controller/show/1` zeigt etwa die Detailansicht des Datensatzes mit der ID 1 an.

New: Ruft eine Seite mit leeren Formularfeldern zum Eingeben eines neuen Datensatzes auf. Ist in der Datenbank z. B. ein Feld mit einem Datumstyp vorhanden, so werden auf der Seite auch die Pull-Down-Menüs zum Auswählen des Datums angezeigt, mit der aktuellen Zeit als Voreinstellung.

Create: Diese Aktion arbeitet im Hintergrund. Von ihr merkt der Benutzer fast nichts, da sie keine neue View auf den Bildschirm bringt. Die Aktion ist dafür Verantwortlich, den eingegebenen Datensatz in die Datenbank zu schreiben. Wurde die Aktion aufgerufen, so wird die List-Aktion ausgeführt, wobei im Kopf der angezeigten View eine Meldung angezeigt wird, die dem Benutzer Feedback über die Erledigung des Vorganges gibt.

Edit: Mit Hilfe dieser Aktion ist es möglich, einen vorhandenen Datensatz zu bearbeiten. Beim Aufruf der Aktion wird im Prinzip dieselbe View wie bei der New-Aktion aufgerufen, jedoch sind bei diesem Vorgang die angezeigten Formularfelder nicht leer, sondern mit den Daten des ausgewählten Datensatzes. Nun können die Daten beliebig geändert werden. Wirft man einen Blick auf das Views-Verzeichnis des Modells, so sieht man, dass es für die beiden Aktionen New und Edit zwei verschiedene Views gibt, obwohl die Funktionsweisen sehr ähnlich sind. Der Grund

sind die unterschiedlichen Aktionen, an die die Formulardaten der Views weitergeleitet werden.

Update: Diese Aktion ist ähnlich wie Create für den Benutzer kaum erkennbar. Sie ist für die Aktualisierung des geänderten Datensatzes in der Datenbank zuständig und speichert wiederum eine Feedbackmeldung die nach Ausführung der Aktion im Kopf der angezeigten View dargestellt wird. Diese View ist wie zuvor bei der Create-Aktion im Standardfall die List-View.

Destroy: Zum Abschluss wird die Destroy-Aktion im Controller des Models definiert. Diese Aktion dient dem Entfernen eines Datensatzes aus der Datenbank. Einerseits sollte man beim Aufruf dieser Aktion vorsichtig sein, da man damit wirklich einen Datensatz aus der Datenbank löscht, andererseits wird bei ihrem Aufruf der Benutzer noch vor die Frage gestellt, ob er den Datensatz wirklich löschen will und erst nach der Bestätigung dieses Dialogs wird der Vorgang abgeschlossen. Im Unterschied zu den beiden erstgenannten „Datenbank-Aktionen“ wird hier keine Meldung an den Benutzer ausgegeben. Das Feedback ist durch das Fehlen des gelöschten Datensatzes in der Tabelle der List-View ersichtlich.

6.1.2 Model

Im Vergleich zu den automatisch erzeugten Controller-Dateien, sind die Model-Dateien nur mit sehr wenig Code ausgestattet. Genau gesagt sind es nur zwei Zeilen, die die Model-Klasse zu Beginn definieren, wie etwa das Modell der Mitglieder-Tabelle:

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

Abbildung 6.1 Klasse Member

Es ist nur die Klassendefinition, die angibt, dass die Klasse von ActiveRecord::Base abgeleitet ist, und damit alle Attribute und Operationen der Basisklasse erbt, sowie das schließende end, das den Abschluss der Klasse signalisiert. Es mag auf den ersten Blick wenig erscheinen, jedoch trägt der Schein, denn durch die Vererbung erhalten die Model-Klassen eine Vielzahl von Methoden aus dem ActiveRecord-Modul, die die Zugriffe auf die Datenbank ermöglichen.

6.1.3 View

Die Anzahl der Views, die vom Generator automatisch generiert wird ist auf den ersten Blick beachtlich, da doch einige Dateien im Views-Verzeichnis vorhanden sind. Dies ist auf die im Controller erstellten Aktionen zurückzuführen. Der Aufruf beinahe jeder Aktion löst die Darstellung einer eigenen View aus. Wie bei der Beschreibung des Controllers erwähnt, gibt es aber auch einige Aktionen, die eher im Hintergrund arbeiten und für die keine eigene View benötigt wird. Betrachtet man den Inhalt einer dieser Views, so wird man feststellen, dass es sich um eine Mischung aus HTML und eingebettetem Ruby-Code handelt. Der Quellcode besitzt jedoch keine Programmlogik, sondern ist nur für die Darstellung von Daten zuständig, was auch für die strikte Einhaltung des MVC-Musters bei der Entwicklung der Webanwendung in Rails zu beachten ist.

6.2 Anforderungen an den erweiterten Rails-Generator

Aufgrund der Tatsache, dass es mit Hilfe des Scaffolding-Mechanismus von Rails so einfach ist, nach ein paar wenigen Schritten eine relativ eindrucksvolle kleine Webanwendung zu bauen, stellt sich die Frage, ob es durch Anpassen oder Erweitern der in Rails integrierten Generatoren realistisch ist, die Funktionalität, die der Prototyp im derzeitigen Zustand besitzt, auszuweiten und mit weiteren wichtigen Features zu ergänzen. Betrachtet man den vom Scaffolding erstellten Prototyp genauer, so stellt man fest, dass besonders zwei wichtige Funktionalitäten fehlen, die zwar vom Framework zur Verfügung gestellt werden, jedoch nicht in dem automatisch erzeugten Prototyp integriert sind:

- Validierung der Modellobjekte
- Assoziationen zwischen den Modellobjekten

Wären diese beiden Punkte in der Endversion des vom Generator erzeugten Prototyps, so erhielte dieser doch ein weiteres Maß an Funktionalität, die der Programmierer bei der weiteren Entwicklung der Applikation nicht mehr selbst integrieren muss. In den folgenden beiden Unterkapiteln wird kurz erläutert, wie sich das Fehlen dieser Funktionalitäten in der Benutzung des Prototyps auswirkt und welche Teile, das Framework in Bezug auf die beiden Funktionalitäten bietet.

6.2.1 Validierung der Modellobjekte

Beim Arbeiten mit dem Scaffold-Generator und dem darauf folgenden Test des generierten Prototyps, fällt dem geschulten Benutzer sofort auf, dass das Programm etwa bei der Erstellung eines neuen Datensatzes jede beliebige Eingabe des Benutzers akzeptiert. Felder, die eigentlich in der Datenbank nicht leer sein dürfen, im Formular aber nicht ausgefüllt werden, oder Buchstaben, die in Feldern, die eigentlich Zahlen erwarten, eingegeben werden, werden vom Programm ohne jede Rückmeldung akzeptiert. Eine der wesentlichen Aufgaben des erweiterten Generators soll die Integration des von Rails zur Verfügung gestellten Validierungsmechanismus in die vom Generator erstellten Modellklassen sein.

Wie bereits oben erwähnt werden die Modelklassen von ActiveRecord::Base abgeleitet. Das Active Record-Modul von Rails stellt eine Reihe von Methoden zur Verfügung, die es ermöglichen, den Inhalt der Modell-Objekte auf seine Gültigkeit zu überprüfen. Diese Überprüfung kann einerseits automatisch beim Speichern des Objekts, andererseits auch manuell angestoßen werden. Je nach dem, ob es sich bei Modellen um einen neuen Datensatz oder einen bereits in der Datenbank vorhandenen Eintrag handelt, werden die Modelle unterschiedlich behandelt. Bei neuen Einträgen führt ActiveRecord eine SQL-INSERT-Operation auf der Datenbank aus, bei bereits vorhandenen Datensätzen eine UPDATE-Operation. Diese Unterscheidung spiegelt sich auch bei der Ausführung des Validierungsmechanismus wider. Es gibt drei Validierungsmethoden auf der untersten Ebene von denen bei jedem Speichern mindestens zwei aufgerufen werden, und die alle überschrieben werden können [Thom06]:

- *validate()* wird bei jedem Speichern aufgerufen.
- *validate_on_create()* wird beim Speichern eines neuen Datensatzes in der Datenbank aufgerufen (SQL-INSERT).
- *validate_on_update()* wird beim Aktualisieren eines Datensatzes in der Datenbank aufgerufen (SQL-UPDATE).

Tritt während der Validierung ein Fehler auf, so wird mittels der Methode *errors.add()* eine Fehlermeldung in die Fehlerliste des Modells aufgenommen. Die Methode erhält als Parameter den Namen des Attributs und die Fehlermeldung. Mit Hilfe dieser Liste können die erhaltenen Fehler (falls vorhanden) an die View weitergegeben und somit dem Benutzer angezeigt werden, wo ein Fehler aufgetreten ist, und worum es sich dabei handelt.

Viele Validierungen kommen innerhalb von Webanwendungen häufig vor. Um nicht für jedes Modell und jeder Art der Validierung die drei genannten Methoden neu implementieren zu müssen, bietet ActiveRecord einige Standardvalidierungsmethoden an, die von den Modellen ganz einfach benutzt werden können (auch Validierungshelfer genannt [Mari06]). Es handelt sich hierbei um Klassenmethoden, die allesamt mit `validates_` beginnen, und mit deren Hilfe sich die Validierung eines Attributs relativ einfach gestaltet. Als Parameter erwarten die Methoden eine Liste von Attributnamen, die validiert werden sollen, und einen Hash von Konfigurationsoptionen. Um beispielsweise zu überprüfen, ob der Titel eines Produkts eingegeben und nicht bereits verwendet wird sowie der Preis einen numerischen Wert darstellt, könnte der Code einer Modell-Klasse `Product` folgendermaßen aussehen:

```
class Product < ActiveRecord::Base
  validates_presence_of :title
  validates_uniqueness_of :title, :on => :create,
    :message => „ist bereits vorhanden!“
  validates_numericality_of :price,
    :message => „muss eine Zahl sein!“
end
```

Abbildung 6.2 Klasse `Product`

Die meisten Validierungshelfer akzeptieren die Optionen `:on` und `:message`. `:on` gibt an zu welchem Anlass die Validierung stattfinden soll (`:save`, `:create` oder `:update`). Die Option muss nicht unbedingt angegeben werden, da `:save` standardmäßig verwendet wird. `:message` trägt eine Fehlermeldung in die Fehlerliste ein.

Das obige Quellcodebeispiel zeigt bereits drei wichtige Validierungsmethoden, die in einer Rails-Applikation häufig eingesetzt werden. Im Anschluss werden die wichtigsten Validierungshelfer und deren Funktionsweise kurz erläutert, die in weiterer Folge bei der Erstellung des erweiterten Generators eine Rolle spielen werden.

`validates_format_of`: Ermöglicht die Überprüfung der angegebenen Attribute anhand eines regulären Ausdrucks. Die Option `:with` muss angegeben werden, sonst wirft Rails eine Exception.

Syntax:

`validates_format_of` Attribut, `:with =>` regexp [Optionen]

Optionen:

:on, :message : Siehe oben.

:with : Erhält einen regulären Ausdruck, gegen den das Attribut getestet wird.

Beispiel:

```
Class Product < ActiveRecord::Base
  validates_format_of :image_url,
    :with => %r{^http:.\.\.(gif|jpg|png)$}
    :message => „muss ein gif, .jpg oder .png Bild sein!“
end
```

Abbildung 6.3 Klasse Product mit Validierungsmethode `validates_format_of`

`validates_numericality_of`: Stellt sicher, dass das Attribut eine gültige Zahl ist.

Syntax:

`validates_numericality_of` Attribut [Optionen]

Optionen:

:on, :message : Siehe oben.

:only_integer : Ist diese Option auf true gesetzt, werden nur Ganzzahlen akzeptiert.

:allow_nil : Ist diese Option auf true gesetzt, werden nil-Objekte nicht überprüft.

Beispiel:

```
Class Product < ActiveRecord::Base
  validates_numericality_of :price,
    :message => „muss eine gültige Zahl sein!“
end
```

Abbildung 6.4 Klasse Product mit Validierungsmethode `validates_numericality_of`

`validates_presence_of`: Stellt sicher, dass das zu überprüfende Attribut nichtleer ist. Auch der Wert nil gilt als leer.

Syntax: `validates_presence_of` Attribut [Optionen]

Optionen:

:on, :message Siehe oben.

Beispiel:

```
Class Product < ActiveRecord::Base
  validates_presence_of :title,
    :message => „muss angegeben werden!“
end
```

Abbildung 6.5 Klasse Product mit Validierungsmethode `validates_presence_of`

`validates_uniqueness_of`: Diese Methode prüft bei jedem Attribut, dass es keine andere Zeile in der Datenbank die in der entsprechenden Spalte den selben Wert hat. Stammt das Modellobjekt aus einer existierenden Datenbankzeile, so wird bei der Prüfung diese Zeile ignoriert.

Syntax: `validates_uniqueness_of` Attribut [Optionen]

Optionen:

`:on`, `:message` siehe oben.

`:scope` Begrenzt die Überprüfung auf die Zeilen, die in der angegebenen Spalte denselben Wert haben, wie die zu überprüfende Zeile.

Beispiel:

```
Class Product < ActiveRecord::Base
  validates_uniqueness_of :title,
    :message => „wird schon verwendet!“
end
```

Abbildung 6.6 Klasse Product mit Validierungsmethode `validates_uniqueness_of`

Dies war natürlich nur ein kurzer Überblick über die in Active Record vorhandenen Standard-Validierungsmethoden. Es gibt noch einige weitere, die für die Validierung der Objekte verwendet werden können. Für eine nähere Betrachtung wird auf [Thom06] und [Mari06] verwiesen, wo noch andere nützliche Validierungshelfer angeführt werden. Eine Übersicht und Beschreibung aller Validierungsmethoden findet sich in der Dokumentation von Ruby on Rails [Hein06b].

6.2.2 Assoziationen zwischen den Modellklassen

Ein weiterer Schwachpunkt, der bei der Arbeit mit dem Prototypen auffällt ist, dass die Beziehungen zwischen den Tabellen (falls vorhanden) in den Modellklassen nicht wiedergegeben werden. Da jedoch die meisten Anwendungen mit mehreren Datenbanktabellen arbeiten und diese normalerweise untereinander in Beziehung stehen, ist es notwendig, diese Beziehungen von der relationalen Datenbanksicht in die objektorientierte Sicht der Webanwendung zu übertragen. Im Datenbankschema werden diese Beziehungen dadurch ausgedrückt, dass man die Tabellen anhand ihrer Primärschlüssel miteinander verknüpft. Bezieht sich etwa ein Einzeleinstellung auf ein Produkt, so erhält die Tabelle *line_items* eine Spalte, die den Primärschlüsselwert des entsprechenden Produkts aus der *products*-Tabelle speichert. Dies nennt man auch Fremdschlüsselbeziehung [Thom06].

Nun arbeiten Rails-Anwendungen nicht mit Datenbankzeilen und Spalten, sondern mit Modellobjekten. Auf der einen Seite ist Active Record im Stande, die Konfiguration des objekt-to-relational-Mappings automatisch durchzuführen, auf der anderen Seite benötigt es in Bezug auf die Darstellung von Beziehungen Unterstützung vom Entwickler. Die Aufgabe der automatischen Darstellung der Relationen ist jedoch nicht so einfach lösbar, da eine korrekte Beschreibung der Beziehungen ausschließlich anhand der vorhandenen Datenbanktabellen nicht eindeutig möglich ist, auch wenn es in den Tabellen Fremdschlüssel gibt, die über die Namenskonvention von Active Record als solche erkannt werden. Enthält eine Tabelle eine Spalte, die den Namen einer anderen Tabelle mit dem Suffix *_id* trägt, und wird diese Spalte mit Primärschlüsselwerten der referenzierten Tabelle gefüllt, so kann Active Record die Verbindung zwischen den Tabellen herstellen. Es gibt drei Arten von Beziehungen, die innerhalb der Modelle darstellbar sind und folgendermaßen vom Framework realisiert werden [Mari06][Thom06]:

1-zu-1-Beziehung (one-to-one): Ein Datensatz einer Tabelle steht mit genau einem Datensatz einer anderen Tabelle in Beziehung. Im Datenbankschema wird die Beziehung durch die Verwendung eines Fremdschlüssels, der auf genau eine Zeile in einer anderen Tabelle zeigt, implementiert. In den Modellklassen der Ruby on Rails Anwendung erhält die Klasse, die die Tabelle mit dem Fremdschlüssel darstellt, das Schlüsselwort *belongs_to* und die referenzierte Klasse die Deklaration *has_one*. Die Abbildung soll die Darstellung der Beziehung veranschaulichen:

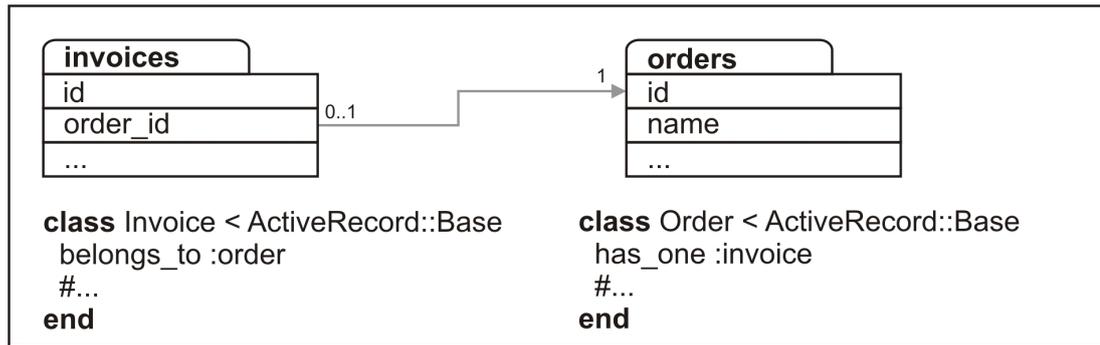


Abbildung 6.7 1:1-Beziehung in Rails [Thom06]

1-zu-n-Beziehung (one-to-many): Ein Datensatz einer Tabelle steht mit mehreren Datensätzen einer anderen Tabelle in Beziehung, die Datensätze der zweiten Tabelle stehen jedoch nur mit einem Datensatz der ersten Tabelle in Beziehung. Die Klasse, die die erste Tabelle repräsentiert erhält die Deklaration *has_many*, die Klasse, die die zweite Tabelle darstellt erhält das Schlüsselwort *belongs_to*. Auch diese Beziehung wird in der Abbildung veranschaulicht.

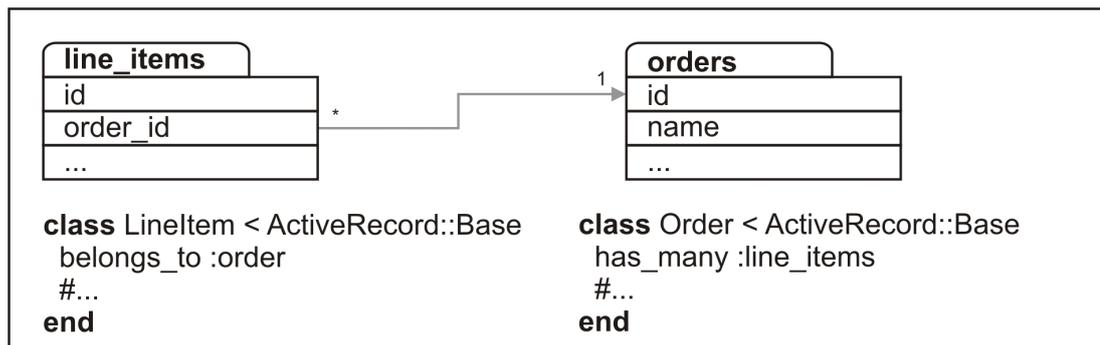


Abbildung 6.8 1:n-Beziehung in Rails [Thom06]

m-zu-n-Beziehung (many-to-many): Eine beliebige Anzahl von Datensätzen einer Tabelle steht mit einer beliebigen Anzahl von Datensätzen einer zweiten Tabelle in Beziehung. Um diese Beziehung im Datenbankschema abzubilden, wird eine Join-Tabelle eingeführt, die ausschließlich Paare von Fremdschlüsseln enthält, die jeweils einen Eintrag der einen Tabelle mit einem Eintrag der anderen Tabelle verknüpft. Die Tabelle wird nicht von ActiveRecord erstellt und muss daher manuell angelegt werden. Die an der Relation beteiligten Klassen erhalten jeweils die Deklaration *has_and_belongs_to_many*. ActiveRecord nimmt an, dass der Name der Join-Tabelle den in alphabetischer Reihenfolge aneinander gehängten Namen der beiden Tabellen entspricht, durch einen Unterstrich getrennt. Man kann der Deklaration auch über den Parameter *join_table* den Namen der Join-Tabelle manuell angeben.

Um sich den Sachverhalt besser vor Augen führen zu können, wird auch die Darstellung dieser Beziehung in einer Abbildung gezeigt:

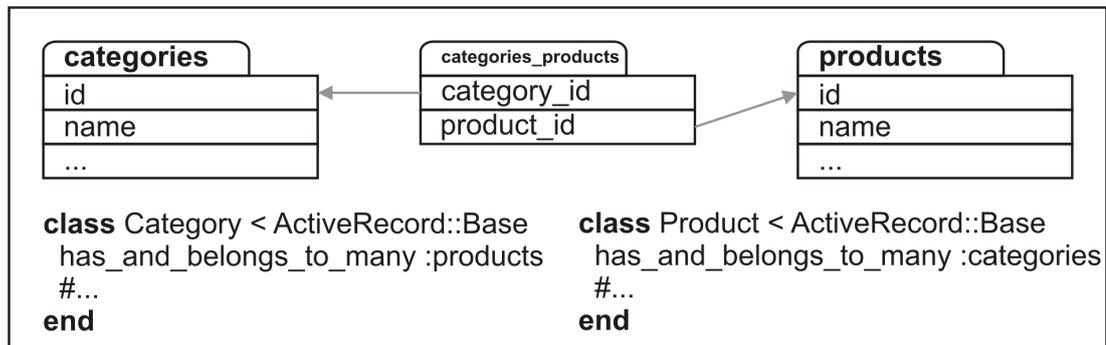


Abbildung 6.9 m:n-Beziehung in Rails [Thom06]

Durch die Verwendung der beschriebenen Deklarationen *has_one*, *belongs_to*, *has_many* sowie *has_and_belongs_to_many* erhalten die Modellklassen eine Reihe von Methoden, die den Zugriff auf die Daten in der Datenbank ermöglichen. Auf eine genaue Beschreibung welche Deklaration welche Methoden bereitstellt wird an dieser Stelle verzichtet. Sie findet sich in [Mari06] und in [Thom06].

Es würde nun einen großen Vorteil und eine immense Zeitersparnis für den Entwickler bedeuten, würden diese Anforderungen, die in den beiden Abschnitten beschrieben wurden, schon vom Generator zu Beginn der Erstellung der Webanwendung automatisch integriert.

6.3 Erstellen eines eigenen Rails-Generators

Die Erstellung eines Generators in Ruby on Rails stellt sich als weniger komplex dar, als es auf den ersten Blick erscheinen mag. In [Rail06b] findet man eine Beschreibung der ersten Schritte, die bei der Erstellung eines eigenen Generators notwendig sind und im folgenden Abschnitt näher ausgeführt werden:

Jeder Generator ist ein kleines Programm, das in Ruby geschrieben wurde. Die Hauptklasse sollte von einer der beiden folgenden Klassen abgeleitet werden:

- `Rails::Generator::Base`
- `Rails::Generator::NamedBase`

Wobei *NamedBase* selbst von *Base* abgeleitet ist und daher im Prinzip alle Eigenschaften besitzt, die auch die Oberklasse enthält. Der Unterschied zwischen diesen beiden Klassen besteht darin, dass die Klasse *NamedBase* (und ihre Unterklassen)

bei ihrem Aufruf den Namen des Generators gefolgt von einer Liste von Argumenten erwartet. So wird beispielsweise ein Controller mit Hilfe des Controller-Generators erzeugt, dessen Aufruf folgendermaßen aussieht, wobei hier ein Controller namens „controller_name“ erstellt wird, der drei Aktionen erhält:

```
ruby script/generate controller [controller_name] [action1]
[action2] [action3]
```

Möchte man einen Generator erstellen, der beim Aufruf diesem Muster folgt, so soll man laut [Rail06b] `Rails::Generator::NamedBase` als Basisklasse verwenden, ansonsten ist die Verwendung der Klasse `Rails::Generator::Base` zu bevorzugen. Da der im Zuge dieser Arbeit zu erstellende Generator ebenfalls Eingabeparameter aus der Kommandozeile akzeptieren werden muss, wird dieser von der Klasse *NamedBase* abgeleitet sein:

```
Class ExtendedGenerator < Rails::Generator::NamedBase
  ...
end
```

Abbildung 6.10 Klassendefinition von ExtendedGenerator

Wie jede Klasse in Ruby besitzt auch der Generator eine *initialize()*-Methode, der die in der Kommandozeile angegebenen Argumente übergeben, und in dem Array `@args` gespeichert werden. Weiters enthält diese Methode Anweisungen, die beim Aufruf des Generators ausgeführt werden.

Ein weiterer Bestandteil, den jede Generator-Klasse enthalten muss ist die *manifest()*-Methode. Sie ist verantwortlich für das Erstellen von Verzeichnissen und Dateien, sowie das Kopieren von vorhandenen Templates in die dafür vorgesehenen Verzeichnisse. Diese Aufgaben werden von den Methoden *m.directory()* und *m.templates()* erledigt, die innerhalb der *manifest()*-Methode aufgerufen werden. Sie geben dem Benutzer auch die Möglichkeit vorhandene Dateien beim Aufruf des Generators durch neue Dateien zu ersetzen oder nicht.

Auch die *banner()*-Methode kann durch eine eigene Version ersetzt werden. Sie dient dazu eine einzeilige Kurzbeschreibung für die richtige Verwendung des Generators auszugeben, z.B.:

```
def banner
  "Verwenden Sie den #{$0} Generator folgendermas-
  sen:\n\n#{$0} #{spec.name}"
End
```

Abbildung 6.11 Methode ExtendedGenerator::banner()

Das sind im Großen und Ganzen die wichtigsten Bestandteile, die ein Generator in Ruby on Rails besitzen muss. Nun ist es an der Zeit mit der Beschreibung des erweiterten Generators zu beginnen.

6.3.1 Vorgehensweise bei der Erstellung des erweiterten Rails-Generators

In Abschnitt 6.2 wurden die beiden wesentlichen Funktionalitäten erläutert, aufgrund derer die Erstellung eines eigenen oder die Erweiterung des bestehenden Scaffold-Generators als sinnvoll erscheint. Nun geht es darum zu beschreiben, ob die technische Umsetzung dieser Probleme möglich ist und wie die tatsächliche Realisierung aussehen könnte. Da in Bezug auf die Integration der Beziehungen zwischen den Datenbanktabellen in die Modellklassen die Lösung nicht ausschließlich aufgrund des vorhandenen Datenbankschemas möglich ist, stellt sich die Umsetzung dieser Funktionalität im erweiterten Generator als relativ aufwendig dar und wird im Zuge dieser Arbeit nicht weiter behandelt. Sie beschränkt sich vielmehr auf die automatische Integration von Validierungsmethoden in die Klassen der Modelle.

Für die Integration der Funktionalität von Attributs-Validierungen, die schon beim Aufruf des Scaffold-Generators in die Modellklassen eingetragen werden sollen, wurde bei der Entwicklung folgende Vorgehensweise ins Auge gefasst:

Innerhalb der Datenbank werden zwei weitere Tabellen, die für die Validierung verwendet werden, angelegt. Eine der beiden Tabellen, die so genannte *validations*-Tabelle beinhaltet die Validierungsregeln, die vom Generator in die Modellklassen eingefügt werden sollen. Das Schema dieser Tabelle sieht so aus:

- *id*...stellt das Primärschlüsselattribut der Tabelle dar.
- *name* ... gibt den Namen der Validierungsregel an.
- *method* ... beinhaltet die Validierungsmethode, die auf das Attribut angewendet werden soll.
- *withtext* ... die Option *:with*, die bei der Validierungsmethode *validates_format_of* verwendet werden muss.

Um diese Validierungsregeln auf die korrekten Attribute anwenden zu können, muss eine Verbindung zwischen den Attributen und den Regeln hergestellt werden. Dies

geschieht mit Hilfe der Mapping-Tabelle *mappings*. Diese Tabelle stellt die Verbindung zwischen den zu überprüfenden Attributen, die ja selbst aus der Sicht des Datenbankschemas Spalten von Tabellen und keine Datensätze sind, mit den in der *validations*-Tabelle definierten Validierungsregeln. Um eine eindeutige Zuweisung der Validierungen zu den Attributen zu gewährleisten, enthält die Tabelle auch die Namen der Tabellen zu denen die Attribute gehören. Somit ergibt sich folgendes Schema für die Mapping-Tabelle:

- *id*...stellt das Primärschlüsselattribut der Tabelle dar.
- *table_name* ... speichert den Tabellennamen.
- *attribute_name* ... speichert den Namen des zu validierenden Attributs.
- *validation_name* ... enthält einen Fremdschlüssel auf die zugehörige Validierungsregel.

Die Aufgabe des erweiterten Generators ist es nun alle in der *mappings*-Tabelle gespeicherten Validierungsregeln durchzugehen und für jedes Attribut jeder gespeicherten Tabelle die zugehörige Validierungsregel aus der *validations*-Tabelle zu holen. Danach sollen sie in die Modellklasse, die die jeweilige Datenbanktabelle abbildet, eingetragen werden. Die folgende Abbildung stellt das Mapping der Validierungsregeln auf die jeweiligen Attribute schematisch dar:

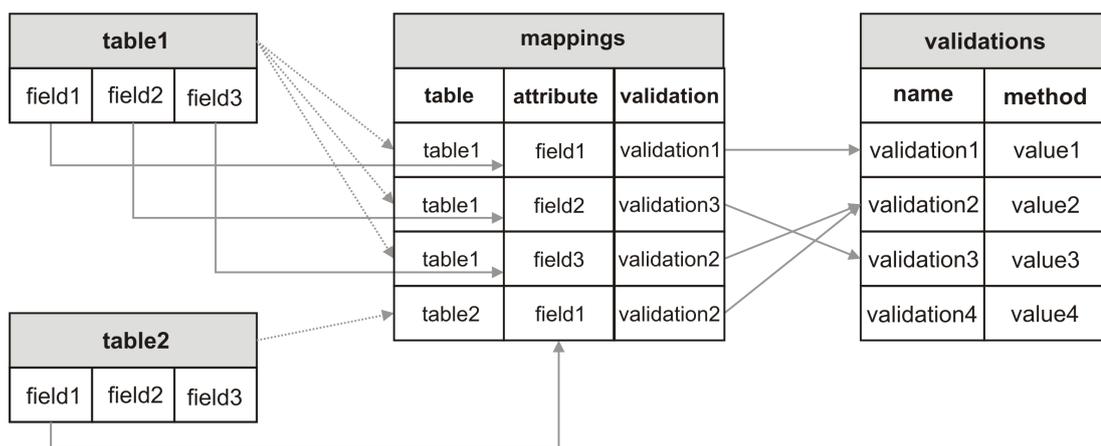


Abbildung 6.12 Mapping der Validierungstabellen

6.3.2 Funktionsweise des erweiterten Rails-Generators

Um den erweiterten Generator für die Erstellung einer ersten kleinen Webanwendung verwenden zu können, muss zuerst durch Eingabe des Befehls `rails we-`

bapp die grundlegende Verzeichnisstruktur der Applikation angelegt werden. Danach kann aus dem Verzeichnis der Applikation der Generator gestartet werden.

Der erweiterte Generator wird über die Kommandozeile aufgerufen. Der Benutzer muss die Angaben für die Datenbank, den Host, den Benutzernamen und (falls vorhanden) das Passwort als Argumente an den Befehl anhängen, also z.B.

```
ruby script/generate extended_generator mydatabase localhost
root [mypassword]
```

Werden die Angaben unvollständig getätigt (d.h. ein benötigter Parameter nicht angegeben), so wird der Benutzer darauf aufmerksam gemacht, seine Angaben zu vervollständigen. Sind alle notwendigen Angaben vorhanden, so beginnt der Generator damit, die vom Benutzer angegebene Datenbank in die Rails-Konfigurationsdatei der Anwendung database.yml einzutragen, da bei Ausführen des rails-Befehls hier eine Standarddatenbank eingetragen wird, die in den meisten Fällen nicht mit der gewünschten Datenbank übereinstimmt.

Für das Einfügen von Textzeilen in Dateien ist die Methode *update_file()* zuständig, der bei ihrem Aufruf drei Parameter übergeben werden. Der erste Parameter gibt die Datei an, die aktualisiert werden soll, der zweite den Text, der geschrieben werden soll und der dritte Parameter gibt an, ob die Konfigurationsdatei oder Modelldateien aktualisiert werden.

```
def update_file(file_name, lines, config)
  @logger.info "-----Aktualisiere Datei-----"
  @logger.info "Versuche Datei #{file_name} zu aktualisieren.."

  begin
    file = File.new(file_name, "r")
    fileHandler = FileHandler.new(file)
    if config == 0 # Schreiben der Modell-Klassen
      fileHandler.read_file()
      fileHandler.write_file(lines)
    else # Schreiben des Config-Files
      fileHandler.write_db_to_config(@dbparams['database'],
        @dbparams['user'], @dbparams['password'])
    end

    @logger.info "Datei #{file_name} aktualisiert."
    @logger.info ""
    @logger.info "-----Aktualisierung abgeschlossen----"
  rescue
    $stderr.print "\nFehler beim Oeffnen der Datei
      #{file_name}.\n"
    @error = true
  end
end #update_file
```

Abbildung 6.13 Methode ExtendedGenerator::update_file()

In diesem Fall wird die Konfigurationsdatei aktualisiert und daher diese beim Aufruf der Methode als erster Parameter angegeben. Der zweite Parameter ist nur ein Leerstring, da die Datenbankparameter in der Instanzvariable *@dbparams* gespeichert werden und für die Verarbeitung auf dieses Array zurückgegriffen wird. Der dritte Parameter erhält den Wert 1 und symbolisiert damit, dass die Konfigurationsdatei zur Bearbeitung an der Reihe ist. Wie man dem obigen Quellcode entnehmen kann, ist für die tatsächliche Bearbeitung der Dateien die Klasse *FileHandler* zuständig, deren Funktionsweise weiter unten im Text noch genauer erläutert wird.

Im nächsten Schritt aktualisiert der Generator die Datenbankverbindung von Active Record, da diese noch die ursprüngliche Einstellung aus der Konfigurationsdatei enthält. Diese Umstellung ist notwendig, damit einerseits die Funktionalität, die das Active Record zur Verfügung stellt, voll eingesetzt werden kann, andererseits verwendet auch der Scaffold-Generator die Datenbankverbindung über Active Record und würde somit bei einer falschen Konfiguration nicht das gewünschte Ergebnis liefern. Natürlich verwendet auch der erweiterte Generator Active Record um auf die Datenbank zuzugreifen wodurch er unabhängig ist vom zu Grunde liegenden DBMS.

Es folgt nun der Aufruf der Methode *get_data_from_db()*, die dafür verantwortlich ist, dass alle Tabellennamen aus der Datenbank gelesen (mit Ausnahme von *mappings* und *validations*) und im Array *table_names* gespeichert werden. Bevor dies geschieht, wird überprüft, ob in der Datenbank die beiden Tabellen *mappings* sowie *validations* vorhanden sind, die für die Erweiterung der Modell-Klassen um die Validierungsfunktionalität benötigt werden. Ist dies der Fall, werden in weiterer Folge für jede Tabelle seine Attribute aus der Datenbank geholt und diese gemeinsam mit dem Namen der Tabelle in einem neuen Table-Objekt gespeichert. Das Instanz-Array *@tableobjects* stellt die Datenstruktur dar, die die Table-Objekte und damit die Namen der Tabellen und deren Attribute speichert.

```
def get_data_from_db
  begin

    # Auslesen der Tabellennamen aus dem ResultSet.
    # Es werden alle Tabellennamen gespeichert, außer die von
    # mappings und validations.
    # Für diese für die Validierung zuständigen Tabellen sollen
    # keine Modelle angelegt werden.
    # Es wird jedoch geprüft, ob diese Tabellen vorhanden sind.
    # Wenn nicht, wird eine Meldung ausgegeben.
    @logger.info "-----Holen der Daten aus der Datenbank---"

    statement="SHOW tables"
    begin
      result = db_connection.execute(statement)
    rescue
      ActiveRecord::StatementInvalid
    end

    boValidations = false
    boMappings = false

    table_names = []
    result.each do |array|
      array.each do |value|
        #Speichern der Werte im Array der Tabellennamen
        table_names << value unless(
          value == "mappings" or value == "validations")
        if(value == "validations")
          boValidations = true
        end
        if(value == "mappings")
          boMappings = true
        end
      end
    end

    # Das Array attributes speichert die Namen der Felder jeder
    # Tabelle. Jedes Tabellenobjekt erhält den Tabellennamen und
    # das Array mit den gespeicherten Attributen.

    if(boValidations and boMappings)
      @logger.info "Datenbanktabellen fuer die Validierung
        vorhanden!\nTable-Objekte koennen erstellt werden..."

      table_names.each do |table_name|
        attributes = []
        statement="SELECT * FROM #{table_name}"
        begin
          result = db_connection.execute(statement)
        rescue
          ActiveRecord::StatementInvalid
        end

        fields = result.fetch_fields
        fields.each do |field|
          attributes << field.name
        end
        @table_objects << Table.new(table_name, attributes)
      end
    end
  end
end
```

Abbildung 6.14 Methode ExtendedGenerator::get_data_from_db() – Abschnitt 1

Abbildung 6.14 zeigt den Vorgang für das Speichern der Tabellennamen im Array `table_names` mit Ausnahme der beiden Tabellen, die für die Validierung verwendet werden. Für diese sollen natürlich in weiterer Folge keine Modellklassen erzeugt werden. Weiters werden die Namen aller Attribute jeder Tabelle gespeichert und das Array `@table_objects` befüllt.

```
# Suche aus der Mapping-Tabelle die Tabellen, deren
# Attribute und aus der Validierungs-Tabelle die
# zugehörigen Validierungsregeln.

statement="SELECT table_name, attribute_name, method,
withtext FROM mappings m, validations v WHERE
m.validation_name=v.name"
begin
  result = ActiveRecord::Base.connection.execute(statement)
rescue
  ActiveRecord::StatementInvalid
end

result.each_hash do |rulehash|
  rulehash['withtext'] = "" if rulehash['withtext'] == nil
  @constraint_objects << Constraint.new(
    rulehash['table_name'], rulehash['attribute_name'],
    rulehash['method'], rulehash['withtext'])
end

else
  @logger.info "Die Datenbanktabellen fuer die Validierung
sind nicht vorhanden!\n"
  @logger.info "Es muessen Tabellen mit Namen \"mappings\"
und \"validations\" existieren."
end

# Wenn Daten- und Validierungstabellen vorhanden sind,
gib eine Meldung aus.
if(@table_objects.size > 0)
  @logger.info "Table-Objekte vorhanden."
end
if(@constraint_objects.size > 0)
  @logger.info "Constraint-Objekte vorhanden."
end

rescue
  $stderr.print "\nFehler beim Holen der Daten aus der
Datenbank - ueberpruefen Sie ihre Angaben!\n"
  @error = true
end
end #get_data_from_db
```

Abbildung 6.15 Methode ExtendedGenerator::get_data_from_db() – Abschnitt 2

Abbildung 6.15 schließlich zeigt den Teil des Quellcodes, der für das Mapping der Validierungsregeln auf die Attributnamen verantwortlich ist. Im Array

@constraint_objects werden Constraint-Objekte gespeichert, die die Namen der Tabelle, der Attribute, der Validierungsmethoden und – falls vorhanden – den Text für die *:with*-Option enthalten.

Nachdem alle Tabellen aus der Datenbank gelesen worden sind, wird die Methode *manifest()* ausgeführt und damit beginnt die eigentliche Erstellung des Prototyps der Webanwendung. Innerhalb dieser Methode macht sich der erweiterte Generator die Möglichkeit zu nutze über einen Aufruf der Methode *dependency()* den in Ruby on Rails integrierten Scaffold-Generator zu starten und so für die gespeicherten Tabellennamen die Modell-Klassen zu erzeugen und damit auch Controller und Views zu erstellen. Nun ist bereits ein Grundgerüst vorhanden, in dem jedoch zusätzlichen Funktionalitäten noch nicht implementiert sind.

Mit Hilfe der Klasse *FileHandler* wird nun versucht, die gerade erzeugten Modell-Klassen zu aktualisieren und um die für die Validierung notwendigen Zeilen zu erweitern. Sie besitzt im Großen und Ganzen zwei Methoden, die für die Aktualisierung von Dateien verantwortlich sind: *FileHandler::read_file()* sowie *FileHandler::write_file(insert_lines)*. Die Funktionsweisen der beiden Methoden sehen folgendermaßen aus:

FileHandler::read_file() liest den gesamten Inhalt der geöffneten Datei in ein Array ein, das die Zeilen der Datei speichert. Weiters wird überprüft, an welchem Index sich die Zeile der Klassendefinition befindet, da die einzufügenden Methoden natürlich zwischen der Definition und ihrem Ende eingetragen werden müssen.

```
def read_file()
  @logger.info " Lese Datei..."
  current_index = 0
  class_found = false
  @file.each_line do |line|
    current_index += 1
    @lines << line
    # @logger.info "Zeilenindex: #{current_index}"
    if find_class_line(line)
      @class_line_index = current_index
      class_found = true
    end
  end
  if class_found == false
    @logger.info " Keine class-Zeile gefunden. Datei kann
nicht aktualisiert werden."
  end
end # read_file
```

Abbildung 6.16 Methode *ExtendedGenerator::read_file()*

`FileHandler::write_file(insert_lines)` schreibt den aus der geöffneten Datei gelesenen Text wieder in die Datei zurück, ergänzt um den Inhalt der Variable `insert_lines`, die der Methode bei ihrem Aufruf übergeben wird. Dieser Inhalt wird unmittelbar nach der Zeile der Klassendefinition in die Modell-Klasse geschrieben.

```
def write_file(insert_lines)
  @file = File.open(@file.path, "w")
  @logger.info " Datei zum Schreiben geoeffnet..."
  current_index = 0
  @lines.each do |line|
    current_index += 1
    @logger.info " Schreibe in Datei: " + line
    @file.puts line
    if current_index == @class_line_index
      insert_lines.each do |insert_line|
        @logger.info " Schreibe in Datei: " + insert_line
        @file.puts insert_line
      end
    end
  end
  @file.close()
  @logger.info " Datei geschlossen."
  @logger.info ""
end # write_file
```

Abbildung 6.17 Methode `ExtendedGenerator::write_file(insert_lines)`

Wurden alle Zeilen erfolgreich in die Dateien der Modell-Klassen geschrieben, so ist die Aufgabe des erweiterten Generators beendet und seine korrekte Funktionsweise kann durch die Ausführung des erstellten Prototyps im Browser überprüft werden.

Kapitel 7

Models on Rails

7.1 Überblick

Um die Erstellung von prototypischen Implementierungen von Webanwendungen in Ruby on Rails noch weiter zu automatisieren, sollte es möglich sein, auch das Datenbankschema, das als Grundlage für die Webapplikation dient, mit Hilfe eines Codegenerators zu erzeugen. Dazu soll der Benutzer die Möglichkeit erhalten, das Domänenmodell der Anwendung als ER-Diagramm mit Hilfe eines graphischen Model-Editors zu erstellen. Dieser Editor basiert auf dem Open Source-Softwareframework *Eclipse*. Es verkörpert nach [Shav04] drei Dinge: Eine Java-Entwicklungsumgebung, eine Plattform für Tool-Integration und eine Open Source-Gemeinde, wobei vor allem das zweite Charakteristikum für diese Arbeit von Bedeutung ist, da einige Erweiterungen von Eclipse bei der Erstellung des Model-Editors eingesetzt werden.

Da es mit erheblichem Aufwand verbunden wäre, den Codegenerator von Ruby on Rails so zu erweitern, dass er als Eingabe die erstellten Modelle akzeptieren kann, ist ein Zwischenschritt notwendig. In diesem Zwischenschritt werden die Modelle in eine derartige Form gebracht, dass in weiterer Folge mit dem erweiterten Codegenerator von Rails eine Webanwendung generiert werden kann.

Die Aufgaben des Frameworks „*Models on Rails*“, das somit entstehen soll, kann man in folgenden drei Schritten zusammenfassen, die in Abbildung 7.1 veranschaulicht werden:

1. Erstellen des ER-Modells als Entwurf für die Webanwendung.

2. Transformation des Modells in eine für den erweiterten Generator akzeptable Form (Modell-zu-Code).
3. Erstellen des Prototyps einer Webanwendung mit Hilfe des erweiterten Codegenerators von Ruby on Rails.

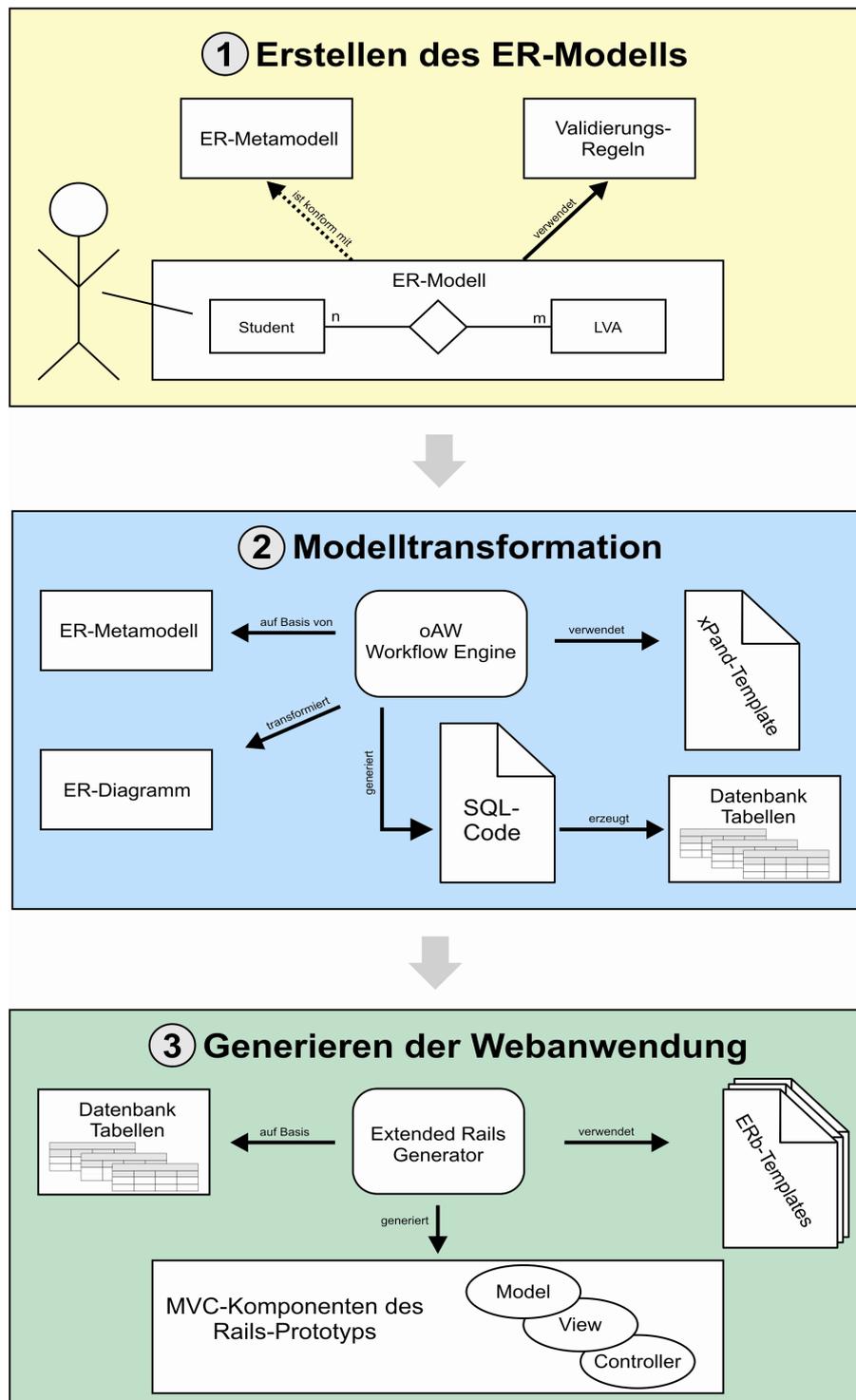


Abbildung 7.1 Die Funktionsweise von Models on Rails

7.2 Erstellen des ER-Modells

Wie in der Einleitung erwähnt, soll der Benutzer die Möglichkeit erhalten mit Hilfe des Frameworks ER-Diagramme zu erstellen. Anhand vorgegebener Komponenten, die der Editor zur Verfügung stellt, soll er so rasch einen Entwurf für die Datenbank, auf der die zu erstellende Webanwendung basiert, erzeugen können.

Um diesen ersten Schritt zu erfüllen, wurde ein Metamodell für die ER-Sprache erstellt. Während ein Modell die Struktur der Elemente einer zu modellierenden Domäne beschreibt, beschreibt ein Metamodell wie gültige Modelle aufgebaut sind. Das hier entwickelte ER-Metamodell enthält die wichtigsten Konzepte der ER-Modellierungstechnik sowie deren Zusammenhänge und wurde um einen weiteren Aspekt ergänzt, der für die automatische Erstellung der Validierungsregeln, die später vom Rails-Codegenerator verwendet werden, zuständig ist.

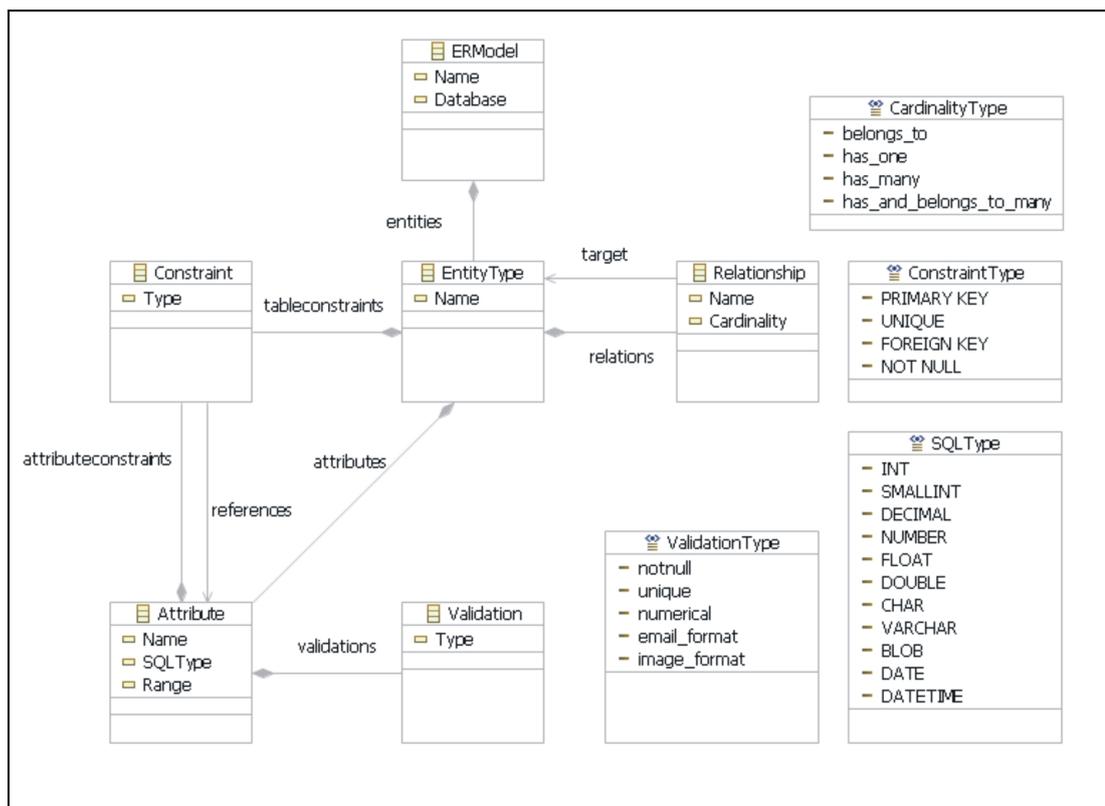


Abbildung 7.2 Das ER-Metamodell

Wie aus der Abbildung ersichtlich gibt es ein Wurzelement: *ERModel*. Man kann es auch als das Element bezeichnen, das durch das Metamodell beschrieben wird, das heißt die Sprache selbst. Weiters gibt es die grundlegenden Elemente der ER-Modellierung: Die Entität (*Entity Type*), das Attribut (*Attribute*) und die Beziehung

(*Relationship*). Jede Beziehung besitzt eine Kardinalität (*Cardinality*). Die hier angegebenen Kardinalitätstypen beziehen sich auf die in Ruby on Rails verwendeten Assoziationen. Ein weiteres Element der Sprache stellt das *Constraint*-Konzept dar. Constraints werden mit Ausnahme des Primärschlüssels üblicherweise nicht in der ER-Modellierung dargestellt. Da sie aber bei der Erstellung des Implementierungsdatenmodells angegeben werden, ist es sinnvoll, sie schon an dieser Stelle zu integrieren. Sie können entweder als *Tableconstraints* angegeben, dann müssen sie das jeweilige Attribut referenzieren, oder aber direkt mit dem Attribut verknüpft werden (*Attributeconstraints*). Ein weiteres Element, das nicht von der ursprünglichen ER-Modellierung in das Metamodell übertragen wurde ist das *Validation*-Element. Jedes Attribut kann keines, eines oder mehrere dieser *Validations* besitzen. Diese Elemente haben einen bestimmten Typ. Dieser Typ bezieht sich auf die Validierungsmethoden, die für die Validierung der Modelle in Ruby on Rails zuständig sind. Somit können die Validierungen schon bei der ER-Modellierung berücksichtigt und vom Codegenerator automatisch in die Mapping-Tabelle eingetragen werden.

Eclipse Modeling Framework

Es gibt verschiedene Ansätze, wie Metamodelle formuliert werden können. Das hier verwendete *Ecore*-Format basiert auf dem von der *Object Management Group* (OMG) eingeführten *Meta Object Facility* (MOF) [OMG06] und wird innerhalb des *Eclipse Modeling Frameworks* (EMF) für die Beschreibung von Metamodellen eingesetzt. Ziel von EMF ist es, Metamodelle im *Ecore*-Format zu erstellen und aus den daraus entwickelten Modellen Code zu generieren [EMF06]. Um ausgehend von dem Metamodell konkrete ER-Modelle gestalten zu können bietet EMF die Möglichkeit diese dynamisch zu erstellen und zu bearbeiten. Im Moment ist es jedoch nur möglich, die konkreten Diagramme unter Verwendung eines Baumeditors zu entwickeln. Eine benutzerfreundlichere Variante für die Erstellung von ER-Diagrammen ist jedoch ein grafischer Editor, mit dem sich die Zusammenhänge der Bestandteile des Modells bedeutend übersichtlicher darstellen lassen. Weitere Vorteile, die die Erstellung eines solchen Editors mit sich bringen, ist eine höhere Geschwindigkeit bei der Erstellung der Entwürfe durch eine geringere Fehleranfälligkeit und eine bessere Dokumentation für das aus dem Entwurf generierte Datenbankschema.

Eclipse Graphical Modeling Framework

Um den Aufwand für die Erstellung des Modell-Editors möglichst gering zu halten, kommt das *Eclipse Graphical Modeling Framework* (GMF) zum Einsatz. Es ermöglicht ausgehend von dem in EMF erstellten Metamodell unter zu Hilfenahme von

Komponenten des *Eclipse Graphical Editing Framework* (GEF) [GEF06] die Erstellung eines grafischen Editors für beliebige Modelle mit selbst ausgewählten Notationselementen. Auf die genaue Vorgehensweise bei der Erstellung des grafischen Editors wird hier nicht eingegangen. Es sei an dieser Stelle nur auf [GMF06] verwiesen, wo man sich anhand einiger Einführungen ein Bild über die Funktionsweise des Frameworks machen kann.

Um die Funktionsweise des fertigen Editors zu erläutern, wird als kleines Beispiel eine Mitgliederstammdatenverwaltung eines Vereins entwickelt. Dieses Beispiel wird auch in den folgenden Abschnitten bei der Generation des Prototyps einer Webanwendung weiterverwendet. Abbildung 7.3 stellt die Erstellung eines konkreten ER-Modells mit dem EMF-Baumeditor der Modellierung mit dem GMF-Modell-Editor gegenüber.

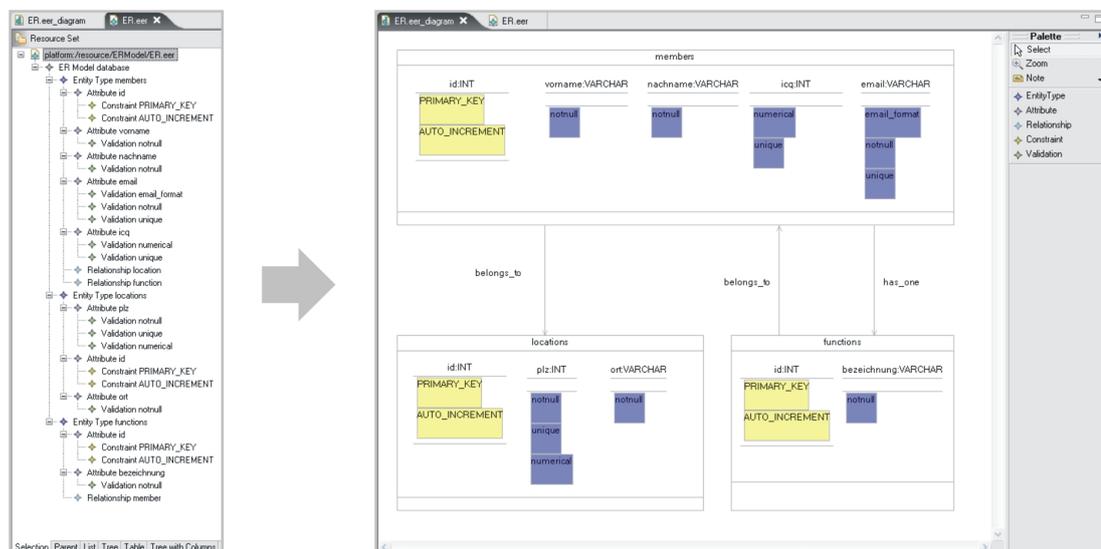


Abbildung 7.3 Gegenüberstellung EMF-Baumeditor – GMF-Modell-Editor

7.3 Modelltransformation

Damit der erweiterte Generator von Rails zur Anwendung kommen kann, muss ein Zwischenschritt erfolgen, der mit Hilfe des grafischen ER-Modell-Editors erstellte Modelle in eine Form bringt, die auf eine Datenbank abgebildet werden kann. Dazu ist es notwendig, aus den erstellten ER-Diagrammen ein Implementierungsdatenmodell, also in diesem konkreten Fall ein SQL-Skript, zu generieren, aus dem die Datenbanktabellen angelegt werden können. Die Tabellen stellen in weiterer Folge den Ausgangspunkt für den Einsatz des erweiterten Generators dar.

OpenArchitectureWare

OpenArchitectureWare (oAW) wurde ebenfalls als Erweiterung von Eclipse realisiert. Das Ziel von oAW ist ausgehend von EMF-Modellen mit Hilfe von Templates Code zu generieren. Da EMF selbst auf die Erzeugung von Java-Quellcode beschränkt ist, ist der Einsatz von oAW und seinen Templates auch besonders interessant, wenn es um die Erstellung von Quellcode- oder Textdateien geht, die keinen Java-Code enthalten sollen. Dadurch ist es auch möglich aus einem logischen Datenbankmodell automatisch ein Implementierungsdatenmodell generieren zu lassen. Diese Templates werden in der in oAW integrierten Sprache *Xpand* [Efft06b] formuliert, einer Sprache, die es erlaubt, die automatische Codegenerierung zu kontrollieren.

Xpand-Template

Das eben erstellte Metamodell ist der Ausgangspunkt für die Codegenerierung mit Hilfe von OpenArchitectureWare. oAW bietet mit Xpand eine flexible Template-Sprache, die für die Steuerung des erzeugten Codes verantwortlich ist. Das Xpand-Template nimmt ein konkretes Modell, dem das Metamodell im Ecore-Format zu Grunde liegt, als Eingabe und kann anhand festgelegter Regeln die weitere Codeerstellung steuern.

Eine Xpand Template-Datei besteht aus einem oder mehreren Templates, die mit dem Schlüsselwort DEFINE eingeleitet werden. Ein Template hat einen Namen und ist an einen bestimmten Typ gebunden. Innerhalb des Templates wird mit dem Schlüsselwort FILE ein Block definiert, der eine Datei mit dem angegebenen Namen öffnet. Der Text, der innerhalb dieses FILE...ENDFILE Blocks generiert wird, wird in diese Datei geschrieben. Die Verwendung von Kontrollstrukturen wie FOREACH oder IF ist genauso möglich, wie der Aufruf weiterer Templates über das Schlüsselwort EXPAND [Efft06a]. Für eine genaue Beschreibung der Syntax von Xpand sei auf [Efft06b] verwiesen.

Um nun aus einem konkreten ER-Diagramm automatisch ein Implementierungsdatenmodell, das heißt ein SQL-Script, das auf der Datenbank ausgeführt werden kann, erzeugen zu können, muss ein Xpand-Template erstellt werden, das für jede Komponente des Metamodells genau festlegt, wie der Code für das konkrete Modell auszu-sehen hat. In diesem speziellen Fall muss das Template auch schon die Erstellung der Tabellen für den Validierungsmechanismus der Webapplikation enthalten, da diese auch vom Generator erstellt werden sollen. Sie werden jedoch nicht als eigene

Entitäten im ER-Diagramm modelliert. Die Abbildungen 7.4 und 7.5 zeigen das fertige Xpand-Template für die Codegenerierung mit OpenArchitectureWare. Der blau gedruckte Text zeigt hart codierte Passagen, die so wie sie sind in die generierte Datei geschrieben werden. Die in rot gehaltenen Textstellen signalisieren den anhand des erstellten ER-Modells generierten Text.

```
«IMPORT eer»

«DEFINE sqlScript FOR eer::ERModel»
«FILE Name+".sql"-»

CREATE DATABASE IF NOT EXISTS «Database»;
USE «Database»;

«FOREACH entities AS entity -»
  DROP TABLE IF EXISTS «entity.Name»;
«ENDFOREACH»
«FOREACH entities AS entity-»

  CREATE TABLE «entity.Name» (
    «FOREACH entity.attributes AS attr SEPARATOR ','-»
      «attr.Name» «attr.SQLType»
      «IF attr.Range.matches("")==false»
        («attr.Range»)
      «ENDIF»
    «FOREACH attr.attributeconstraints AS const-»
      «const.Type»
    «ENDFOREACH»
  «ENDFOREACH-»
  «FOREACH entity.relations AS rel-»
    ,«rel.Name»_id «rel.target.attributes.get(0).SQLType»
  «ENDFOREACH-»
  );
«ENDFOREACH»
```

Abbildung 7.4 Xpand-Template – Abschnitt 1

Abbildung 7.4 zeigt den Teil des Xpand-Template, der zuerst für die Definition des Templates sowie der Datei, in die der Code geschrieben wird, verantwortlich ist. Doch vor allem ist er zuständig für die automatische Generierung der SQL-Statements, die für jede Entität eine Datenbanktabelle mit ihren Spalten und Fremdschlüsseln, sowie Constraints anlegen.

```

DROP TABLE IF EXISTS mappings;
CREATE TABLE mappings (
  id INT PRIMARY KEY auto_increment,
  table_name VARCHAR(50),
  attribute_name VARCHAR(50),
  validation_name VARCHAR(100)
);

«FOREACH entities AS entity-»
  «FOREACH entity.attributes AS attr-»
    «FOREACH attr.validations AS valid-»
      INSERT INTO mappings VALUES (NULL,
        '«entity.Name»', '«attr.Name»', '«valid.Type»');
    «ENDFOREACH-»
  «ENDFOREACH-»
«ENDFOREACH-»

DROP TABLE IF EXISTS validations;

CREATE TABLE validations (
  id INT PRIMARY KEY auto_increment,
  name VARCHAR(50),
  method VARCHAR(200),
  withtext VARCHAR(250)
);

INSERT INTO validations VALUES (NULL, 'notnull',
  'validates_presence_of', NULL);
INSERT INTO validations VALUES (NULL, 'unique',
  'validates_uniqueness_of', NULL);
INSERT INTO validations VALUES (NULL, 'numerical',
  'validates_numericality_of', NULL);
INSERT INTO validations VALUES (NULL, 'email_format',
  'validates_format_of', '%r{^[_a-zA-Z0-9-]+(\.[_a-zA-Z0-9-
  ]+)*@[a-zA-Z0-9-]+\.[a-zA-Z]{2,4}$}');
INSERT INTO validations VALUES (NULL, 'image_url_format',
  'validates_format_of', '%r{^http:.\.(gif|jpg|png)$}');
«ENDFILE»
«ENDDEFINE»

```

Abbildung 7.5 Xpand-Template – Abschnitt 2

Der erste Teil von Abbildung 7.5 beinhaltet hart codiert die Erstellung der Mapping-Tabelle, die die Validierungen, die im ER-Modell für Attribute von Entitäten festgelegt wurden, mit den Validierungsmethoden aus der Validierungstabelle verbindet. Den Abschluss des Templates macht die Erstellung der Validierungstabelle mit einigen vorgegebenen Validierungsmethoden. Man sieht hier die Verwendung von Regulären Ausdrücken bei der Überprüfung des E-Mail-Formats und dem Format der URL für die Datei eines Bildes.

Workflow Engine

Im Moment stellen sich die vorgestellten Dinge noch als lose Teile auf dem Weg zur automatischen Codegenerierung dar. Die Verbindung zwischen dem abstrakten ER-Metamodell, den konkreten ER-Diagrammen und dem darauf anzuwendenden Xpand-Template stellt die so genannte *Workflow Engine* her. Sie dient im Großen und Ganzen der Konfiguration und dem Ausführungsprozess der Codegenerierung. In der Workflow Engine, die im Wesentlichen eine XML-Konfigurationsdatei darstellt, wird festgelegt, für welche Instanz des Metamodells die Codegenerierung stattfindet, welches Metamodell für den Generierungsprozess eingesetzt werden und welches Template bei der Generierung zur Anwendung kommen soll. Die Workflow Engine, die das ER-Metamodell verwendet, auf das Beispielmmodell angewendet werden soll und das Xpand-Template verwendet, könnte wie in Abbildung 7.6 aussehen. Der blau gedruckte Text markiert die Referenzen auf das ER-Metamodell, das konkrete ER-Modell und das Template:

```
<workflow>
  <property name="model" value="ER.eer" />
  <property name="src-gen" value="src-gen" />

  <component class="org.openarchitectureware.emf.XmiReader">
    <metaModelFile value="metamodel/er.ecore" />
    <modelFile value="{model}" />
    <outputSlot value="model" />
    <firstElementOnly value="true" />
  </component>

  <component class="org.openarchitectureware.xpand2.Generator">
    <metaModel
      class="org.openarchitectureware.type.emf.EmfMetaModel">
      <metaModelFile value="metamodel/er.ecore" />
    </metaModel>
    <expand value="template::Simple::sqlScript FOR model" />
    <genPath value="{src-gen}" />
  </component>
</workflow>
```

Abbildung 7.6 Workflow Engine

Wird nun die Workflow Engine gestartet, so wird damit der Codegenerierungsprozess in Gang gesetzt und die Dateien erstellt, wie sie im Template angegeben sind. Das Ziel der generierten Dateien ist ebenfalls in der Datei der Workflow Engine anzugeben. Ist der Prozess ohne Auftreten von Fehlern abgeschlossen, so werden die automatisch generierten Dateien im angegebenen Verzeichnis abgelegt. In diesem Beispiel findet sich nur eine Datei im Ausgabeordner nämlich das SQL-Script zum Anlegen der Datenbanktabellen, das die Abbildungen 7.7 und 7.8 zeigen:

```
CREATE DATABASE IF NOT EXISTS mydb;
USE mydb;

DROP TABLE IF EXISTS members;
DROP TABLE IF EXISTS locations;
DROP TABLE IF EXISTS functions;

CREATE TABLE members (
  id INT PRIMARY KEY AUTO_INCREMENT
  ,vorname VARCHAR(50) NOT NULL
  ,nachname VARCHAR(50) NOT NULL
  ,email VARCHAR(100)
  ,icq INT(12)
  ,location_id INT);

CREATE TABLE locations (
  id INT PRIMARY KEY AUTO_INCREMENT
  ,plz INT(4)
  ,ort VARCHAR(50)
);

CREATE TABLE functions (
  id INT PRIMARY KEY AUTO_INCREMENT
  ,bezeichnung VARCHAR(50) NOT NULL UNIQUE
  ,member_id INT);
```

Abbildung 7.7 Automatisch generiertes SQL-Script – Abschnitt 1

Der erste Teil des automatisch generierten SQL-Scripts in Abbildung 7.7 beinhaltet die Statements für das Anlegen der Entitäts-Tabellen. Als Beispiel dient die Mitgliederstammdatenverwaltung. Der dafür verantwortliche Code des Templates ist oben in Abbildung 7.4 dargestellt.

```
DROP TABLE IF EXISTS mappings;
CREATE TABLE mappings (
  id INT PRIMARY KEY auto_increment,
  table_name VARCHAR(50),
  attribute_name VARCHAR(50),
  validation_name VARCHAR(100));
INSERT INTO mappings VALUES(NULL, 'members', 'vorname', 'notnull');
INSERT INTO mappings VALUES(NULL, 'members', 'nachname',
  'notnull');
INSERT INTO mappings VALUES(NULL, 'members', 'email',
  'email_format');
INSERT INTO mappings VALUES(NULL, 'members', 'email', 'notnull');
INSERT INTO mappings VALUES(NULL, 'members', 'email', 'unique');
INSERT INTO mappings VALUES(NULL, 'members', 'icq', 'numerical');
INSERT INTO mappings VALUES(NULL, 'members', 'icq', 'unique');
INSERT INTO mappings VALUES(NULL, 'locations', 'plz', 'notnull');
INSERT INTO mappings VALUES(NULL, 'locations', 'plz', 'unique');
INSERT INTO mappings VALUES(NULL, 'locations', 'plz', 'numerical');
INSERT INTO mappings VALUES(NULL, 'locations', 'ort', 'notnull');
INSERT INTO mappings VALUES(NULL, 'functions', 'bezeichnung',
  'notnull');
```

Abbildung 7.8 Automatisch generiertes SQL-Script – Abschnitt 2

Die Abbildung 7.8 zeigt das Ergebnis der Codegenerierung, für den der zweite Abschnitt des Xpand-Templates verantwortlich ist: Die Mapping-Tabelle wird mit Datensätzen befüllt. Natürlich ist auch der Code der Validierungstabelle in der Datei vorhanden, er wird aber an dieser Stelle nicht noch einmal wiederholt.

Nun kann das automatisch generierte SQL-Script in die Datenbank übertragen werden, bei MySQL etwa durch den Befehl

```
mysql -u user -p password < database.sql
```

Damit werden die Datenbanktabellen erstellt und der erweiterte Generator kann ausgeführt werden.

7.4 Generieren der Webanwendung

Der dritte und letzte Schritt, den das Framework dem Benutzer bietet ist die Erstellung eines ersten Prototyps einer Webanwendung mit Hilfe des in Kapitel 6 beschriebenen erweiterten Generators, der sich die Funktionalität des Webapplikationsframeworks Ruby on Rails zu nutze macht und dieses um die oben beschriebene Funktionalität erweitert. Mit dem aus dem Zwischenschritt generierten SQL-Script können die Datenbanktabellen generiert werden, auf denen die Webanwendung aufbaut. Durch den Aufruf des Application Generators und der anschließenden Ausführung des erweiterten Generators über den Befehl

```
ruby script/generate extended mydb localhost username password
```

erstellt dieser unter Verwendung seiner integrierten eRB-Templates die MVC-Komponenten des Prototyps. Aufgrund der automatisch generierten Validierungstabelle werden nun auch automatisch die Validierungsmethoden in die Modellklassen geschrieben. Für die genaue Funktionsweise des erweiterten Generators sei auf Kapitel 6 verwiesen. Abbildung 7.9 zeigt den Code der Modellklasse *Member*:

```
class Member < ActiveRecord::Base
  validates_presence_of :vorname
  validates_presence_of :nachname
  validates_presence_of :email
  validates_format_of :email,
    :with =>%r{^[a-zA-Z0-9-]+(\.[a-zA-Z0-9-]+)*@[a-zA-Z0-9-]+\.[a-zA-Z]{2,4}$}
  validates_uniqueness_of :email
  validates_numericality_of :icq
end
```

Abbildung 7.9 Klasse Member mit generierten Validierungsmethoden

Da in Ruby on Rails der WeBRick Web Server integriert ist, lässt sich die eben serstellte Anwendung sofort problemlos testen. Nach einem Aufruf von „*ruby script/server*“ kann man im Browser die URL „*http://localhost:3000/members*“ aufrufen und bekommt die noch leere Liste der Mitglieder zu Gesicht. Durch einen Klick auf „*New member*“ erscheint das Formular zum Anlegen eines neuen Mitgliedes. Lässt man nun alle Eingabefelder leer und bestätigt die Eingabe, so erhält man das Ergebnis, das Abbildung 7.10 zeigt:

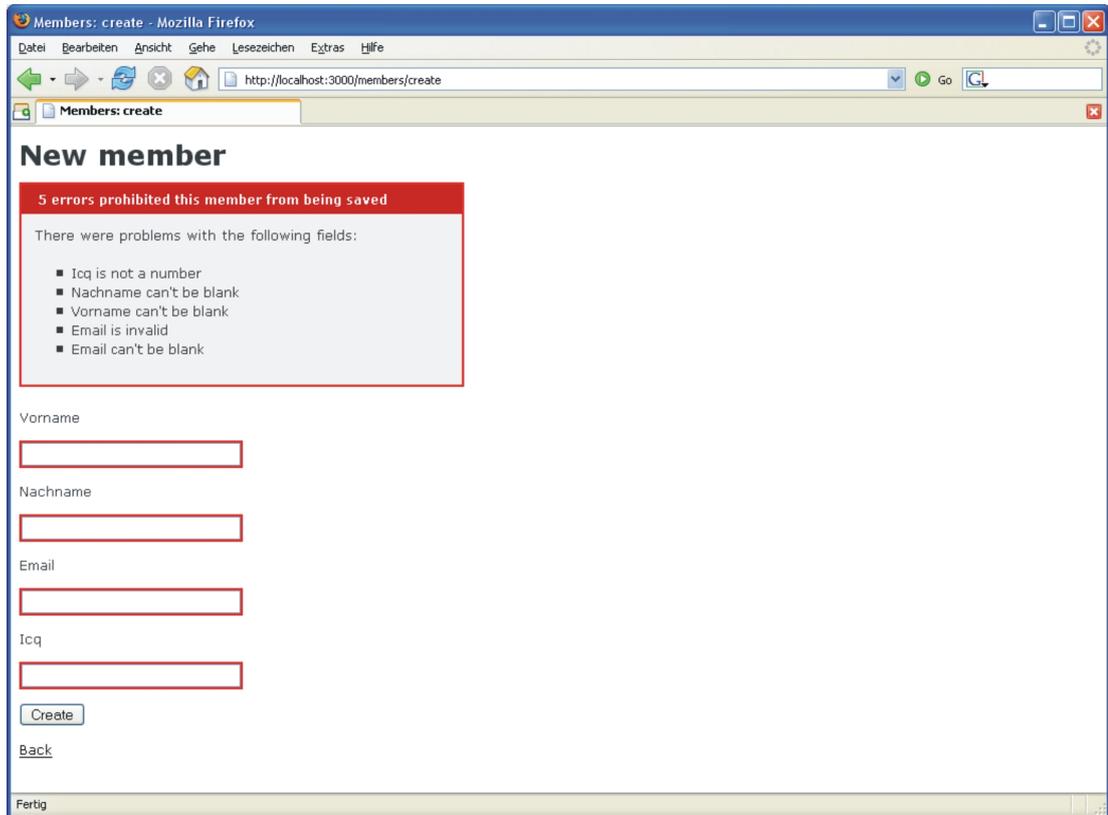


Abbildung 7.10 Fehlerhafte Eingabe der Mitgliederdaten

Erst nach der vollständigen und korrekten Eingabe aller Daten, wird das neue Mitglied angelegt und es erscheint auch sofort in der Liste der Mitglieder (Abbildung 7.11).

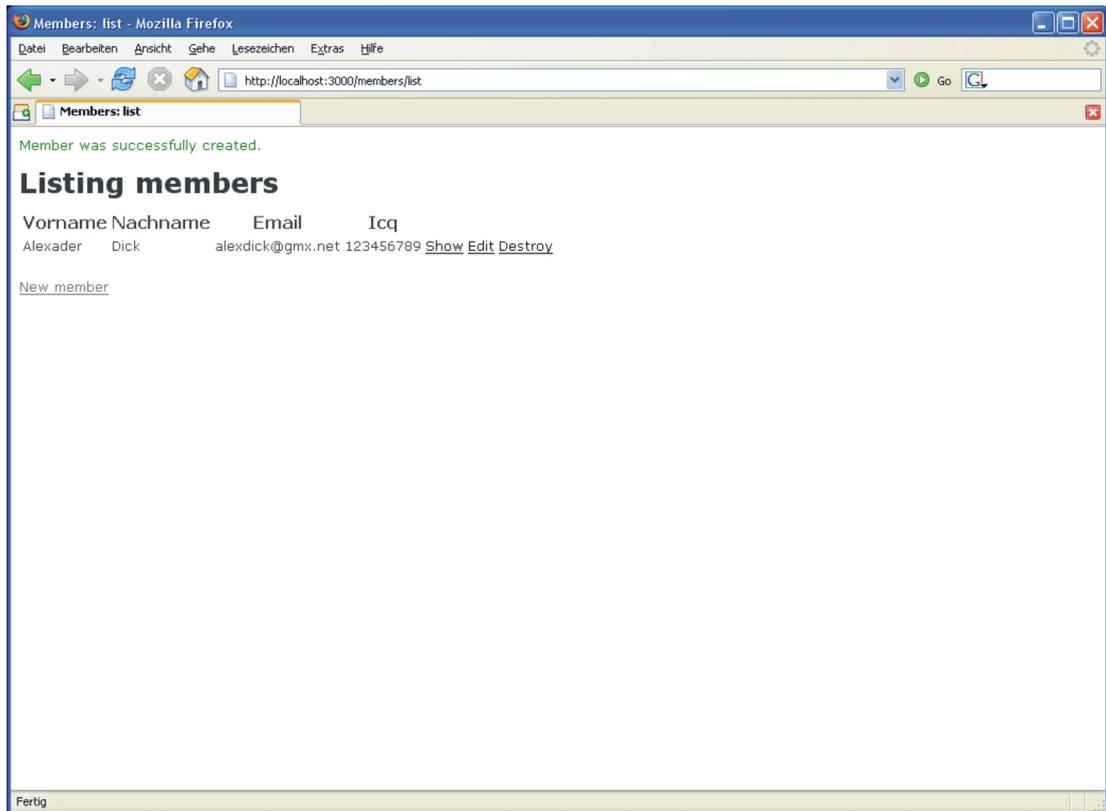


Abbildung 7.11 Liste der Mitglieder nach Eingabe eines neuen Datensatzes

Das Bemerkenswerte an dieser Vorgehensweise ist, dass keine einzige Zeile Code geschrieben wurde, um dieses Ergebnis zu erzielen. Weder wurden Templates für die Codegenerierung entwickelt, noch irgendeine Klasse der Webanwendung erstellt. Die Grundlage bildet einzig und allein das grafische ER-Modell.

Kapitel 8

Zusammenfassung und Ausblick

Zusammenfassung

Durch den ständigen Anstieg der Anforderungen an moderne Webanwendungen wird es immer wichtiger, den Entwicklern Arbeiten, die nicht unmittelbar mit der eigentlichen Problemlösung zusammenhängen aber doch häufig anfallen, abzunehmen. Im Bereich der Webanwendungen etablieren sich immer mehr Frameworks, die zu diesem Zweck geschaffen werden. In dieser Arbeit wurden die zwei Webframeworks Apache Struts und Ruby on Rails miteinander verglichen und einige Unterschiede zwischen den beiden herausgearbeitet. Aufgrund der besonderen Fähigkeit von Ruby on Rails wurde sein integrierter Scaffold-Codegenerator um die Funktionalität der automatischen Validierung von Benutzereingaben erweitert.

Da Rails-Webanwendungen auf einer vorhandenen Datenbank basieren, wurde beschlossen dem Entwickler auch die Arbeit der Erstellung von SQL-Scripts abzunehmen. Und so wurde mit Hilfe des Eclipse Modeling Frameworks ein grafischer Editor für die Modellierung von ER-Diagrammen als logischen Datenbankentwurf entwickelt, wodurch aus der Kombination von Datenmodellierung, Codegenerierung und Prototyping das Framework „*Models on Rails*“ entstand.

Ziel der Erstellung dieses Frameworks war es, dem Entwickler die Möglichkeit zu bieten mit Hilfe eines grafischen Modell-Editors ein ER-Diagramm als Datenentwurf für eine Webanwendung zu erstellen. Das Framework generiert daraus automatisch den Prototyp einer Webanwendung in Ruby on Rails, den der Programmierer für die Weiterentwicklung der Applikation verwenden kann. Um zu diesem Prototyp zu kommen sind nur wenige, einfache Schritte notwendig:

1. Der Anwendungsentwickler erstellt mit Hilfe des erstellten GMF-Modell-Editors ein ER-Diagramm als Basis für die Webanwendung.
2. Mit Hilfe von OpenArchitectureWare und dem in der Xpand-Language erstellten Template wird aus dem ER-Modell automatisch der Code für das SQL-Script generiert, das für die Erstellung der Datenbanktabellen benutzt wird. In diesem Template befinden sich vorgefertigte Validierungsregeln für die spätere Überprüfung von Benutzereingaben.
3. Der nächste Schritt ist die Erstellung der Datenbanktabellen mit Hilfe des automatisch generierten SQL-Scripts und die Erstellung des Prototyps in Rails. Dazu wird zuerst die Struktur einer neuen Rails Applikation mit Hilfe des Application-Generators und dem Befehl `rails webapp` erstellt. Danach kann der Extended Rails-Generator ausgeführt werden und die prototypische Implementierung einer Webanwendung basierend auf den erstellten Datenbanktabellen generieren.

Ausblick

Die Funktionalität, die dieses Framework besitzt, sollte natürlich noch erweitert werden. Ein erster Ansatz wäre die Umsetzung der zweiten Anforderung an den erweiterten Rails-Generator, nämlich die automatische Integration der Beziehungen zwischen den Entitäten aus dem ER-Diagramm in die Modellklassen der Webanwendung und die Anpassung der automatisch generierten Views.

Ein weiterer Punkt für die Verbesserung des Frameworks stellt die Flexibilisierung der verwendenden Validierungsregeln dar. Die Tabellen, die für die Validierung benötigt werden, sind momentan hart codiert in das Xpand-Template integriert. Dadurch muss eine Änderung der Standardvalidierungsregeln in diesem Template vorgenommen werden. Es wäre jedoch von Vorteil, diese Validierungsmethoden auch in den Schritt der Modellierung zu ziehen und somit dem Entwickler eine benutzerfreundliche Anpassungsmöglichkeit zu bieten.

Da die einzelnen Schritte im aktuellen Status des Frameworks manuell angestoßen werden müssen, wäre es sinnvoll diese Vorgänge etwa durch die Erstellung eines ANT-Skripts vollständig zu automatisieren. Die Ausführung dieses Skripts könnte in Folge so in den Modell-Editor integriert werden, dass sich der gesamte Codegenerierungsprozess durch einen einzigen Knopfdruck in Gang setzen lässt.

Anhang

Ruby on Rails – Installationsanweisung

Installation von Ruby

Um Rails verwenden zu können muss natürlich zuerst eine (am besten die neueste) Version von Ruby installiert werden. Die zu diesem Zeitpunkt aktuelle und stabile Version ist Version 1.8.5 und steht zum Download auf [Ruby06] bereit.

Installation von Ruby on Rails

Beim Windowsinstallationspaket von Ruby ist RubyGems inkludiert. Bei RubyGems handelt es sich um einen Package-Manager, mit dessen Hilfe sich Erweiterungen für Ruby einfach herunterladen und installieren lassen, so auch Rails.

Um Rails mit Hilfe von RubyGems zu installieren muss nur in einem Eingabeaufforderungsfenster der Befehl

```
gem install rails --remote
```

einggegeben werden. Die notwendigen Dateien werden automatisch herunter geladen, und das Framework wird installiert. Die derzeit aktuelle Version von Ruby on Rails ist Version 1.1.6.

RubyGems installiert auch alle anderen Bibliotheken, die von Rails benötigt werden. Bei jeder dieser Abhängigkeiten fragt das Installationsprogramm, ob sie installiert werden sollen, was man einfach mit JA (y) bestätigt. Damit ist die Installation im Prinzip schon beendet und Ruby on Rails kann verwendet werden.

Natürlich wird auch eine Datenbank benötigt, um mit dem Framework produktiv arbeiten zu können. Wie in Kapitel 4 erwähnt werden von Rails sehr viele Datenbanken unterstützt. Die Applikationen in dieser Arbeit verwenden das freierhältliche Datenbanksystem MySQL (Hinweise zur Installation sind auf [Mysq06] nachzulesen).

Inhalt der beiliegenden CD-ROM

GMF-Model-Editor

Vorliegend als Eclipse-Projekt in den Verzeichnissen

CD-ROM\model-editor\er2,

CD-ROM\model-editor\er2.edit,

CD-ROM\model-editor\er2.editor,

CD-ROM\model-editor\er2.diagramm

Xpand-Template und generiertes SQL-Script

Vorliegend als Eclipse-openArchitectureWare-Projekt im Verzeichnis

CD-ROM\sql.generator

Extended Rails-Generator

Vorliegend im Verzeichnis

CD-ROM\rails\generators\extended

Text-Datei mit Installationshinweisen

Vorliegend im Verzeichnis

CD-ROM\ReadMe.txt

PDF-Dateien der Diplomarbeit und des Posters

Vorliegend im Verzeichnis

CD-ROM\Diplomarbeit.pdf

CD-ROM\Poster.pdf

Abkürzungsverzeichnis

API	Application Programming Interface
CGI.....	Common Gateway Interface
DAO	Data Access Object
DBMS	Datenbankmanagementsystem
DRY-Prinzip	Don't Repeat Yourself
EJB	Enterprise Java Bean
EMF.....	Eclipse Modeling Framework
ER-Model.....	Entity-Relationship-Model
ERb.....	Embedded Ruby
FTP	File Transfer Protocol
GEF	Eclipse Graphical Editing Framework
GMF	Eclipse Graphical Modeling Framework
HTML	Hypertext Markup Language
HTTP.....	Hypertext Transfer Protocol
JDBC.....	Java Database Connectivity
JDO	Java Data Object
JSP.....	JavaServer Pages
MDA	Model Driven Architecture
MVC.....	Model-View-Controller
oAW	OpenArchitectureWare
ORM.....	objekt-relationales Mapping
RDBMS.....	Relationales Datenbankmanagementsystem
RPC	Remote Procedure Call
RUP	Rational Unified Process
SOAP.....	Simple Object Access Protocol
SQL	Structured Query Language
UML.....	Unified Modeling Language
W3C	World Wide Web Consortium
WWW.....	World Wide Web
XML.....	Extensible Markup Language
XP.....	Extreme Programming

Abbildungsverzeichnis

Abbildung 2.1 Schichtenarchitekturen.....	17
Abbildung 3.1 Das Model View Controller-Entwurfsmuster [Fowl03]	23
Abbildung 3.2 Model 1-Ansatz für Java-Webanwendungen [Weße05]	24
Abbildung 3.3 Das Model 2 für Java Webanwendungen [Weße05].....	24
Abbildung 3.4 Front Controller-Entwurfsmuster [Alur03].....	26
Abbildung 3.5 Application Controller-Entwurfsmuster [Alur03]	26
Abbildung 3.6 Active Record-Entwurfsmuster [Fowl03].....	27
Abbildung 3.7 Data Access Object-Entwurfsmuster [Alur03]	28
Abbildung 4.1 Hauptkomponenten einer Struts-Applikation [Bond03]	33
Abbildung 4.2 Ruby on Rails – Web Applications (geändert) [Rail06d]	45
Abbildung 5.1 Standard-Startseite von Ruby on Rails	59
Abbildung 5.2 Klasse TestController	60
Abbildung 5.3 Rails: Datenbank-Konfigurationsdatei database.yml.....	61
Abbildung 5.4 Klasse MembersController	65
Abbildung 6.1 Klasse Member	68
Abbildung 6.2 Klasse Product.....	71
Abbildung 6.3 Klasse Product mit Validierungsmethode validates_format_of.....	72
Abbildung 6.4 Klasse Product mit Validierungsmethode validates_numericality_of	72
Abbildung 6.5 Klasse Product mit Validierungsmethode validates_presence_of.....	73
Abbildung 6.6 Klasse Product mit Validierungsmethode validates_uniqueness_of	73
Abbildung 6.7 1:1-Beziehung in Rails [Thom06].....	75
Abbildung 6.8 1:n-Beziehung in Rails [Thom06].....	75
Abbildung 6.9 m:n-Beziehung in Rails [Thom06]	76
Abbildung 6.10 Klassendefinition von ExtendedGenerator	77
Abbildung 6.11 Methode ExtendedGenerator::banner()	78
Abbildung 6.12 Mapping der Validierungstabellen.....	79
Abbildung 6.13 Methode ExtendedGenerator::update_file()	80
Abbildung 6.14 Methode ExtendedGenerator::get_data_from_db() – Abschnitt 1 ..	83
Abbildung 6.15 Methode ExtendedGenerator::get_data_from_db() – Abschnitt 2 ..	83
Abbildung 6.16 Methode ExtendedGenerator::read_file()	84

Abbildung 6.17 Methode ExtendedGenerator::write_file(insert_lines)	85
Abbildung 7.1 Die Funktionsweise von Models on Rails	87
Abbildung 7.2 Das ER-Metamodell.....	88
Abbildung 7.3 Gegenüberstellung EMF-Baumeditor – GMF-Modell-Editor	90
Abbildung 7.4 Xpand-Template – Abschnitt 1	92
Abbildung 7.5 Xpand-Template – Abschnitt 2	93
Abbildung 7.6 Workflow Engine	94
Abbildung 7.7 Automatisch generiertes SQL-Script – Abschnitt 1.....	95
Abbildung 7.8 Automatisch generiertes SQL-Script – Abschnitt 2.....	96
Abbildung 7.9 Klasse Member mit generierten Validierungsmethoden.....	97
Abbildung 7.10 Fehlerhafte Eingabe der Mitgliederdaten.....	97
Abbildung 7.11 Liste der Mitglieder nach Eingabe eines neuen Datensatzes	98

Tabellenverzeichnis

Tabelle 4.1 Gegenüberstellung Ruby on Rails – Struts	49
Tabelle 5.1 Rails: Modellssoziationen	63

Literaturverzeichnis

- [Alex77] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, Shlomo Angel: „A Pattern Language“, Oxford University Press, New York, USA, 1977.
- [Alex79] Christopher Alexander: „The Timeless Way Of Building“, Oxford University Press, New York, USA, 1979.
- [Agil01] The Agile Alliance: „Manifesto for Agile Software Development“, <http://www.agilemanifesto.org> (2006-09-25), 2001.
- [Agil06] The Agile Alliance: <http://www.agilealliance.org> (2006-09-25), 2006.
- [Alur03] Deepak Alur, John Crupi, Dan Malks: „Core J2EE Patterns Best Practices and Design Strategies“, Second Edititon, Prentice Hall, Palo Alto, USA, 2003.
- [Andr06] AndromDA, <http://www.andromda.org> (2006-10-20), 2006.
- [Anas01] Michalis Anastopoulos, Tim Romberg: „Referenzarchitekturen für Web-Applikationen - Finale Version“, Projekt Application2Web, Forschungszentrum Informatik (FZI), <http://app2web.fzi.de> (2006-09-25), 2001.
- [Apac06] The Apache Software Foundation: „Struts Framework“, <http://struts.apache.org> (2006-09-25), 2006.
- [Balz96] Helmut Balzert: „Lehrbuch der Softwaretechnik – Software Entwicklung“, Spektrum Akademischer Verlag, Hedelberg, Deutschland, 1996.
- [Balz98] Helmut Balzert: „Lehrbuch der Softwaretechnik – Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung“, Spektrum Akademischer Verlag, Hedelberg, Deutschland, 1998.
- [Beck04] Kent Beck: „Extreme Programming Explained – Embrace Change“, 2.Auflage, Addison-Wesley, USA, 2004.
- [Bond03] Martin Bond, Debbie Law: „Tomcat Kickstart“, Sams Publishing, USA, 2003.
- [Cava04] Chuck Cavaness: „Programming Jakarta Struts“, Second Edition, O'Reilly Media, Inc., Sebastopol, USA, 2004.
- [Cekv04] Vic Cekvenich, Wolfgang Gehner: „Struts - Best Practices“, 1. Auflage, dpunkt.verlag, Heidelberg, Deutschland, 2004.
- [Ceri03] Stefano Ceri, Piero Fraternali, Aldo Bongio, Marco Brambilla, Sara Comai, Maristella Matera: „Designing Data-Intensive Web Applications“, Morgan Kaufmann Publishers, San Francisco, USA, 2003.

- [Chen76] Peter Pin-Shan Chen: „The Entity Relationship Model – Toward a Unified View of Data“, ACM Transactions on Database Systems, Vol.1, No.1, March 1976, pp. 9-36.
- [Dalg06] Mark Dalgarno: „Code Generation Network“, <http://www.codegeneration.net> (2006-09-12), 2006.
- [Efft06a] Sven Effttinge, Markus Volter, Arno Haase Bernd Kolb: “The Pragmatic Code Generator Programmer”, <http://www.theserverside.com/tt/articles/article.tss?l=PragmaticGen> (2006-10-16), 2006.
- [Efft06b] Sven Effttinge, Clemens Kadura: „Xpand Language Reference“, http://www.eclipse.org/gmt/oaw/doc/4.1/r20_xPandReference.pdf (2006-10-16), 2006
- [Elma02] Ramez Elmasir, Shamkant B. Navante: „Grundlagen von Datenbanksystemen“, 3. Auflage, Pearson Studium, München, Deutschland, 2002
- [EMF06] Eclipse Modeling Framework, <http://www.eclipse.org/emf> (2006-10-16), 2006.
- [Fowl03] Martin Fowler: „Patterns of Enterprise Application Architecture“, Addison Wesley, Boston, USA, 2003.
- [Fowl97] Martin Fowler: „Analysis Patterns: Reusable Object Models“, Addison Wesley, USA, 1997.
- [Frau06] Fraunhofer ISIE: „Virtuelles Software Engineering Kompetenzzentrum“, <http://www.software-kompetenz.de> (2006-09-12), 2006.
- [Froe98] Garry Froehlich, H. James Hoover, Ling Liu, Paul Sorenson: „Designing object-oriented frameworks“, in CRC Handbook of Object Technology, CRC Press, 1998, pp. 25-1 - 25-22.
- [Gamm95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: „Design Patterns – Elements of Reusable Object-Oriented Software“, Addison Wesley, USA, 1995.
- [Gini01] Athula Ginige, San Murugesan: „Web Engineering: An Introduction“, IEEE Multimedia 8 (1), January–March 2001, pp. 14-18.
- [GEF06] Eclipse Graphical Editing Framework, <http://www.eclipse.org/gef> (2006-10-16), 2006.
- [GMF06] Eclipse Graphical Modeling Framework, <http://www.eclipse.org/gmf> (2006-10-16), 2006.
- [Hein06a] David Heinemeier Hansson: „Ruby on Rails“, <http://www.rubyonrails.org> (2006-09-13), 2006.
- [Hein06b] David Heinemeier Hansson: „Rails Framework Dokumentation“, <http://api.rubyonrails.org> (2006-09-14), 2006.
- [Hibe06] Hibernate: „Relational Persistence for Java and .NET“, <http://www.hibernate.org> (2006-09-28), 2006.

- [Hunt03] Andrew Hunt, Dave Thomas: „Der Pragmatische Programmierer“, Carl Hanser Verlag, München, Deutschland 2003.
- [Jabl04] Stefan Jablonski, Ilia Petrov, Christian Meiler, Udo Mayer: „Guide To Web Application and Platform Architectures“, Springer-Verlag, Heidelberg, Deutschland, 2004.
- [Jaco99] Ivar Jacobson, Grady Booch, James Rumbaugh „The Unified Software Development Process“, Addison-Wesley, USA, 1999.
- [Kapp04] Gerti Kappel, Birgit Pröll, Siegfried Reich, Werner Retschitzegger: „Web Engineering – Systematische Entwicklung von Web-Anwendungen“, dpunkt.verlag, Heidelberg, Deutschland, 2004.
- [Mari06] Martin Marinschek, Wolfgang Radinger: „Ruby on Rails – Einstieg in die effiziente Webentwicklung“, dpunkt.verlag, Heidelberg, Deutschland, 2006.
- [Mysq06] MySQL Website, „MySQL: The World's Most Popular Open Source Database“, <http://www.mysql.org> (2006-10-05), 2006.
- [Omg06] Object Management Group, „OMG's MetaObject Facility“, <http://www.omg.org/mof> (2006-10-16), 2006.
- [Qcod06] Qcodo – PHP Development Framework, <http://www.qcodo.com> (2006-10-20), 2006.
- [Rail06a] Rails Users: „Ruby on Rails – Howtos“, <http://wiki.rubyonrails.org/rails>, (2006-09-14), 2006.
- [Rail06b] Rails Users: „Ruby on Rails – Understanding Generators“, <http://wiki.rubyonrails.org/rails/pages/UnderstandingGenerators> (2006-09-14), 2006.
- [Rail06c] Rails Users: „Ruby on Rails – How To Extend Scaffolding“, <http://wiki.rubyonrails.com/rails/pages/HowToExtendScaffolding> (2006-09-14), 2006.
- [Rail06d] Rails Users: „Ruby on Rails – UnderstandingRailsMVC“, <http://wiki.rubyonrails.com/rails/pages/UnderstandingRailsMVC> (2006-10-02), 2006.
- [Röhr02] Armin Röhr, Stefan Schmiedl, Clemens Wyss: „Programmieren mit Ruby – Eine praxisorientierte Einführung“, 1. Auflage, dpunkt.Verlag, Heidelberg, Deutschland, 2002.
- [Rose03] Nikolaus Rosenmayer: „Evaluierung von Opensource Software als Entwicklungsbasis für ein Web-basiertes, generisches Anmeldesystem“, Diplomarbeit Universität Linz, Linz, Österreich, 2003.
- [Ruby06] Ruby Community: „Ruby Programming Language“, <http://www.ruby-lang.org> (2006-09-13), 2006.
- [Shav04] Sherry Shavor, Jim D'Anjou, Scott Fairbrother, Dan Kehn, John Kellerman, Pat McCarthy, „Eclipse – Anwendungen und Plug-Ins mit Java entwickeln“, Addison-Wesley Verlag, München, Deutschland, 2004.

- [Sun03] Sun Microsystems, Inc.: „Java Data Objects (JDO)“, <http://java.sun.com/products/jdo> (2006-09-28), 2003.
- [Sun06a] Sun Microsystems, Inc.: „Enterprise JavaBeans Technology (EJB)“, <http://java.sun.com/products/ejb> (2006-09-28), 2006.
- [Sun06b] Sun Microsystems, Inc.: „Java Technology“, <http://java.sun.com> (2006-09-28), 2006.
- [Sun06c] Sun Microsystems, Inc.: „JavaServer Pages Technology (JSP)“, <http://java.sun.com/products/jsp> (2006-09-28), 2006.
- [Sun06d] Sun Microsystems, Inc.: „Java Database Connectivity (JDBC)“, <http://java.sun.com/javase/technologies/database.jsp>, (2006-10-05), 2006.
- [Thom04] Dave Thomas, Chad Fowler, Andrew Hunt: „Programming Ruby – The Pragmatic Programmer’s Guide“, Second Edition, Pragmatic Bookshelf, USA, 2004.
- [Thom06] Dave Thomas, David Heinemeier Hansson: „Agile Webentwicklung mit Rails“, Carl Hanser Verlag, München, Deutschland 2006.
- [W3C06] W3C World Wide Web Consortium: „SOAP Specification“, <http://www.w3.org/TR/soap> (2006-10-02), 2006.
- [Weße05] Matthias Weßendorf: „Struts - Websites effizient entwickeln“, W3L GmbH Herdecke, Bochum, Deutschland, 2005.
- [Wird05] Ralf Wirdemann, Thomas Baustert: „Einführung in die Webentwicklung mit Ruby on Rails“, Javamagazin 10/2005, pp. 92-99.
- [Wird06] Ralf Wirdemann, Thomas Baustert: „Ruby on Rails – Alternative zur Web-Entwicklung mit Java?“, JavaSpektrum 4/2006, pp. 20-24.
- [Wiki06a] Wikipedia - Die Freie Enzyklopädie: „Agile Softwareentwicklung“, <http://de.wikipedia.org> (2006-09-25), 2006.
- [Wiki06b] Wikipedia - Die Freie Enzyklopädie: „Framework“, <http://de.wikipedia.org/wiki/Framework> (2006-09-28), 2006.
- [Wimm05] Manuel Wimmer: „Model Driven Architecture in der Praxis Evaluierung aktueller Entwicklungswerkzeuge und Fallstudie“, Diplomarbeit, TU Wien, Wien, Österreich, 2005.
- [XML99] XML-RPC Homepage, <http://www.xmlrpc.com> (2006-10-03), 1999.
- [Zuse01] Zuser Wolfgang, Biffel Stefan, Grechenig Thomas, Köhle Monika: „Software-Engineering mit UML und dem Unified Prozess“, Pearson Studium, München, Deutschland, 2001.