JOHANNES KEPLER
UNIVERSITÄT LINZ
Netzwerk für Forschung, Lehre und Praxis

Ruby on Rails with Roles

Ein verteiltes Rollenmodell für RESTful Webservices

DIPLOMARBEIT

zur Erlangung des akademischen Grades
„Magister der Sozial- und Wirtschaftswissenschaften“
(Mag.rer.soc.oec.)
im Diplomstudium Wirtschaftsinformatik

Eingereicht am
Institut für Wirtschaftsinformatik –
Data & Knowledge Engineering

Eingereicht von
Klaus Ettmayer

Begutachter
o. Univ.-Prof. Dr. Michael Schrefl

Mitbetreuung
Mag. Bernd Neumayr

Linz, im Jänner 2010

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit mit dem Titel „Ruby on Rails with Roles – Ein verteiltes Rollenmodell für RESTful Webservices“ 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.

Linz, im Jänner 2010 _____

(Klaus Ettmayer)

Danksagung

An dieser Stelle möchte ich mich herzlich bei all jenen bedanken, die mich bei der Entstehung dieser Diplomarbeit unterstützt haben.

Insbesondere Herrn o. Univ.-Prof. Dr. Michael Schrefl und Herrn Mag. Bernd Neumayr möchte ich für ihre fachliche Unterstützung, und zahlreichen Ideen aus gemeinsamen Gesprächen danken. Herrn Mag. Bernd Neumayr für die Aufopferung zahlreicher Stunden und seine engagierten Betreuung die wesentlich zu dieser Arbeit beigetragen haben.

Weiters möchte ich mich bei meiner Familie bedanken, die mich moralisch unterstützte und durch ihre motivierenden Worte die Entstehung dieser Arbeit förderten. Hier möchte ich mich besonders meine Eltern hervorheben, die mir auch durch ihre finanzielle Unterstützung das Studium ermöglichten und mir in allen Belangen behilflich waren.

Kurzfassung

In objektorientierten Systemen bilden Rollenmodelle wichtige Konstrukte zur Modellierung bzw. Programmierung. In Bezug auf das World Wide Web wurden Rollenkonzepten jedoch noch wenig Aufmerksamkeit gewidmet. Doch genau dort finden sich Unmengen an Informationen, welche anhand von Rollenmodellen besser strukturiert werden können.

Rollenmodelle müssen für den Einsatz in dezentralen Informationssystemen, wie dem World Wide Web, zusätzlichen Anforderungen genügen. Rollen sollen nicht nur durch Spezialisierung, sondern auch durch Generalisierung erzeugt werden können. Weiters sollen Rollen unabhängig existieren können und eine spätere Eingliederung in eine Rollenhierarchie ermöglichen. Auch die Navigation innerhalb einer Rollenhierarchie muss bestmöglich unterstützt werden, wodurch Einschränkungen durch Zugriffsbeschränkungen minimiert werden sollen. Eine weitere Anforderung besteht in einer einheitlichen Kommunikation innerhalb eines Informationssystems, wobei URIs zur Identifikation und XML zur Repräsentation von Rollen bestens geeignet sind.

Die Hauptaufgabe dieser Diplomarbeit besteht in der Erstellung eines Erweiterungs-Packages für das Web Application Frameworks Ruby on Rails, welches die Verwendung von Rollenklassen und –instanzen innerhalb einer Web-Applikation erlaubt. Ruby on Rails verwendet den Model-View-Controller Architekturstil (MVC) und bietet einen so genannten scaffold-Generator an, mit dessen Hilfe das Model, die View, der Controller und die Datenbanktabelle einer Klasse halbautomatisch generiert werden. Das im Rahmen dieser Diplomarbeit entwickelte Package beinhaltet einen solchen scaffold-Generator für Rollenklassen für Ruby on Rails. Die Erstellung dieses erweiterten Packages erfolgt schrittweise. Zu Beginn wird ein aus der Literatur ausgewähltes Rollenkonzept in der Programmiersprache Ruby implementiert, in Schritt zwei wird das Konzept und die Implementierung um die Anforderungen für den dezentralen Einsatz erweitert. Dieses erweiterte Rollenkonzept samt Implementierung wird schließlich in Schritt 3 an die Spezifika von Ruby on Rails angepasst und in dieses Framework integriert.

Abstract

Roles are an important construct in conceptual modelling and object-oriented systsms. Concerning the World Wide Web role concepts did not attract much attention so far. However, especially in the World Wide Web there is a plethora of information, which could be better structured and be modelled more naturally if a role construct was available.

In decentralized information systems, like the World Wide Web, role models have to fulfil additional requirements: Roles can be generated by generalization as well as by specialization, roles have to exist independently and can be integrated to a role hierarchy at any time, the navigation within a role hierarchy has to be supported to minimize access restrictions. A further requirement includes an uniform communication within an information system. Therefore URIs are best qualified for identification and XML for representation of roles.

The main contribution of this thesis is an extension package for the web application framework Ruby on Rails that allows to work with role classes and instances within a web application. Ruby on Rails uses the model-view-controller architectural pattern (MVC) and provides for semi-automatic generation of model, view, controller and database table of a class by using a so-called scaffold generator. The role-extension package described in this thesis comes with a scaffold generator for role classes that intuitively integrates with the existing application framework. This extension package is developed in three steps. In the first step a prominent role concept will be implemented in the programming language Ruby. In the second step this implementation will be enhanced by requirements for its decentralized use, as described in the previous paragraph. In step three Ruby on Rails will be extended accordingly.

Inhaltsverzeichnis

Abkürzungen

bzw.	beziehungsweise
bzgl.	bezüglich
CRUD	Create, Read, Update, Delete
d.h.	das heißt
ERb	Embedded Ruby
HTML	Hypertext Markup Language
PHP	PHP: Hypertext Preprocessor
REST	Representational State Transfer
RDF	Resource Description Framework
SQL	Structured Query Language
u.a.	unter anderem/n
UML	Unified Modelling Language
URI	Uniform Resource Identifier
usw.	und so weiter
u.v.m.	und viele mehr
vgl.	vergleiche
vs.	versus
XML	Extensible Markup Language
z.B.	zum Beispiel

Abbildungsverzeichnis

Listingverzeichnis

1 Einleitung

Rollen sind in der realen Welt von großer Bedeutung. Nicht nur Menschen, auch andere Dinge können Rollen einnehmen. Das Einnehmen von Rollen ist dabei sehr flexibel. Personen, sowie auch andere Dinge, können ihre Rollen ständig wechseln und zu einem Zeitpunkt mehrere Rollen inne haben. In der objektorientierten Modellierung bzw. Programmierung wird versucht die Welt so realitätsgetreu wie möglich abzubilden, wodurch ein Rollenmodell bereits zu einem wichtigen Konstrukt in der Objektorientierung avancierte. In der Literatur finden sich daher viele Ansätze zu Rollenmodellen und zu derem Einsatz in objektorientierten Systemen.

Im World Wide Web liegen Unmengen von Informationen die anhand eines Rollenmodells strukturiert dargestellt bzw. neue Informationen gewonnen werden können. Das World Wide Web dient als Informationsplattform und bietet Informationen in Form von Dokumenten an, welche für den Menschen gut lesbar sind. Das World Wide Web ist ein dezentrales Informationssystem indem unzählige Quellen Informationen anbieten. Ein Rollenmodell soll dabei Informationen von verschiedenen Quellen verwenden können, wofür eine Kommunikation zwischen diesen Quellen erfolgen muss. Um eine Kommunikation zwischen verschiedenen Quellen im World Wide Web zu gewährleisten, muss eine gemeinsame Sprache gesprochen werden um Informationen auszutauschen. Fielding [9] beschreibt den Architekturstil REST nachdem dezentrale Informationssysteme Informationen einfach austauschen können. Dieser Architekturstil enthält Regeln, die auf die Besinnung der grundlegenden Ideen des Datenaustausches über das World Wide Web zurückführen, wodurch ein einfacher Informationstransfer, ohne zusätzliche Funktionalitäten, wie z.B. Sessions, gewährleistet wird. Durch den REST-Architekturstil können Ressourcen über URIs angesprochen werden, wodurch der Zugriff auf Informationen über einfache HTTP-Methoden erfolgt.
Für die Anforderung bzw. Darstellung von Informationen dienen dem Benutzer[1] bzw. weiteren Maschinen Web Services. Das Web Application Framework Ruby on Rails bietet die Möglichkeit Web Services in der objektorientierten Programmiersprache Ruby zu erstellen, wobei ab der Version 1.2 standardmäßig RESTful Webapplikationen generiert werden.

[1] Da in der deutschen Sprache die Verwendung des Maskulinums beide Geschlechter gleichermaßen mit einbezieht, wird in dieser Arbeit auf die Konstruktion der „geschlechterneutralen Formulierung“ verzichtet.

1.1 Rollen

Rollen bezeichnen in der Soziologie einen Zustand, den ein Objekt einnehmen kann. Mit Rollen gehen auch Erwartungen und Verhaltensweisen einher. Rollen haben in der realen Welt eine wichtige Bedeutung. Sie sind allgegenwärtig. So nimmt beispielsweise eine Person im Laufe ihres Lebens viele Rollen ein. Ob als Mitglied einer Familie oder im Arbeitsleben, wo eine Person eine bestimmte Position inne hat. Rollen repräsentieren also Personen und je nachdem welche Rolle gerade eingenommen wird kann sich das Auftreten bzw. Verhalten ändern. Nicht nur Personen, sondern auch andere Objekte können Rollen einnehmen. Ein Unternehmen bietet ein bestimmtes Produkt seinen Kunden an, während dieses Unternehmen Rohstoffe bei seinen Lieferanten kauft. Das Unternehmen fungiert somit einerseits als Verkäufer und andererseits als Käufer.

Objektorientierte Systeme haben das Ziel die Wirklichkeit so direkt als möglich abzubilden. Deswegen bestehen in der Literatur unzählige Arbeiten, die sich damit beschäftigen eine bestmögliche Abbildung von Rollen zu schaffen. In Kapitel 2 wird auf einige, solcher Rollenkonzepte näher eingegangen. Darunter befindet sich auch das Rollenkonzept von Gottlob et al. [11], welches für diese Diplomarbeit von großer Bedeutung ist, weshalb es auch detailliert beschrieben wird.

1.2 Dezentrale Informationssysteme

Ein Informationssystem ist nach Roithmayr [31] ein in sich grundsätzlich geschlossenes System, indem unterschiedliche Elemente Informationen beinhalten und miteinander in Beziehung stehen, sodass zwischen den Elementen eine Kommunikation stattfinden kann. Innerhalb eines Informationssystems werden Informationen erfasst, gespeichert, verarbeitet, analysiert, gewartet, ausgetauscht und auch produziert. Der Zweck eines Informationssystems besteht darin Informationen zur Verfügung zu stellen. Wenn Elemente eines Informationssystems auf unterschiedlichen Rechnern liegen, also verteilt sind, und keine zentrale Steuerungseinheit vorliegt, spricht man von einem dezentralen Informationssystem. Ein derartiges dezentrales Informationssystem stellt das World Wide Web dar, auf das im folgenden Abschnitt näher eingegangen wird. Elemente des World Wide Web sind unterschiedliche Systeme, die miteinander kommunizieren können. Wenn Web-Elemente, wie Web Services, die Richtlinien des REST-

Architekturstils einhalten, spricht man von RESTful Web Services. Diese Richtlinien von REST werden in Abschnitt 1.2.2. beschrieben.

1.2.1 Das Web

Das World Wide Web, kurz Web oder WWW, ist grundsätzlich ein über das Internet verfügbares dezentrales Informationssystem. Die bereitgestellten Informationen stellen dabei Dokumente dar, welche mittels Links mit anderen Dokumenten verknüpft sein können, wodurch eine nahezu unüberschaubare Verkettung von Dokumenten bestehen kann. Die Grundidee lag für Berners-Lee – er gilt als Gründer des World Wide Web – darin (vgl. Berners-Lee und Cailliau [5]), Informationen auszutauschen, wobei dem anfragenden Client egal ist, wo und wie die gewünschte Information gespeichert ist und welches System für die Verwaltung der Information verantwortlich ist. Dem Client interessiert nur die Information, welche in einem oder mehreren miteinander verlinkten Dokumenten repräsentiert wird. Das World Wide Web selbst stellt somit einfach ausgedrückt eine Unzahl von miteinander verlinkten Dokumenten dar. Für die Nutzung des Webs ist ein Webbrowser nötig, welche die von den Webserver(n) zur Verfügung gestellten Dokumente anfordert und anzeigt. Ein Webserver stellt dabei die Informationen von ein oder mehreren Informationssystemen zur Verfügung, die dezentral verteilt sein können.

1.2.2 RESTful Web Services

Representational **S**tate **T**ransfer, kurz REST, ist ein Architekturstil für dezentrale Informationssysteme. Geprägt wurde dieser Begriff von Thomas Roy Fielding [9] in seiner Dissertation im Jahr 2000. Er empfiehlt, dass jede Ressource anhand einer eigenen URI identifiziert werden soll. Dadurch soll ermöglicht werden, dass sämtliche Daten mittels HTTP gelesen, aber auch verändert werden können ohne zusätzliche Verfahren (wie z.B. Cookies) mit einzubeziehen, was die Kommunikation erleichtern soll. Der Zugriff auf die Ressourcen erfolgt über die HTTP-Methoden GET, PUT, POST und DELETE, mit denen nach Bayer [3] grundsätzlich alle Anwendungsfälle abgedeckt werden können.

Wenn ein Client über eine URI eine Ressource abruft, erhält er vom Server ein HTML-Dokument, welches die Repräsentation dieser Ressource, inklusive deren

Ressourcenzustand, darstellt. Diese Repräsentation kann auch Verweise auf weitere Ressourcen enthalten. Ein solches Netzwerk aus Ressourcen bildet einen virtuellen Zustandsautomaten, wobei eine Zustandsveränderung durch die Verwendung der Webapplikation durch einen User erfolgt. Sobald der User einen Verweis verfolgt oder eine Dateneingabe über ein Formular durchführt, hat diese Aktion eine Überleitung in den nächsten Zustand einer Webapplikation zur Folge. Die Zustandsveränderung erfolgt durch die Übermittlung der Repräsentation dieses neuen Zustandes zum Client (vgl. Fielding und Taylor [10]). Daraus leitet sich der Begriff Representational State Transfer ab. Dem Server ist dabei der Anwendungszustand des Clients egal. Sämtliche Informationen die der Server zur Interpretation benötigt sind in der übertragenen Nachricht enthalten.

Sobald ein Dienst im Web auf dem REST-Architekturstil aufbaut, wird dieser als RESTful bezeichnet. Im Web befinden sich bereits zahlreiche Web Services, die auf dem REST-Architekturstil aufbauen. So sind beispielsweise die APIs von Amazon, eBay oder myVideo als RESTful verfügbar.

In diesem Abschnitt wurden somit schon auf die Prinzipien einer RESTful Webanwendung eingegangen. Dabei handelt es sich um die Prinzipien (vgl. Bayer [3]):

- Adressierbarkeit:
  Jede Ressource verfügt über eine eigene URI
- Zustandslosigkeit:
  Für den Server spielt der Status des Clients keine Rolle
- Wohldefinierte Operationen:
  Der Zugriff auf Ressourcen erfolgt über HTTP-Methoden
- Verwendung von Hypermedia:
  Zur Repräsentation von Ressourcen werden Hypermedia-Formate, wie beispielsweise HTML oder XML verwendet.

Einen großen Vorteil, den man durch REST erhält, ist, dass die Kommunikation mit anderen Systemen ohne weiteren Aufwand möglich ist. Durch die Identifikation der Ressourcen über eigene URIs werden nach Bayer [3] grundsätzlich keine Web Services angeboten sondern nur die Ressourcen. Somit kann eine Webanwendung auch einfach auf andere Ressourcen verweisen. Auch die Kombination mehrere Webanwendungen ist

durch die Zustandlosigkeit gegeben. Zwischen Servern müssen daher keine Zusatzinformationen über den derzeitigen Status transferiert werden. Es reicht lediglich eine Übermittlung eines HTTP-Requests zu dem anderen Server.

1.3 Motivation/Problembeschreibung

Rollen haben sich in objektorientierten Programmiersprachen bzw. in objektorientierten Modellierungen bewährt, da diese oft eine adäquatere Abbildung der Wirklichkeit ermöglichen, als es ohne Rollen möglich wäre. In der Literatur finden sich unzählige Konzepte über Rollen, wobei in dieser Diplomarbeit ein Rollenkonzept herangezogen werden soll, welches für den Einsatz in dezentralen Informationssystemen gerüstet werden soll. Dezentrale Informationssysteme spielen schon seit geraumer Zeit eine wichtige Rolle. Einerseits gewinnen, durch die immer stärker ansteigenden Datenmengen, Informationssysteme mit verteilten Quellen an Bedeutung. Andererseits tendieren Informationssysteme immer mehr zur Dezentralisierung. Auch das World Wide Web stellt ein riesiges dezentrales Informationssystem dar, indem Informationen ausgetauscht, bearbeitet und generiert werden. Das World Wide Web eignet sich auch hervorragend, um ein Rollenkonzept einer breiten Masse zur Verfügung zu stellen. Zur Bereitstellung eines Rollenkonzeptes eignen sich Web Services, die weit verbreitete Regeln einhalten. Der REST-Architekturstil beschreibt solche Regeln, die bereits in vielen Web Services inkludiert sind. Ein Vorteil solcher RESTful Web Services liegt in deren Einfachheit zur Kommunikation von Informationen. Diese Diplomarbeit hat demnach zum Ziel ein bestehendes Framework zur Erstellung von Webapplikationen zu erweitern, wodurch die Verwendung von Rollen im Web erleichtert wird.

Um Rollen in dezentralen Informationssystemen zu verwenden, muss ein Rollenmodell einigen Anforderungen entsprechen. Innerhalb einer Rollenhierarchie muss die Navigation zwischen Rollen gegeben sein. Wurden auf eine Rolle Zugriffsbeschränkungen durch ein System gesetzt, sollte diese Rolle trotz allem in der Rollenhierarchie eingebunden werden können. Wenn auf eine Rolle beispielsweise nur lesend zugegriffen werden darf, muss mindestens eine Beziehung zu dieser Rolle erstellt werden – unabhängig von ihrer Position in der Rollenhierarchie.
Weiters muss in einem dezentralen Informationssystem die Möglichkeit bestehen Teilhierarchien zu einer gemeinsamen Rollenhierarchie zu bündeln. Dafür ist eine

separate Existenz der Teilhierarchien auf unterschiedlichen Quellen erforderlich. Die Verbindung solcher Teilhierarchien bzw. einzelner Rollen muss auch auf unterschiedliche Arten gegeben sein. Einerseits kann eine Rollenhierarchie anhand des Top-Down Prinzips erstellt werden – hierbei handelt es sich um eine Spezialisierung einer Rolle – andererseits kann eine Rollenhierarchie durch das Bottom-Up Prinzip erzeugt werden. Hier wird eine Rolle aus einer oder mehrerer Rollen erstellt – dies entspricht einer Generalisierung.

Werden Rollen innerhalb eines dezentrallen Informationssystems erzeugt, kann nicht davon ausgegangen werden, dass ihre Entwickler Absprachen bzgl. Namensgebungen für Attribute und Methoden getroffen haben. Es können demnach gleiche Attribute unterschiedliche Namen (Synonyme) bzw. verschiedene Attribute dieselben Namen (Homonyme) aufweisen. Hier muss ein Mapping verfolgt werden, um eine korrekte Verwendung des Rollenmodels zu gewährleisten.

Viele, jedoch nicht alle Attribute sollen innerhalb einer Rollenhierarchie auf Instanzebene weitervererbt werden können. Attribute müssen demnach gekennzeichnet werden um deren Weitervererbung zu erlauben bzw. zu verhindern.

Eine wichtige Anforderung besteht auch in der Identifikation von Rollen. Ihnen muss ein universeller Identifikator zugewiesen werden, um eine eindeutige Identifizierung innerhalb des dezentralen Informationssystems zu ermöglichen. Auch deren Repräsentation soll im gesamten Informationssystem verwertbar sein.

1.4 Beispiel

In diesem Kapitel wird ein einfaches Beispiel beschrieben, auf welches im Zuge dieser Diplomarbeit mehrmals verwiesen wird. Zunächst wird ein fiktives Szenario präsentiert und anschließend dessen Repräsentation als Rollenhierarchien, sowohl auf Schema- als auch auf Instanzebene vorgestellt.

Mr. Black ist eine Person die am 11.7.1971 geboren wurde und seine private Telefonnummer lautet 533-4842. Er studiert an der Universität Harward und erhält ein Stipendium in Höhe von 600. Seine Telefonnummer am Campus ist 483-3840. Weiters ist Mr. Black an einem Unternehmen als Angestellter beschäftigt. Dabei verdient er 900. In seinem Büro ist er über die Telefonnummer 342-8764 erreichbar. In diesem Unternehmen leitet er die Verkaufsabteilung und verdient dafür zusätzlich 650.

Außerdem ist Mr. Black in diesem Unternehmen der Leiter des Projektes 1. Diese Projektleitung verdankt er seiner Erfahrung, welche für das Projekt wichtig ist.

Für dieses Beispiel wird eine Rollenhierarchie auf Klassenebene benötigt, welche in Abbildung 1 dargestellt wird. Hierbei ist neben dem Strukturaufbau auch ersichtlich, welche Attribute für die jeweiligen Klassen bzw. Rollen benötigt werden.

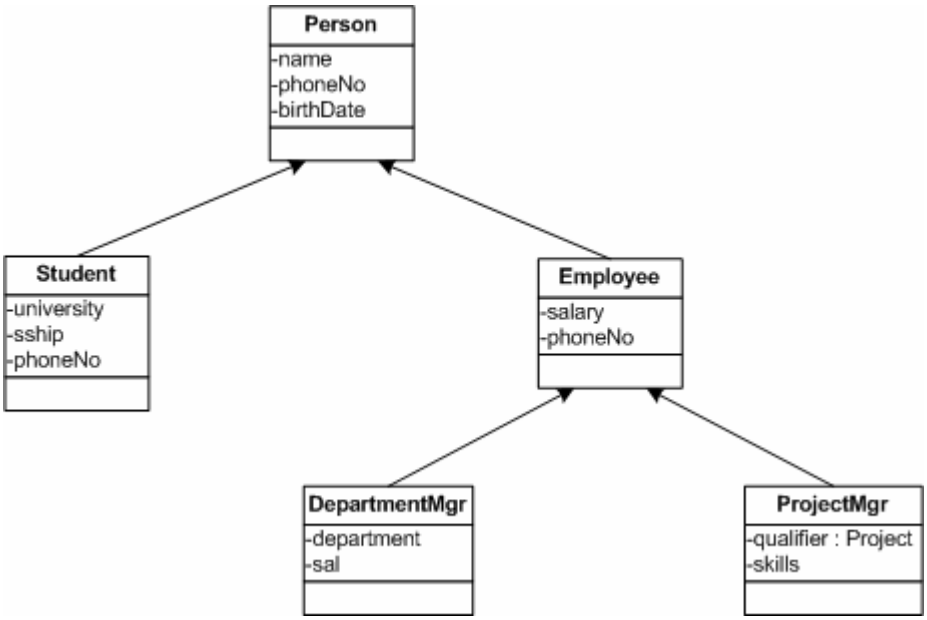

Abbildung 1: Rollenhierarchie auf Klassenebene – Beispiel

In Abbildung 2 werden die Instanzen für das Beispiel bildlich dargestellt. Sie repräsentieren eine einfache Rollenhierarchie aufbauend auf der der Klassenebene.

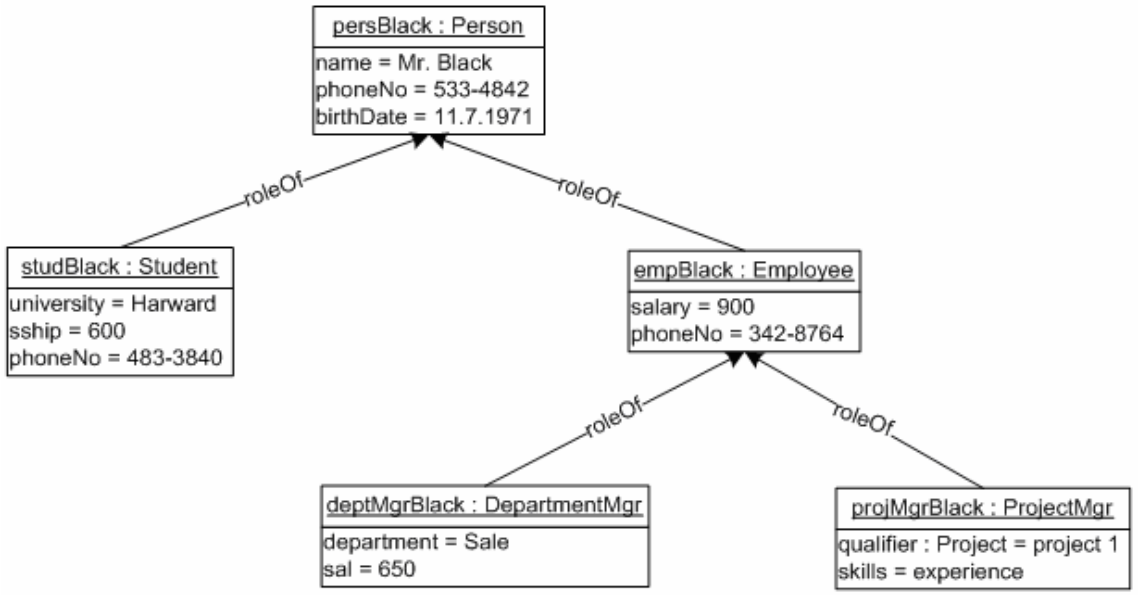

Abbildung 2: Rollenhierarchie auf Instanzebene – Beispiel

In diesen Abbildungen werden einige Eigenschaften, wie sämtliche Methoden, vernachlässigt. Der Grund dafür liegt einerseits in einer überschaubareren Darstellung und andererseits um zu diesem Zeitpunkt keine Verwirrung wegen neuer Elemente zu erzeugen.

Im Folgenden wird dieses Beispiel, in Bezug auf die objektorientierte Modellierung, nun näher erklärt:

`persBlack` ist eine Instanz der Klasse Person und repräsentiert als Wurzelknoten dieser Rollenhierarchie das Objekt aus der Wirklichkeit. Dieses wird als Entität bezeichnet. Diese Person verfügt über die Attribute `name`, `phoneNo` und `birthDate`, welche auf die Werte „Mr. Black“, 533-4842“ bzw. „11.7.1971“ gesetzt sind. Weiters nimmt die Person `persBlack` mehrere Rollen ein. In direkter Beziehung werden dabei die Rollen eines Studenten (`studBlack`) und eines Angestellten (`empBlack`) eingenommen. Indirekt verfügt die Person auch über Rollen eines `DepartmentMgrs` (`deptMgrBlack`) und eines `ProjectMgrs` (`projMgrBlack`), wobei diese Rollen über die `Employee`-Rolle eingenommen werden.

Sämtliche Knoten in der Rollenhierarchie verfügen über mehrere Attribute, wobei manche Knoten das gleiche Attribut, aber mit einer anderen Ausprägung, aufweisen (z.B. `phoneNo`). Weiters bestehen auch Attribute, welche das gleiche aussagen jedoch anders benannt wurden. So gibt z.B. das Attribut `salary` des Rollentyps Employee ebenso Auskunft über das Gehalt wie das Attribut `sal` vom Rollentyp `DepartmentManager`.

1.5 Vorgehensweise/Aufbau

Der Aufbau dieser Diplomarbeit spiegelt auch die Entwicklung der implementierten Lösungen wider. In Kapitel 2 werden, neben Rollenkonzepte, die sich in der Literatur finden, die objektorientierte Programmiersprache Ruby und das Web Application Framework Ruby on Rails vorgestellt. Bei den Rollenkonzepten wird im Speziellen auf Rollenkonzepte für objektorientierte Systeme eingegangen. Im Detail wird in diesem Kapitel das Rollenkonzept von Gottlob et al. [11] erläutert, da dies das Rollenkonzept ist, auf welches die Implementierungen dieser Diplomarbeit aufbauen. In Kapitel 3 wird die Verwendung von Rollen in Ruby beschrieben und deren Umsetzung – aufbauend auf dem Rollenmodell von Gottlob et al. [11] – detailliert vorgestellt. Im Anschluss wird diese Implementierung mit Lösungen für die Programmiersprachen Smalltalk (von Gottlob et al. [11]) und Java (von Schrefl und Thalhammer [35]) verglichen.

Da die Motivation dieser Diplomarbeit darin besteht, ein Rollenkonzept als Webapplikation zur Verfügung zu stellen, müssen für den Einsatz in dezentralen Informationssystemen Anpassungen an der Umsetzung vorgenommen werden. In Kapitel 4 wird, aufbauend auf der Implementierung von Rollen in Ruby, ein erweitertes Rollenmodell entwickelt, welches den Anforderungen an einen dezentralen Einsatz genügen. Anschließend wird die Verwendung des erweiterten Rollenmodells in Ruby beschrieben und weiters wiederum ein detaillierter Einblick in die Implementierung des erweiterten Rollenmodells gewährt.

Damit nun dieses erweiterte Rollenkonzept als Webapplikation angeboten werden kann, wird dieses in einen Generator von Ruby on Rails, einem Web Application Framework, umgewandelt. In Kapitel 5 werden Anpassungen an das, in Kapitel 4 erstellte, erweiterte Rollenmodells vorgenommen. Anschließend erfolgt die Implementierung eines

Generators, welcher eine Webapplikation generiert, die die Grundstruktur zur Verwendung des erweiterten Rollenkonzeptes aufweist. Dieser Generator und der aus ihm entstehende Rails-Code wird ebenso in diesem Kapitel erläutert, wie die Einschränkungen, die dieser Generator mit sich bringt.

2 State of the Art

In diesem Kapitel werden rollenbasierte Ansätze aus der Literatur beschrieben. Dabei werden ausschließlich Ansätze aus der objektorientierten Welt dargestellt. Insbesondere auf das Rollenkonzept von Gottlob et al. [11] wird näher eingegangen, da es für diese Diplomarbeit als Ausgangspunkt dient. Anschließend wird die objektorientierte Programmiersprache Ruby und wichtige Aspekte dieser Programmiersprache beschrieben. Weiters wird das Web Application Framework Ruby on Rails näher erläutert.

2.1 Rollenbasierte Ansätze

In der Literatur finden sich unzählige Rollenkonzepten zu den unterschiedlichsten Systemen. In diesem Kapitel werden unterschiedliche Ansätze aus der Literatur präsentiert, wie Rollenmodelle in objektorientierten Systemen umgesetzt werden können. Insbesondere wird auf das Rollenkonzept von Gottlob et al. [11] näher eingegangen. Wobei in diesem Abschnitt eine Gegenüberstellung von Klassen- und Rollenhierarchien erfolgt und danach eine beispielhafte Erläuterung des Rollenkonzeptes beschrieben wird. Den Abschluss dieses Teilabschnittes bildet die Erklärung weiterer Elemente, die für die Erstellung einer Rollenhierarchie benötigt werden und zur Verwendung des Rollenkonzeptes von Nöten sind.

2.1.1 Rollenkonzept nach Gottlob et al.

Das Rollenkonzept, auf dem diese Diplomarbeit basiert, wurde von Gottlob et al. [11] entwickelt, wonach Rollen über folgende charakteristische Eigenschaften verfügen:

- Unterschiedliche Rollen können gemeinsame Eigenschaften teilen (z.B. Die Rollen Employee und Student erben beide die Attribute der Klasse Person).
- Rolleninstanzen können dynamisch erstellt und gelöscht werden.
- Rollen können voneinander unabhängig erstellt und gelöscht werden, wobei auf den Ansatz der Mehrfachvererbung verzichtet werden kann.
- Rollenspezifische Eigenschaften werden in der Rollenhierarchie weitergegeben.
- Eigenschaften einer Rolle sind nicht für Rollen eines anderen Hierarchiezweiges verfügbar (z.B. kann ein Student nicht auf das Einkommen eines Employees zugreifen).

- Für ein Wurzelobjekt können mehrere Instanzen eines Rollentyps bestehen. Diese Rollentypen werden als qualifizierte Rollen bezeichnet.

Doch worin bestehen nun die gravierenden Unterschiede zu einer Klassenhierarchie eines objektorientierten Systems? Diese werden im nächsten Abschnitt näher erläutert.

2.1.1.1 Rollenhierarchie vs. Klassenhierarchie

Im Gegensatz zu einer Klassenhierarchie, in der verschiedene Objekte klassifiziert werden, beschreibt eine Rollenhierarchie auf Instanzebene nur ein Objekt in dessen unterschiedlichen Kontexten.

Ein weiterer Unterschied besteht bei der Vererbung. Während in einer Klassenhierarchie sämtliches Verhalten vom Vorgänger an den Nachfolger auf Schemaebene vererbt wird, erbt eine Rolle in einer Rollenhierarchie die Attribute und Methoden der übergeordneten Rolle nur auf Instanzebene.

Ein Rollenkonzept sollte aber auch die wichtigsten Konzepte eines objektorientierten Systems beinhalten. Im Folgenden werden diese Konzepte aufgelistet und deren Inkludierung in das Rollenkonzept von Gottlob et al. [11] aufgezeigt:

- Klassifizierung:
  Jede Objektklasse kann als Wurzelklasse einer Rollenhierarchie angesehen werden. Wobei die Struktur und das Verhalten der Rollenhierarchie für alle Instanzen der Objektklasse gleich ist. Weiters beschreibt ein Rollentyp die Struktur und das Verhalten aller Instanzen dieses Rollentyps.

- Objektidentifizierung:
  Da jede Entität auch durch mehrere Rollen repräsentiert werden kann, muss näher auf die Identifizierung eingegangen werden. Systemintern wird jede Entität, sowie jede Rolleninstanz durch eine Objekt-ID (OID) identifiziert. Für den Anwender kann eine Entität anhand der Wurzelinstanz einer Rollenhierarchie identifiziert werden. Eine Rolleninstanz kann mithilfe des Wurzelobjektes identifiziert werden, welches die gesuchte Rolle einnimmt. Wenn also überprüft werden soll, ob zwei Rolleninstanzen die selbe Entität

repräsentieren, muss lediglich die Gleichheit der Wurzelobjekte kontrolliert werden.

- Spezialisierung und Vererbung:
  Eine Rolleninstanz entspricht einer Spezialisierung einer Entität. Somit soll die Rolleninstanz auch die Attribute und Methoden dieser Entität erben.

- Polymorphismus:
  Polymorphismus bedeutet, dass ein Bezeichner, je nach Kontext, unterschiedliche Datentypen annimmt und einen dementsprechenden Wert zurückliefern kann. In der Rollenhierarchie muss es möglich sein, je nach Rollentyp den korrekten Wert der jeweiligen Instanz zu ermitteln.

Zum besseren Verständnis wird nun dieses Rollenkonzept mittels einer beispielhaften Erläuterung untermauert, welche in Abbildung 3 näher erläutert wird. Diese Abbildung zeigt anhand eines Beispiels einen möglichen Aufbau einer Rollenhierarchie, wobei sämtliche Arten von Rollentypen inkludiert sind.

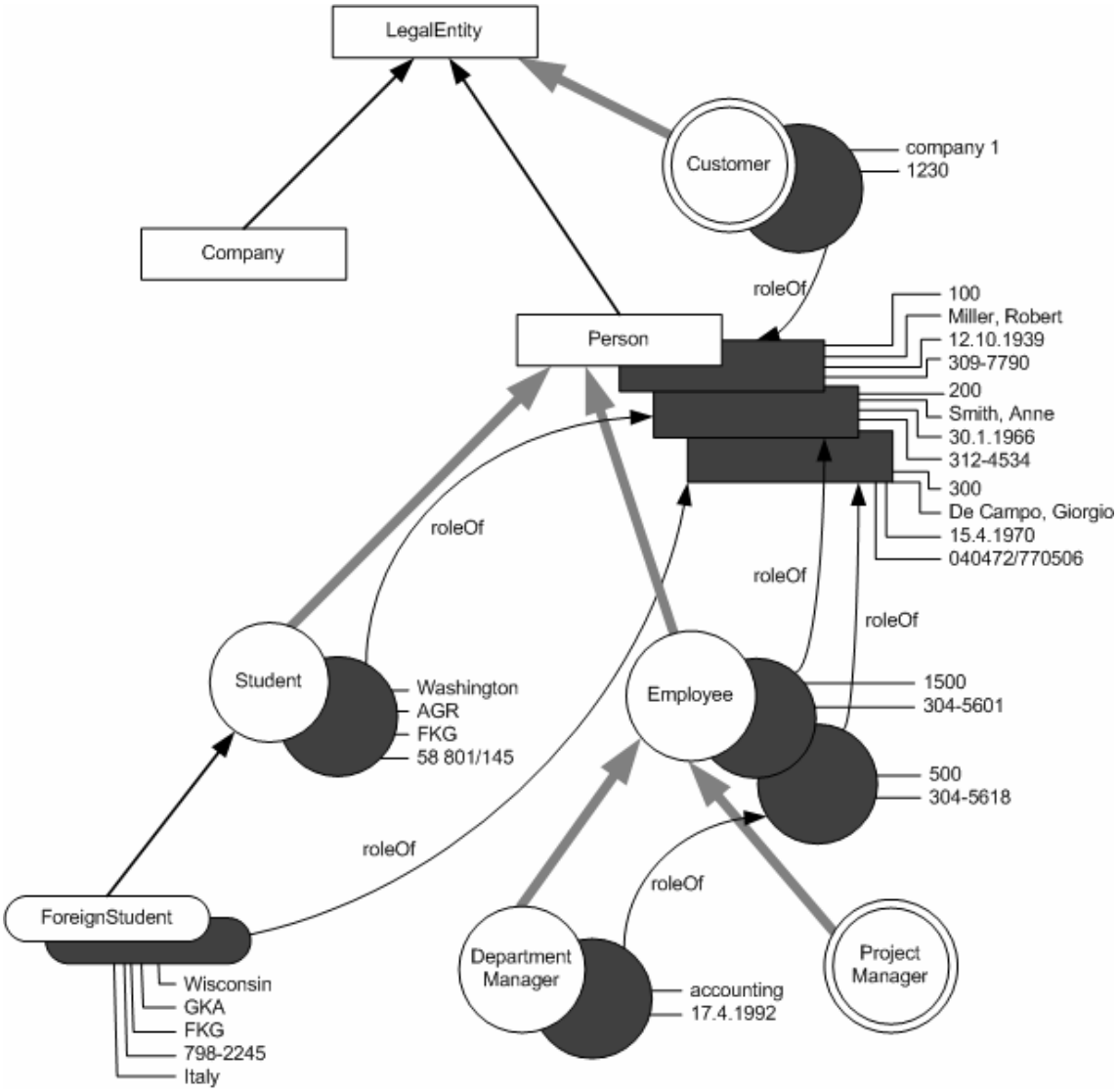

Abbildung 3: Beispiel einer Rollenhierarchie

Im oberen Bereich der Grafik sind drei Klassen von Objekten aus der realen Welt abgebildet. Diese werden als Rechtecke dargestellt. Aufgrund ihrer Verknüpfung über einen dünnen, schwarzen Pfeil – dieser stellt eine Vererbungsbeziehung dar – bilden diese eine in objektorientierten Systemen übliche Klassenhierarchie. In diesem Fall erben also die Klassen Company und Person sämtliche Eigenschaften der Klasse LegalEntity. Die Klasse Person selbst bildet wiederum den Wurzelknoten einer weiteren Hierarchie. Diese wird als Rollenhierarchie bezeichnet, da alle darunter liegenden Knoten Rollentypen repräsentieren. Einfache Rollentypen werden dabei als Kreise dargestellt, wohingegen qualifizierte Rollen als doppelte Kreise abgebildet werden (z.B. ProjectManager). Die ovale Darstellungsform, welche beispielsweise den ForeignStudent repräsentiert, ist eine spezielle Art einer Rolle. Dieser Rollentyp erbt – wie eine Subklasse von derer Superklasse – sämtliche Eigenschaften von der

übergeordneten Rolle (Student). Hier wird also eine Vererbung auf Schemaebene dargestellt.

Die Beziehungen zwischen den Rollentypen bzw. der Wurzelklasse (dicker, grauer Pfeil) wird als roleSuperType bezeichnet und identifiziert den übergeordneten Rollentyp. Hierbei besteht keine Vererbungsbeziehung.

Wird nun die Instanzebene dieser Hierarchie betrachtet, ist ersichtlich, dass die einzelnen Rolleninstanzen lediglich über eine roleOf-Beziehung miteinander verbunden sind (dünner, grauer Pfeil). So ist beispielsweise erkennbar, dass die erste Instanz des Rollentyps Employee ein Rollenobjekt der Personeninstanz Anne Smith repräsentiert. Möchte man nun die Telefonnummer von Anne Smith ermitteln, muss zuerst geklärt werden, ob man die Nummer der Person selbst erhalten möchte – also sozusagen die private Telefonnummer – oder ob man die Nummer der Angestellten Anne Smith benötigt, welche in der Instanz des Rollentyps Employee hinterlegt wurde. Diese entspreche der geschäftlichen Telefonnummer.

Weiters nimmt die Person Anne Smith eine Rolle des Typs Student ein. Dies wird durch die Rolleninstanz von Student mit deren roleOf-Beziehung zur Person Anne Smith dargestellt.

Eine besondere Rolle nimmt die Person Giorgio De Campo ein. Er verfügt über eine Rolleninstanz des Typs ForeignStudent, wobei dieser Rollentyp sämtliche Eigenschaften von Student erbt und zusätzlich das Attribut country bereitstellt. Zu den vererbten Eigenschaften zählt auch der roleSuperType, wodurch auch der Rollentyp ForeignStudent als übergeordnete Klasse die Person aufweist. Somit bezieht sich die roleOf-Beziehung einer Instanz von ForeignStudent, wie auch einer Instanz vom Rollentyp Student, auf eine Personeninstanz. Mit anderen Worten ist ForeignStudent lediglich eine Spezialisierung von Student, was für die Rollenhierarchie grundsätzlich keinen Unterschied macht.

Auch die Person Giorgio De Campo verfügt über eine Instanz des Rollentyps Employee und weiters auch eine des Rollentyps DepartmentManager, welche einen Sub-Rollentyp von Employee darstellt. Auf derselben Hierarchieebene wie der Rollentyp DepartmentManager befindet sich auch der qualifizierte Rollentyp ProjectManager. Würde die Person Giorgio De Campo – oder besser gesagt die Instanz der Employee-Rolle – einen solchen Rollentyp einnehmen, müsste zusätzlich zu den Attributen eines ProjectManagers auch ein qualifier gesetzt werden, der für eine qualifizierte Rolle zur eindeutigen Identifikation von Nöten ist. In der Abbildung 3 ist jedoch eine weitere

qualifizierte Rolle, und zwar der Rollentyp Customer, dargestellt. Dieser Rollentyp gehört zur Rollenhierarchie der Klasse LegalEntity, wobei die roleOf-Beziehung auf eine Instanz der Klasse Person zeigt. Dies ist möglich, da eine Person eine Spezialisierung von LegalEntity darstellt und somit auch ein LegalEntity ist. Somit nimmt die Person Robert Miller die Rolle Customer ein. Der qualifier der Customer-Instanz ist dabei company 1, wodurch der Customer Robert Miller einen Kunden von company 1 repräsentiert. Der qualifier würde üblicherweise durch eine Beziehung zu einer Instanz der Klasse Company aufgezeigt, was aber zu Gunsten einer besseren Überschaubarkeit vernachlässigt wurde.

2.1.1.2 Weitere Elemente einer Rollenhierarchie

Weitere Elemente, welche in einer Rollenhierarchie implementiert sind, aber nicht in der Abbildung 3 dargestellt werden, sind einerseits das Rollenverzeichnis `roleDictionary`, über welches jede Wurzelknoteninstanz verfügt, und andererseits die Rollenmethoden, auf die später eingegangen wird. Das Rollenverzeichnis ist eine Liste sämtlicher Rollentypen, die ein Wurzelobjekt einnimmt. Sie unterstützt das schnelle Auffinden der eingenommenen Rollen einer Entität.
Sobald eine neue Rolleninstanz angelegt wird, wird diese Liste um den Rollentyp der neuen Instanz erweitert. Soll hingegen eine Rolleninstanz eliminiert werden, wird auch deren Eintrag im Rollenverzeichnis gelöscht. Besitzt diese zu löschende Rolleninstanz jedoch Sub-Rollen, werden auch diese aus der Liste und aus der Rollenhierarchie entfernt.

Die Rollenmethoden sind hingegen Methoden, über welche alle Rollentypen in einer Rollenhierarchie verfügen müssen. Bei diesen Methoden handelt es sich um folgende:

- `root`:
  Diese Methode ermittelt das Wurzelelement einer Rollenhierarchie.
- `roleOf`:
  Dadurch wird das übergeordnete Rollenobjekt zurückgeliefert.
- `as: aRoleType`:
  Mithilfe dieser Methode wird auf das Rollenobjekt des Typs `aRoleType` verwiesen, falls ein derartiges in der Rollenhierarchie besteht. Diese Methode dient demnach zum Rollenwechsel

- `existsAs: aRoleType`:
  Anhand von `existsAs` wird überprüft, ob ein Rollenobjekt des Typs `aRoleType` in der Rollenhierarchie besteht und liefert den entsprechenden booleschen Wert zurück.
- `entityEquiv: anObject`:
  Diese Methode überprüft, ob das jeweilige Rollenobjekt und `anObject` derselben Entität angehören, sprich ob sie dem selben Wurzelobjekt unterliegen.
- `abandon`:
  `abandon` löscht ein Rollenobjekt aus der Rollenhierarchie. Diese Methode ist jedoch nicht für das Wurzelobjekt verfügbar.

Weiters zählen zu diesen Methoden auch die der qualifizierten Rollen. Also den folgenden Methoden:

- `as: aQualifiedRoleType of: qualifyingObj`:
  Hiermit wird auf das Rollenobjekt des Typs `aQualifiedRoleType` verwiesen, welches auch durch das `qualifyingObj` qualifiziert ist. Der Rückgabewert ist, wie bei der `as`-Methode, das gefundene Rollenobjekt falls dieses vorhanden ist.
- `existsAs: aQualifiedRoleType of: qualifyingObj`:
  Mit dieser Methode wird überprüft, ob ein Rollenobjekt des Typs `aQualifiedRoleType` existiert, welcher mithilfe des `qualifyingObj` identifiziert werden kann. Zurückgegeben wird hier wiederum ein boolescher Wert.
- `qualifier`:
  `qualifier` gibt jenes Objekt zurück, mithilfe dessen die qualifizierte Rolle identifiziert wird. Diese Methode steht nur qualifizierten Rollen zur Verfügung.

2.1.2 Weitere rollenbasierte Ansätze

In der Literatur finden sich zahlreiche weitere Ansätze, die sich mit der konzeptionellen Umsetzung von Rollen in unterschiedlichen Systemen beschäftigen. In diesem Unterkapitel werden speziell Konzepte betrachtet, die über Rollenmodelle in objektorientierten Systemen handeln. In objektorientierten Systemen nehmen Objekte Rollen ein, wodurch ein Objekt Rollenverhalten übernimmt. Dabei verfügen Rollen über

folgende Eigenschaften, welche alle von Zhu und Zhou [42] betrachteten rollenbasierten Ansätze vorweisen:

- Rollen stellen eine Sichtweise auf ein Objekt dar.
- Rollen können unabhängig von einander erstellt und gelöscht werden.
- Rollen können durch Generalisierung oder Spezialisierung erzeugt werden, wobei sich Rollen eines Objektes eine allgemeine Struktur und ein Verhalten teilen können.
- Eine Rolle bezieht sich auf ein Objekt, wobei ein Objekt aber mehrere Rollen gleichzeitig einnehmen kann.
- Rollen zeigen, wie Entitäten miteinander interagieren können.
- Rollen sind dynamisch. Sie können jederzeit eingenommen aber auch abgeben werden.
- Der Status eines Objektes ist von seinen Rollen abhängig.

All diese aufgezählten Eigenschaften von Rollen treffen auch für Rollen der nachfolgenden Rollenkonzepte zu. Optionale Eigenschaften, bei denen sich die Expertenmeinungen unterscheiden sind folgende:

- Ein Objekt und deren Rollen haben unterschiedliche Identitäten oder eine gemeinsame.
- Rollen können oder können nicht weitere Rollen einnehmen.
- Der Zugriff auf ein Objekt ist von seinen Rollen abhängig oder nicht.
- Eine Rolle hat entweder eine allgemeine Bedeutung oder nur in der Rollenhierarchie.
- Rollen haben oder haben kein eigenes Verhalten.

Zho und Zhou [42] unterscheiden Rollenkonzepte nach ihrer Einbindung in ein objektorientiertes System. Rollenkonzepte werden einerseits als Evolution von Objekten gesehen, andererseits als Teil eines Modellierungskonzeptes implementiert. Weiters werden Rollen als Interface von Objekten betrachtet oder sie werden zur Interessensgruppierung von Objekten verwendet.

Werden Rollen als Evolution von Objekten gesehen, können Objekte dabei in den Status einer Rolle wechseln. Rollen stellen somit eine Grundstruktur, wie eine Klasse, dar, die ein Objekt einnimmt, sobald es eine Rolle repräsentiert. Reimer [27] konstruiert eine

Datenstruktur und Albano et al. [1] eine objektorientierte Programmiersprache, in denen Objektinstanzen in Instanzen von Rollen umgewandelt bzw. kopiert werden. Ein Kritikpunkt dieser Lösungsansätze liegt in der Veränderung der Identität eines Objektes, sobald es eine Rolle einnimmt.

In anderen Arbeiten werden Rollenkonzepte als Teil eines Modellierungskonzeptes vorgestellt. Brachman und Schmolze [6] implementieren in ihrem Wissensrepräsentationssystem KL-ONE eine Rolle als Komponente. Eine Rolle entspricht dabei einer Beziehung zwischen zwei Objekten. Kristensen [15] inkludiert sein Rollenkonzept in objektorientierten Systemen anhand mehrerer Variablen und Methoden die einem Objekt zugewiesen werden, sobald es eine Rolle einnimmt.
Steimann [36] gibt eine Übersicht über verschiedene Ansätze zu Rollen, die er in folgende Gruppen einteilt:

- Rollen als benannte Enden von Beziehungen:
  Dies ist die einfachste Form der Rollenrepräsentation. In vielen Modellen werden nach Steimann [36] bereits Beziehungen zwischen Objekten, wie z.B. „is-parent-of“ in Rollennamen umgeformt (z.B. „parent“). Dadurch ist erkennbar, dass ein Objekt in Bezug zu einem anderen die Rolle „Parent“ einnimmt.
- Als Spezialisierung oder als Generalisierung:
  In dieser Kategorie fallen Ansätze in denen Rollentypen als Subklassen (z.B. „Employee“) oder als Superklassen (z.B. „Person“) des Entitässtyps modelliert werden.
- Rollen als an Objekte angehängte Hilfsinstanzen:
  Bei diesen Ansätzen werden Rollen als eigene Instanzen repräsentiert, die ihre eigene Strukur und Verhalten haben (z.B. Rolleninstanz „played-by“ Klasseninstanz).

In all diesen Fällen ist eine Rolle vollständig vom jeweiligen Objekt abhängig und muss dadurch über keine eigene Identität verfügen.
Manche Experten betrachten Rollen als ein Interface von Objekten, das vorgibt welche Eigenschaften eine Rolle vorweisen muss, wobei die Implementierung dieser Eigenschaften beim Objekt selbst liegt. Genilloud und Wegmann [12] beschreiben Rollen als abstrahiertes Verhalten eines Objektes, bei denen das Verhalten einer Rolle

von dem Verhalten anderer Rollen abhängig sein kann. Es besteht also ein allgemeines Verhalten über alle eingenommenen Rollen.

Pernici [25] erzeugt zusätzlich zu einer normalen Klasse eine eigenständige Rollenklasse deren Instanzen neben einer eigenen Struktur und einem eigenen Verhalten auch eine eigene ID aufweisen. Objekte, welche Rollen einnehmen, verweisen demnach auf diese Rolleninstanzen. Je nach Struktur und Verhalten der eingenommenen Rollen können Objekte gruppiert werden. Diese Sichtweise von Rollen hat jedoch nach Zho und Zhou [42] in der Praxis wenig Anhang gefunden. Demsky und Rinard [8] bzw. Kuncak et al. [18] entwickelten ein System, indem Rollen zur Beschreibung einer Beziehung zwischen zwei Klassen (z.B. „Employee“ und „Person“) dienen. Jede Rollenbeschreibung beinhaltet sämtliche Informationen für jedes Objekt, welches in dieser Beziehung involviert ist. Weiters beinhaltet ihr System eine Analyse-Funktion, welches Rollenbeziehungen bzw. -beschreibungen analysiert, wodurch eine Gruppierung nach Interessen möglich ist.

Im Gegensatz zu Rollen in objektorientierten Systemen wurden Rollen in dezentralen Informationssystemen, insbesondere dem World Wide Web, bislang nur wenig Beachtung geschenkt. Rossi et al. [32] analysieren den Einsatz von Rollen im Web Engineering. Sie zeigen, dass die Stärken von Rollen (z.B. Flexibilität) auch in diesem Bereich hilfreich sind und Vorteile bei der Modellierung mit sich bringen.
Auch Huemer [13] beschäftigt sich mit dem Einsatz von Rollen in dezentralen Informationssystemen. Er entwickelt ein RDF-basiertes Rollenmodell für das Web of Data und analysiert dafür Anforderungen für den dezentralen Einsatz.

2.2 Die Programmiersprache Ruby

Ruby ist eine objektorientierte, plattformunabhängige, interpretierbare Programmiersprache. Sie wurde 1993 von Yukihiro Matsumoto entworfen. Eine erste Version erschien 1995 in Japan. Erst im Jahr 2000 erschienen englischsprachige Dokumentationen und Bibliotheksbeschreibungen, wodurch Ruby immer mehr Akzeptanz in der westlichen Welt fand. Den Durchbruch erreichte Ruby jedoch erst im Jahre 2006, indem „aktive Usergroups“ entstanden und „Konferenzen rund um Ruby“ ausverkauft waren (vgl. über Ruby [39]). Im November 2009 vermeldet das wichtigste

Forum zu Ruby[2] rund 200 Einträge pro Tag (vgl. über Ruby [39]) und erreicht in der Liste der populärsten Programmiersprachen nach dem TIOBE Community Index [38] bereits den zehnten Rang. Ruby ist als Open Source frei verfügbar, was auch zur Verbreitung beigetragen hat.

Die Grundidee hinter Ruby lag für Matsumoto in der Schaffung einer Programmiersprache, welche die Vorteile vieler anderer Sprachen inkludiert. Laut Matsumoto wollte er eine Sprache entwickeln, die „mächtiger als Perl und objektorientierter als Python“ (vgl. Röhrl et al. [30]) ist. Neben diesen beiden Programmiersprachen wurde Ruby auch von Smalltalk, Scheme, Lisp, CLU und Eiffel beeinflusst. Weiters legte Matsumoto großen Wert darauf, dass Ruby einige Prinzipien verfolgt. Die wichtigsten dabei sind (vgl. Rubys Prinzipien [33]):

- POLS-Prinzip:
  Das Prinzip der geringsten Überraschung („**P**rinciple **o**f **l**east **s**urprise“) soll das Programmieren intuitiv machen. Es soll den Entwicklern also nichts überraschen, sondern ihren Erwartungen entsprechen. Auch die Syntax soll einfach gehalten werden, um die Anwendung von Ruby zu erleichtern.
- EIAO-Prinzip:
  Hierbei handelt es sich um das Prinzip „**E**verything **i**s **a**n **o**bject“ – alles ist ein Objekt. In Ruby wird grundsätzlich alles als Objekt gehandhabt. Die oberste Superklasse in Ruby ist `Object`, wodurch jedes erzeugte Ding von dieser Klasse erbt.
- Duck typing-Prinzip:
  Das duck typing-Prinzip beschreibt die Behandlung von Objekten aufgrund ihrer Methoden, nicht anhand ihrer Typisierung. D.h. das Ruby nicht auf die Klasse eines Objektes achtet, sondern nur darauf, ob dieses Objekt den gewünschten Methodenaufruf abarbeiten kann oder nicht.

Ruby ist vielseitig einsetzbar. Es können komplexe Projekte genauso wie einfache Skripten erstellt werden. Dadurch und aufgrund der großen Akzeptanz von Ruby wurden auch viele Erweiterungen dafür entwickelt. So wurde beispielsweise ein Ruby-Interpreter in Java, namens JRuby, entwickelt, der die Kombination von Java und Ruby ermöglicht. Oder auch die Entwicklung des Web Application Frameworks Ruby on

[2] http://www.ruby-forum.com/

Rails, welches im Kapitel 2.3 näher beschrieben wird. Weiters bestehen zahlreiche Softwarepakete für Ruby – so genannte Gems – die als Bibliothek in Ruby einfach eingebunden werden können.

Ruby verfügt über interessante Konzepte, welche für die Erstellung eines Rollenkonzeptes von Bedeutung sind. Zu diesen zählen u. a. folgende Möglichkeiten, auf die im Anschluss näher eingegangen wird:

- Eingriff in das Method Dispatching
- Vortäuschen einer Mehrfachvererbung
- Erstellen von objektspezifischen Klassen

2.2.1 Eingriff in das Method Dispatching

Method Dispatching ist eine spezielle Form des Dynamic Dispatch Ansatzes. Nach Milton und Schmidt [21] ist der Prozess des Dynamic Dispatch ein sehr wichtiger für objektorientierte Sprachen. Sie beschreiben die Aufgabe des Dynamic Dispatching ganz allgemein durch folgenden Satz: „Given a class name c and a feature name f, determine the actual feature location.“, wobei ein Feature ein Attribut oder eine Methode darstellen kann. Es stellt also grob ausgedrückt nur einen internen Suchalgorithmus dar. Etwas detaillierter formuliert beschäftigt sich das Dynamic Dispatching grundsätzlich mit dem Senden bestimmter Nachrichten an die richtigen Objekte und zwar zur Laufzeit. Dies soll ermöglichen, dass Objekte deren Typ zur Kompilierzeit noch nicht bekannt sind trotzdem eine Struktur und ein Verhalten verwenden können. Wenn z.B. zwei unterschiedliche Klassen jeweils eine gleichnamige Variable beinhalten und nun ein Objekt, deren Klassenzugehörigkeit noch nicht bekannt ist, diesen Variablennamen aufruft, soll dies kompilierbar sein da im Laufe der Ausführung dem Objekt die Klasse noch zugewiesen werden kann. Das Dynamic Dispatching ist in diesem Beispiel nun dafür verantwortlich, dass, je nach Klassenzugehörigkeit, die korrekte Methode ausgeführt wird.

Dynamic Dispatching kann auch Probleme mit sich bringen. Insbesondere durch deren zusätzlich anfallenden Kosten. Je nach Programmiersprache und der Umsetzung dieses Ansatzes sind die einzelnen Probleme mehr oder weniger stark ausgeprägt. Wobei in vielen Programmiersprachen zusätzliche Optimierungsverfahren für das Dynamic Dispatching entwickelt wurden, um diese Probleme weiter zu reduzieren. Milton und

Schmidt [21] gehen dabei näher auf diese Probleme und deren Optimierungs- und Lösungsansätze ein. Lippman [19] beschreibt ausführlich Lösungsmöglichkeiten und Optimierungen von Dynamic Dispatching in Bezug auf C++.

Im Gegensatz zum Dynamic Dispatching wird beim Method Dispatching lediglich auf das Auffinden von Methoden eingegangen. In Ruby macht dies jedoch keinen Unterschied, da dabei Attribut- und Methodenaufrufe gleich gehandhabt werden. Somit kann das Method Dispatching in Bezug auf Ruby mit dem Dynamic Dispatching gleichgesetzt werden.

Das Method Dispatching wird in Ruby anhand der `method_missing`-Methode durchgeführt. Sobald ein Methodenaufruf erfolgt und diese Methode nicht beim jeweiligen Objekt oder deren Vorfahren gefunden werden kann, wird die `method_missing`-Methode aufgerufen. Diese kann jedoch überschrieben werden, was dazu führt, dass die `method_missing` der jeweiligen Klasse aufgerufen wird, sobald eine Methode in dieser Klasse nicht gefunden werden kann. Somit hat der Programmierer die Möglichkeit Einfluss auf den Dispatch-Vorgang zu nehmen (vgl. Thomas et al. [37]).
Der Eingriff in das Method Dispatching ist ein äußerst hilfreicher Ansatz in Bezug auf die Implementierung von Erweiterungsmechanismen. So kann beispielsweise eine Vererbung auf Instanzebene einfach umgesetzt werden.

2.2.2 Simulation einer Mehrfachvererbung

Ruby unterstützt grundsätzlich keine Mehrfachvererbung. Jedoch verfügt Ruby über die Möglichkeit eine Mehrfachvererbung zu simulieren. Eine Klasse kann zwar nur von genau einer Superklasse erben, sie kann aber zusätzlich mehrere Module einbinden, wobei ein Modul eine Gruppierung von Variablen und Methoden ist. Durch die Einbindung stellt ein Modul ein so genanntes Mix-In dar. Mit deren Hilfe erbt die Klasse sämtliche Variable und Methoden des Moduls, wodurch sich das Modul wie eine Superklasse verhält. Der einzige Unterschied zu einer tatsächlichen Mehrfachvererbung besteht darin, dass nur von der echten Superklasse Instanzen erzeugt werden können, nicht aber von den Mix-Ins. Durch diese simulierte Mehrfachvererbung können einerseits die Vorteile einer Mehrfachvererbung, wie z.B. die Vermeidung von Code-

Duplikation, genutzt werden und andererseits bleibt die Überschaubarkeit einer Einfachvererbung bestehen (vgl. Thomas et al. [37]).

Wenn ein Mix-In in einer Klasse eingebunden wird, erweitert sich die interne Klassenstruktur von Ruby wie in Abbildung 4 dargestellt.

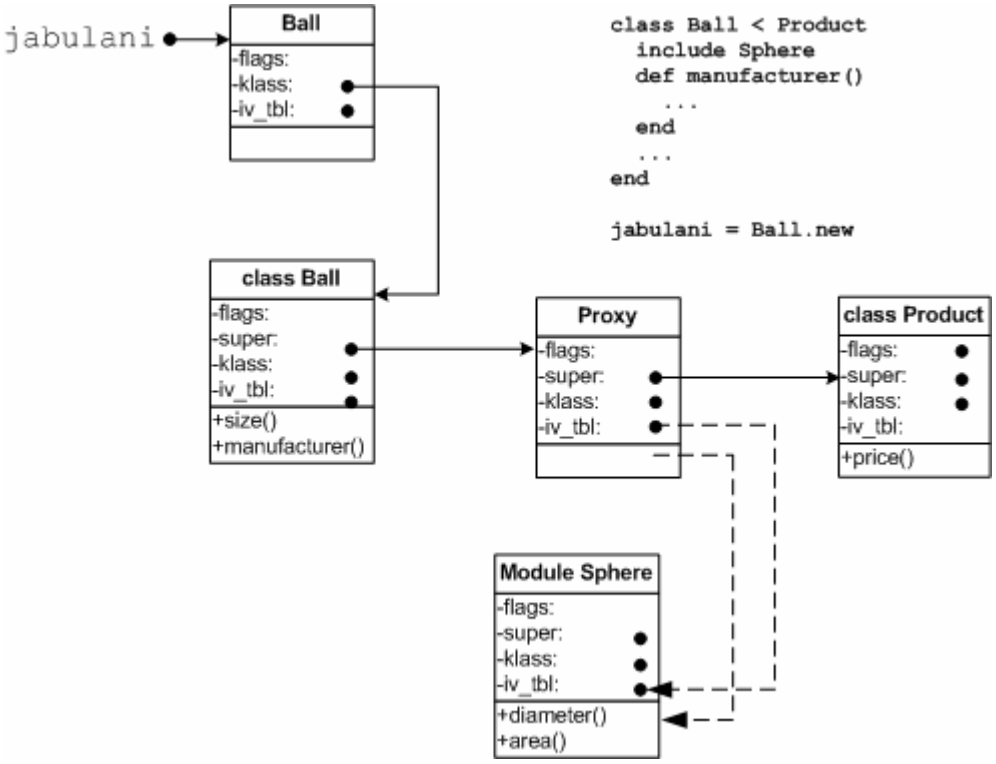

Abbildung 4: Erstellung eines Mix-In

In dieser Abbildung soll die simulierte Mehrfachvererbung anhand eines Beispiels verdeutlicht werden. Der Fußball der kommenden Weltmeisterschaft 2010, der Jabulani, ist eine Instanz der Klasse `Ball`. Ein Ball stellt wiederum ein Produkt eines Sportgeschäftes dar und wird dabei durch seine Größe und seinem Hersteller beschrieben. Den Preis soll ein Ball von der Klasse `Product` erben. Jedoch ist ein Ball aus mathematischer Sicht eine Kugel, wodurch auch beispielsweise sein Durchmesser und seine Fläche berechnet werden können. Daher sollte ein Ball einerseits von der Klasse `Product` und andererseits von der Klasse `Sphere` erben. In der Abbildung ist ersichtlich, dass `Sphere` als Modul in die Klasse `Ball` inkludiert wird. Somit verfügt das erzeugte `Ball`-Objekt, zusätzlich zu den `Ball`-Methoden und den Methoden von `Product`, über die `Sphere`-Methoden, was im Sinne einer Mehrfachvererbung ist. Trotzdem ist die Klasse `Ball` nur einer einzigen Klasse, der `Product`-Klasse,

zugeordnet. Durch die Einbindung des Moduls entsteht Ruby-intern eine Proxy-Klasse zwischen der Klasse `Ball` und ihrer Superklasse `Product`. Der Proxy verweist dabei auf das entsprechende Modul. Der Grund, warum ein Proxy erzeugt und nicht das gesamte Modul einfach der Klasse `Ball` angehängt wird, liegt im dafür benötigten Speicherbedarf. Wenn mehrere Klassen das gleiche Modul einbinden, entstehen mehrere Kopien vom selben Code, was unnötigen Speicher kostet. Deswegen wird für jede Einbindung eine Proxy-Klasse erstellt, welche eine Referenz auf genau das eine Modul beinhaltet. Somit verweisen alle entsprechenden Klassen auf das Original-Modul über einen separaten Proxy.

2.2.3 Erstellen von objektspezifischen Klassen

In manchen Szenarien ist es sinnvoll, wenn sich ein Objekt einer Klasse von den anderen Objekten dieser Klasse aufgrund deren Eigenschaften unterscheidet. Wenn beispielsweise ein Produkt aus einem Sortiment beschädigt wurde, sollte nur dieses Produkt gekennzeichnet werden (z.B. durch eine neue Beschreibung in der der Schaden beschrieben wird). Dies ist möglich, indem dem einen Objekt eine Klasse zugewiesen wird, in der die Ausgabe der Beschreibung überschrieben wird. Ähnliches wird in Abbildung 5 dargestellt.

Rubyintern wird bei diesem Aufruf eine Singleton-Klasse erzeugt, welche die dadurch erstellten Variable und Methoden beinhaltet. Die Positionierung dieser neuen Singleton-Klasse in der Klassenhierarchie wird in Abbildung 5 dargestellt. Im oberen Bereich der Abbildung wird die Struktur eines einfachen Stringobjektes aufgezeigt. Wird hier die String-Methode `to_s` aufgerufen, wird für das Objekt `a` der Wert `hello` ausgegeben.

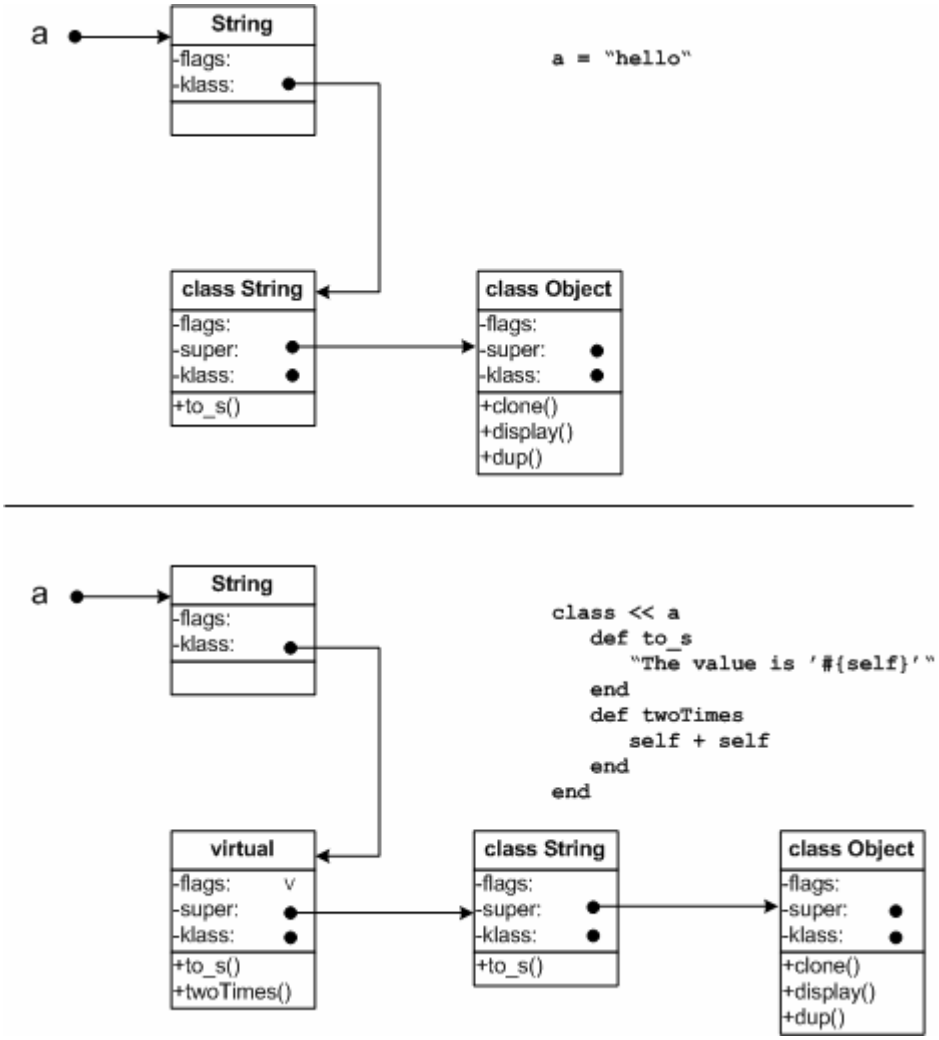

Abbildung 5: Erstellung einer objektspezifischen Klasse nach Thomas et al. [37]

Im unteren Teil der Abbildung wird diesem Objekt `a` eine Singleton-Klasse zugewiesen (entspricht der `virtual`-Klasse), welche einerseits die Methode `to_s` überschreibt und andererseits eine zusätzliche Methode `twoTimes` beinhaltet. Die neue Singleton-Klasse wird dabei, wie zuvor schon erwähnt, zwischen dem Objekt und seiner ursprünglichen Klasse hinzugefügt. Wenn nun der Aufruf `a.to_s` erfolgt, wird der Text „`The value is 'hello'`“ ausgegeben. Für alle anderen String-Objekte gilt aber immer noch die standardmäßige Ausgabe. Für die Programmierer ist diese neue Klasse jedoch nicht ersichtlich. D.h. dass der Aufruf `a.class` nicht, wie der Abbildung entsprechend, die Singleton-Klasse zurückgibt, sondern die ursprüngliche Klasse `String`.

2.3 Das Web Application Framework Ruby on Rails

Ruby on Rails – kurz Rails oder auch RoR genannt – ist ein auf Ruby basierendes Web Application Framework. Es wurde von David Heinemeier Hansson entwickelt und im Jahr 2004 veröffentlicht. Seit jeher ist es als Open-Source verfügbar, was zu einer großen Verbreitung führte. Das Ziel von Ruby on Rails liegt in einer schnellen, direkten und vor Allem unkomplizierten Erstellung einer Web-Applikation.

In der Literatur (vgl. Wirdemann und Baustert [41] bzw. Carl [7]) ist zu lesen, dass Ruby on Rails zwei wesentliche Philosophien verfolgt:

- „**D**on't **r**epeat **y**ourself" – DRY:
  Um einerseits ein Projekt nicht unnötig zu vergrößern und andererseits um redundante Codestücke bzw. Informationen zu vermeiden. Dabei wird auch die Datenbank miteinbezogen. Daten, welche in einer Datenbank abgespeichert sind, sollen bei der Implementierung nicht zusätzlich als Attribut neu angelegt werden. Mit Ruby on Rails besteht grundsätzlich die Möglichkeit direkt auf die Daten aus der Datenbank zuzugreifen, worauf aber erst später eingegangen wird.

- „Convention over Configuration":
  „Convention over Configuration" oder „Konvention geht über Konfiguration" bedeutet, dass Ruby on Rails standardmäßig anhand von Namenskonventionen innerhalb eines Projektes kommuniziert. So muss beispielsweise der Programmierer keine eigene Konfigurationsdatei für den Datenbankzugriff erzeugen um auf eine Tabelle zuzugreifen, da Ruby on Rails den Zugriff über die Modelklassen der MVC-Architektur bereitstellen kann und auf die Inhalte der Tabelle über deren Namen möglich ist. Dafür ist jedoch eine bestimmte, jedoch einfach verständliche und somit gut lesbare, Namenskonvention einzuhalten. Ein Beispiel dafür wäre, dass für das Model Employee eine Tabelle in der Datenbank zur Verfügung stehen muss, die denselben Namen in der Mehrzahl trägt (Employees). Weiters muss diese Tabelle über den Primärschlüssel id verfügen. Fremdschlüssel hingegen haben den Namen des Models mit einem angehängten „_id" (z.B. person_id). Diese Namenskonventionen sind jedoch nicht zwingend und die jeweiligen Verbindungen können auch konfiguriert werden. Jedoch muss eine derartige Implementierung dadurch bei der einfachen Lesbarkeit Abstriche verbuchen.

Die Grundarchitektur einer Ruby on Rails Webapplikation verfolgt die MVC-Struktur (Model View Controller), wobei das Model die Daten bzw. Ressourcen einer Webapplikation verwaltet, die View für die Repräsentation der Daten verantwortlich ist und der Controller die Schnittstelle zwischen Model und View bereitstellt. Reenskaug et al. [26] geben einen detaillierten Einblick in die MVC-Archtiektur. In Abbildung 6 wird die Architektur von Ruby on Rails, entsprechend Wirdemann und Baustert [41], dargestellt und anschließend erklärt. Ab der Ruby on Rails Version 1.2 werden standardmäßig RESTful Webapplikationen erstellt, worauf die Beschreibung auch abzielt.

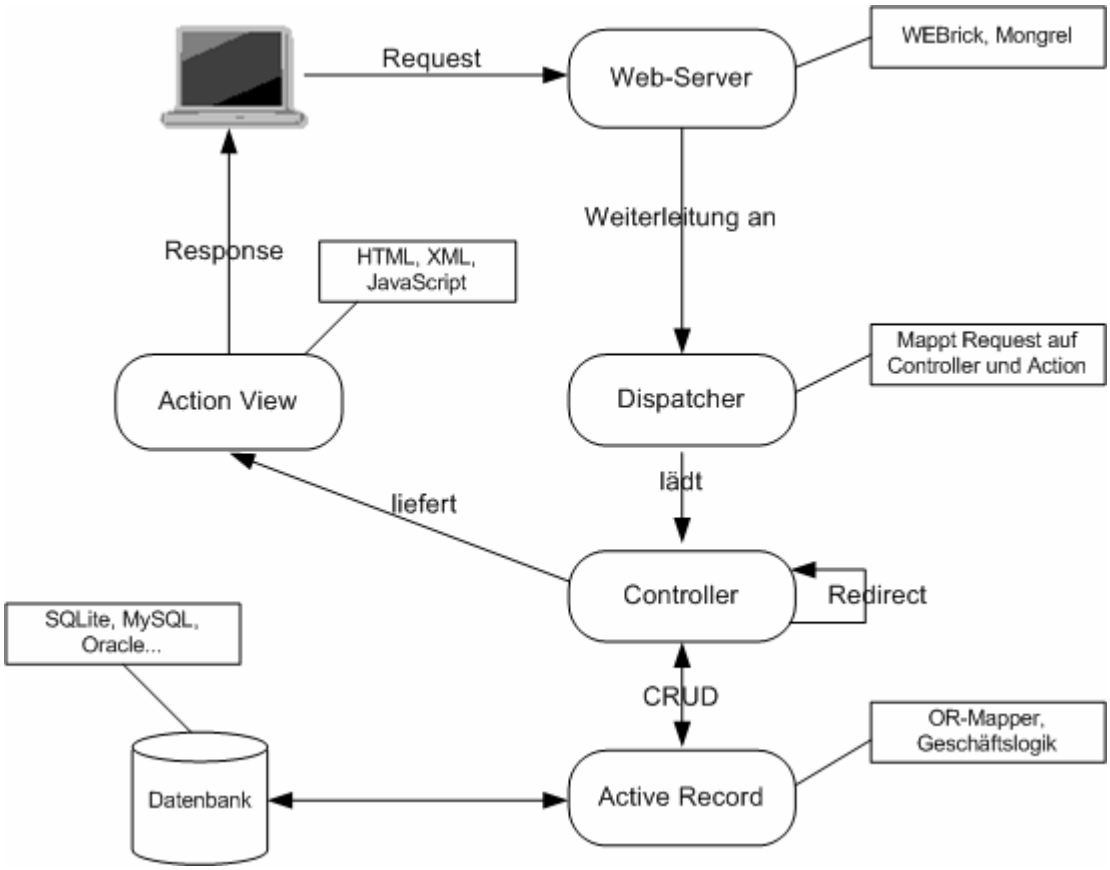

Abbildung 6: Ruby on Rails Architektur

Diese Abbildung zeigt sämtliche Komponenten und deren Zusammenwirken. Erfolgt dabei eine Anfrage aus dem Netz wird diese an den Web-Server (z.B. WEBrick oder Mongrel) gestellt und von diesem an den Dispatcher des Ruby on Rails-Projekts weitergeleitet. Jedes einzelne Ruby on Rails-Projekt besitzt einen eigenen Dispatcher, der aufgrund des Requests den zuständigen Controller lädt. Genauer gesagt wird dabei die angeforderte Action eines Controllers ausgeführt. Ein Controller kann über mehrere Actions verfügen, wobei jede Methode eines Controllers eine Action darstellt. Ruby on

Rails kann bei der Generierung Actions für die HTTP-Methoden bereitstellen, wodurch die CRUD-Methoden (Create, Read, Update und Delete) für den Zugriff auf das Model verfügbar sind. Actions eines Controllers können dabei andere Actions desselben Controllers aufrufen, wodurch ein Redirect ausgeführt wird. Im Model erfolgen Datenbankzugriffe üblicherweise über das ActiveRecord-Framework, welches auch die Geschäftslogik inkludiert. Es beinhaltet somit einen objektrelationalen Mapper, der für die Umwandlung eines Tabelleneintrags zu einem Objekt und umgekehrt, zuständig ist. Natürlich muss nicht immer eine Datenbank am Back-End zur Verfügung stehen. Stattdessen können andere Frameworks bereitstehen, über die ein Model verfügen kann:

- Action Mailer – für den Email-Versand über Ruby on Rails
- ActiveResource – zum Verweis auf entfernte Ressourcen
- ActiveRDF – zur Handhabung von RDF-Dateien
- u.v.m.

Wurde der gestellte Request abgearbeitet, werden die Daten vom Controller an die jeweilige View weitergeleitet und von dort an die ausgehende Quelle ein entsprechender Response zurückgeliefert. Die View kann hier unterschiedlich gestaltet werden. Standardmäßig werden rhtml-Dokumente (Ruby HTML[3]) für die Repräsentation mittels Browser generiert. Nach Carl [7] können diese rhtml-Dokumente durch das Framework Action View in JavaScripts umgewandelt werden.

Bei der Erstellung einer Ruby on Rails Webapplikation wird zuerst das Grundgerüst des Projektes erzeugt. Dabei werden neben der Verzeichnisstruktur auch schon sämtliche Dateien erzeugt, die zum Start der Webapplikation von Nöten sind (siehe Kapitel 7.2 – Anleitung). Um nun für das gewählte Back-End die richtigen Model-Klassen zu erzeugen, stehen verschiedene Generatoren zur Verfügung. Wird beispielsweise als Back-End eine Datenbank ausgewählt, so stehen grundsätzlich zwei Generatoren zur Verfügung. Einerseits der model-Generator und andererseits der scaffold-Generator. Ersterer erzeugt lediglich die Model-Klassen, welche zum Datenbankzugriff benötigt werden. Die dazugehörigen Controller, welche die Benutzereingaben an das Model weiterleitet und die passenden View-Dateien zur Präsentation der Daten werden von letzteren generiert [vgl. Marinschek und Radinger [20]). Wird der scaffold-Generator verwendet, kann bereits ohne weiteres Zutun seitens des Programmierers eine lauffähige

[3] RHTML ist ein erweitertes HTML-Dokument um zusätzlichen Ruby-Code.

Webapplikation erzeugt werden, welche die CRUD-Funktionen auf der Datenbank abarbeiten kann. Nach Carl [7] ist diese effiziente Erstellung des Grundgerüstes ein großer Vorteil von Ruby on Rails gegenüber anderen Web-Frameworks und um den diese Ruby on Rails „beneiden“.
Betrachtet man diese Generatoren näher, so ist ersichtlich, dass es sich hier um Metaprogrammierung handelt: Generatoren sind Ruby-Programme, die Ruby-Code erzeugen.

2.4 Open Issues

In vielen objektorientierten Programmiersprachen können Rollen bereits inkludiert und zur Implementierung verwendet werden. Für die Programmiersprache Ruby trifft dies allerdings noch nicht zu.

Durch den Vorteil des Frameworks Ruby on Rails ein Grundgerüst für Webapplikation zu generieren, ergibt sich die Möglichkeit Rollenfunktionalitäten zu einem Grundgerüst zu inkludieren. Derzeit besteht noch kein Generator für Ruby on Rails, welcher ein Rollenmodell in der Struktur einer Webapplikation einbettet. Durch die Bereitstellung eines Generators für Rollenklassen könnte die Entwicklung von rollenbasierten Webapplikationen deutlich erleichtert werden. Wichtig dabei ist jedoch, dass inkludierte Rollenmodell an die Anforderungen des dezentralen Einsatzes anzupassen. Diese beziehen sich auf folgende Punkte, welche anschließend näher erläutert werden:

- Navigation im dezentralen Einsatz
- Dezentrale Teilhierarchien
- Top-Down und Bottom-Up Erstellung von Rollenhierarchie
- Namensgebung (Synonyme und Homonyme)
- Unterscheidung in private und öffentliche Attribute
- Identifikation und Repräsentation

2.4.1 Navigation im dezentralen Einsatz

Beim Rollenkonzept von Gottlob et al. [11] wird davon ausgegangen, dass sämtliche Instanzen einer Rollenhierarchie uneingeschränkt zugänglich sind, sodass zur Navigation in der Hierarchie lediglich die Beziehung `roleOf` zwischen zwei Instanzen

und eine Liste aller Rollentypen eines Real-World-Objektes (`roleDictionary`) notwendig ist. Durch die mögliche Verteilung der inkludierten Instanzen im dezentralen System kann jedoch nicht ausgeschlossen werden, dass auf einige Instanzen nur beschränkt ein Zugriff möglich ist. Verfügt man über eine Instanz beispielsweise nur über einen Lesezugriff kann die `roleOf`-Beziehung nicht gesetzt werden. Dadurch kann auch keine spätere Navigation über diese Beziehung erfolgen. Es wäre nur eine Navigation über das Wurzelobjekt und deren Rollenverzeichnis möglich, wobei aber auch der Schreib-Zugriff auf dieses Verzeichnis verwehrt werden könnte, was eine Navigation dadurch verhindern würde.

Durch die Zugriffsbeschränkungen ergeben sich somit folgende Szenarien, welche anhand eines Beispieles näher erklärt werden.

Beispiel:
Eine Instanz der Klasse `Person` soll eine Rolle des Rollentyps `Employee` einnehmen. Wobei zwischen den beiden Objekten die Beziehung `roleOf` (von `Employee` zu `Person`) bestehen soll.

Szenarien:

1. Die Instanz der Klasse `Person` ist lokal gespeichert, die des Rollentyps `Employee` jedoch auf einer nicht vertrauenswürdigen Quelle.
   In diesem Szenario kann nur ein Eintrag in das Rollenverzeichnis der Personen-Instanz erstellt werden, da der Zugriff auf die Rolle nur lesend gestattet wird. Dadurch ist zwar keine Navigation von der Instanz von `Employee` zur derer von Person mehr gewährleistet, aber durch den Eintrag in das Rollenverzeichnis dafür eine Navigation in der entgegen gesetzten Richtung. Weiters muss davon ausgegangen werden, dass die Rollenmethoden, über die jede Rolle verfügen soll (siehe Abschnitt 2.1.1.2), auch nur für die Instanz der Klasse `Person` verfügbar sind, nicht aber für die `Employee`-Instanz. Deren Rollenmethoden können, müssen aber nicht vorhanden sein. Auch die Rückgabewerte einiger Methoden der Instanz von `Person` sind mit Vorsicht zu genießen, da z.B. der Methodenaufruf `as(Employee)` – diese soll die existierende Rolleninstanz von `Employee` zurückgeben – entweder keinen Wert oder keine Rollen-Instanz zurückgeben kann. Der Grund dafür ist einerseits, dass der „entfernte“ Klasse

`Employee` keinem Rollentyp entspricht und somit eben nicht zurückgegeben wird, und andererseits, wenn deren Typ ignoriert werden würde, muss man davon ausgehen, dass die Klasse `Employee` alles andere als eine Rolleklasse sein kann.

2. Die Instanz des Rollentyps `Employee` liegt am lokalen Rechner, die der Klasse `Person` jedoch nicht.

   Bei dieser Ausgangslage ist sichergestellt, dass die `Employee`-Instanz alle notwendigen Methoden besitzt, jedoch nicht die der Klasse `Person`. Weiters besteht auch entweder kein Rollenverzeichnis für die Instanz der Person oder zumindest kein Eintrag in dieser Liste über die `Employee`-Instanz, wodurch das `roleDictionary` nicht verwendbar ist. Die Beziehung zwischen den beiden Objekten kann hier, im Gegensatz zum vorherigen Szenario, nur mittels der `roleOf`-Beziehung erstellt werden. Somit ist die Navigation von der Instanz Person zu der Instanz von `Employee` nicht sichergestellt. Aber auch hier können Probleme mit den Methoden der Rolle auftreten. Zwar treten in diesem sehr einfachem Beispiel keine Fehler auf, sofern nur die Methoden der Rolle verwendet werden. Sobald aber eine Rollenstruktur an der entfernten Instanz angefügt ist, kann auf diese nicht unbedingt zugegriffen werden. Dies wäre nur möglich, wenn die entfernte Personen-Instanz und deren Rollenstruktur dieselben Methoden vorweisen können, die das Rollenkonzept von Gottlob et al. [11] vorgibt. Ansonsten wären nur diese zwei Objekte in der Rollenhierarchie verfügbar.

3. In diesem Szenario sollen beide Objekte Rolleninstanzen darstellen, wobei der Rollentyp `Person` (Super-Rolle) wiederum auf einer entfernten Quelle zu finden ist.

   Hierbei vermischen sich Problematiken von den ersten beiden Szenarien. Einerseits ist die Navigation nur von der Instanz der Sub-Rolle `Employee` zu der des Rollentyps `Person` möglich. Weiters ist mit Sicherheit kein Rollenverzeichnis verfügbar, da die Instanz der Person nun kein Wurzelobjekt mehr repräsentiert. Dadurch kann wiederum nur dieser kleine Ausschnitt aus der womöglich größeren Rollenstruktur betrachtet werden. Andererseits können dabei Probleme bei den Methoden von `Employee` auftreten – sofern die Super-

Rolle `Person` nicht wieder exakt die benötigten Methoden anbietet. Beispielsweise kann die Rollenmethode `root` – diese soll die Instanz der Wurzelklasse einer Rollenhierarchie zurückgeben – nicht den gewünschten Rückgabewert liefern. Entweder es liefert einen falschen Wert, da die entfernte Rolleninstanz nicht über eine `roleOf`-Beziehung auf deren übergeordnete Rolleninstanz verweist. Dies hätte zur Folge, dass die Instanz des Rollentyps `Person` als Wurzelobjekt zurückgeliefert werden würde, oder es kann eine Exception geworfen werden, weil z.B. die benötigten Methoden nicht in der Super-Rolle implementiert sind.

4. Beide Objekte – nun wieder eine Instanz einer Rolle und einer Klasse – sind auf entfernten Quellen.
   In diesem Fall besteht die Möglichkeit die Rollenbeziehung auf einer weiteren Quelle festzuhalten, wobei die beiden Objekte nichts von dieser Beziehung wissen. Wenn hierbei innerhalb der Rollenhierarchie navigiert werden soll, muss über dies über die dritte Quelle erfolgen. Huemer beschreibt in seiner Diplomarbeit [13] eine Lösung dieses Szenarios, wobei dieses Szenario in dieser Diplomarbeit jedoch nicht berücksichtigt wird.

Aufgrund dieser Szenarien wird ersichtlich, dass die Navigation einige Probleme mit sich bringt, wenn Objekte nur beschränkte Zugriffsrechte aufweisen.

2.4.2 Dezentrale Teilhierarchien

Eine weitere Anforderung, bei der das Rollenkonzept von Gottlob et al. [11] an seine Grenzen stößt, ist, dass Teilhierarchien in dezentralen Systemen einer Rollenhierarchie auch unabhängig voneinander existieren können. Ein Beispiel dafür wäre, wenn die Rollenhierarchie einer Universität (`Student`) und die eines Unternehmens (`Employee`) zusammengeführt werden sollen, zu der im Beispiel aus Abschnitt 1.4 beschriebenen Rollenhierarchie. Damit aber zuvor die Existenz der beiden Teilhierarchie mit den Vorgaben des Rollenkonzeptes von Gottlob et al. [11] konform gehen, müssen die Wurzelobjekte der beiden Rollenhierarchien der Metaklasse `ObjectWithRoles` angehören. Wenn jedoch diese Teilhierarchien zusammengeführt werden sollen, muss die Klassenzugehörigkeit der beiden Wurzelobjekte auf die

Metaklasse `RoleType` geändert werden. Eine dynamische Änderung der Klassenzugehörigkeit kann aber zu Komplikationen führen und wird nicht von allen objektorientierten Systemen unterstützt. Die andere Möglichkeit dabei bestünde darin, die Wurzelobjekte der Teilhierarchien gleich als Objekte der Metaklasse `RoleType` zu generieren, was aber laut Gottlob et al. [11] nicht zulässig wäre. Jedoch ist der Bestand von Teilhierarchien ein essentieller Bestandteil bei der Verwendung eines Rollenkonzeptes im dezentralen Einsatz und muss deshalb auch verfügbar gemacht werden.

2.4.3 Top-Down und Bottom-Up Erstellung von Rollenhierarchie

Beim Ansatz von Gottlob et al. [11] wird davon ausgegangen, dass Rollenhierarchien nach dem Top-Down-Prinzip erstellt werden. Zuerst wird dabei die generelle Instanz der Wurzelklasse erzeugt und anschließend deren direkt darunter liegenden Rolleninstanzen usw. Es findet also eine Spezialisierung statt. Bei der Generierung einer Instanz einer Sub-Rolle ist einfach zu erkennen, welche Attribute bzw. Methoden dieses von ihrem Wurzelobjekt erbt und welche (noch) zu erstellen sind.

Beobachtungen aus der realen Welt zeigen jedoch, dass der Generalisierungsansatz häufig angewendet wird. Insbesondere beim dezentralen Einsatz ist eine Bottom-Up Erstellung von Rollenhierarchien oft zweckmäßig. Wie auch das Beispiel im vorherigen Abschnitt zeigt, bei dem zwei Teilhierarchien zu einer Gesamthierarchie zusammengeführt werden, wobei das Finden bzw. Generieren von gemeinsamen Nennern von Objekten eine wichtige Rolle spielt. Aus diesem Grund muss auch die Möglichkeit bestehen eine Rollenstruktur von unten nach oben zu erzeugen.

2.4.4 Namensgebung (Synonyme und Homonyme)

Ein weiteres Problem durch einen dezentralen Einsatz besteht in der Namensgebung von Attributen und Methoden von Objekten einer Rollenhierarchie. Da die einzelnen Objekte oder Teilhierarchien von unterschiedlichen Quellen erstellt werden, kann die Namensvergabe nicht kontrolliert werden. Somit besteht die Möglichkeit, dass gleiche Attribute bzw. Methoden unterschiedliche Namen aufweisen (Synonyme) oder verschiedene Attribute bzw. Methoden mit denselben Namen belegt werden (Homonyme). Wenn jedoch Objekte mit solchen Attributen bzw. Methoden in eine

Rollenhierarchie zusammengeführt werden, kann dies zu Komplikationen führen. Insbesondere Synonyme müssen behandelt werden, wenn beispielsweise alle Telefonnummern einer Person ermittelt werden sollen, aber nicht alle Rollenattribute für die Telefonnummer den Namen `phoneNo` aufweisen. Dadurch besteht die Möglichkeit, dass nicht alle verfügbaren Telefonnummern in der Rollenhierarchie gefunden werden. Homonyme können hingegen vernachlässigt werden, da der Aufruf des Attributes bzw. der Methode von der jeweiligen Instanz abhängig ist. Problematisch kann dies nur in der Interpretation des Rückgabewertes werden, welche jedoch dem Anwender vorbehalten ist und somit nicht weiter behandelt werden.

2.4.5 Unterscheidung in private und öffentliche Attribute

Grundsätzlich können sämtliche Attributwerte von Rollen an die Subrollen weitergereicht werden. Und durch die Anpassung der Erstellung einer Rollenhierarchie mittels Bottom-Up-Ansatz (siehe Abschnitt 2.4.3) kann eine derartige Nachricht auch an die Superrolle weitergeleitet werden. Um jedoch die Verwendungsmöglichkeiten zu erweitern, sollen auch private Attribute an eine Rolle inkludiert werden können. Darunter werden solche Attribute verstanden, welche nicht weitergegeben werden sollen. Diese sollen somit nur über die eigene Rolleninstanz abgefragt werden können. Ein Beispiel könnte das Glaubensbekenntnis einer Person darstellen. Dabei soll nicht möglich sein dieses Attribut über deren Angestelltenrolle abzufragen, weil das Glaubensbekenntnis eines Angestellten keine Relevanz in dieser Rolle haben sollte. Ein weiteres Beispiel für ein derartiges Attribut, welches sich auf das Beispiel aus Abschnitt 1.4 bezieht, könnte das Gehalt eines `DepartmentManagers` sein. Wenn also das Gehalt von `deptMgrBlack` abgefragt werden soll, ist das nur über diese Rolle möglich. Somit sollte der Zugriff über den Angestellten `empBlack` auf deren Gehalt als `DepartmentManager` nicht möglich sein.

Anhand dieser Voraussetzungen ist ersichtlich, dass für einen dezentralen Einsatz des Rollenkonzeptes von Gottlob et al. [11] einige Änderungen vorgenommen werden müssen, um hier eine Verwendung zu gewährleisten.

2.4.6 Identifikation und Repräsentation

Aufgrund des Einsatzes im Web bzw. der Einhaltung der REST-Anforderungen muss die Identifikation von Objekten und deren Repräsentation noch zusätzlich ins Auge gefasst werden. Üblicherweise werden Objekte in objektorientierten Systemen durch ihre interne Objekt-ID (OID) identifiziert. Im dezentralen Einsatz bei unterschiedlichen Informationssystemen muss jedoch eine andere Form der Identifikation gefunden werden, da die OID außerhalb des eigenen Systems nicht als Identifikation verwendbar ist. Es muss also eine globale Eindeutigkeit eines Objektes geschaffen werden.

Auch die Repräsentation der Objekte muss näher betrachtet werden. Die Darstellung eines Objektes soll für den Menschen, aber auch für Maschinen, gut lesbar sein. Hierbei soll wiederum eine standardisierte Repräsentation verwendet werden, um die Einsatzmöglichkeiten einer zu erstellender Webapplikation nicht unnötig einzuschränken.

3 Rollen in Ruby

Im vorherigen Kapitel wurden einige Rollenkonzepte erwähnt, wobei das Rollenkonzept von Gottlob et al. [11] detailliert beschrieben wurde. Dieses Konzept wurde bereits in einigen objektorientierten Programmiersprachen umgesetzt (vgl. Gottlob et al. [11] bzw. Schrefl und Thalhammer [35]). In diesem Kapitel soll nun dieses Rollenkonzept in der Programmiersprache Ruby umgesetzt werden. Im nachfolgenden Abschnitt wird die Verwendung von Rollen in Ruby beschrieben und anschließend auf die Umsetzung des Rollenkonzeptes in Ruby näher eingegangen. Abschließend folgt ein Vergleich der erstellten Ruby-Implementierung mit den zuvor erwähnten Umsetzungen in Smalltalk und Java.

3.1 Verwendung von Rollen in Ruby

In diesem Abschnitt wird die Verwendung von Rollen in Ruby erklärt. Insbesondere auf die Erstellung von Rollenklassen und deren Instanzen wird näher eingegangen und anhand von Beispielen verdeutlicht.

3.1.1 Erstellen von Rollenklassen

Klassen werden in Ruby üblicherweise durch das Konstrukt „`class ClassName`“ erzeugt. Dieses Konstrukt wird auch bei der Erstellung einer Wurzelklasse einer Rollenhierarchie verwendet, wobei eine Wurzelklasse von der Pseudo-Metaklasse (siehe Abschnitt 3.2.1) `ObjectWithRoles` erben muss (z.B. `class Person < ObjectWithRoles`). Rollentypen hingegen werden über die Klassenmethoden `defRoleType`, `defSpecializedRoleType` oder `defQualifiedRoleType` der Pseudo-Metaklassen `Role` bzw. `QualifiedRole` angelegt[4]. Der Grund dafür liegt u. a. in der Definition der Klassenvariable `roleSuperType`, welcher dynamisch gesetzt werden soll (siehe Abschnitt 3.2.2). In Listing 1 wird die Erstellung eines Rollentyps am Beispiel `Employee` dargestellt.

[4] Auf eine Methode zum Anlegen einer spezialisierten, qualifizierten Rolle (defSpecializedQualifiedRoleType) wurde verzichtet. Eine solche Methode kann analog zu defSpecializedRoleType erstellt werden.

```
01 Employee = Role.defRoleType(Person) do
02   attr_reader :salary, :phoneNo
03   def init(salary, phoneNo)
04     @salary = salary
05     @phoneNo = phoneNo
06   end
07 end
```

Listing 1: Erstellen der Rollenklasse Employee

Der Klassenmethode `defRoleType` wird die übergeordnete Klasse in einer Rollenhierarchie mitübergeben (`roleSuperType`). Durch den nachstehenden Befehl `do` wird der Klassenmethode das nachstehende Codestück als Block mitgegeben. Dies ermöglicht einen dynamischen Zugriff bzw. Änderungen auf dieses Codestück. Innerhalb des Codestücks befindet sich hier die Deklaration von Attributen und deren Definition in der Initialisierungsmethode.
Das Anlegen einer spezialisierten Rollenklasse funktioniert analog zum Erstellen einer normalen Rollenklasse. Der einzige Unterschied liegt im Übergabeparameter für die Klassenmethode `defSpecializedRoleType`. Bei dieser Klassenmethode wird nicht der `roleSuperType`, sondern die Superklasse übergeben, von dem die neu zu erstellende Rollenklasse erben soll (z.B. `ForeignStudent = Role.defSpecializedRoleType(Student)`).

Die Erstellung einer qualifizierten Rollenklasse erfolgt über die Klassenmethode `defQualifiedRoleType` der Pseudo-Metaklasse `QualifiedRole`. Das Konstrukt zu dieser Erstellung wird im folgenden Listing dargestellt.

```
01 ProjectMgr = QualifiedRole.defQualifiedRoleType(Project,
   Employee) do
02   attr_reader :skills
03   def init(skills)
04     @skills = skills
05   end
06 end
```

Listing 2: Erstellung der qualifizierten Rollenklasse ProjectMgr

In diesem Listing wird die qualifizierte Rolle `ProjectMgr` erzeugt. Die Klassenmethode `defQualifiedRoleType` hat, wie hier ersichtlich, zwei Übergabeparameter. Der erste stellt die Klasse des `qualifiers` dar, während der zweite Parameter wiederum den `roleSuperType` kennzeichnet.

3.1.2 Erstellen von Rolleninstanzen

Die Erstellung sämtlicher Instanzen erfolgt über die `new`-Methode. Hier muss nur auf die Übergabeparameter geachtet werden. Instanzen der Wurzelklasse einer Rollenhierarchie benötigen keine zustätzlichen Parameter (siehe Listing 3, Zeile 01), wohingegen Rolleninstanzen, je nach ihren Rollentyp, unterschiedliche Parameter vorweisen müssen. Normale und spezialisierte Rolleninstanzen benötigen an erster Stelle der Parameterliste die Instanz der übergeordneten Rolle bzw. der Wurzelinstanz einer Rollenhierarchie (siehe Listing 3, Zeile 02). Qualifizierte Rollen weisen die Instanz der übergeordneten Rolle bzw. der Wurzelinstanz an zweiter Stelle auf. An erster Stelle der Parameterliste muss die Instanz des qualifiers stehen (siehe Listing 3, Zeile 03).

```
01 personBlack = Person.new('Mr. Black', '533-4842', '11.7.1971')
02 empBlack = Employee.new(personBlack, 900, '342-8764')
03 projMgrBlack = ProjectMgr.new(project1, empBlack, 'experience')
```

Listing 3: Erstellung der Instanzen einer Rollenhierarchie

Anhand dieser Instanzerstellungen wurde ein Teil der Rollenhierarchie aus dem Beispiel von Unterkapitel 1.4 erzeugt. Rollenmethoden, wie z.B. ein Rollenwechsel durch die `as`-Methode, können nun durch einen Aufruf wie `empBlack.as(ProjectMgr)` erfolgen.

3.2 Implementierung

In diesem Kapitel erfolgt die Beschreibung der direkten Umsetzung des Rollenkonzeptes von Gottlob et al. [11] in Ruby. Hier werden zuvor konzeptionelle Ideen für die Implementierung betrachtet und anschließend werden die Möglichkeiten der Erstellung der drei Arten von Rollenhierarchieklassen beschrieben. Abschließend werden wichtige Elemente erläutert, die zur Verwendung des Rollenkonzeptes von Nöten sind.

3.2.1 Strukturaufbau

Bei der Umsetzung des Rollenkonzeptes von Gottlob et al. [11] würde grundsätzlich die Trennung vom Meta-Modell und dem eigentlichen Rollenmodell als sinnvoll erachtet. Atkinson und Kühne [2] verfolgen in ihrer Arbeit ebenfalls den Ansatz der schichtweisen Metamodellierung, wobei sie diese über Metaebene M2 noch hinaus erweitern. Ein, für das Rollenkonzept, nahe liegender Ansatz wird in Klas und Schrefl [14] bzw. in Kühne und Schreiber [16] und Kühne und Steimann [17] beschrieben. Sie verwenden dabei den Ansatz der tiefen Instanzierung. Ein ähnlicher Ansatz wird auch von Neumayr et al. [22] verfolgt. Die tiefe Instanzierung gewährleistet nicht nur eine Beschreibung der direkt untergeordneten Instanzen, sondern auch von jenen aus tieferen Ebenen. In Abbildung 7 wird ein dafür ausgelegter Lösungsansatz dargestellt. Hier ist die Trennung zwischen Meta-Modell und Klassenmodell gut ersichtlich, wobei das Meta-Modell (entspricht laut UML der M2-Schicht) klar von der Rollenstrukturierung (entspricht laut UML der M1-Schicht) abgegrenzt wird. In dieser Schicht befinden sich nach Klas und Schrefl [14] sämtliche Instanztypen, während die dazugehörigen Instanz-Instanzen in Schicht M0 abgebildet werden würden. Auf deren Abbildung wurde hier aber verzichtet, da sie für den Aufbau der Klassenstrukturierung keine Rolle spielen.

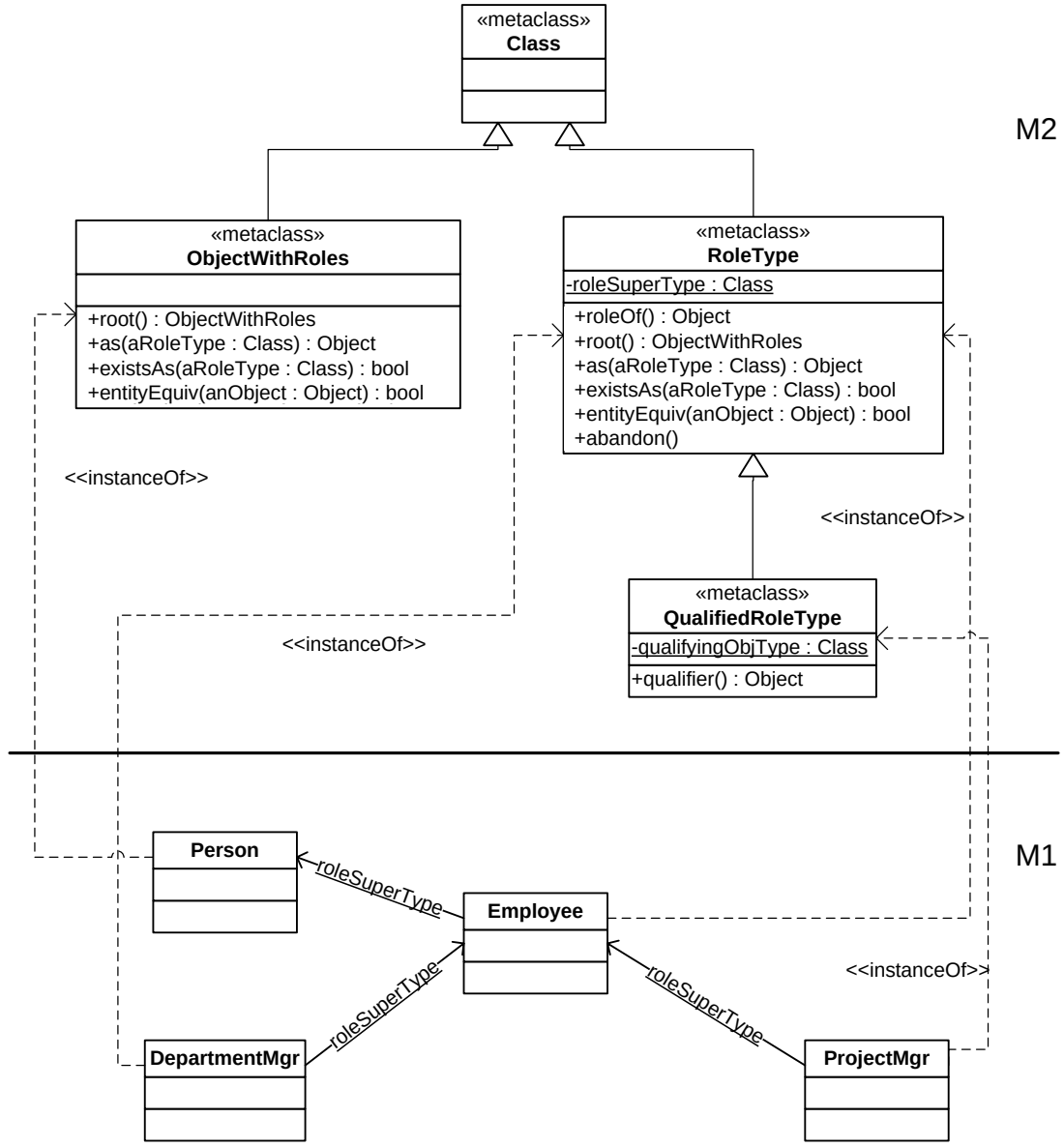

Abbildung 7: Umsetzung des Rollenkonzeptes – Ideallösung mittels tiefer Instanzierung

In dieser Abbildung werden in der Metaebene M2 vier Klassen abgebildet. Die dargestellte Metaklasse `Class` ist in Ruby die ausgehende Klasse. Sämtliche Klassen sind Instanzen von dieser. Sie ist in Ruby somit eine allgegenwärtige Metaklasse und daher auch die Metaklasse jeder Klassenhierarchie (vgl. Thomas et al. [37]). Weiters befinden sich auf dieser Ebene die Metaklassen `ObjectWithRoles`, `RoleType` und `QualifiedRoleType` aus Gottlob et al. [11]. Diese werden als Subklassen von `Class` dargestellt und bilden somit Spezialisierungen von `Class`. In diesen Metaklassen kommt die tiefe Instanzierung zum Tragen. Direkt unterhalb des Klassennamens befinden sich Attribute und Methoden. Diese stellen den jeweiligen Instanztyp dar, also Struktur und Verhalten der Instanzen. Darunter befinden sich

weitere Attribute und Methoden, diese stellen den jeweiligen Instanz-Instanztyp dar, also Struktur und Verhalten der Instanzen von Instanzen. Durch die tiefe Instanzierung können demnach Attribute und Methode für Klassen unterschiedlicher Ebenen deklariert werden. Die Instanzen der Metaklassen werden in der darunter liegenden Ebene M1 aufgezeigt. Dabei ist ersichtlich, dass die Klasse `Person`, welche eine Wurzelklasse einer Rollenhierarchie darstellt, eine Instanz der Klasse `ObjectWithRoles` ist. Diese enthält sämtliche Methoden für die Instanz-Instanztypen, über die eine Entität einer Rollenhierarchie verfügen muss. Alle dargestellten Rollentypen der Ebene M1 sind direkte oder indirekte Instanzen der Metaklasse `RoleType`. Diese enthält neben allen Rollenmethoden für die Instanz-Instanztypen auch eine Klassenvariable für die Instanztypen. In dieser Metaklasse werden die Vorteile der tiefen Instanzierung sichtbar. Einerseits ist somit ein extensionaler Zugriff auf Rollentypen bzw. qualifizierten Rollentypen möglich und andererseits ist eine kompakte Strukturierung sämtlicher Rolleneigenschaften gegeben.

Das Konstrukt einer tiefen Instanzierung ist jedoch in Ruby nicht umsetzbar, wodurch eine Anpassung der Struktur vorgenommen werden muss.
Das Ergebnis dieser Anpassungen wird in Abbildung 8 abgebildet. Hierbei werden in der Klassenebene M1 Superklassen eingeführt, welche über die nötigen Rollenmethoden verfügen und diese an ihre Subklassen weitervererben. Diese neuen Superklassen zählen dabei nicht zu den Klassen einer Rollenhierarchie, wodurch diese mit Hilfe einer gedanklichen Linie von den Rollenhierarchieklassen getrennt dargestellt werden. Die Klassen bzw. Rollentypen selbst sind dabei weiter Instanzen der Metaklassen aus M2, wodurch weiterhin die volle Metaklassen-Unterstützung gewährleistet wird. Es ergibt sich jedoch der Nachteil der Fragmentierung der Rolleneigenschaften auf zwei Ebenen.

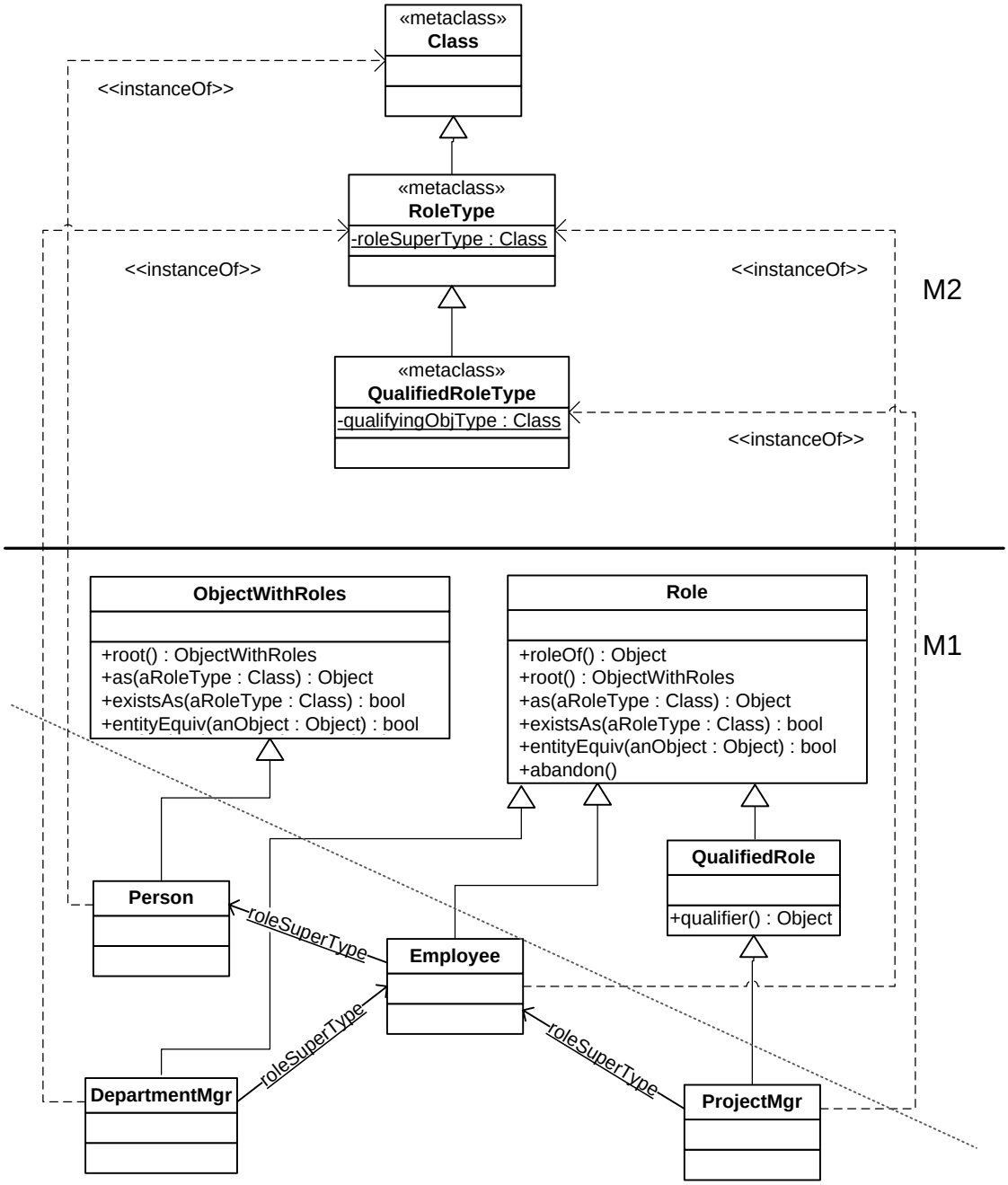

Abbildung 8: Umsetzung des Rollenkonzeptes – Lösung ohne tiefer Instanzierung

Aber auch diese Abbildung zeigt noch keine umsetzbare Lösungsvariante, da dieser in vielen Programmiersprachen nicht implementierbar ist. Deswegen muss dieser auch wiederum teilweise verändert werden. In Ruby besteht die Einschränkung, dass die Metaebene nicht erweitert werden kann. Es kann also keine Klasse von `Class` erben. Lediglich Instanzen dieser Metaklasse können erzeugt werden. Dadurch ergibt sich ein Lösungsansatz der in Abbildung 9 dargestellt wird.

In dieser befindet sich nur die Metaklasse `Class` in der Metaebene, was auch den Voraussetzungen von Ruby entspricht. Die Metaklassen-Eigenschaften, wie die Erzeugung von Rollentypen, werden in dieser Abbildung von den Superklassen in der Ebene M1 übernommen bzw. unterstützt, wodurch diese Superklassen in der Folge als Pseudo-Metaklassen bezeichnet werden. Die inkludierten Pseudo-Metaklassen repräsentieren nun die Instanzen von `Class`. Wird nun beispielsweise ein Rollentyp erzeugt, erfolgt dies über die Methode `defRoleType` der Pseudo-Metaklasse `Role`, welche u. a. die Klassenvariable `roleSuperType` für den neuen Rollentyp setzt. Die Rollentypen bzw. die Wurzelklasse einer Rollenhierarchie sind wiederum Subklassen der erstellten Pseudo-Metaklassen.

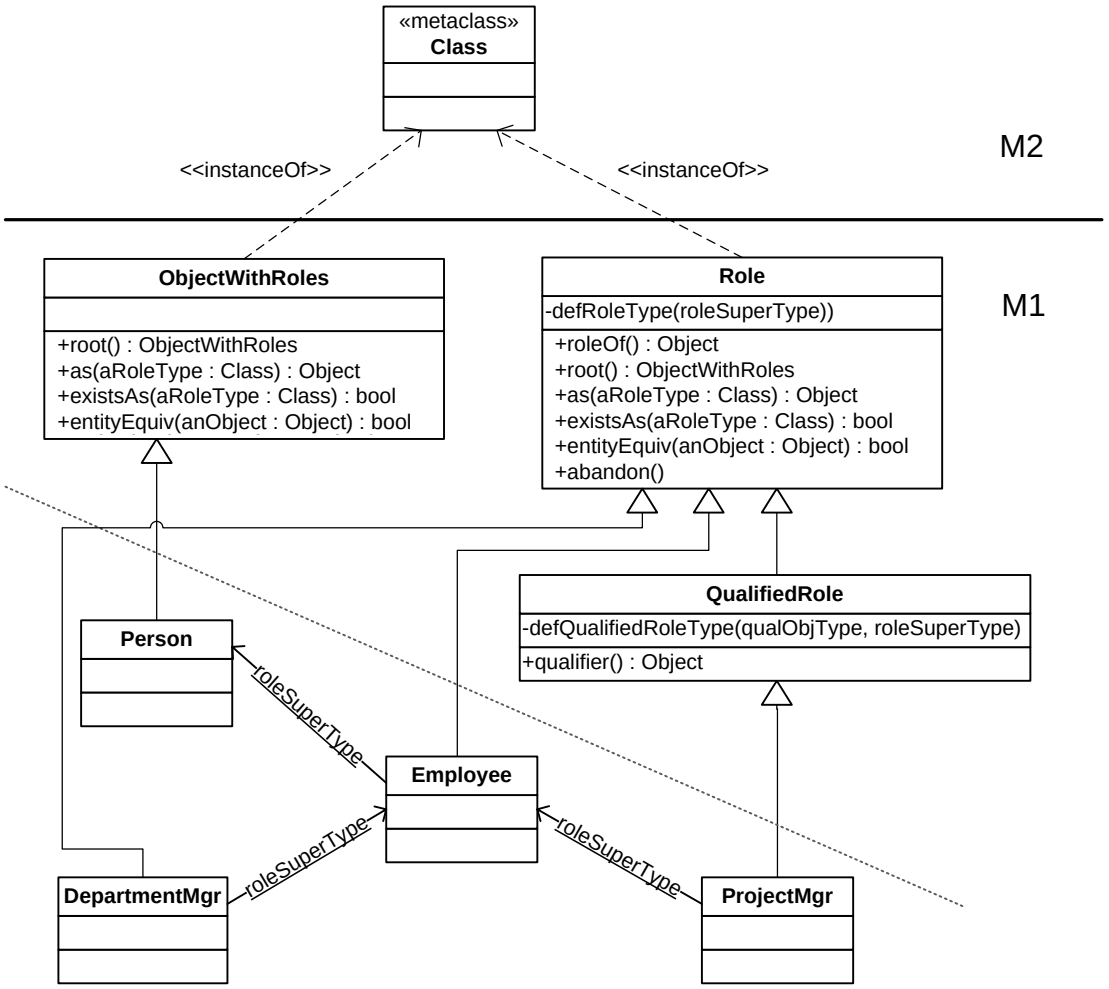

Abbildung 9: Umsetzung des Rollenkonzeptes – Istlösung

3.2.2 Dynamische Generierung von Rollentypen

Um zu Gewährleisten, dass eine Rollenhierarchie jederzeit um weitere Rollentypen erweitert werden kann, soll ein neuer Rollentyp dynamisch in eine Rollenhierarchie hinzugefügt werden können. Somit muss auch gewährleistet sein, dass die Klassenvariable `roleSuperType` eines Rollentyps dynamisch zugewiesen werden kann. Ein Rollentyp soll daher, wie auch in der Abbildung 9 ersichtlich, einerseits von der Klasse `Role` erben und gleichzeitig auf die übergeordnete Rolle bzw. die jeweilige Wurzelklasse verweisen. Bei der Erstellung eines Rollentyps (z.B. `Employee = Role.new(Person,…)`) muss demnach die `new`-Methode überschrieben oder durch eine andere Methode ersetzt werden, wodurch eine einfache Erstellung eines Rollentyps, sowie die Einhaltung des Strukturaufbaus gewährleistet wird. In dieser Implementierung wird die `new`-Methode durch die Methoden `defRoleType`, `defSpecializedRoleType` und `defQualifiedRoleType` ersetzt.

3.2.3 Die Klassenvariable `roleSuperType`

Die Methoden `defRoleType`, `defSpecializedRoleType` und `defQualifiedRoleType` erzeugen grundsätzlich die jeweiligen Rollenklassenkonstrukte mit den übergebenen, übergeordneten Rollenklassen zur Laufzeit und setzen deren Klassenvariable (insbesondere die Variable `roleSuperType`). Somit kann bei der Erstellung der Rolleninstanzen sichergestellt werden, dass jede Rolle an der richtigen Stelle in der Rollenhierarchie eingefügt wird, denn die Einhaltung der Rollenhierarchie ist für die Umsetzung des Rollenkonzeptes von Gottlob et al. [11] von großer Bedeutung. Würden die Rollenklassen auf den üblichen Weg angelegt (wie die Wurzelklasse), könnte die Klassenvariable `roleSuperType` erst nach der Initialisierung gesetzt werden, wodurch eine Überprüfung bei der Initialisierung nicht möglich wäre. Eine weitere Möglichkeit wäre, den `roleSuperType` bei der Initialisierung mitzugeben, was aber eine Überprüfung auf diesen sinnlos machen würde.

Wie bereits in Abschnitt 3.2.1 erläutert, würde eine Rollenklasse idealerweise durch eine Subklasse von `Class` realisiert werden (siehe Listing 4), wodurch eine Metaklassenstruktur erzeugt werden könnte und somit die Klassenvariable `roleSuperType` einfach bei der Initialisierung einer Rollenklasse gesetzt werden könnte.

```
01   class RoleClass < Class
02     def initialize(roleSuperType)
03       @roleSuperType = roleSuperType
04     end
05   end
```

Listing 4: Ideallösung zur Struktureinhaltung

Doch da das Erstellen einer Subklasse von `Class` in Ruby nicht möglich ist, erfolgt die Definition der Klassenvariable `roleSuperType` über die Methoden `defRoleType`, `defSpecializedRoleType` und `defQualifiedRoleType`, wodurch die Struktureinhaltung wiederum über die Klassenvariable gewährleistet wird. Sie stellt somit den Aufbau einer Rollenhierarchie dar (siehe Abbildung 10), wodurch die gewünschte Metaklassenstruktur simuliert werden kann.

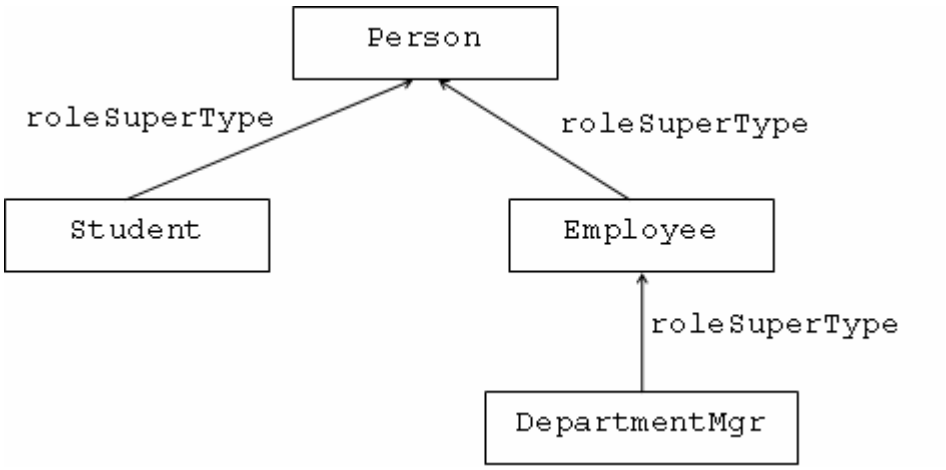

Abbildung 10: Rollenhierarchie

Damit das Attribut `roleSuperType` auf Klassenebene definiert werden kann, darf dieses einerseits nicht als übliche Instanzvariable deklariert werden, da ansonsten der Zugriff bzw. die Zuweisung auf Instanzebene erfolgen müsste. Andererseits kann es auch nicht als Klassenvariable angelegt werden, da sonst die dynamische Zuweisung Probleme mit sich bringt. Denn diese Klassenvariable kann erst zur Laufzeit für das jeweilige Klassenobjekt gesetzt werden.

Um diesem Problem entgegen zu steuern besteht in Ruby die Möglichkeit eine Singleton-Klasse zu einem Objekt zu erzeugen (siehe Abschnitt 2.2.3). Eine Singleton-Klasse ist eine Klasse, zu der nur eine einzige Instanz verfügbar ist. Natürlich kann eine solche Singleton-Klasse auch zu einer anderen Klasse, anstatt eines Objektes, generiert

werden. Es wird also eine Singleton-Klasse auf Klassenebene anstatt auf Instanzebene eingesetzt. Dies erfolgt durch das Codestück aus Listing 5.

```
01   class << Role
02     attr :roleSuperType, true
03   end
```

Listing 5: Erzeugen einer Singleton-Klasse

Zur Erinnerung wird im Detail dabei zur Klasse `Role` eine Singleton-Klasse erzeugt, welche das Attribut `roleSuperType` enthält. Betrachtet man die interne Klassenstruktur von Ruby befindet sich die neu erstellte bzw. erweiterte Singleton-Klasse zwischen der Klasse `Role` und derer bisherigen Metaklasse (hier `Class` – siehe Abbildung 11). Somit fungiert die Singleton-Klasse nun als Metaklasse von `Role` (vgl. Thomas et al. [37]). Ein Grund für diese Positionierung ist, dass dadurch der Aufruf der zusätzlichen Attribute bzw. Methoden Ruby-intern unverändert weiterverfolgt werden kann ohne dabei die Klasse `Role` zu verändern.

Eine derart erstellte Singleton-Klasse in Ruby entspricht einer virtuellen Metaklasse einer anderen Klasse. Zu jeder Klasse kann nur eine Singleton-Klasse bestehen, welche aber zur Laufzeit erweitert oder deren Attribute und Methoden überschrieben werden können (durch wiederholten Aufruf von `class << Role`). Eine weitere Eigenschaft einer Singleton-Klasse in Ruby ist, dass ein direkter Zugriff auf diese Metaklasse nicht möglich ist. Somit können deren Methoden und Attribute nur von den Instanzklassen der Singleton-Klasse aufgerufen werden.

Dadurch ergibt sich die in Abbildung 11 dargestellte Klassenstruktur, die im Anschluss näher beschrieben wird.

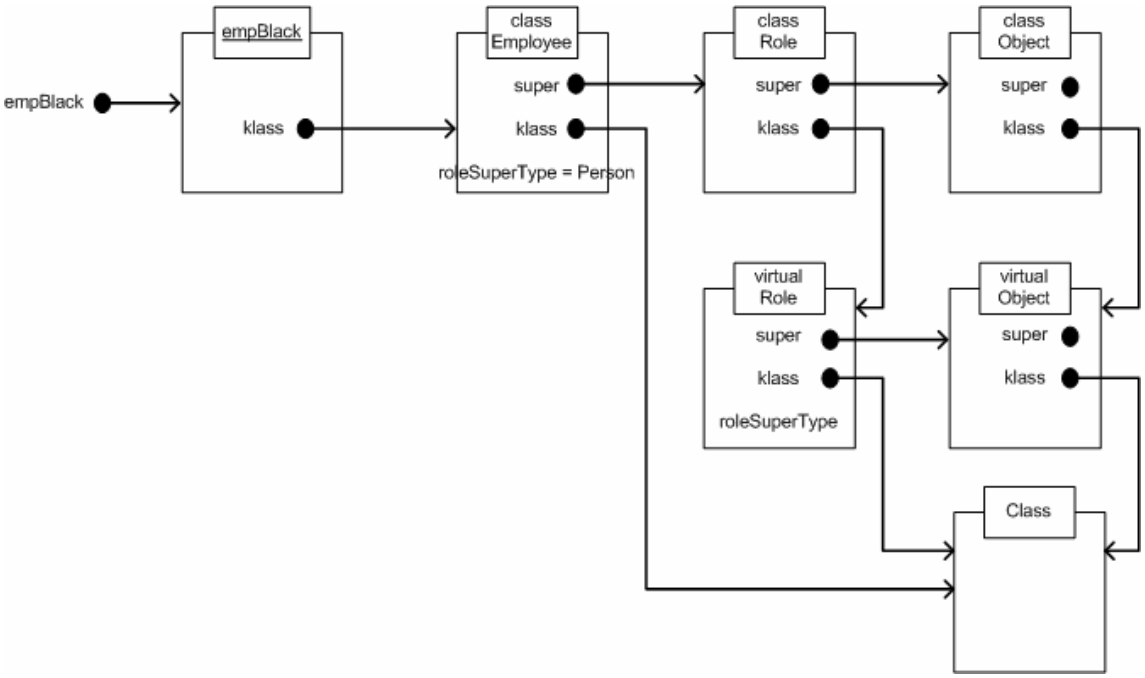

Abbildung 11: Interne Klassenstruktur in Ruby

Das Objekt `empBlack` ist eine Instanz der Klasse `Employee`. Die Klasse `Employee` hingegen ist eine Subklasse von `Role`, welche wiederum eine Subklasse von `Object` ist. Alle diese drei Klassen enthalten grundsätzlich eine Klassenreferenz auf `Class` (siehe Zeiger `klass`). Durch das Ausführen der Anweisung `class << Role` (siehe Listing 5) erzeugt Ruby zur Laufzeit eine virtuelle Klasse zu `Role`. Diese virtuelle Klasse entspricht einer Singleton-Klasse, welche nun das Attribut `roleSuperType` beinhaltet. Weiters generiert Ruby automatisch eine zusätzliche Singleton-Klasse für die Superklasse `Object`, welche daraufhin die Superklasse von `virtual Role` darstellt (vgl. Thomas et al. [37]).

Bei der späteren Initialisierung von `Employee` wird der Wert von `roleSuperType` in `Employee` auf die übergeordnete Klasse „`Person`“ gesetzt. Dabei erbt die Klasse `Employee` das Attribut `roleSuperType` von der Superklasse `virtual Role` über die Klasse `Role`.

Ein Nachteil dieses Lösungsansatzes ist jedoch, dass der Pseudo-Metaklasse `Role` ebenfalls ein `roleSuperType` zugewiesen werden kann, was jedoch nicht sinnvoll ist.

3.2.4 Erstellung eines Wurzelobjektes

Die Wurzelklasse einer Rollenhierarchie ist eine Instanz der Pseudo-Metaklasse `ObjectWithRoles`. Wie aber im vorherigen Kapitel beschrieben wurde, trifft der Ausdruck Metaklasse bei der Implementierung in Ruby nicht ganz zu. Somit entspricht `ObjectWithRoles` einer Superklasse der erzeugten Wurzelklasse. Der erste Codeteil der Klasse `ObjectWithRoles` wird in der folgenden Abbildung dargelegt.

```
01 class ObjectWithRoles
02   include AbstractObjectOrRole
03   attr_reader :roleDictionary
04
05   def initialize()   # constructor
06     @roleDictionary = Hash.new   # initRoleDictionary
07   end   # end initialize
```

Listing 6: ObjectWithRoles – initialize

In diesem Ausschnitt ist ersichtlich, dass die Klasse `ObjectWithRoles` das Modul `AbstractObjectOrRole` inkludiert, welches sämtliche Struktur und Verhalten beinhaltet, die alle Rollenhierarchieklassen vorweisen müssen. Diese Klassen sind neben allen Subklassen von `ObjectWithRoles` auch alle Rollenklassen. Auf dieses Modul wird jedoch erst später näher eingegangen

Weiters beinhaltet die Klasse `ObjectWithRoles` die Instanzvariable `roleDictionary`, in der sämtliche Subrollen einer Instanz eingetragen werden. Im Konstruktor dieser Klasse wird das `roleDictionary` initialisiert, wobei sie als Hash fungiert. Ein Eintrag in diesem Hash sieht folgendermaßen aus: Die Rollenklasse einer Rolleninstanz bezeichnet den Key eines Hash-Eintrags, während die Referenz auf die Rolleninstanz selbst als Value zu diesem Key abgespeichert wird. Bestehen zu einer Rollenklasse mehrere Instanzen, was bei qualifizierten Rollen der Fall sein kann, so werden die jeweiligen Instanzen einer qualifizierten Rollenklasse als Liste im Value abgelegt. Die Initialisierung des Rollenverzeichnisses im Konstruktor entspricht dabei der `initRoleDictionary`-Methode aus Gottlob et al. [11].

Die Klasse `ObjectWithRoles` verfügt auch über einige Methoden, welche die Handhabung die Rollenhierarchie erleichtern soll. Die nachfolgenden Abbildungen zeigen diese Methoden und werden anschließend näher erläutert.

```
09   def getAllSubRoles
10     allRoles = Array.new
11     @roleDictionary.keys.each do |aKey|
12       if @roleDictionary[aKey].class == Array then
13         allRoles += @roleDictionary[aKey]
14       else
15         allRoles.push(@roleDictionary[aKey])
16       end
17     end
18     return allRoles
19   end   # end getAllSubRoles
```

Listing 7: ObjectWithRoles – getAllSubRoles

Die Methode `getAllSubRoles` gibt, wie der Name schon ausdrückt, alle Subrollen des Wurzelobjektes zurück. Grundsätzlich könnte diese mit einer Getter-Methode von `roleDictionary` verglichen werden, was aber nicht ganz zutrifft, da der Rückgabewert dieser Funktion nicht mehr vom Typ Hash ist. Hierbei wird zuerst das Rollenverzeichnis durchlaufen und überprüft, ob der Wert eines Eintrages vom Typ Array ist. Bei solchen Einträgen handelt es sich um qualifizierte Rollen, was bedeutet, dass dieser Wert möglicherweise mehrere Instanzen aufweist, die alle zurückgeliefert werden müssen. Wenn ein solcher Array gefunden wird, werden alle seine Einträge dem späteren Rückgabewert angehängt und der nächste Eintrag des Rollenverzeichnisses wird überprüft.
Handelt es sich bei einem Eintrag jedoch nicht um einen vom Typ Array, muss es sich um eine Rolle handeln, welche in der Liste des Rückgabeparameters eingefügt wird. Dadurch erhält man durch den Aufruf von `getAllSubRoles` eine einfache Liste, welche sämtliche Subrolleninstanzen eines Wurzelknotens zurückliefert.

Als Nächstes wird auf die Methode `addSubRole` eingegangen. Ihre Aufgabe scheint auf den ersten Anblick einfach zu sein, nämlich lediglich das Anfügen einer Rolle im Rollenverzeichnis. Da hier aber auch wiederum Unterscheidungen bei der Speicherung zwischen einfachen Rollen und qualifizierten Rollen gemacht werden müssen, wird diese Methode ebenfalls näher erläutert.

```
20   def addSubRole(role)
21     if role.class.superclass == QualifiedRole then
22       tempArray = Array.new
23       if @roleDictionary[role.class] != nil then
24         tempArray = @roleDictionary[role.class]
25       end
26       tempArray.push(role)
27       @roleDictionary[role.class] = tempArray
28     else
29       @roleDictionary[role.class] = role
30     end
31   end   # end addSubRole
```

Listing 8: ObjectWithRoles – addSubRole

In Listing 8 wird die Implementierung der `addSubRole`-Methode dargestellt. Hier wird ebenfalls, wie in der `getAllSubRole`, zuerst der Rollentyp einer Rolle überprüft. Handelt es sich dabei um eine qualifizierte Rolle, wird die Rolleninstanz im Rollenverzeichnis an der Stelle beigefügt, an der der Key der jeweiligen Rollenklasse entspricht. Dabei werden zuvor alle Einträge im Value – es handelt sich hierbei um eine Liste – gesichert, falls welche vorhanden sind. Anschließend wird die neue Rolleninstanz zu dieser Liste beigefügt und diese als neuer Value gesetzt. Im Gegensatz zu qualifizierten Rolleninstanzen werden einfache Rollen sofort in das Rollenverzeichnis eingefügt. Diese unterschiedlichen Handhabungen des Einfügens einer Subrolle wird von Gottlob et al. [11] mittels den Methoden `recordNewRole` bzw. `recordNewQualifiedRole` erreicht.

Eine weitere Methode der Klasse `ObjectWithRoles` ist die `deleteSubRole`-Methode. Ihre Aufgabe besteht darin, eine mit übergebener Rolleninstanz aus dem Rollenverzeichnis zu entfernen. Listing 9 zeigt ihre Implementierung.

```
32   def deleteSubRole(role)
33     if role.class.superclass == QualifiedRole then
34       tempArray = @roleDictionary[role.class]
35       i = 0
36       tempArray.each do |aQualRole|
37         if role == aQualRole then
38           @roleDictionary[role.class].delete_at(i)
39         end
40         i += 1
41       end
42       if @roleDictionary[role.class] == nil then
43         @roleDictionary.delete(role.class)
44       end
45     else
46       @roleDictionary.delete(role.class)
47     end
48   end
49 end   # end ObjectWithRoles
```

Listing 9: ObjectWithRoles – deleteSubRole(role)

Auch in dieser Methode wird zuvor der Rollentyp auf deren Superklasse überprüft, da auch hier unterschieden werden muss, ob eine Rolleninstanz direkt aus dem Rollenverzeichnis gelöscht werden kann oder ob diese aus der Liste eines qualifizierten Rollentyps entfernt werden muss.

3.2.5 Erstellung einer Rolle

In Gottlob et al. [11] ist jeder Rollentyp eine Subklasse von `RoleType`. Analog, wie bei der Superklasse `ObjectWithRoles`, wird auch die Pseudo-Metaklasse `Role` zur Superklasse umgestaltet. Somit ist jeder Rollentyp eine Subklasse der Klasse `Role`. Zunächst wird auf die Erstellung eines Rollentyps eingegangen und anschließend auf die Erzeugung einer Instanz eines derartigen Rollentyps. Abschließend wird noch auf die einzige Rollenmethode eingegangen, welche lediglich durch eine Rolle ausgeführt werden kann.

Im Gegensatz zu einer Subklasse von `ObjectWithRoles` wird ein Rollentyp jedoch anders angelegt. Während eine Wurzelklasse nach dem üblichen Muster der Klassenerstellung erfolgt (z.B. `class Person`), wird ein Rollentyp über die Klassenmethode `defRoleType` erzeugt (z.B. `Employee = Role.defRoleType(Person)`). Diese Methode übernimmt somit Aufgaben einer Metaklasse und wird u. a. in nachstehender Abbildung gezeigt.

```ruby
01 class Role
02   include AbstractObjectOrRole
03   attr_reader :roleOf
04
05   class << Role
06     attr :roleSuperType, true
07   end
08
09   def Role.defRoleType(roleSuperType, &body)
10     newRoleType = Class.new(Role, &body)
11
12     newRoleType.roleSuperType = roleSuperType
13     return newRoleType
14   end   # end defRoleType
```

Listing 10: Role –übernommene Metaklassen-Aufgaben

In Listing 10 wird die grundsätzliche Metaprogrammierung dargestellt. Durch die Zuweisung `class << Role` (siehe Abschnitt 3.2.3) wird zur Klasse `Role` eine Klasse erzeugt. Diese beinhaltet dabei die Variable `roleSuperType`, welche somit als Klassenvariable fungiert die dynamisch zur Laufzeit gesetzt werden kann.

Wenn nun ein neuer Rollentyp erzeugt werden soll, wird dabei die Klassenmethode `defRoleType` aufgerufen. Diese legt im ersten Schritt eine neue Klasse an, der die Superklasse `Role` zugewiesen wird. Anschließend werden deren `roleSuperType`, welcher sich in der durch `class << Role` erstellten Klasse befindet, für diesen Rollentyp gesetzt und zuletzt der neu generierte Rollentyp zurückgegeben. Eine ähnliche Funktionalität beinhaltet die Klassenmethode `defSpecializedRoleType`. Sie legt einen Rollentyp an, welche von einem anderen erbt. Diese Methode erstellt somit einen Rollentyp zu einem Rollentyp (z.B. `ForeignStudent` von `Student`) und setzt deren `roleSuperType`.

Wird zu diesem neu erzeugten Rollentyp eine Instanz erzeugt, ruft dies den Konstruktor von `Role` auf. Er ist dafür verantwortlich, dass pro Rolle innerhalb der Rollenhierarchie maximal nur eine Instanz erzeugt wird. Ein Wurzelobjekt kann also eine Rolle nur einmal einnehmen. Die Ausnahme bildet dabei eine qualifizierte Rolle, da bei ihrer Identifikation neben der Rolle selbst auch ihr `qualifier` miteinbezogen wird. Der Konstruktor wird in Listing 11 abgebildet und entspricht der von Gottlob et al. [11] definierten Funktion `newRoleType`.

```
14   def initialize(roleOf, *rest)  #Rolleninstanz-Konstruktor
15     if self.class.roleSuperType == roleOf.class then
16       roleExists = roleOf.root.roleDictionary.include?(self.class)
17       if roleExists == false ||
18           self.class.superclass == QualifiedRole then
19         @roleOf = roleOf
20         init(*rest)
21         self.root.addSubRole(self)
22       end
23     end
24   end   # end initialize
```

Listing 11: Role – initialize

Zunächst wird in dieser Methode überprüft, ob die angegebene übergeordnete Rolle auch einer Instanz des `roleSuperTypes` dieser Rollenklasse entspricht. Ist dies der Fall, wird im nächsten Schritt das Rollenverzeichnis dieser Rollenhierarchie nach einer bereits bestehenden Rolle dieses Typs durchsucht. Wenn keine derartige Rolleninstanz gefunden wird und es sich auch nicht um eine qualifizierte Rolle handelt, wird deren `roleOf`-Beziehung gesetzt. Anschließend erfolgen noch die Initialisierung aller Rollentypattribute und die Erweiterung des Rollenverzeichnisses um diese neue Rolleninstanz.

Gottlob et al. [11] beschreiben nur eine einzige Rollenfunktion, über die nur Klassen von `RoleType` verfügen. Dabei handelt es sich um die Methode `abandon`. Sie soll alle Subrollen einer Instanz und anschließend die Instanz selbst aus der Rollenhierarchie entfernen. In Gottlob et al. [11] werden für das Löschen sämtlicher Subrollen die Methoden `cancelRoleAndSubRoles` und `cancelQualifiedRoleAndSubRoles` aufgerufen. Wie in Listing 12 ersichtlich wird hier nicht zwischen dem Löschen von einfachen bzw. qualifizierte Rollen unterschieden, da diese Unterscheidung erst in der zuvor vorgestellten Methode `deleteSubRole` getroffen wird. In `abandon` wird lediglich die Liste aller Subrollen nach den betroffenen Instanzen durchforstet und diese aus der Rollenhierarchie entfernt. Abschließend wird auch die Rolleninstanz selbst entfernt.

```
25   def abandon
26     self.root.getAllSubRoles.each do |tempRole|
27       if tempRole.roleOf == self then
28         self.root.deleteSubRole(tempRole)
29       end
30     end
31     self.root.deleteSubRole(self)
32   end  # end abandon
33 end   # end Role
```

Listing 12: Role – abandon

3.2.6 Erstellung einer qualifizierten Rolle

Die Pseudo-Metaklasse einer qualifizierten Rolle `QualifiedRole` ist eine Subklasse von `Role`. Somit verfügt eine qualifizierte Rollenklasse über sämtliche Funktionalitäten einer einfachen Rolle. Darunter fällt auch die `method_missing` zur Nachrichtenweiterleitung. Die Unterschiede zu `Role` bestehen grundsätzlich in der Erweiterung der Metaprogrammierung. So wird einerseits die zu erstellende Singletonklasse um die Klassenvariable `classOfQualifyingObj` ergänzt. Diese Klassenvariable dient der Überprüfung der Klassenzugehörigkeit des `qualifiers`. Es hat also eine ähnliche Funktion wie die Klassenvariable `roleSuperType`. Auch die Definition von `classOfQualifyingObj` erfolgt wie beim `roleSuperType` bei der Erzeugung des Rollentyps, also in der Klassenmethode `defQualifiedRole` (siehe Listing 13).

```
01 class QualifiedRole < Role
02   attr_reader :qualifier
03
04   class << QualifiedRole
05     attr :classOfQualifyingObj, true
06     attr :roleSuperType, true
07   end
08
09   def QualifiedRole.defQualifiedRoleType(classOfQualifyingObj,
10     roleSuperType, &body)
11     newQualifiedRole = Class.new(QualifiedRole, &body)
12
13     newQualifiedRole.classOfQualifyingObj = classOfQualifyingObj
14     newQualifiedRole.roleSuperType = roleSuperType
15     return newQualifiedRole
16   end   # end defQualifiedRoleType
17   ...
18 end   # end QualifiedRole
```

Listing 13: QualifiedRole

Weiters existieren zu `QualfiedRole` noch der Konstruktor und die Methode `qualifier`. Erstere ist ähnlich dem Konstruktor von `Role`. Zuvor wird die Klassenzugehörigkeit des als übergeordnete Rolle mitgegebene Objektes überprüft, wobei hier auch zusätzlich die Klassenzugehörigkeit des mitgegebenen `qualifiers` kontrolliert wird, und anschließend das Rollenverzeichnis des Wurzelobjektes nach einer solchen, bereits bestehenden Rolle überprüft. Zu dieser Überprüfung wird aber auch zusätzlich der `qualifier` miteinbezogen. Wurde keine derartiger Eintrag gefunden, wird einerseits der Konstruktor der übergeordneten Klasse `Role` aufgerufen und andererseits die Instanzvariable `qualifier` gesetzt. Die zweite Methode, welche `QualifiedRole` noch vorweist ist lediglich die Getter-Methode für die Instanzvariable `qualifier`.

3.2.7 Das Modul AbstractObjectOrRole

Das Modul `AbstractObjectOrRole` wird als Mix-In (siehe Abschnitt 2.2.2) in die Superklassen sämtlicher Klassen einer Rollenhierarchie miteingebunden. Also in den Klassen `ObjectWithRoles` und `Role`. Aber auch in `QualfiedRole`, da diese von `Role` erbt. Dieses Modul enthält neben der Instanzvariablen `roleOf` auch folgende Methoden. Diese entsprechen den Rollenmethoden nach Gottlob et al. [11] (vgl. Kapitel 2.1.1), wobei hier kurz auf deren Implementierung eingegangen wird:

- `roleOf`:

  Dies ist lediglich eine Getter-Methode der Variablen `roleOf`.
- `root`:

  Hierbei wird in der Rollenhierarchie über die `roleOf`-Beziehung solange nach oben navigiert bis der Wurzelknoten erreicht wurde. Dieser wurde erreicht, sobald `roleOf` den Wert `nil` aufweist.
- `entityEquiv(anObject)`

  In dieser Methode werden die Wurzelobjekte von `anObject` und jenes von `self` miteinander verglichen und das Resultat zurückgegeben.
- `as(otherRoleType)`

  Hier wird zuvor der `otherRoleType` mit der Wurzelklasse und anschließend mit allen Subrollenklassen verglichen. Bei einer Übereinstimmung wird die jeweilige Instanz zurückgeliefert.

- `existsAs(otherRoleType)`

  Diese Methode entspricht exakt der `as`-Methode, nur mit dem Unterschied, dass bei einer Übereinstimmung der boolesche Wert `true` zurückgeliefert wird.

- `as_of(aQualifiedRoleType, qualObj)`

  Auch diese Methode funktioniert ähnlich. Hier wird zwar auf die Überprüfung des Wurzelobjektes verzichtet, da es sich dabei keinesfalls um eine qualifizierte Rollen handeln kann, aber auch hier werden die Subrollenklassen und `aQualifiedRoleType` mit denselben Algorithmus verglichen. Zusätzlich wird hier aber auch der `qualifier` mit `qualObj` auf deren Gleichheit überprüft. Rückgabewert ist wiederum die gefundene Instanz.

- `existsAs_of(aQualifiedRoleType, qualObj)`

  Deren Abarbeitung gleicht wiederum der `as_of`-Methode, wobei auch hier bei erfolgreicher Suche der boolesche Wert `true` zurückgegeben wird.

3.2.8 Struktur und Verhalten innerhalb der Rollenhierarchie

Da in einer Rollenhierarchie zwischen den Rollen keine Vererbung stattfindet, wohl aber eine Weiterleitung von Nachrichten, muss dies auch in dieser Implementierung der Fall sein. Ruby unterstützt dies mittels der `method_missing`-Methode, welche in Abschnitt 2.2.1 bereits erläutert wurde. Die Umsetzung der Nachrichtenweitergabe anhand dieser Methode ist gleichermaßen simpel wie effektiv. Mit Hilfe der drei Codezeilen aus Listing 14 wird diese grundsätzlich komplizierte Funktionalität implementiert. Die Überschreibung der `method_missing` befindet sich dabei in der Klasse `Role`. Wird demnach ein Attribut bzw. eine Methode nicht gefunden, wird die `method_missing`-Methode aufgerufen, welche die gesuchte Methoden-Id zur übergeordneten Rolle weiterleitet.

```
01  def method_missing(methId, *rest)
02    @roleOf.send(methId, *rest)
03  end
```

Listing 14: method_missing v1.0

3.3 Evaluierung

Die Umsetzung des Rollenkonzeptes von Gottlob et al. [11] in der objektorientierten Programmiersprache Ruby wurde hier auf prototypische Programmierweise vollzogen, wodurch eine Implementierung entstand, welche sämtliche Anforderungen von Gottlob et al. [11] entspricht. Dieses Rollenkonzept wurde bereits in anderen Programmiersprachen implementiert. In diesem Unterkapitel sollen dabei die Unterschiede zwischen der in diesem Kapitel implementierten Ruby-Lösung und den Lösungen in Smalltalk (von Gottlob et al. [11]) und Java (Schrefl und Thalhammer [35]) dargebracht werden.

3.3.1 Vergleich mit Smalltalk-Lösung

Gottlob et al. [11] implementierten ihr Rollenkonzept für die objektorientierte Programmiersprache Smalltalk. Zur Verwendung ihres Rollenkonzeptes müssen in Smalltalk lediglich drei Klassen inkludiert werden. Dies sind die Klassen `ObjectWithRoles`, `RoleType` und `QualifiedRoleType`. Im Gegensatz zur Implementierung aus Kapitel 3.2 verfügt die Klasse `ObjectWithRoles` über interne Methoden, welche vom Anwender nicht aufrufbar sind. Diese Methoden sind `initRoleDictionary` (zur Initialisierung des Rollenverzeichnisses), `recordNewRole` und `recordNewQualifiedRole` (zur Speicherung einer neu eingenommenen Rolle) bzw. `cancelRoleAndSubroles` und `cancelQualifiedRoleAndSubrols` (um Rollen und deren Subrollen aus dem Rollenverzeichnis zu entfernen). Diese Methoden werden von den, für den Anwender aufrufbaren, Methoden `newRoleOf`, `newRoleOf:qualifiedBy` (zur Erzeugung einer neuen Rolle zu einem Objekt) bzw. `cancelRole` und `cancelQualifiedRole` (zum Löschen von Rollen) aufgerufen. Ihre Funktionalitäten werden in der Ruby-Implementierung einerseits von den `initialize`-Methoden und andererseits von `addSubRole` bzw. `deleteSubRole` übernommen.

Eine weitere Unterscheidung besteht in den Variablen der Klassen. Während in der Smalltalk-Lösung die Klasse `RoleType` neben `roleOf` auch eine Referenz auf das Wurzelobjekt enthält, wird das Wurzelobjekt in der Ruby-Lösung durch einen Methodenaufruf ermittelt. In der Smalltalk-Lösung wird die Variable `roleOf`

hauptsächlich für das Weiterleiten von Nachrichten verwendet und die Variable `root` für die Aufgabenbewältigung der Rollenmethoden. In der Ruby-Lösung steht hingegen die `roleOf`-Referenz im Mittelpunkt, wobei neben den Rollenmethoden auch `root` über diese Variable ermittelt wird.

Die Funktionalität des Method Dispatchings ist jedoch in beiden Lösungen identisch. Während die Ruby-Lösung eine Nachricht anhand der `method_missing`-Methode auffängt und über die `roleOf`-Referenz weiterleitet, geschieht in Smalltalk derselbe Vorgang mit der `DoesNotUnterstand`-Methode, welche mit der `method_missing`-Methode gleichzusetzen ist.

Auch das Anlegen einer neuen Rollenklasse ist ähnlich. Beide Lösungen erzeugen Subklassen zu einer Rolle (`RoleType` bzw. `Role` - der Grund für die unterschiedliche Bezeichnung liegt darin, dass Gottlob et al. [11] den Metaklassen-Aspekt hervorhebt) und setzen deren `roleSuperType`, welcher u. a. als Parameter übergeben wird. Während in der Smalltalk-Lösung die Klassenvariable `roleSuperType` hier auch erst deklariert wird, erfolgt in der Ruby-Lösung lediglich die Definition dieser Klassenvariable, welche von der Klasse `Role` geerbt wird.

Ein weiterer Unterschied besteht in der Generierung einer Subklasse zu einem Rollentyp. Gottlob et al. [11] erzeugen solche Rollentypen ebenfalls über die Methode `defRoleType`, während in der Ruby-Lösung dies über die Methode `defSpecializedRoleType` erstellt wird.

Die Ruby- und die Smalltalk-Implementierung sind demnach sehr ähnliche Lösungen, da auch Ruby und Smalltalk sehr viele Gemeinsamkeiten vorweisen.

3.3.2 Vergleich mit Java-Lösung

Aufgrund des Typisierungs-Unterschiedes zwischen den Programmiersprachen Ruby und Java sind auch die Lösungen zur Umsetzung des Rollenkonzeptes von Gottlob et al. [11] verschieden. Durch die dynamische Typisierung von Ruby ist das Versenden von Nachrichten an unterschiedliche Objekten, also das Method Dispatching, schnell und

einfach umzusetzen. Java unterstützt hingegen Typsicherheit, wodurch der Typ eines Objektes zur Kompilierzeit bekannt sein muss.

Schrefl und Thalhammer [35] entwickelten ein Java Package, welches zur Verwendung eines Rollenkonzeptes genutzt werden kann. Dieses beinhaltet grundsätzlich wiederum die drei Klassen `ObjectWithRoles`, `RoleType` und `QualifiedRoleType`. Ihr Rollenverhalten wird dabei in einem Interface namens `RoleProtocol` definiert.

Wenn dabei eine Rolle erzeugt werden soll, wird eine Java-Klasse erstellt, die von der Klasse `RoleType` erbt. Die Rollenbeziehung wird hier ebenfalls über die Instanzvariable `roleOf` beschrieben. Um diese Instanzvariable zu setzen, muss jede Rollenklasse einen Konstruktor vorweisen, der einerseits als Parameter den `roleSuperType` übergibt und andererseits die `super`-Methode mit dem übergeordneten Objekt ausführt. Die Ruby-Lösung ist in dieser Hinsicht flexibler, da sie keine Anweisungen bezüglich Inhalte des Konstruktors vorgibt.

Wie auch in der Java-Lösung von Schrefl und Thalhammer [35] besteht in der Ruby-Lösung eine Methode `getRoot` bzw. `root` für die Klassen `RoleType` bzw. `Role` und `ObjectWithRoles`, um das Wurzelobjekt einer Rollenhierarchie zu ermitteln. Dies wird in der Smalltalk-Lösung hingegen über eine Instanzvariable festgehalten. Es bestehen jedoch Unterschiede bei den anderen Rollenmethoden, wie `as` und `existsAs`. In der Java-Lösung sind deren mitgegebene Parameter vom Typ String, während in der Ruby-Lösung diese Klassen darstellen. Auch der Rückgabewert der Rollenmethoden `as` und `entityEquiv` sind verschieden. Während in der Ruby-Lösung der Wert `nil` zurückgegeben wird, wirft die Java-Lösung entsprechende Exceptions.

Der wohl schwerwiegendste Unterschied der Lösungen liegt beim Wechseln von einer Rolleninstanz zu einer anderen. Die Ruby-Lösung behilft sich dabei mit der Möglichkeit des Method Dispatching, welches implizit abläuft. Während in Java ein Rollenwechsel explizit angegeben werden muss, da Java keine dynamische Typisierung unterstützt. Hierbei muss ein explizites Typecasting vorgenommen werden (z.B.: `((Employee)deptMgrBlack.as(„Employee“)).salary()`). In dieser Hinsicht erlaubt Ruby eine flexiblere Lösung als Java.

Java definiert nach Schrefl und Thalhammer [35] eine so genannte „primary inheritance hierarchy“. Um diese Hierarchie um Rollen zu erweitern ohne sie jedoch zu ändern, wird in der Java-Lösung eine Proxy-Klasse für Klassen, welche Subklassen von `ObjectWithRoles` darstellen sollen, eingeführt. Diese ist für alle rollenspezifischen Aufgaben, wie das Erstellen neuer Rollen, einem Rollenwechsel etc., verantwortlich. Wenn sich beispielsweise die Klasse `Person` in einer Klassenhierarchie befindet und dieser Klasse eine Rolle zugewiesen werden soll, wird ihr eine derartige Proxy-Klasse angehängt. Dadurch bleibt die bestehende Klassenhierarchie unverändert und die Klasse `Person` erhält trotzdem Rolleneigenschaften.

4 Rollen in dezentralen Informationssystemen

Für die Verwendung von Rollen bzw. eines Rollenkonzeptes in dezentralen Informationssystemen müssen bestimmte Anforderungen (siehe Abschnitt 2.4) erfüllt werden. Aufgrund dieser Anforderungen wird ein erweitertes Rollenmodell entwickelt (siehe Abschnitt 4.1). Die Verwendung dieses erweiterten Rollenmodells wird anschließend in Abschnitt 4.2 vorgestellt. In Abschnitt 4.3 wird abschließend die Implementierung des erweiterten Rollenmodells beschrieben und detailliert auf die Veränderungen zur Implementierung aus Kapitel 3 näher eingegangen.

4.1 Entwicklung eines erweiterten Rollenmodells

Aufgrund der Einschränkungen und Möglichkeiten, welche ein dezentraler Einsatz des Rollenkonzeptes von Gottlob et al. [11] mit sich bringt, muss das Konzept angepasst werden. Diese Einschränkungen und Möglichkeiten werden in den folgenden Unterkapiteln beschrieben und die daraus resultierenden, konzeptionellen Änderungen dargelegt.

4.1.1 Einführung der Beziehung `roles`

Zugriffsbeschränkungen durch andere Systeme auf einzelne Knoten einer Rollenhierarchie können die Navigation erheblich beeinträchtigen. Insbesondere Einschränkungen des Zugriffs auf das Wurzelobjekt kann beim Rollenkonzept von Gottlob et al. [11] beträchtliche Auswirkungen haben, da hier das Rollenverzeichnis mit sämtlichen eingenommenen Rollentypen hinterlegt ist. Kann auf dieses Verzeichnis gar nicht zugegriffen werden, würde dies eine Navigation von oben nach unten gänzlich unmöglich machen. Wenn jedoch nur der Schreibzugriff verweigert wird, würde zwar eine Navigation möglich sein, dennoch bringt dies Probleme mit sich. Würde beispielsweise eine neue Rolleninstanz von dieser Entität eingenommen, könnte deren Rollentyp nicht in das Rollenverzeichnis mit aufgenommen werden, wodurch später bei der Top-Down-Navigation diese neue Rolleninstanz nicht mitberücksichtigt werden würde.

Um die hier erwähnte Abhängigkeit vom Wurzelobjekt etwas zu schwächen wird anstelle eines Rollenverzeichnisses eine weitere Beziehung zwischen zwei Konten in der Rollenhierarchie eingeführt (siehe Abbildung 12). Diese neue Beziehung wird `roles` genannt und ist die inverse Beziehung zu `roleOf`.

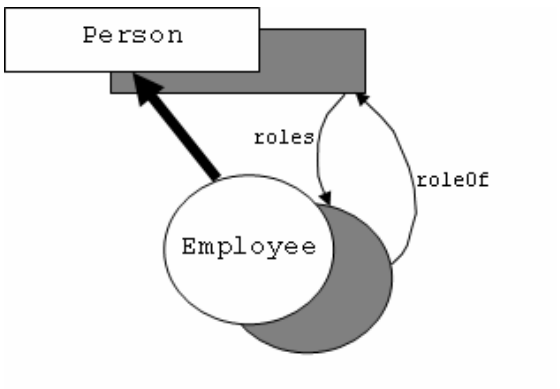

Abbildung 12: Einführung der Beziehung roles

Im Grunde ist die `roles`-Beziehung ähnlich eines Rollenverzeichnisses. Es ist eine 1-zu-n-Beziehung zwischen Rollen, was in objektorientierten Systemen auch als Auflistung aller Subrollen angesehen werden kann. Somit handelt es sich bei der `roles`-Beziehung um eine Liste. Im Gegensatz zum Rollenverzeichnis, welches als Hashmap fungiert und deren Einträge Rollentypen (Key) mit den dazugehörigen Rolleninstanzen (Values) darstellen, sind die Elemente in `roles` lediglich Rolleninstanzen. Eine weitere Unterschied besteht darin, dass nicht nur das Wurzelobjekt, sondern jede Instanz in der Rollenhierarchie über eine solche Liste verfügen kann. Auch die zu inkludierenden Listenelemente unterscheiden sich. Während im Rollenverzeichnis sämtliche eingenommenen Rollentypen und deren Rolleninstanzen enthalten sind, werden in der `roles`-Beziehung lediglich die direkten Subrolleninstanzen aufgenommen. In Abbildung 13 werden diese Unterschiede veranschaulicht, wobei `roles`-Beziehung als Listen zu den jeweiligen Objekten dargestellt werden.

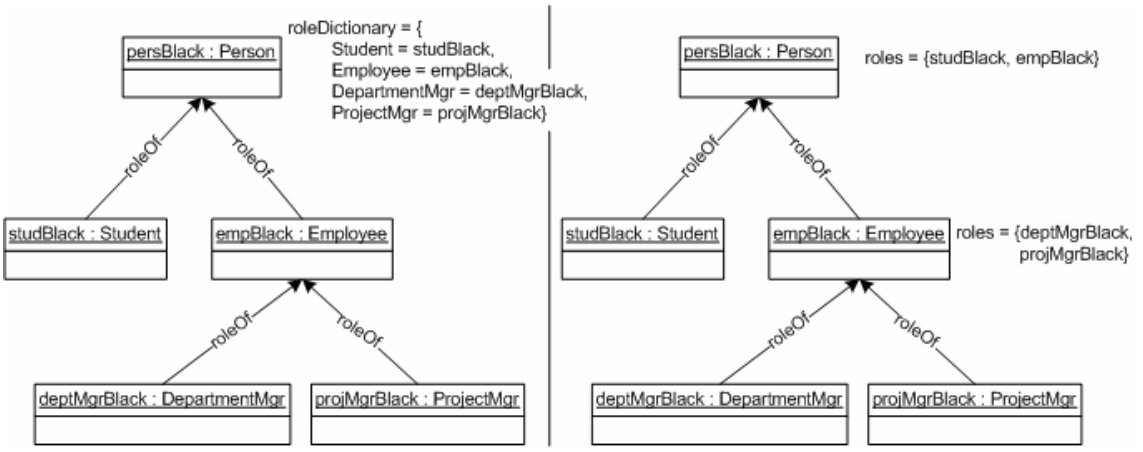

Abbildung 13: Unterschied zwischen roleDictionary und roles

Wenn nun der Zugriff auf das Wurzelobjekt verweigert wird, steht zwar die Liste deren Subrollen nicht zur Verfügung, dafür aber die Liste derer bei denen der Zugriff gestattet ist. Das bedeutet, dass die Top-Down-Navigation innerhalb der Rollenhierarchie zumindest teilweise zur Verfügung steht.

4.1.2 Einführung der Klasse `ObjectOrRole`

Da im dezentralen Einsatz die Möglichkeit gegeben werden muss auch Teilhierarchien zu erstellen und diese anschließend zusammenzuführen, muss dies auch in der Implementierung berücksichtigt werden. Um dabei der Änderung der Klassenzugehörigkeit zu umgehen, wird eine einfache, jedoch effektive Anpassung des Rollenkonzeptes vorgenommen. Es werden schlicht die Klassen `ObjectWithRoles` und `Role` zu einer gemeinsamen Klasse namens `ObjectOrRole` zusammengeführt. Somit ist jede Klasse in einer Rollenhierarchie eine Subklasse von `ObjectOrRole` (siehe Abbildung 14 im Vergleich zu Abbildung 9 aus Kapitel 3.2).

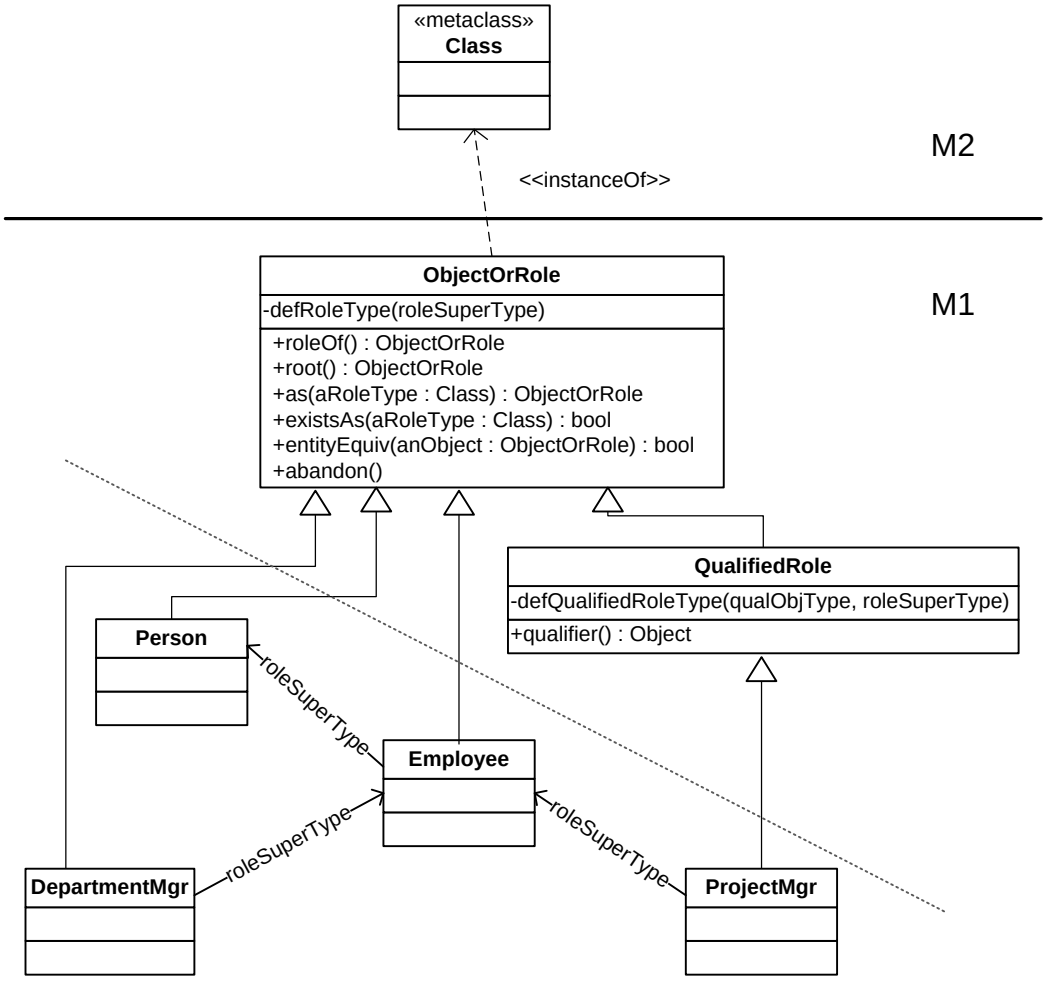

Abbildung 14: Zusammenführung zur Metaklasse ObjectOrRole

Ein Wurzelobjekt ist dadurch nur daran zu erkennen, dass es keine übergeordnete Rolle aufweist. Sie besitzt also keine `roleOf`-Beziehung. Wird nun eine Teilhierarchie zu einer anderen angehängt, muss lediglich die `roleOf`-Beziehung des Wurzelobjektes der Teilhierarchie auf deren neue, übergeordnete Rolleninstanz gesetzt werden. Weiters muss durch den Wegfall des Rollenverzeichnisses (siehe Einführung der Beziehung `roles`) auf dieses keine Rücksicht genommen werden. Stattdessen muss aber die `roles`-Beziehung der übergeordneten Rolleninstanz um die neue Subrolleninstanz erweitert werden, was aber im Gegensatz zur Verschiebung des `roleDictionary` einfacher zu bewerkstelligen ist.

4.1.3 Upward Inheritance

Eine Rollenhierarchie sollte nicht nur anhand des Top-Down-Prinzips erstellt werden können, sondern aus durch das Bottom-Up-Prinzip. Um die Erstellung einer Rollenhierarchie anhand des Bottom-Up-Prinzips zu gewährleisten, muss eine Art Generalisierung möglich sein. Diese wird durch einen Ansatz der Upward Inheritance erzielt, welcher an jenen von Schrefl und Neuhold [34] angelehnt ist. Dabei muss eine neue, übergeordnete Rolleninstanz mithilfe von ein oder mehreren Rolleninstanzen erstellt werden können. Damit dies erreicht werden kann, muss die Initialisierung eines Rollenobjektes um eine derartige Funktionalität erweitert werden.

Ein weiterer Schritt der Upward Inheritance besteht in der Weiterleitung von Nachrichten zu den Subrollen, da die Superrolle auch Attribute und Methoden ihrer Subrollen ausführen kann, diese aber nicht direkt erbt. Bisher wurden Nachrichten, wie Methodenaufrufe, nur zur übergeordneten Rolleninstanz weitergereicht, was auch im Sinne des Rollenkonzeptes von Gottlob et al. [11] ist. In Ruby erfolgt dabei das Method Dispatching (siehe Abschnitt 2.2.1) mittels der `method_missing`-Methode. In Kapitel 3.2 hatte diese Methode dabei die Aufgabe eine Nachricht an die übergeordnete Rolle in einer Rollenhierarchie weiterzuleiten. Die Implementierung der Upward Inheritance muss dabei auch in der `method_missing`-Methode mit eingebunden werden. So muss diese Methode derart geändert werden, dass die Nachricht zuerst in die Sub-Hierarchie der jeweiligen Rolleninstanz weitergeleitet werden, bevor die Nachricht erst zur übergeordneten Rolleninstanz weitergereicht wird.
Durch eine derartige Anpassung des Method Dispatching kann also eine Upward Inheritance erzeugt werden, welche eine Generalisierung imitiert.

4.1.4 Mapping-Ansatz

Aufgrund der Tatsache, dass unter anderem nicht gewährleistet werden kann, dass in allen Subklassen für dasselbe Attribut der gleiche Name vergeben wird, muss in den durch Upward Inheritance erzeugten Klassen ein Mapping-Verfahren zur Verfügung stehen.

Das Mapping-Verfahren dieser Diplomarbeit ist an jenes von Schrefl und Neuhold [34] bzw. Klas und Schrefl [14] angelehnt. Dabei ist für jede Generalisierungsklasse eine

Corresponding-List von Nöten, welche für jedes Mapping-Attribut bzw. -Methode einen Eintrag über deren Navigierbarkeit aufweist. Schrefl und Neuhold [33] bezeichnen diese Corresponding-List als Protocol. Sobald ein Attribut- bzw. Methodenaufruf über die Generalisierungsklasse erfolgt und dieser nicht direkt über das Objekt selbst aufgerufen werden kann, wird die Corresponding-List nach einen solchen Eintrag durchsucht. Bei erfolgreicher Suche werden die entsprechenden Attribute bzw. Methoden in den untergeordneten Objekten aufgerufen und dementsprechend abgearbeitet, damit für das übergeordnete Objekt das gewünschte Ergebnis zurückgeliefert werden kann.

Beispiel:

Es soll das gesamte Einkommen von `personBlack` ermittelt werden, wobei `personBlack` mehrere Rollen einnimmt bei denen er ein Gehalt bezieht (siehe Abbildung 15). Darüber hinaus verwenden die einzelnen Klassen unterschiedliche Attributnamen für das Gehalt (`salary` für `Employee` und `sal` für `DepartmentMgr`). Für das dafür benötigte Mapping-Verfahren muss also eine Corresponding-List für die Klasse Person bestehen. Diese Corresponding-List muss weiters einen Eintrag enthalten, der für das Attribut `income` eine dementsprechende Abarbeitung der Attribute `Employee.salary` und `DepartmentMgr.sal` bereitstellt.

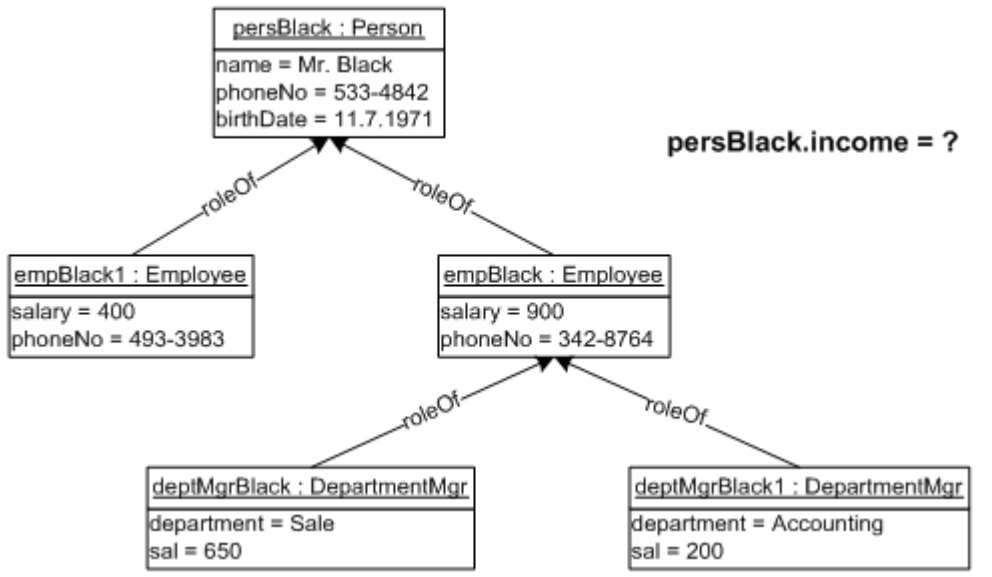

Abbildung 15: Beispiel für das Mapping-Verfahren

Jeder Eintrag in der Corresponding-List beinhaltet auch ein so genanntes Relationship-Flag, welches Auskunft über die Verarbeitung der Subklassen-Attribute bzw. -Methoden liefert, damit die Super-Klasse das gewünschte Ergebnis zurückliefert. Hierzu gibt es vier Ausprägungen für das Relationship-Flag:

- ID-Relationship:
  Dabei wird davon ausgegangen, dass die jeweiligen Attribute bzw. Methoden aller Subklassen denselben Wert zurückgeben (z.B. Name). Somit wird in der Generalisierungsklasse nur einer dieser Werte zurückgegeben.
- ALL-Relationship:
  Bei dieser werden alle Werte der entsprechenden Attribute bzw. Methoden aller Subklassen aufgelistet und diese Liste bei der ausgeführten Klasse zurückgeliefert (z.B. alle geschäftlichen und privaten Telefonnummern einer Person).
- SUM-Relationship:
  Dadurch werden alle Werte der jeweiligen Attribute bzw. Methoden aller Subklassen aufsummiert und bei der übergeordneten Klasse zurückgegeben (z.B. Summe aller bezogenen Gehälter einer Person – wie im obigen Beispiel).
- GENERAL-Relationship:
  Deren Abarbeitung soll je nach Bedarf individuell implementiert werden. Es handelt sich hierbei lediglich um einen Platzhalter für weitere Verarbeitungsmöglichkeiten.

Diese Arten von Relationships unterscheiden sich jedoch zu denen aus der Literatur. So wurde beispielsweise das HISTORY-Relationship von Klas und Schrefl [14] nicht miteinbezogen. Dieses Relationship soll nur die aktuellsten Daten zur Berechnung des Rückgabewertes verwenden. Da aber die Daten direkt von den jeweiligen Speicherorten eingelesen werden, wird davon ausgegangen, dass diese immer am aktuellen Stand sind. Jedoch ist dem Anwender dieser Arbeit freigegeben dieses Relationship anstelle der GENERAL-Relationship zu implementieren.

Mithilfe des Corresponding List-Eintrags aus Abbildung 16 kann nun geschlossen werden, dass der Rückgabewert von `personBlack.income` sich aus der Summe (durch den Flag-Wert 2) aller Instanzmethodenaufrufe von `Employee.salary` und

`DepartmentMgr.sal` ergibt, welche `personBlack` einnimmt. Somit ergibt der Aufruf `personBlack.income` den Wert 2150 (900 + 400 + 650 + 200).

```
Person.correspondingList.element = {
                "income",
                2,
                Employee, "salary",
                DepartmentMgr, "sal" }
```

Abbildung 16: Eintrag einer Corresponding-List

4.1.5 Einführung einer Private-List

Nach Gottlob et al. [11] ist eine Weiterleitung von Nachrichten zwischen Rollentypen von der Sub- zur Superrolle möglich. Nach der Implementierung der bisher vorgestellten Erweiterungen bzw. Anpassungen ist eine derartige Weitergabe auch in entgegen gesetzter Richtung möglich. Um jedoch die Verwendungsmöglichkeiten zu erweitern soll eine Weiterleitung von Attributen auch verhindert werden können, um Attribute somit „privat“ zu halten. Private Attribute sollen dabei nur durch ihre Instanzen aufgerufen werden können.

Wenn jedoch auf einen Attributwert nur von derer eigenen Rolleninstanz zugegriffen werden soll, muss ein derartiges Attribut gekennzeichnet werden. Diese Markierung kann einerseits über ein Flag für jedes Attribut erfolgen oder aber auch über eine Liste, welche solche Attribute beinhaltet. Diese Liste wird als Private Liste bezeichnet. Jeder Rollentyp soll über eine derartige Liste verfügen, die beim Method Dispatching Vorgang durchlaufen werden soll.

4.2 Verwendung von erweiterten Rollen in Ruby

In diesem Abschnitt wird die Verwendung des erweiterten Rollenmodells beschrieben. Zu Beginn wird auf das Anlegen einer Rollenklasse und anschließend auf deren Instanzen eingegangen. In den Abschnitten 4.2.3 und 4.2.4 wird abschließend die Verwendung der Corresponding List bzw. der Private List dargestellt.

4.2.1 Anlegen von Rollenklassen

Rollenklassen werden, wie auch in Kapitel 3 mittels einer der Methoden `defRoleType`, `defSpecializedRoleType` oder `defQualifiedRoleType` deklariert. Die Ausnahme davon ist die Wurzelklasse einer Rollenhierarchie. Diese kann auf die übliche Art beschrieben werden, wobei sie von `ObjectOrRole` erben muss. Der Grund dafür liegt in der Definition der Klassenvariable `roleSuperType`, denn diese muss bei der Wurzelklasse nicht gesetzt werden. Alle anderen Hierarchieklassen werden, wie in Listing 15 beschrieben, angelegt, wodurch beim Aufruf von z.B. `Employee.new(...)` zuvor die Methode `defRoleType` der Klasse `ObjectOrRole` aufgerufen wird. Dadurch wird das Klassenkonstrukt der Rolle erzeugt und anschließend die Funktionalität der Methode `new` abgearbeitet.

```
01 Employee = ObjectOrRole.defRoleType(Person, true) do
02   attr_reader :salary, :phoneNo
03   ...
04 end
```

Listing 15: Beschreibung einer Rollenklasse

Der Unterschied bei der Erstellung von Rollenklassen gegenüber der Erstellung aus Abschnitt 3.1.1 liegt, neben der Verwendung der Pseudo-Metaklasse `ObjectOrRole` anstelle von `Role`, im zusätzlichen Übergabeparameter. Dieser Parameter beschreibt das Flag `isSingleInstRole`. Diese Flag gibt Auskunft darüber, ob eine Rollenklasse innerhalb einer Rollenhierarchie mehrere Instanzen aufweisen darf. Eine derartige Rollenklasse entspricht einer qualifizierten Rolle ohne qualifier. Nach Gottlob et al. [11] darf dies zwar nicht der Fall sein, jedoch sollte in Bezug auf den dezentralen Einsatz diese Möglichkeit in Betracht gezogen werden. Wird das Flag auf `true` gesetzt, kann nur eine Instanz zu einer Rollenklasse existieren, wodurch keine weiteren Instanzen erzeugt werden können.

4.2.2 Anlegen von Rolleninstanzen

Bei der Erzeugung von Instanzen sämtlicher Rollenklassen werden verschiedene Möglichkeiten zur Verfügung gestellt. Diese Möglichkeiten beziehen sich grundsätzlich auf den Zeitpunkt ihrer Erstellung. Da aufgrund der Anforderungen des dezentralen Einsatzes nicht davon ausgegangen werden kann, dass die Erstellung der gesamten

Rollenhierarchie nach dem Top-Down-Prinzip erfolgt, muss auch sichergestellt werden, dass eine Rolleninstanz auch nach dem Bottom-Up-Prinzip erzeugt werden kann. Natürlich muss auch gegeben sein, dass eine Rolleninstanz alleine existieren kann, also ohne in einer Hierarchie eingebettet zu sein. Dies ist notwendig, da bei der Erstellung einer neuen Hierarchie der erste Knoten ebenfalls erzeugt werden muss.

Nach welcher dieser drei Arten eine Rolleninstanz erstellt wird, richtet sich nach dem ersten Übergabeparameter beim Aufruf der Funktion `new`. Entspricht der erste Übergabeparameter einer anderen Rolleninstanz wird diese als übergeordnete Rolle betrachtet und das Attribut `roleOf` der zu erzeugenden Rollenobjektes auf diese übergeordnete Rolle gesetzt (siehe Listing 16, Zeile 01).

Besteht der erste Parameter aus einem Array von mehreren Rollenobjekten. Dies wird derart interpretiert, dass die zu erzeugende Rolleninstanz die übergeordnete Rolle darstellt und sämtliche Rollenobjekte in der Liste deren Subrollenobjekte repräsentieren (siehe Listing 16, Zeile 03). Dadurch werden die `roleOf`-Attribute der Subrollen auf die neu erstellte Rolle gesetzt.

Handelt es sich beim ersten Parameter um einen anderen Wert, wird davon ausgegangen, dass diese Rolleninstanz keiner Rollenhierarchie angehört und somit isoliert besteht (siehe Listing 16, Zeile 04).

```
01 empBlack = Employee.new(personBlack, 900, '342-8764')

02 roleArray = Array.new(deptMgrBlack, projMgrBlack)
03 empBlack = Employee.new(roleArray, 900, '342-8764')

04 empBlack = Employee.new(900, '342-8764')
```

Listing 16: Möglichkeiten zur Erstellung von Rolleninstanzen

4.2.3 Verwendung der Corresponding List

Die Corresponding List ist Ausgangspunkt für einen Mapping-Ansatz zur Bewältigung von Synonymen bei Attributen und Methoden innerhalb einer Rollenhierarchie. Wenn beispielsweise in einer Rollenhierarchie für dieselben Attribute unterschiedliche Bezeichnungen verwendet worden sind (zB. „sal“ und „salary“), können diese Attribute durch einen Corresponding List Eintrag zusammengefasst werden. Sollen beispielsweise, aus dem Beispiel in Abschnitt 1.4, alle Einkommen der Person Black aufgelistet werden, muss der Benutzer folgende Aufrufe sebstständig tätigen: `studBlack.sship`, `empBlack.salary` und `deptMgrBlack.sal`. Bestehen

zu einer Rollenklasse mehrere Instanzen, muss er jede davon einzeln aufrufen. Durch einen Eintrag in der Corresponding List kann dies automatisch erfolgen. Zusätzlich können über die Corresponding List auch Angaben zur Bearbeitung der Attribute gemacht werden. In der Implementierung des erweiterten Rollenmodells werden drei unterschiedliche Bearbeitungsarten unterstützt, welche durch den Programmierer erweitert werden kann. Diese sind:

- Ausgabe eines Attributwertes bzw. Rückgabe einer Methode,
- Auflistung aller Attributwerte bzw. Methodenrückgabewerte und
- Summierung aller Attributewerte bzw. Methodenrückgabewerte.

Für das angeführte Beispiel ist folgender Corresponding List Eintrag nötig (siehe Listing 17). Durch den Aufruf von `persBlack.income` wird dasselbe Ergebnis erzielt wie zuvor.

```
corrListPers.push(CorrClass.new("income", 1,
                                Employee, "salary",
                                Student, "sship",
                                DepartmentMgr, "sal"))
```

Listing 17: Eintrag in die Corresponding List

Eine Corresponding List kann für jede Klasse innerhalb einer Rollenhierarchie bestehen. Ihre Einträge sind Objekte von CorrClass. Ein CorrClass-Objekt besteht aus einer Bezeichnung für die Attribute (z.B. „income“), einen Flag, welches die oben erwähnten Bearbeitungsarten beschreibt und beliebig viele Parameter, welche abwechselnd die Rollenklasse, den Methodennamen dieser Rollenklasse und optional eine Liste von Parametern für diese Rollenklassenmethode, enthält. Eine inkrementelle Erweiterung eines CorrClass-Objektes ist derzeit noch nicht möglich, wodurch in solchen Fällen ein neues CorrClass-Objekt erstellt werden muss.

4.2.4 Verwendung der Private List

Im erweiterten Rollenmodell besteht auch die Möglichkeit den Aufruf von Attributen durch andere Rolleninstanzen innerhalb der Rollenhierarchie zu verweigern. Somit wird ein Attribut „privat“ gehalten und kann nur von der eigenen Rolleninstanz aufgerufen werden. Um den Aufruf durch andere Rolleninstanzen zu verweigern, genügt ein

Eintrag des Attributnames in der Private List der jeweiligen Rollenklasse (siehe Listing 18).

```
DepartmentMgr.privateList.push("sal")
```

Listing 18: Eintrag in eine Private List

Durch diesen Eintrag in der Private List der Rollenklasse `DepartmentMgr` kann das Attribut `sal` nur mehr der `DepartmentMgr`-Instanz aufgerufen werden. Zugriffe von anderen Rolleninstanzen oder durch die Corresponding List sind dadurch nicht mehr möglich.

4.3 Implementierung

In diesem Abschnitt werden zunächst die unterschiedlichen Arten von Klassen innerhalb einer Rollenhierarchie beschrieben. Anschließend wird auf die Erstellung von Rolleninstanzen näher eingegangen und abschließend wichtige Klassen und Methoden präsentiert, welche zur Implementierung des erweiterten Rollenmodells von Nöten sind.

4.3.1 Die unterschiedlichen Arten von Klassen

In Kapitel 3 wurden drei Arten von unterschiedlichen Klassen entwickelt: `ObjectWithRoles`, `Role` und `QualifiedRole`. Bei der Implementierung eines erweiterten Rollenmodells in Ruby mussten jedoch konzeptuelle Änderungen vorgenommen werden, welche einerseits in den Anpassungen für den dezentralisierten Einsatz (siehe Abschnitt 4.1) und andererseits in der Implementierung selbst begründet sind. Nach Gottlob et al. [11] ist das Wurzelobjekt einer Rollenhierarchie immer ein Objekt der Klasse `ObjectWithRoles`. Dies führt jedoch bei einer möglichen späteren Zusammenführung mit anderen Rollenhierarchien zu Problemen (falls beispielsweise ein gemeinsamer Wurzelknoten erzeugt wird und somit die Wurzelknoten der beiden Teilhierarchien auf Objekte der Klasse `RoleType` geändert werden müssen), da die Klassenzugehörigkeit eines Objektes in Ruby nicht geändert werden kann. Aus diesem Grund werden die Klassentypen `ObjectWithRoles` und `RoleType` zu einer gemeinsamen Klasse `ObjectOrRole` zusammengeführt (siehe Abschnitt 4.1.2).

Wie in Kapitel 3 werden auch im erweiterten Rollenmodell die Rollenklassen mittels der Klassenmethoden `defRoleType`, `defSpecializedRoleType` und `defQualifiedRoleType` erstellt. Diese haben wiederum die Aufgaben die Klassenvariable `roleSuperType` zu setzen und die jeweilige Klasse zu erzeugen. Diese Funktionalität wird durch das Definieren der Klassenvariable `isSingleInstRole` erweitert. Klassen der neuen Pseudo-Metaklasse `ObjectOrRole` verfügen im erweiterten Rollenmodell neben der Klassenvariable `roleSuperType` auch über die Klassenvariablen `isSingleInstRole`, `corrList` und `privateList`. Das Flag isSingleInstRole gibt Auskunft darüber, ob eine Rollenklasse mehrere Instanzen vorweisen darf oder nicht. Die beiden Listen `corrList` und `privateList` sind optionale Klassenvariable. Sie werden gesetzt, sofern eine Rollenklasse über eine Corresponding List (siehe Abschnitt 4.1.4) bzw. einer Private List (siehe Abschnitt 4.1.5) verfügt.

4.3.2 Erstellen von Rolleninstanzen

Rolleninstanzen, die in Abschnitt 4.2.2 durch den Anwender angelegt werden, werden aus der Implementierungssicht ebenfalls wie in Kapitel 3 erzeugt. Es wird jedoch aufgrund der Übergabeparameterliste unterschieden woraus eine Rolleninstanz erstellt wird. In der Implementierung des erweiterten Rollenmodells werden drei Arten unterschieden:

- Erstellung einer Rolleninstanz durch Spezialisierung:
  Dabei handelt es sich um eine Erstellung einer Rolleninstanz durch die Übergabe der übergeordneten Rolleninstanz bzw. des Wurzelobjektes. In dieser Form wird die `roleOf`-Beziehung der neuen Rolleninstanz auf die übergeordnete Rolleninstanz bzw. des Wurzelobjektes gesetzt und die `roles`-Liste der übergeordneten Rolleninstanz bzw. des Wurzelobjektes um eine Referenz auf die neue Rolleninstanz erweitert.
- Erstellung einer Rolleninstanz durch Generalisierung:
  Bei dieser Art der Erzeugung wird der neuen Rolleninstanz eine Liste ihrer untergeordneten Rolleninstanzen mitgegeben. Bei all diesen Rolleinstanzen wird deren `roleOf`-Verweis auf die neue Rolleninstanz gesetzt und die `roles`-Liste der neuen Rolleninstanz mit Referenzen auf deren untergeordneten Rolleninstanzen gefüllt. Dies entspricht einer Generalisierung von Rollen zu

dieser neuen Rolleninstanz. Eine inkrementelle Generalisierung ist dabei jedoch nicht möglich, d.h. es müssen vor der Generalisierung alle untergeordneten Rolleninstanzen bereits bestehen.

- Erstellung einer isolierten Rolleninstanz:
  Entspricht der erste Parameter der Übergabeparameterliste weder einer Rolleninstanz (Spezialisierung) noch einer Liste von Rolleninstanzen (Generalisierung), so wird die neue Rolleinstanz ohne Verweis auf andere Rollen erzeugt. Sie stellt demnach eine isolierte Rolle ohne Rollenhierarchie dar. Durch eine derartige Erstellung ist es möglich, Rolleninstanzen zu erzeugen, welche erst zu einem späteren Zeitpunkt zu einer Rollenhierarchie hinzugefügt werden.

Neben den `roleOf`-Attributen werden auch die jeweiligen `roleOfType`-Flags der einzelnen Rollenobjekte gesetzt. Ein solches Flag gibt Auskunft über die Entstehung der `roleOf`-Beziehung zwischen zwei Rollenobjekte. Erfolgt die Generierung nach dem Top-Down-Prinzip wird das Flag auf `0` gesetzt. Wurde die Beziehung mittels Bottom-Up-Prinzip erstellt, erhält das Flag den Wert `1`. Es besteht auch die Möglichkeit das `roleOf`-Attribut später zu ändern, wobei das Flag auf den Wert `2` gesetzt wird. Somit ist nachvollziehbar wie eine Rollenhierarchie aufgebaut wurde.

Diese Unterscheidung der Erstellungsart und das Setzen des `roleOfType`-Flags erfolgt durch den Konstruktor der Pseudo-Metaklasse `ObjectOrRole`.

4.3.3 Wichtige Klassen und Methoden

In diesem Abschnitt werden nun die wichtigsten Klassen und Methoden beschrieben, die die Erstellung einer Rollenhierarchie nach Gottlob et al. [11] gewährleisten. Die dabei wichtigste Klasse stellt die Klasse `ObjectOrRole` dar, auf die zunächst näher eingegangen wird. Anschließend folgt eine Beschreibung derer Methoden und zuletzt ein Darstellung der weiteren Klassen bzw. Hilfsmethoden, welche für die Verwendung der Rollenhierarchie von Nöten sind.

4.3.3.1 Die Klasse `ObjectOrRole`

Die Klasse `ObjectOrRole` ist das Herzstück der Implementierung des erweiteren Rollenkonzepts. Jede Rollenklasse in einer Rollenhierarchie ist eine Subklasse von `ObjectOrRole` und erbt somit sämtliche Funktionalitäten dieser Klasse. Zu diesen Funktionalitäten zählen neben der Generierung der Rollenklassen – die zuvor schon näher erläutert wurde – folgende Methoden:

- `method_missing`,
- Navigationsmethoden innerhalb einer Rollenhierarchie und
- Rollenmethoden, welche laut Gottlob et al. [11] jede Rolle beinhaltet.

Von der Klasse `ObjectOrRole` erbt die speziellere Klasse `QualifiedObjectOrRole`, welche eine weitere Klassenvariable (`classOfQualifyingObj`), eine Instanzvariable (`qualfier`) und eine Getter-Methode für den `qualifier` enthält. `QualifiedObjectOrRole` wird zur Erzeugung von qualifizierten Rollen benötigt. Die dabei erwähnte Klassenvariable hat dabei die Funktion der Überprüfung der Klassenzugehörigkeit des `qualifiers`. Die speziellen Methoden für qualifizierte Rollen (nach Gottlob et al. [11]) werden hingegen in der Klasse `ObjectOrRole` implementiert, damit auch normale Rollen diese Methoden ausführen können.

4.3.3.2 Die Methode `method_missing`

Die Methode `method_missing` ist eine vordefinierte Methode der Klasse `Object`. Somit verfügt jedes in Ruby erstelltes Objekt grundsätzlich über diese Methode. Sie wird aufgerufen, sobald eine Methode in der Klasse des Objektes nicht gefunden wird. Der Sinn, diese Methode zu überschreiben, liegt in der individuellen Handhabung von Abfragen auf fehlende Methoden. Verfügt die Klasse des Objektes aber weder über die aufgerufene Methode, noch über eine eigene `method_missing`-Methode, wird in der übergeordneten Klasse nach den beiden Methoden gesucht. Wobei natürlich zuerst die von vorne herein gewünschte Methode gesucht wird. Werden diese Methoden in der gesamten Klassenstruktur des Objektes nicht gefunden, erfolgt der Aufruf der `method_missing` der generellen Superklasse `Object`. Diese wirft eine `NoMethodError`-Exception.

Im Ruby-Code dieser Diplomarbeit wird eine eigene method_missing-Methode in der Klasse `ObjectOrRole` erstellt. Dabei übernimmt die Methode die Aufgaben des Durchsuchens der Rollenhierarchie nach der gesuchten Methode. Es wird jedoch zuvor die gewünschte Methode in der darunter liegenden Rollenhierarchie gesucht. Blieb die Suche dabei erfolglos, wird die Suche erst bei der übergeordneten Rolle fortgesetzt und gegebenenfalls deren Rollenhierarchie begutachtet.

Bei der Implementierung dieser Methode in Ruby wird zu Beginn der zweite Übergabeparameter auf deren Typ überprüft (der erste Übergabeparameter entspricht der Methoden-Id, welche gefunden und aufgerufen werden soll. In der Liste `*rest` befinden sich alle übrigen Parameter, die üblicherweise den Übergabeparametern der gesuchten Methode darstellen. Da jedoch in der `method_missing`-Methode bei deren rekursiven Aufruf ein weiterer Parameter mit übergeben wird – auf diesen wird später näher eingegangen – und dieser somit an erster Stelle der `*rest`-Liste festgehalten wird, wird dieser überprüft.). Entspricht dieser Typ einem Array, so wird davon ausgegangen, dass es sich dabei um die mit übergebene `searchedRoleList`-Liste handelt. Diese Liste beinhaltet sämtliche Rollenklassen, in denen bereits nach der gewünschten Methode gesucht wurde. Durch diese Liste wird also verhindert, dass eine Rollenklasse mehrmals nach ein und derselben Methode überprüft wird und somit auch die Gefahr einer Endlosschleife gebannt wird. Wurde die `searchedRoleList` als Übergabeparameter identifiziert, erfolgt durch den `shift`-Befehl das Löschen dieses Parameters aus der `*rest`-Liste, wodurch die original übergebene `*rest`-Liste wiederhergestellt wird (siehe Listing 19).

```
01   def method_missing(methId, *rest, &block)
02     searchedRoleList = Array.new
03     methName = methId.id2name
04
05     if rest[0].kind_of? Array then
06       searchedRoleList = rest[0]
07       rest.shift
08     end
```

Listing 19: method_missing – Ermitteln der searchedRoleList

Nach anschließender Überprüfung der `searchedRoleList` nach der Rollenklassen der derzeit aktuellen Rolle (z.B. beim Aufruf von `empBlack.income` wird überprüft, ob die Klasse `Employee` in der `searchedRoleList` vorhanden ist), wird die

Corresponding-List (siehe Abschnitt 4.3.3.5) der Rollenklasse nach der gewünschten Methode durchsucht, sofern eine solche Liste besteht und die Methode nicht in der `privateList` (siehe Abschnitt 4.3.3.7) gefunden wird (siehe Listing 20). Ist ein Eintrag dieser Methode in der Corresponding-List verfügbar, erfolgt der Aufruf der `getCorrValue`-Methode, welche den gewünschten Wert zurückgibt der wiederum von der `method_missing`-Methode zurückgegeben wird. Somit entstand ein erfolgreicher Aufruf der ursprünglichen Methode (z.B. `empBlack.income`).

```
09      # Already searched in actual role?
10      if !searchedRoleList.include?(self.class) then
11        searchedRoleList.push(self.class)
12
13        # Looking for method in corrList of the actual role
14        if self.class.corrList && !findInPrivateList(self,
    methName)then
15          self.class.corrList.each do |aCorr|
16            if aCorr.varName == methName then
17              return aCorr.getCorrValue(self, *rest)
18            end
19          end
20        end
```

Listing 20: method_missing – Überprüfung der Corresponding-List

Befindet sich kein entsprechender Eintrag in der Corresponding-List bzw. steht die Methode in der `privateList` – und darf somit nicht vererbt werden – dann wird die Methode in den Subrollen des derzeitigen Rollenobjekts gesucht (siehe Listing 21). Ein Beispiel dafür ist der ursprüngliche Aufruf von `empBlack.birthDate`, wodurch nun die Methode `birthDate` im Subrollenobjekt `deptMgrBlack` gesucht wird. Diese Suche erfolgt durch den Aufruf dieser Methode durch dieses Rollenobjekt inklusive der aktuellen `searchedRoleList`, die nun um die Klasse `Employee` erweitert wurde. Dadurch ergeben sich nun eine Fallunterscheidung: Beim ersten Fall wird die Methode nicht in der Subklasse gefunden, was zum wiederholten Aufruf der `method_missing`-Methode beim `deptMgrBlack` führt. Im zweiten Fall wirft dieser Aufruf eine `ArgumentError`-Exception, was bedeutet, dass die gesuchte Methode in der Subrollenklasse existiert aber die Anzahl der mit übergebenen Parameter nicht korrekt ist – da die `searchedRoleList` auch übergeben wird. Somit wird hier die Exception abgefangen und der Methodenaufruf ohne `searchedRoleList` wiederholt. Da im derzeitigen Beispiel die Suche nach der Methode `birthDate` beim Subklassenobjekt `deptMgrBlack` fehlschlägt und keine

weiteren Subklassenobjekte vorhanden sind, wird durch den zuvor erwähnten, abermaligen Aufruf von `method_missing` lediglich die `searchedRoleList` um die Rollenklasse `DepartmentMgr` erweitert.

```
21        # method not found in role and corrList
22        self.getRoles.each do |aRole| # search at role.getRoles
23          begin
24            return aRole.send(methId, searchedRoleList, *rest,
   &block)
25          rescue ArgumentError # Exception throws when method found
26            if !findInPrivateList(aRole, methName) then
27              return aRole.send(methId, *rest, &block)
28            end
29          end
30        end
31      end
```

Listing 21: method_missing – Überprüfung aller Subrollen

Blieb auch die Suche in allen Subrollen erfolglos wird in der `method_missing`-Methode das übergeordnete Rollenobjekt (`roleOf`) auf die gewünschte Methode hin untersucht. Dies erfolgt analog zur Durchsuchung eines Subrollenobjektes (siehe Listing 22). Auf das Beispiel bezogen wird nun die `method_missing`-Methode wiederum für das Klassenobjekt `empBlack` ausgeführt. Da jedoch in der `searchedRoleList` die Rollenklasse Employee bereits vorhanden ist, erfolgt die weitere Suche nun gleich beim übergeordneten Rollenobjekt (`personBlack`). Dort wird endlich die gewünschte Methode gefunden – oder besser gesagt das Attribut, was aber für Ruby keinen Unterschied macht – und die zuvor erwähnte Exception geworfen, wodurch der direkte Methodenaufruf auf `personBlack.birthDate` erfolgt und somit der ersehnte Rückgabewert ermittelt werden konnte.

Wenn jedoch die Suche in der gesamten Rollenhierarchie erfolglos bleibt wird die Exception `NoMethodError` geworfen (siehe Listing 22).

```
32      # method not found in role, corrList and sub-roles
33      if @roleOf then
34        begin
35          return @roleOf.send(methId, searchedRoleList, *rest,
    &block)
36        rescue ArgumentError # Exception throws when method found
37          if !findInPrivateList(@roleOf, methName) then
38            return @roleOf.send(methId, *rest, &block)
39          end
40        end
41      end
42
43      # method not found/available in role-hierarchy
44      raise NoMethodError.new("Method '#{methName}' not found ...")
45    end   # end method_missing
```

Listing 22: method_missing – Überprüfung der übergeordneten Rolle

Diese Implementierung der `method_missing`-Methode unterscheidet sich zu jener aus der im Ruby on Rails Projekt, da Unterschiede bei der allgemeinen Verwendung dieser Methode zwischen Ruby und Ruby on Rails bestehen (siehe Kapitel 5.3 – Implementierung in Ruby on Rails).

4.3.3.3 Navigationsmethoden

Um eine Navigation in der Rollenhierarchie zu gewährleisten muss auf Instanzebene grundsätzlich eine 1-zu-n-Assoziation nachgebildet werden. Dies geschieht bei dieser Implementierung durch die zwei Attribute `roleOf` und `roles`. Die Methode `roleOf` beschreibt dabei die Verbindung von einem Rollenobjekt zu deren Superobjekt, wohingegen `roles` eine Liste aller Subrollen eines Rollenobjektes darstellt. In Gottlob et al. [11] wird lediglich die `roleOf`-Beziehung aufgeführt. Diese ist jedoch nur dann ausreichend, wenn man die Rollenhierarchie zu jeden Zeitpunkt vom untersten Element bis zum Wurzelobjekt durchsuchen kann. Durch die Vorgaben des dezentralen Einsatzes (siehe Abschnitt 2.4.1) kann aber nicht davon ausgegangen werden. Demzufolge wurde das Attribut `roles` eingeführt. Für diese Attribute gibt es jeweils eine Getter-Methode (`roleOf` und `roles`). Weiters wurde eine Setter-Methode für das Attribut `roleOf` implementiert, wodurch das Attribut selbstverständlich neu gesetzt wird und auch das Flag `roleOfType` auf den Wert `2` gesetzt wird.

Zusätzlich zu den beiden Getter-Methoden wurde noch die Methode `getAllSubRoles` inkludiert, welche alle Rollenobjekte der darunter liegenden Teilhierarchie als Liste zurückliefert. Diese Methode dient als Hilfsmethode, wodurch

das Durchsuchen einer Teilrollenhierarchie erleichtert wird. Somit ergeben sich folgende Navigationsmethoden:

- `roleOf` mit Rückgabewert vom Typ `ObjectOrRole` oder `nil`,
- `roles` mit Rückgabewert vom Typ Array und
- `getAllSubRoles` ebenfalls mit einem Rückgabewert vom Typ Array.

4.3.3.4 Rollenmethoden

Sämtliche Rollenmethoden, die in Gottlob et al. [11] beschrieben werden, wurden im Ruby-Code umgesetzt. Dabei handelt es sich einerseits um die Methoden für Objekte der Superklassen `ObjectWithRoles` und `RolesType`. Diese werden im Sinne der besseren Lesbarkeit hier nochmals aufgeführt und kurz beschrieben (vgl. Abschnitt 2.1.1.2):

- `root`:
  Dies ist eine Methode zur Ermittlung des Wurzelobjektes einer Rollenhierarchie.
- `roleOf`:
  `roleOf` liefert den übergeordneten Knoten einer Rollenhierarchie zurück. Sie wird auch als Navigationsmethode verwendet.
- `as(aRoleType)`:
  Diese Methode vergleicht das jeweilige Rollenobjekt mit dem Rollentyp `aRoleType`. Besteht in der Rollenhierarchie des aktuellen Objektes eine Instanz von aRoleType wird diese zurückgeliefert. Ansonsten wird der Wert `nil` zurückgegeben.
- `existsAs(aRoleType)`:
  Mit Hilfe dieser Methode wird überprüft, ob ein Rollenobjekt des Typs `aRoleType` in der Rollenhierarchie besteht und liefert den entsprechenden booleschen Wert zurück.
- `entityEquiv(anObject)`:
  Die `entityEquiv`-Methode vergleicht die Wurzelobjekte des aktuellen Objektes mit dem von `anObject`. Sind diese beiden Wurzelobjekte identisch wird der jeweilige boolesche Wert zurückgegeben.
- `abandon`:
  `abandon` löscht ein Rollenobjekt aus der Rollenhierarchie.

- `as_of(aQualifiedRoleType, qualifyingObj)`:
  Diese Methode verweist auf das Rollenobjekt des Typs `aQualifiedRoleType` und dem `qualifyingObj` als entsprechenden `qualifier` Der Rückgabewert ist, wie bei der `as`-Methode, entweder das gefundene Rollenobjekt oder `nil`.
- `existsAs_of(aQualifiedRoleType, qualifyingObj)`:
  Mit dieser Methode wird überprüft, ob ein Rollenobjekt des Typs `aQualifiedRoleType` in der Rollenhierarchie besteht, welcher mithilfe des `qualifyingObj` identifiziert werden kann. Zurückgegeben wird hier wiederum ein boolescher Wert.

Im Anschluss werden nun weitere Klassen und Methoden beschrieben, welche für die Umsetzung bestimmter Funktionalitäten notwendig sind. Dies sind einerseits die CorrespondingClass, kurz `CorrClass`, und deren Methoden zur Implementierung des Mapping-Ansatzes und andererseits die Private List, kurz `privateList`, und deren Umsetzung zur Unterscheidung von Rollenattributen und Nicht-Rollenattributen.

4.3.3.5 Die Klasse `CorrClass`

Die Klasse `CorrClass` ist eine Hilfsklasse zur Umsetzung des an Schrefl und Neuhold [34] bzw. Klas und Schrefl [14] angelehnten Mapping-Ansatzes. Beim Mapping-Ansatz sollen Attribute oder Methoden zu einer übergeordneten Rolle generiert werden können, welche deren Wert bzw. Rückgabewert aus den Aufrufen von Attributen und/oder Methoden derer Sub-Rollen ermittelt (siehe Kapitel 4.1.4). So soll beispielsweise der Rolle `Employee` die Methode `income` hinzugefügt werden, welche das gesamte Einkommen eines Angestellten zurückgibt – also das Gehalt des Angestellten selbst plus der Gehälter, die dieser Angestellte durch seine `DepartmentMgr`-Rollen einnimmt.

Für die Umsetzung dieses Mapping-Ansatzes wird zuerst eine Liste aller solcher Attribute und Methoden angelegt – dies ist die sogenannte Corresponding-List, wobei jede Rollenklasse eine derartige Liste besitzt. Die einzelnen Elemente dieser Corresponding-List sind Objekte der Klasse `CorrClass`. Eine Instanz dieser Klasse entspricht einer Beschreibung einer derartig gewünschten Methode, welche Subrollen-

Attribute oder -Methoden aufruft. Diese Beschreibung enthält dabei folgende Informationen (siehe Listing 23 – `attr_reader`):

- Attribut- bzw. Methodennamen (`varName`)
- Relationship-Flag: Ein Flag (`relationship`), welches Auskunft darüber gibt, wie die einzelnen Subrollen-Attribute bzw. -Methoden aufgerufen und deren Ergebnisse zu einem gemeinsamen Rückgabewert verknüpft werden.
- Rollenliste: Die Rollenliste (`roleArray`) beinhaltet sämtliche Rollenklassen von denen Attribute bzw. Methoden aufgerufen werden.
- Methodenliste (`methArray`): Diese enthält alle Attribut- bzw. Methodennamen, die über die Rollenklassen aus der Rollenliste aufgerufen werden.
- Parameterliste (`argsArray`): Darin können benötigte Parameter zu den jeweiligen Methoden aus der Methodenliste mit übergeben werden. Das Anführen einer Parameterliste ist jedoch optional.

Durch das Zusammenspiel der drei Listen (`roleArray`, `methArray` und `argsArray`) wird bei der Initialisierung darauf geachtet, dass zusammengehörige Einträge in allen Listen dieselbe Indizierung aufweisen. Da jedoch eine Parameterliste nicht zwingend angegeben werden muss, wird in solchen Fällen ein `nil`-Wert im `argsArray` an der jeweiligen Stelle eingefügt.

```
01   class CorrClass
02     attr_reader :varName, :relationship, :roleArray, :methArray,
03       :argsArray
04
05     def initialize(varName, relationship, *rest)
06       @roleArray = Array.new
07       @methArray = Array.new
08       @argsArray = Array.new
09       @varName = varName
10       @relationship = relationship # Type of relationship (0..3)
11
12       rest.each do |variable|
13         @roleArray.push(variable) if variable.instance_of?(Class)
14         @methArray.push(variable) if
   variable.instance_of?(String)
15         @argsArray.push(variable) if variable.instance_of?(Array)
16
17         @argsArray.push(nil) if @roleArray.size-1 >
   @argsArray.size
18       end
19       @argsArray.push(nil) if @roleArray.size > @argsArray.size
20     end
```

Listing 23: CorrClass – Initialisierung

Da ein Corresponding-List Eintrag – und somit eine Instanz von `CorrClass` – beliebig viele Subrollen und deren Attribute bzw. Methoden beinhalten kann, muss zur korrekten Initialisierung auf die Reihenfolge der Eingangsparameter geachtet werden.

4.3.3.6 Die Methode `getCorrValue`

Die Methode `getCorrValue` generiert aus den Vorgaben eines Corresponding-List Eintrages den erwünschten Rückgabewert. Wird beispielsweise die in der Corresponding-List von `Employee` eingetragene Methode `income` vom `empBlack` aufgerufen, erfolgt die Ermittlung des Rückgabewertes nach folgendem Schema: Es wird die Liste aller im Corresponding-List Eintrag inkludierten Rollenklassen – gespeichert im `roleArray` – durchlaufen und zuerst überprüft, ob auch die eigene Rollenklasse miteinbezogen werden muss. Also, ob zur Berechnung von `income` auch eine Methode von `Employee` selbst aufgerufen werden soll. Wie in Abbildung 17 ersichtlich soll in diesem Beispiel zum Gesamteinkommen `income` auch das Gehalt (`salary`) der Rollenklasse `Employee` hinzuaddiert werden (also beispielsweise `empBlack.salary`, falls dieser existiert). Für alle anderen Rollenklassen in `roleArray` wird überprüft, ob das Rollenobjekt diese Rollen als Sub-Rollen einnimmt. Bei jeder eingenommenen Rolle wird anschließend kontrolliert, ob die

dazugehörige Methode – aus dem `methArray` – auch übergeben werden darf. D.h. ob für diese Methode kein Eintrag in der `privateList` der jeweiligen Rollenklasse besteht (siehe Abschnitt 4.3.3.7). Ist dies nicht der Fall wird die Methode beim jeweiligen Rollenobjekt ausgeführt (Aufruf von z.B. `deptMgrBlack.sal`). Durch die optionale Angabe von Parametern zu einer Methode aus dem `methArray` muss die `ArgumentError`-Exception abgefangen werden, da zuerst versucht wird, die Methode mit Parametern aufzurufen. Benötigt die Methode keine Parameter wird die zuvor erwähnte Exception geworfen und dieselbe Methode ohne Parameter aufgerufen.

```
Employee.corrList[i] = (CorrClass.new("income", 2,
                                      Employee, "salary",
                                      DepartmentMgr, "sal"))
```

Abbildung 17: Eintrag in der Corresponding-List von Employee

Dadurch, dass die Generierung des Rückgabewertes für eine Methode aus der Corresponding-List auf unterschiedliche Arten erfolgen kann, muss die Ermittlung durch die `getCorrValue`-Methode auf verschiedene Arten erfolgen. Für diese Unterscheidung wird in der Corresponding-List das Relationship-Flag verwendet. Wodurch die Art der Generierung vom Relationship-Flag abhängig ist. In dieser Arbeit wurden vier Abwicklungsarten zur Verfügung gestellt (siehe Kapitel 4.1.4). Diese werden aufgrund der besseren Lesbarkeit auch hier kurz angeführt und deren Implementierung dargestellt:

- `ID`-Relationship:

  Hier wird der Wert des ersten Attributs bzw. der ersten Methode zurückgegeben, welche(s) nicht in der `privateList` der jeweiligen Rollenklasse eingetragen wurde (siehe Listing 24).

```
21      if @relationship == 0 then      # id relationship
22        i = 0
23        @roleArray.each do |aRoleClass|
24          if role.class == aRoleClass then
25            if !findInPrivateList(role, @methArray[i])
    then
26              begin
27                return role.send(@methArray[i],
    *@argsArray[i])
28              rescue ArgumentError
29                return role.send(@methArray[i])
30              end
31            end
32          else
33            role.getAllSubRoles.each do |aRole|
34              if aRole.class == aRoleClass then
35                if !findInPrivateList(aRole,
    @methArray[i]) then
36                  begin
37                    return aRole.send(@methArray[i],
38                                      *@argsArray[i])
39                  rescue ArgumentError
40                    return aRole.send(@methArray[i])
41                  end
42                end
43              end
44            end
45          end
46          i += 1
47        end
```

Listing 24: getCorrValue – ID-Relationship

- `ALL`-Relationship:

  Dabei wird eine Liste zurückgegeben, welche alle Attributwerte bzw. Rückgabewerte der einzelnen Methoden aus der Corresponding-List beinhaltet, wobei wiederum alle Werte ignoriert werden, welche sich in der `privateList` der jeweiligen Rollenklassen befinden (siehe Listing 25).

```
47      elsif @relationship == 1 then      # all relationship
48        valueArray = Array.new
49        i = 0
50        @roleArray.each do |aRoleClass|
51          if role.class == aRoleClass then
52            if !findInPrivateList(role, @methArray[9]) then
53              begin
54                valueArray.push(role.send(@methArray[9],
55                                          *@argsArray[9]))
56              rescue ArgumentError
57                valueArray.push(role.send(@methArray[9]))
58              end
59            end
60          else
61            role.getAllSubRoles.each do |aRole|
62              if aRole.class == aRoleClass then
63                if !findInPrivateList(aRole, @methArray[9])
   then
64                  begin
65                    valueArray.push(aRole.send(@methArray[9],
66                                               *@argsArray[9]))
67                  rescue ArgumentError
68                   valueArray.push(aRole.send(@methArray[9]))
69                  end
70                end
71              end
72            end
73          end
74          i += 1
75        end
76        return valueArray
```

Listing 25: getCorrValue – ALL-Relationship

- SUM-Relationship:

  Die Generierung dieses Rückgabewertes erfolgt durch den Aufruf sämtlicher Attribute bzw. -Methoden und deren Aufsummierung. Der Rückgabewert in dieser Implementierung entspricht einem Integer und arbeitet dementsprechend auch mit Integer-Werten. Daher muss bei der Verwendung darauf geachtet werden oder dieser Codeteil angepasst werden (siehe Listing 26).

```
77      elsif @relationship == 2 then     # sum relationship
78        value = 0
79        i = 0
80        @roleArray.each do |aRoleClass|
81          if role.class == aRoleClass then
82            if !findInPrivateList(role, @methArray[9]) then
83              begin
84                number = @argsArray[9].to_s.to_i
85                value += role.send(@methArray[9], number)
86              rescue ArgumentError
87                value += role.send(@methArray[9])
88              end
89            end
90          else
91            role.getAllSubRoles.each do |aRole|
92              if aRole.class == aRoleClass then
93                if !findInPrivateList(aRole, @methArray[9])
    then
94                  begin
95                    number = @argsArray[9].to_s.to_i
96                    value += aRole.send(@methArray[9],
    number)
97                  rescue ArgumentError
98                    value += aRole.send(@methArray[9])
99                  end
100               end
101             end
102           end
103         end
104         i += 1
105       end
106       return value
```

Listing 26: getCorrValue – SUM-Relationship

- GENERAL-Relationship:

  Hierbei wird lediglich ein `nil`-Wert zurückgeliefert. In diesem Codestück soll dem Programmierer die Möglichkeit gegeben werden eine individuelle Generierungsart anzufügen.

Da die Implementierung des Mappings sehr allgemein gehalten wurden, muss bei speziellerer Verwendung diese Methode möglicherweise angepasst werden.

4.3.3.7 Die Methode `privateList`

Bei der Erstellung einer Rollenhierarchie bzw. bei der Zusammenführung mehrerer Teilhierarchien, insbesondere wenn diese aus unterschiedlichen, dezentralen Quellen stammen, muss davon ausgegangen werden, dass nicht alle Attribute und/oder Methoden als Rollenattribute angesehen werden können. Mit anderen Worten können

Attribute bzw. Methoden inkludiert werden, die in der Rollenhierarchie nicht weitergereicht werden dürfen, also nicht von anderen Rollen aus gelesen werden dürfen. Nimmt man beispielsweise an, dass das Gehalt eines `DepartmentMgr` ein Nicht-Rollenattribut darstellt und somit nur vom Rollenobjekt `deptMgrBlack` betrachtet werden kann, so muss dies festgehalten werden.

Die Private-List – im Ruby-Code kurz `privateList` genannt – ist eine Liste zur Unterscheidung von Rollenattributen und Nicht-Rollenattributen. Jedes Element in dieser Liste stellt ein Nicht-Rollenattribut einer Rollenklasse dar, wobei jede Rollenklasse ihre eigene `privateList` besitzt. Wird das zuvor angesprochene Beispiel herangezogen, so muss zu deren Implementierung in der `privateList` der Rollenklasse `DepartmentMgr` lediglich das Attribut `sal` hinzugefügt werden. Schon kann das Gehalt eines `DepartmentMgrs` nicht mehr über die Methode `method_missing` gefunden werden und daher auch nicht von anderen Rollenobjekten in der Rollenhierarchie. Der Grund dafür liegt im Aufruf der `findInPrivateList`-Methode in der `method_missing` (siehe Abschnitt 4.3.3.2). Diese Methode durchsucht die `privateList` der jeweiligen Rollenklasse und gibt, je nach erfolgreichem Suchen, den dementsprechenden booleschen Wert zurück (siehe Listing 27).

```
01   def findInPrivateList(role, meth)
02     if role.class.privateList then # privateList
   available
03       role.class.privateList.each do |elem|
04         if elem == meth then
05           return true
06         end
07       end
08     end
09     return false
10   end
```

Listing 27: findInPrivateList

Weiters wird die Methode `findInPrivateList` auch bei der Ermittlung des Rückgabewertes eines Attributs aus der Corresponding-List (siehe Abschnitt 4.3.3.6) verwendet. Deshalb befindet sich die Methode in einem eigenen Modul namens `Helpers`, wodurch sie einerseits in der Klasse `ObjectOrRole` (siehe Abschnitt 4.3.3.1) sowie auch in der Klasse `CorrClass` (siehe Abschnitt 4.3.3.5) zur Verfügung

steht. Die Einführung dieses Moduls hat lediglich den Vorteil, dass die darin verwendeten Methoden nicht mehrmals implementiert werden müssen.

4.4 Evaluierung

In diesem Kapitel wurde ein erweitertes Rollenmodell entwickelt, welches die Anforderungen für den dezentralen Einsatz unterstützt. Dieses erweiterte Rollenmodell verfügt neben der roleOf-Beziehung auch über die inverse roles-Beziehung. Weiters wurden die Pseudo-Metaklassen ObjectWithRoles und Roles, aus Kapitel 3 zu einer gemeinsamen Pseudo-Metaklasse zusammengefügt. Eine Erstellung einer Rollenhierarchie ist nun auch mittels Bottom-Up-Prinzip möglich, wodurch auch der Ansatz der Upward Inheritance verfügbar ist. Das erweiterte Rollenmodell inkludiert weiters einen Mapping-Ansatz, sowie die Möglichkeit Attribute privat zu halten.
Die Verwendung dieses erweiterten Rollenmodells wurde ebenso beschrieben, wie die wichtigsten Implementierungen dieses Rollenmodells.
Durch diesen neu implementierten Rubycode sind die Grundsteine gelegt, um das Rollenkonzept von Gottlob et al. [11] in einem dezentralen Informationssystem zu verwenden.

5 Erweiterung von Ruby on Rails um Rollen

Ein Ziel dieser Diplomarbeit ist es, das Rollenkonzept von Gottlob et al. [11] in dezentralen Systemen zur Verfügung zu stellen. Rossi et al. [32] wählten den Weg von Webapplikationen, wobei Huemer [13] einen entsprechenden Linked Data Browser entwickelte, um die Verwendung von Rollen in RDF zu unterstützen. Bis zu dieser Stelle wurde ein Ruby-Projekt erstellt, welches für den dezentralen Einsatz gerüstet ist. Als Basis für die Entwicklung eines Webapplikationframeworks mit Rollenunterstützung ist das an Ruby nahe liegende Framework Ruby on Rails geeignet. Durch die Vorgaben von Ruby on Rails muss das erweiterte Rollenmodell angepasst werden. Diese Anpassungen werden im Abschnitt 5.1 beschrieben. Die Verwendung von Ruby on Rails mit Rollen wird in Abschnitt 5.2 erläutert.

Da Ruby on Rails mit seinen Generatoren ein nützliches Grundgerüst für eine Webapplikation erstellen kann, wird auch versucht, ein Grundgerüst zu erstellen, welches die Eigenschaften eines Rollenkonzeptes beinhaltet. In Abschnitt 5.3 wird einerseits dieser Generator vorgestellt und andererseits ein Beispiel dargelegt, welchen Code dieser Generator erzeugt. Anschließend erfolgt noch die Beschreibung der Einschränkungen, welche der generierte Code mit sich bringt.

5.1 Konzeptionelle Anpassungen des erweiterten Rollenmodells

Aufgrund von Rahmenbedingungen durch Ruby on Rails werden in diesem Kapitel notwendige, konzeptionelle Anpassungen getroffen, wodurch die Implementierung eines Generators ermöglicht wird, der ein Grundgerüst für Webapplikationen schafft, die die Anforderungen eines Rollenkonzeptes zur Verfügung stellt.

5.1.1 Dynamische Klassenerzeugung

In der vorherigen Implementierung erfolgte die Rollenklassenerzeugung mittels der Methode `defRoleType`, die für das Setzen der Klassenvariablen zuständig war. Diese Klassenvariablen wurden später bei der Initialisierung einer Rolleninstanz verwendet. Eine Rollenklasse kann in Ruby on Rails nicht mehr dynamisch über Klassenmethoden erzeugt werden, da die Rollen als Ressourcen gelten und somit persistent abgespeichert

werden müssen, wodurch eine Rollenklasse genau einer Klasse im Model einer Webapplikation entsprechen muss. Schrefl und Thalhammer müssen in ihrer Java-Lösung [35] ebenfalls auf eine dynamische Rollenklassenerstellung verzichten, da dies die Typsicherheit von Java nicht zulässt. Ruby on Rails erwartet, dass jede Ressource eine Instanz einer Model-Klasse darstellt, wodurch eine Art Typsicherheit gegeben ist. Model-Klassen werden bei der Erstellung einer Webapplikation mit Ruby on Rails über die Konsole angelegt, wobei dies auch somit für Rollenklassen gelten muss. Aus diesem Grund muss auch die Definition der Klassenvariablen, wie z.B. des `roleSuperTypes`, welche zuvor in der Methode `defRoleType` definiert wurden, auf eine andere Art und Weise erfolgen. Weiters ist auch die Deklaration dieser Klassenvariablen betroffen. Zuvor erfolgte dies über die Generierung einer objektspezifischen Klasse. Nun muss die Deklaration und Definition der Klassenvariablen bereits bei der Generierung der Modelklasse über die Konsole erfolgen.

Bei der Generierung einer neuen Rollenklasse wird diese als Tabelle in der Datenbank angelegt, wodurch sie persistent gespeichert wird. Die, für die Rollenklasse benötigten, Klassenvariablen, werden dabei im erzeugten Model, als übliche Klassenvariablen, hinterlegt und anhand des Generators zugewiesen. Die Werte der Klassenvariablen müssen dabei beim Erstellungsbefehl über die Konsole mitgegeben werden (z.B. `generate Employee Person...`). Die Erzeugung einer objektspezifischen Singleton-Klasse für eine Rollenklasse ist somit nutzlos und fällt daher weg.

5.1.2 Identifikation und Repräsentation

Bei der Implementierung des Rollenkonzeptes in Ruby wird noch keine Rücksicht auf eine globale Identifikation genommen. Hier erfolgt die Identifikation von Objekten noch anhand der OID. Der Grund dafür liegt in der Zweckmäßigkeit der Implementierung. Da diese als Zwischenschritt zur Implementierung eines Generators für Ruby on Rails dient, wäre hier der Aufwand der Generierung eines neuen Identifikators unnötig.

Für einen Einsatz im Web ist jedoch eine Anpassung des Identifikators eines Objektes unumgänglich. Da Ruby on Rails REST unterstützt und auch RESTful Webapplikationen erstellt werden sollen ist die Änderung des Identifikators auf die

RESTkonforme URI nahe liegend. Dadurch entsteht auch kein weiterer Aufwand, das Ruby on Rails ohnehin mit URIs arbeitet und somit jede Ressource (und daher auch jedes Objekt) eine URI zugewiesen bekommt.

Trotzdem muss ein akzeptabler Weg zur Präsentation von Objekten noch gefunden werden. Ruby on Rails stellt unterschiedliche Möglichkeiten zur globalen Repräsentation zur Verfügung. Diese sind beispielsweise die Verwendung von XML (siehe Wintschel [40]) oder von RDF (siehe Oren et al. [24]). Die Präsentation anhand der ersten Variante kann dabei durch die Ruby-Bibliotheken REXML oder XML-Builder erfolgen. Oren et al. [24] bzw. Oren und Delbru [23] beschreiben für die zweite Variante die Verwendung von ActiveRDF. ActiveRDF hat dabei dieselben Aufgaben für RDF-Dokumente, wie ActiveRecord für Datenbankentabellen. Da jedoch der Administrator von ActiveRDf Eyal Oren im zweiten Halbjahr 2009 dieses Amt zurückgelegt hat, wurde die Unterstützung für die Implementierung von ActiveRDF eingestellt und somit deren Zukunft offen steht, wird die Verwendung von XML-Bibliotheken bevorzugt.

5.2 Verwendung von Rollen in Ruby on Rails

In diesem Abschnitt wird die Verwendung von Rollen über eine Ruby on Rails-Webapplikation erklärt. In Abschnitt 5.2.1 wird zunächst die Erstellung einer Rollenhierarchie auf Schemaebene und in Abschnitt 5.2.2 auf Instanzebene beschrieben. Abschnitt 5.2.3 gibt Auskunft über die Verwendung von Rollenmethoden über eine Webapplikation.

5.2.1 Erstellen von Rollenklassen

Um Rollen anhand einer Ruby on Rails-Webapplikation zu verwenden, muss zuerst eine derartige Webapplikation erzeugt werden. Dies erfolgt über den Konsolenbefehl „rails roleApp“, wobei roleApp für den Namen der zu erzeugenden Webapplikation steht. Anschließend kann die Rollenhierarchie auf Schemaebene für diese Webapplikation mit weiteren Konsolenbefehlen erstellt werden. Ein derartiger Konsolenbefehl wird in Abbildung 18 dargestellt. Im Anhang befindet sich eine Auflistung sämtlicher Konsolenbefehle zur Erstellung einer Webapplikation, welche dem Beispiel aus Abschnitt 1.4 entspricht.

```
>ruby script/generate rolescaffold Employee Person name:string
salary:integer phoneNo:string person_id:integer
```

Abbildung 18: Konsolenbefehl zur Erstellung der Rollenklasse Employee

Anhand dieses Konsolenbefehles wird die Rollenklasse Employee erzeugt, welche als roleSuperType die Klasse Person aufweist und die Attribute name, salary und phoneNo beinhaltet. Die person_id dient als ein, der Namenskonventionen von Ruby on Rails entsprechender, Fremdschlüssel für die Datenbank.

5.2.2 Erstellen von Rolleninstanzen

Eine durch den Generator erzeugte Ruby on Rails-Webapplikation verfügt u.a. über die Funktionalität zur Erstellung von Instanzen, welche über einen Webbrowser aufgerufen werden kann. Sobald der Server gestartet ist, können Instanzen erzeugt werden. Eine Instanz der Klasse Employee kann beispielsweise über die Adresse „http://localhost:3000/employees/new“ angelegt werden. Je nach Klassenattributen werden dementsprechende Eingabefelder zur Verfügung gestellt über die Daten zu Rolleninstanzen eingegeben werden können (siehe Abbildung 19).

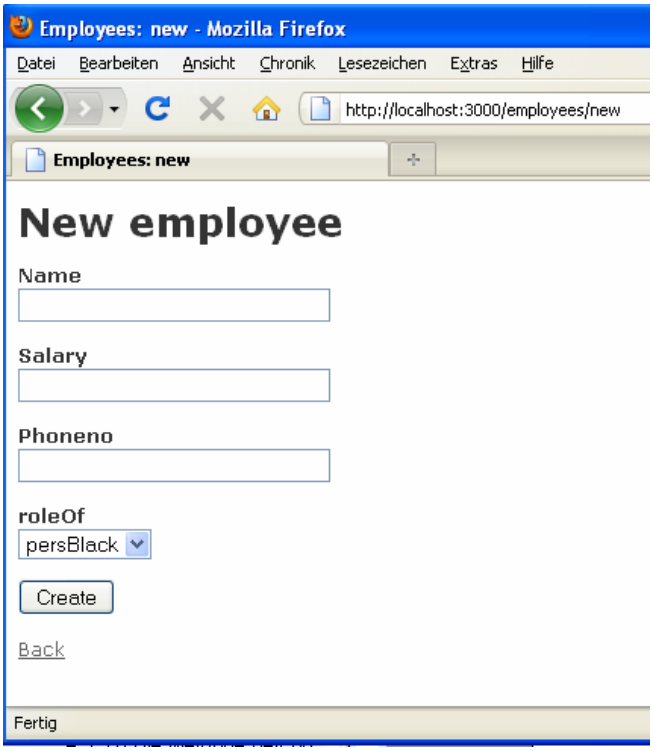

Abbildung 19: View zum Anlegen einer Employee-Instanz

Im roleOf-Feld werden sämtliche Instanzen der roleSuperType-Klasse aufgelistet, wobei hier eine Instanz oder keine ausgewählt werden können.

5.2.3 Verwenden von Rollenmethoden

Eine, durch den Generator erstellte Webapplikation, verfügt über sämtliche Rollenmethoden nach Gottlob et al. [11]. Für die Verwendung von Rollenmethoden bestehen zwei Möglichkeiten. Der Rückgabewert einer Rollenmethode kann einerseits über die URL angegeben werden (z.B. „http://localhost:3000/employees/1?as=Student“), andererseits können Rollenmethoden innerhalb der View auf die übliche Weise ausgeführt werden (z.B. „empBlack.as(Student)“). Somit ist ersichtlich, dass in den Views auch auf die Attribute einer Rolle zugegriffen werden kann. Hierbei handelt es sich nicht nur um die eigenen Attribute, sondern auch über Inheritance bzw. Upward Inheritance verfügbare Attribute und Methoden. Weiters können auch durch die Corresponding List erzeugte Attribute in dieser Weise aufgerufen werden.
Einträge in der Corresponding List, sowie in der Private List, können im Model einer Rollenklasse hinzugefügt werden.

Für die Repräsentation von Rolleninstanzen in XML muss in der URL lediglich die Erweiterung „.xml“ angefügt werden. Diese Repräsentation enthält, neben den Tabelleneinträgen aus der Datenbank, auch die rollenspezifischen Informationen, wie die Attribute roleOf, roles und root.

5.3 Implementierung

Aufgrund des bereits entwickelten Rollenkonzeptes für den Einsatz in dezentralen Informationssystemen aus Kapitel 4.3 und den Anpassungen aus dem Kapitel 5.1 wird nun ein Generator implementiert, der das Grundgerüst einer Webapplikation erzeugen soll, welches das Rollenkonzept zur Verfügung stellt. Dieser Generator wird roleScaffold genannt. Mit seiner Hilfe können sämtliche Objekte einer Rollenhierarchie generiert werden. Ferner ist er auch für die Erstellung der MVC-Architektur zu diesem Objekt verantwortlich. Dieer Generator wird im folgenden Abschnitt 5.3.1 beschrieben. Im darauf folgenden Abschnitt 5.3.2 soll der Code einer generierten Rollenklasse verdeutlicht werden. Einschränkungen, welche aufgrund von Ruby on Rails für diese Webapplikation auftreten, werden abschließend im Kapitel 5.3.3 beschrieben.

5.3.1 Generierung von Rollenklassen mit roleScaffold

Der roleScaffold-Generator ist ein im Rahmen dieser Diplomarbeit entworfener Generator. Er ist grundsätzlich eine Erweiterung des scaffold-Generators (siehe Abschnitt 2.3). Neben dem scaffold-Grundgerüst werden einerseits neue Dateien generiert und andererseits Dateien verändert generiert, worauf in diesem Abschnitt noch näher eingegangen wird.

Die Aufgabe des roleScaffold-Generators besteht darin ein Grundgerüst für eine Webapplikation zu erzeugen, dass die Verwendung eines Rollenkonzeptes erlaubt, welches auf jenem von Gottlob et al. [11] aufbaut. Anders ausgedrückt basiert das Rollenkonzept auf das von Gottlob et al. mit den Veränderungen bzw. Erweiterungen für einen dezentralen Einsatz. Wenn also eine Webapplikation mittels des roleScaffold-Generators erzeugt wird, kann bereits eine Rollenhierarchie erstellt werden und ihre die Grundfunktionalität verwendet werden. Standardmäßig verwendet der roleScaffold-Generator, wie der scaffold-Generator, als Front-End einen Browser bei dem die Präsentation der Daten über .erb-Dateien erfolgt. ERb steht dabei für Embedded Ruby und ist eine Template-Sprache für Ruby. Sie ist stark an die Skriptsprache PHP angelehnt.
Als Back-End dient eine SQLite Datenbank. SQLite ist eine Programmbibliothek, welche ein einfaches, relationales Datenbanksystem beinhaltet. Der Einsatzbereich von SQLite besteht in der Einbindung in Systeme. Die Abfragen basiert auf SQL-Befehlen, wobei ein Großteil dieser Befehle unterstützt wird. Natürlich kann auch dieses Back-End jederzeit ersetzt werden.

Änderungen bei der Generierung gegenüber dem scaffold-Generator bestehen zunächst in den neu generierten Dateien object_or_role.rb und role_helper.rb. bzw. in folgenden veränderten Dateien:

- controller.rb,
- model.rb,
- routes.rb und
- sämtlichen Views.

Auf diese Änderungen wird später in den nächsten Unterkapiteln näher eingegangen, doch zunächst wird der roleScaffold-Generator selbst beschrieben, welcher in der Datei roleScaffold.rb implementiert ist.

Sobald also eine neue Rollenklasse für eine Webapplikation erstellt werden soll, erfolgt dies über den Aufruf des roleScaffold-Generators. Seine Aufgaben sind einerseits die Überprüfung der Eingaben über die Konsole und andererseits die Koordination bzw. Erstellung der korrekten Dateien an den jeweiligen Positionen innerhalb der Verzeichnisstruktur der Webapplikation. Für die Bewältigung dieser Aufgaben wurden zusätzliche Variablen eingeführt. Diese sind:

- `role_super_type:`
  Der `role_super_type` definiert auch hier die übergeordnete Klasse innerhalb einer Rollenhierarchie.
- `role_super_type_underscore_name:`
  Dies ist der Name des `roleSuperTypes` in Kleinbuchstaben.
- `role_flag:`
  Dieses Flag gibt Auskunft darüber, ob es sich bei der zu erzeugende Klasse um eine qualifizierte Rolle handelt oder nicht.
- `qualifier_type:`
  Der `qualifier_type` beinhaltet die Klasse des `qualifiers` einer qualifizierten Rolle
- `qualifier_type_underscore_name:`
  Diese Variable hat als Wert den Namen des `qualifiers` einer qualifizierten Klasse in Kleinbuchstaben.

Die im ersten Teil der vom Generator zu erfüllenden Aufgabe wird in den folgenden Abbildungen dargestellt. In Listing 28 wird der Beginn des Konstruktors abgebildet, welcher für die Überprüfung des `role_flags` verantwortlich ist. Hierbei wird untersucht, ob in der Liste der übergebenen Parameter ein Flag gesetzt wurde und ob dieses Flag den Wert „q“ bzw. „Q“ vorweist. Denn nur diese Werte identifizieren bei der Eingabe über die Konsole eine Generierung einer qualifizierten Rolle.

```
01  def initialize(runtime_args, runtime_options = {})
02    @role_flag = false
03    checkFlag = runtime_args[0]
04    if checkFlag.size == 1 &&
05        (checkFlag == "Q" || checkFlag == "q") then
06      @role_flag = true
07      runtime_args.shift
08    end
```

Listing 28: roleScaffold – Überprüfung auf eine qualifizierte Rolle

Wurde ein dementsprechendes Flag gesetzt, wird die Variable `role_flag` auf den booleschen Wert `true` gesetzt und anschließend die Liste der Übergabeparameter präpariert. D.h. dass das Flag aus dieser Liste entfernt wird. Dies ist notwendig, da für den weiteren eine Parameterliste benötigt wird, welche eine bestimmte Reihenfolge vorweist. So wird beispielsweise das erste Element der Liste als Name der späteren Tabelle angesehen. Wenn also ein Flag gesetzt wurde, befindet sich dieses jedoch an erster Stelle der Liste und der gesuchte Name erst an zweiter. Durch die optionale Angabe des Flags muss also eine Überprüfung bzw. eine eventuelle Bereinigung erfolgen.

In Listing 29 wird nun die Zuweisung des `roleSuperTypes` behandelt, falls ein solcher angegeben wurde.

```
09      checkString = runtime_args[10]
10      if checkString[0,1].between?("A","Z") then
11        @role_super_type = checkString
12        @role_super_type_underscore_name =
    @role_super_type.underscore
13        runtime_args[10] = runtime_args[0]
14        runtime_args.shift
15      else
16        @role_super_type = NilClass
17        @role_super_type_underscore_name = nil
18      end
```

Listing 29: roleScaffold – Überprüfung des roleSuperType

Bei der Angabe zur Erstellung einer Rollenklasse – jedoch nicht einer Wurzelklasse – wird die Klasse der übergeordneten Rolle an der zweiten Stelle, nach dem Namen der zu generierenden Rollenklasse, angegeben. Diese befindet sich somit in der übergebenen Parameterliste an der Position 1. Der Wert dieses Elements wird zuerst auf deren Anfangsbuchstabe überprüft. Ist dieser in Großbuchstaben geschrieben handelt es sich um eine Klasse, also um die Angabe des `roleSuperTypes`. Somit wird dieser Wert

auf die dafür vorgesehene Variable gelegt und zusätzlich die Variable `role_super_type_underscore_name` gesetzt. Die Methode `underscore` ist dabei eine vordefinierte Methode, welche den Anfangsbuchstaben eines Strings in den dementsprechenden Kleinbuchstaben ändert. Anschließend wird das Parameterlistenelement an der ersten Stelle auf die zweite Stelle gesetzt und wiederum das Anfangselement von der Liste entfernt. Dadurch wurde die Angabe des `roleSuperTypes` aus der Liste gelöscht, wodurch diese nun für dieselbe Abarbeitung wie beim Scaffold-Generator verwendet werden kann.

Im nächsten Schritt wird nach der Angabe eines Fremdschlüssels für die Klasse eines `qualifiers` gesucht (siehe Listing 30). Denn wenn eine qualifizierte Rollenklasse erstellt werden soll, benötigt diese ein Objekt als `qualifier`, wodurch in der späteren Tabelle ein Verweis auf die Tabelle des entsprechenden Objektes bestehen muss.

```
19      checkAttrs = runtime_args[1..runtime_args.size]
20      qualFkFound = false
21      checkAttrs.each do |aAttr|
22        aAttr = aAttr.split(':')[0]
23        aAttrLength = aAttr.size
24        if aAttr[aAttrLength-3..aAttrLength] == "_id" &&
25            aAttr != "#{@role_super_type_underscore_name}_id" then
26          qualFkFound = true
27          @qualifier_type_underscore_name = aAttr[0..aAttrLength-4]
28          @qualifier_type =
    @qualifier_type_underscore_name.capitalize
29        end
30      end
31
32      if @role_flag && !qualFkFound then
33        raise NameError.new("Error: no foreign-key for qualifier!")
34      end
```

Listing 30: roleScaffold – Überprüfung der Attribute

Deswegen wird zuvor die Liste der übergebenen Parameter ohne den ersten Eintrag als neue Liste gespeichert, welche anschließend durchlaufen wird. Bei jedem Element dieser neuen Liste wird der String nach dem „:“ abgeschnitten und verworfen. Der Grund dafür besteht darin, dass ein Attribut in dem Format „`name:type`“ angegeben wird (z.B. `name:string`). Durch dieses Kürzen des Strings erhält man nun nur den Namen eines Attributs. Dieser Attributname wird hinterher auf seine letzten drei Zeichen überprüft. Gleichen diese letzten drei Zeichen dem String „`_id`“, kann davon ausgegangen werden, dass es sich dabei um einen Fremdschlüssel handelt. Begründet wird dies durch die Namenskonventionen von Ruby on Rails. Wenn nun dieses Attribut

mit der Endung „_id“ auch noch unterschiedlich zum Fremdschlüssel vom `roleSuperType` ist, wird dieses Attribut als Fremdschlüssel für den `qualifier` angesehen. Dadurch werden auch gleich die Variablen `qualifier_type` und `qualifier_type_underscore_name` gesetzt.

Sobald eine qualifizierte Rolle angelegt werden soll, aber kein Fremdschlüssel für den `qualifier` angegeben wurde, wird eine `Name-Error`-Exception mit einer entsprechenden Ausgabe geworfen.

Nach diesen Überprüfungen erfolgen grundsätzlich dieselben Aufgaben, wie der Aufruf zur Generierung der einzelnen Dateien, die auch vom scaffold-Generator erfüllt werden. Es wird nur zusätzlich die Generierung der Dateien object_or_role.rb bzw. role_helper.rb und die Überschreibung der Datei routes.rb befohlen. Die Beschreibungen dieser Dateien erfolgen neben denen der veränderten Dateien für die MVC-Architektur (model.rb, view.rb und controller.rb) in den folgenden Unterkapiteln.

5.3.1.1 Generierung der Model-Klasse

Die Generierung eines Models beinhaltet wohl die gravierendsten Änderungen des roleScaffold-Generators gegenüber dem Scaffold-Generator. Das begründet sich auch dadurch, dass ein Model als Rollenklasse gesehen werden kann und somit das Kernstück einer Rollenhierarchie darstellt. Ein Model bzw. eine Rollenklasse repräsentiert eine Ressource und wird im Webapplikationsverzeichnis im Ordner app/models/ in der Datei <resourceName>.rb abgespeichert.

Eine Klasse im Model, welche einen Rollentyp repräsentiert, muss über die Rollenmethoden verfügen. Beinahe alle diese Methoden sind im Modul `ObjectOrRole` beinhaltet (siehe Abschnitt 5.3.2.3), um dem DRY-Prinzip von Ruby on Rails gerecht zu werden. Lediglich die Methode `roleOf` wird in der Model-Klasse implementiert, worauf später näher eingegangen wird. Gleich zu Beginn der Erstellung eines neuen Models wird demnach das Modul `ObjectOrRole` eingebunden (siehe Listing 31). `ObjectOrRole` wird also als Mix-In verwendet. Wie in Listing 31 ersichtlich, werden auch hier ERb-Tags verwendet, um die Variablen aus dem `roleScaffold`-Generator zu übernehmen.

```
01 class <%= controller_singular_name %> < ActiveRecord::Base
02   include ObjectOrRole
03   <% if role_super_type != NilClass %>
04   belongs_to :<%= role_super_type_underscore_name %> <% end %>
05
06   @@roleSuperType = <%= role_super_type %>
07   @@corrList = []
08   @@privateList = []
09
10   attr_accessor :roleOf
11   attr_accessor :roles
12   attr_accessor :roleOfType
13   <% if role_flag %>attr_accessor :qualifier <% end %>
```

Listing 31: model.rb – Variable

Weiters werden die Klassenvariablen definiert und die Instanzvariablen deklariert. Die Klassenvariablen `corrList` und `privateList` werden dabei auf leere Listen gesetzt. Die Instanzvariable `qualifier` wird dabei nur für qualifizierte Rollen hinzugefügt.

Wie zuvor erwähnt, wird die `roleOf`-Methode direkt im Model generiert. Die nachfolgende Abbildung zeigt diese Methode.

```
14   def roleOf
15     <% if role_super_type == NilClass then %>
16       return nil
17     <% else %>
18       if self.<%= role_super_type_underscore_name %>_id then
19         @roleOf = <%= role_super_type %>.find(
20            self.<%= role_super_type_underscore_name %>_id)
21         return @roleOf
22       end
23     <% end %>
24   end
```

Listing 32: model.rb – roleOf-Methode

Der Grund, warum die `roleOf`-Methode im Model generiert werden muss, liegt in deren Funktion. Der Rückgabewert dieser Methode wird aus einer bestimmten Tabelle der Datenbank ermittelt, was nur über das Model möglich ist, da der `roleSuperType` für jede Rollenklasse unterschiedlich sein kann. Hier wird auch unterschieden, ob es sich bei dieser Klasse um eine Wurzelklasse handelt oder nicht. Ist dies der Fall, erzeugt der Generator lediglich eine Methode, die den Wert `nil` zurückgibt. Eine ähnliche Methode ist die `qualifier`-Methode. Diese wird jedoch nur für qualifizierte Rollen generiert.

Eine weitere Methode, die ebenfalls ihre Daten aus der Datenbank ermittelt und somit in der Model-Klasse erzeugt wird, ist die Navigationsmethode `roles`. Da ihre Metaprogrammierung sehr statisch ist, wird sie erst im Abschnitt 5.3.2 näher beschrieben und ihre Funktionalität näher erläutert. Auch in der method_missing-Methode sind nur wenige ERb-Tags enthalten, wodurch auch ihre Funktionalität erst im Abschnitt 5.3.2 beschrieben wird.

5.3.1.2 Generierung der Views

Der Generator erzeugt auch zusätzlich Dateien zur Präsentation von Daten. Beim Format dieser Views handelt es sich um .html.erb-Dateien, also um .html-Dateien mit inkludierten ERb-Tags. Diese können mit jedem Browser geöffnet werden, welcher HTML unterstützt. Standardmäßig werden zu jeder angelegten Ressource die Views edit.html.erb, index.html.erb, new.html.erb und show.html.erb angelegt. Diese Views einer Ressource befinden sich im Webapplikationsverzeichnis im Ordner app/views/<resourceName>/. Die Anzeigen dieser Views sind folgende:

- edit.html.erb zeigt ein Dokument zum Ändern der Daten einer Rolleninstanz,
- index.html.erb zeigt eine Liste sämtlicher Rolleninstanzen einer Rollenklasse,
- new.html.erb zeigt ein Formular zum Hinzufügen einer neuen Rolleninstanz und
- show.html.erb zeigt ein detailliertes Dokument einer einzelnen Rolleninstanz.

Ihre Generierungsvorlage ist dabei sehr ähnlich, wodurch nur die Datei show.html.erb in den folgenden Abbildungen beschrieben wird.

Grundsätzlich werden in einer View nur die grundlegenden Daten dargestellt. Dabei handelt es sich einerseits um die Inhalte einer Tabelle aus der Datenbank und andererseits um die rollenspezifischen Informationen, wie der übergeordneten Rolle, den untergeordneten Rollen, sowie dem Wurzelobjekt einer Rollenhierarchie. Je nachdem ob in der angegebenen URI auch Queries enthalten sind (durch das Anfügen von z.B. „?existsAs=Student“) werden diese über den Controller weitergeleitet und deren Ergebnis an die show-View geliefert, welche die Ergebnisse ausgibt. Listing 33 zeigt dabei die Generierung der Ausgabe von Tabelleninhalten aus der Datenbank. Hier

werden zum besseren Verständnis Variable aus anderen Dateien (z.B. `role_super_type` aus roleScaffold.rb) blau dargestellt.

```
01 <% roleOfAttr = attributes.pop if role_super_type != NilClass %>
02 <% qualifierAttr = attributes.pop if role_flag %>
03 <% for attribute in attributes %>
04   <p>
05     <b><%= attribute.column.human_name %>:</b>
06     <%%=h @<%= singular_name %>.<%= attribute.name %> %>
07   </p>
08 <% end %>
09 <% attributes.push(qualifierAttr) if role_flag %>
10 <% attributes.push(roleOfAttr) if role_super_type != NilClass %>
```

Listing 33: show.html.erb – Anzeige Tabellenattribute

Im ersten Schritt wird aus der Liste aller Attribute aus der Tabelle der Datenbank die `roleOf`-Beziehung (diese entspricht dem letzten Attribut, z.B. `person_id`) herausgespeichert, sofern es sich um eine untergeordnete Rollenklasse handelt. Als nächstes wird das Attribut aus der Liste entnommen, welches dem `qualifier` einer qualifizierten Rolle entspricht. In der Datenbank verfügt die jeweilige Tabelle dabei wiederum um eine Spalte an vorletzter Stelle, wenn es sich um eine qualifizierte Rolle handelt. Alle anderen Attribute werden in einem eigenen HTML-Paragrafen ausgegeben. Anschließend werden die zuvor entnommenen Attribute der Attributliste wieder angefügt, um diese nicht zu modifizieren.

In der nächsten Abbildung wird die Generierung der Ausgabe von der `roleOf`-Beziehung dargestellt. Wobei die Ausgabe von roleOf nur erfolgt, wenn ein übergeordnete Rolle in der Rollenhierarchie besteht. Ist dies der Fall wird versucht, eine Verlinkung auf dieses übergeordnete Rollenobjekt anzuzeigen. Hier besteht die Möglichkeit, dass die Webapplikation für dieses übergeordnete Rollenobjekt keinen Eintrag in der Routing-Liste vorfindet. Der Grund dafür kann daran liegen, dass dieses Objekt auf einer anderen Quelle zu finden ist, wodurch die URI dieses Objektes erzeugt wird und diese als Link dargestellt wird.

```
11 <% if role_super_type != NilClass %><p>
12   <b>roleOf:</b>
13     <%% begin %>
14       <% if @<%= singular_name %>.roleOf %>
15        <%%= link_to (@<%= singular_name %>.roleOf.name,
16                      roleOf_url) %>
17       <% end %>
18     <%% rescue Exception %>
19      <a href = "<%%= @<%= singular_name %>.roleOf.class.site %>
20                 <%%= @<%= singular_name %>
21                 .roleOf.class.to_s.pluralize.underscore %>/
22                 <%%= @<%= singular_name %>.roleOf.id %>">
23                 <%%= @<%= singular_name %>.roleOf.name %></a>
24     <%% end %>
25 </p><% end %>
```

Listing 34: show.html.erb – Anzeige roleOf

Ähnlich der Darstellung von der Beziehung `roleOf` aus Listing 34 erfolgt die Ausgabe der Methoden `root`, `qualifier` bzw. `roles`, weshalb diese hier nicht näher erläutert werden. Hingegen werden aber die Darstellungen von Rollenmethodenaufrufen in der Listing 35 aufgezeigt.

```
27 <%% if !@paramsHash.empty? then %>
28   <p><b>Methods: </b></p>
29   <%% @paramsHash.each do |aMeth, aParam| %>
30     <p>
31       <%% begin %>
32         <b><%%= aMeth %>(<%%= aParam.name %>): </b>
33       <%% rescue Exception %>
34         <b><%%= aMeth %>(<%%= aParam %>): </b>
35       <%% end %>
36       <%% value = @<%= singular_name %>.send(aMeth, aParam) %>
37       <%% begin %>
38         <%%= link_to (value.name, value) %>
39       <%% rescue Exception %>
40         <%%=h value %>
41       <%% end %>
42     </p>
43   <%% end %>
44 <%% end %>
```

Listing 35: show.html.erb – Anzeige Methoden

Die Variable `paramsHash` wird dabei vom Controller ermittelt und steht für die View zur Verfügung. Sie enthält sämtliche Methodenaufrufe, sofern welche über die URI als Query-Parameter (z.B. „…?existsAs=Student“) angehängt wurden. Diese Variable entspricht dabei einem Hash, welcher als Key den Methodennamen enthält und der dazugehörige Wert den Parameter beinhaltet. In Listing 35 wird dieser Hash durchlaufen und je nach Methodenaufruf die Methode ausgeführt und der Wert

entweder mit oder ohne Link ausgegeben. Ein Rückgabewert wird dabei mit Link ausgegeben, sofern es sich um eine Ressource mit einem Eintrag in der Routing-Liste handelt.

5.3.1.3 Generierung der Controller-Klasse

Zu jeder Ressource wird ein Controller angelegt, welcher für die Kommunikation und Navigation zwischen Model und View verantwortlich ist. Die Controller-Klasse einer Ressource wird dabei in einer Datei mit dem Name <resourceName>_controller.rb im Verzeichnis app/controllers der Webapplikation abgespeichert. Die Erstellung eines solchen Controllers erfolgt auch beim scaffold-Generator, wobei beim roleScaffold-Generator dieser Controller erweitert werden muss.

Die erste Erweiterung der Controller-Klasse besteht in der Inkludierung des REXML-Moduls. Dieses Modul stellt einen in Ruby erzeugten XML-Parser zur Verfügung. REXML steht dabei für **R**uby **E**lectric **XML** und wurde von der Electric XML Bibliothek von Java inspiriert (vgl. [28]). REXML stellt in Ruby on Rails einen Standard-Parser dar, weshalb auch Richardson und Ruby [29] REXML verwenden, obwohl sie auch Zweifel daran hegen. Nach ihnen ist dieses Modul „zu strikt, um schlechtes XML zu parsen, aber nicht strikt genug, um jedes schlechte XML abzulehnen“ (vgl. [29]). Durch das Einbinden dieses Moduls bestehen jedoch später mehrere Möglichkeiten das generierte XML-Dokument besser zu verwenden.

Auch die schon vom scaffold-Generator erstellte Methode `show` wird erweitert. In ihr wird zusätzlich ein Hash generiert, welcher mögliche Query-Parameter, die an der URI angehängt sind, abspeichert. In Listing 36 wird diese Methode dargestellt, wobei bei der Zuweisung eines Hash-Eintrages die Methode `getValue` aus dem Modul `RoleHelper` aufgerufen wird. Diese ermittelt aus dem mitgegebenen String-Wert das jeweilige Objekt bzw. die jeweilige Klasse. Beim Aufruf des XML-Formates wird dabei dieser Hash mit übergeben.

```
01  def show
02    @<%= file_name %> = <%= class_name %>.find(params[:id])
03
04    @paramsHash = Hash.new
05    request.query_parameters.each do |aKey, aValue|
06      @paramsHash[aKey] = RoleHelper.getValue(aValue)
07    end
08
09    respond_to do |format|
10      format.html # show.html.erb
11      format.xml  { render :xml => <%= class_name %>_to_xml(
12        @<%= file_name %>, @paramsHash) }
13    end
14  end
```

Listing 36: controller.rb – Methode show

Die nächste modifizierte Methode stellt `destroy` dar. Diese Action wird aufgerufen, wenn ein Objekt gelöscht werden soll. Durch die Erweiterung in Listing 37 wird ein Objekt erst dann gelöscht, wenn diese auch keine Subrollen mehr aufweist. Sobald Subrollen bestehen wird eine Flash-Nachricht ausgegeben, die darauf hinweist, dass zuerst die Subrollen gelöscht werden müssen. Ansonsten erfolgt eine Mitteilung, dass das jeweilige Objekt gelöscht wurde.

```
15  def destroy
16    @<%= file_name %> = <%= class_name %>.find(params[:id])
17    if @<%= file_name %>.roles.size == 0 then
18      flash[:notice] = "Deleted #{@<%= file_name %>.name}!"
19      @<%= file_name %>.destroy
20    else
21      flash[:notice] = 'Delete all sub-roles first!'
22    end
23
24    respond_to do |format|
25      format.html { redirect_to(<%= table_name %>_url) }
26      format.xml  { head :ok }
27    end
28  end
```

Listing 37: controller.rb – Methode destroy

Eine zusätzliche Erweiterung besteht in der Methode `roleOf` (siehe Listing 38). Mit Hilfe dieser Methode ist die Navigation auf das übergeordnete Rollenhierarchieelement mittels einer Uri der Form „`/:controller/:id/roleOf`“ möglich, wobei `:controller` und `:id` als Platzhalter fungieren.

```
29   def roleOf
30     @<%= file_name %> = <%= class_name %>.find(params[:id])
31
32     respond_to do |format|
33       format.html { redirect_to(@<%= file_name %>.roleOf)   }
34       format.xml  { render :xml => @<%= file_name %>.roleOf }
35     end
36   end
```

Listing 38: controller.rb – Methode roleOf

In dieser Abbildung ist der PHP-ähnliche ERb-Code ersichtlich, welcher hier als Platzhalter für die Variablen aus dem roleScaffold-Generator dienen. Die Metaprogrammierung erfolgt dabei in der Art, dass die Inhalte der Ruby-Datei einfach kopiert werden und der in ERb-Tags stehende Code jedoch abgearbeitet wird. Wobei der `file_name` dem Klassennamen (`class_name`) im Plural entspricht. Die daraus generierte Funktionalität wird später anhand eines Beispiels in Abschnitt 5.3.2 erklärt. Weiters werden zwei ähnliche Methoden erzeugt, welche aber auf den `root` einer Rollenhierarchie bzw. dem `qualifier` einer qualifizierten Rolle verweisen. Diese Methoden werden demnach `root` bzw. `qualifier` genannt. Die Methoden `roleOf`, `root` und `qualifier` unterscheiden sich dabei nur durch die Verweise in der `respond_to`-Schleife. Jedoch wird die Methode `qualifier` nur für Klassen angelegt, welche auf eine qualifizierte Rolle im Model verweisen.

Eine weitere Anpassung des scaffold-Generators liegt in der Generierung einer Methode zur Präsentation eines Objektes in XML-Format. Listing 39 zeigt das Metaprogramm dieser Methode. Die daraus generierte Funktionalität wird wiederum erst in Abschnitt 5.3.2 beschrieben, wobei grundsätzlich ein dem Model entsprechendes REXML-Dokument erstellt wird und dieses zurückgeliefert wird.

```
37   def <%= file_name %>_to_xml(<%= file_name %>, *rest)
38     string = <%= file_name %>.to_xml(:skip_types => true) do
   |xml|
39       <% if role_flag then %>
40         xml.qualifier(<%= file_name %>.qualifer.to_xml(
41                       :skip_instruct => true, :skip_types =>
   true))
42       <% end %>
43       if <%= file_name %>.roleOf then
44         xml.roleOf(<%= file_name %>.roleOf.to_xml(:skip_instruct
   => true, :skip_types => true))
45       end
46       xml.roles do
47         <%= file_name %>.roles.each do |aRole|
48           xml.tag!(aRole.class.to_s, aRole.to_xml(:skip_instruct
   => true, :skip_types => true))
49         end
50       end
51       if <%= file_name %>.root then
52         xml.root(<%= file_name %>.root.to_xml(:skip_instruct =>
53                  true, :skip_types => true))
54       end
55       if rest then
56         rest[0].each do |aMeth, aParam|
57           value = <%= file_name %>.send(aMeth, aParam)
58           if aMeth == "as" then
59             xml.as(value.to_xml(:skip_instruct => true,
60                     :skip_types => true), :param => aParam)
61           elsif aMeth == "existsAs" then
62             xml.existsAs(value, :param => aParam)
63           elsif aMeth == "entityEquiv" then
64             xml.entityEquiv(value, :param => aParam.name)
65           end
66         end
67       end
68     end
69        doc = Document.new(string)
70     return doc
71   end
```

Listing 39: controller.rb – XML-Repräsentation

5.3.1.4 Erweitern des Dispatchers

Der Dispatcher einer Rails-Webapplikation arbeitet mit der Datei routes.rb, welche sich im Verzeichnis /config befindet. Daher wird auch die Routing-List in dieser Datei einer Webapplikation durch den roleScaffold-Generator etwas erweitert (siehe Listing 40). So werden zusätzliche Einträge bezüglich der Verlinkung von `roleOf`, `root` und `qualifier` erstellt. Diese Einträge sagen aus, dass bei dem Aufruf einer URI, wie z.B. „.../employees/1/roleOf“, auf die Methode `roleOf` des `Employee`-Controllers zugegriffen wird. Die URI hat hier, wie oben bereits erwähnt, die Struktur „`/:controller/:id/:action`“ welche in der routes.rb analysiert wird.

```
01   base = '/:controller/:id'
02   map.roleOf '/roleOf', :path_prefix => base,
03              :controller => @controller, :action => 'roleOf'
04   map.roleRoot '/root', :path_prefix => base,
05                :controller => @controller, :action => 'root'
06   map.qualifier '/qualifier', :path_prefix => base,
07                 :controller => @controller, :action =>
```

Listing 40: Erweiterung in routes.rb

Je nachdem welcher Controller in der URI angegeben wird, wird dieser Controller geladen und die dementsprechende Action aufgerufen. In Zeile 04 der Abbildung fällt auf, dass die Action `root` die Bezeichnung `roleRoot` zugewiesen wurde. Dies liegt darin, dass die Bezeichnung `root` bereits von Ruby on Rails belegt wird und auf das Wurzelverzeichnis der Webapplikation verweist.

5.3.2 Generierter Rails-Code

Sobald der roleScaffold-Generator ausgeführt wird, erzeugt er die im vorherigen Kapitel beschriebenen Dateien. In diesem Abschnitt soll nun der dadurch erzeugte Code beschrieben werden. Hierfür wird die Rollenklasse Employee aus dem Beispiel von Abschnitt 1.4 erzeugt. Diese Rollenklasse enthält folgende Attribute `salary` und `phoneNo`. Um das Objekt eindeutig erkennen zu können wird zusätzlich das Attribut `name`[5] angefügt. Um die Namenskonventionen von Ruby on Rails einzuhalten wird auch das Attribut `person_id` hinzugefügt. Dieses Attribut dient der Datenbank als Fremdschlüssel auf das übergeordnete Rollenobjekt der Klasse Person. Anhand dieser Angaben wird der roleScaffold-Generator, mit dem Konsolenbefehl „*ruby script/generate roleScaffold Employee Person name:string salary:integer phoneNo:string person_id:integer*“, ausgeführt. Im Anschluss werden nun die wichtigsten generierten Dateien dargestellt.

[5] WICHTIG: Dieses Attribut soll zwar eindeutig sein, wird aber im generierten Code nicht als Identifikator herangezogen!

5.3.2.1 Generierte Model-Klasse

Die oben erwähnte Ausführung des roleScaffold-Generators erzeugt die Modelklasse employee.rb, welche in den folgenden Abbildungen dargestellt wird.

```
01 class Employee < ActiveRecord::Base
02   include ObjectOrRole
03
04   belongs_to :person
05
06   @@roleSuperType = Person
07   @@corrList = []          # ADD CorrClass-objects for
08                            # mapping-methods
09                            # (e.g. CorrClass.new("methName", 2,
10                            #        SubType, "methName1",
11                            #        SubType, "methName2")
12   @@privateList = []         # ADD attribute- or method-name, if
   it
13                            # should not inherited
14
15   attr_accessor :roleOf       # super-role
16   attr_accessor :roles        # Array of all sub-roles
17   attr_accessor :roleOfType   # type of roleOf (0..specialization,
18                               # 1..generalization, 2..annotation)
19
20   def roleSuperType
21     return @@roleSuperType
22   end
```

Listing 41: Model employee.rb – Variable

In Listing 41 wird neben dem Einbinden des Mix-Ins ObjectOrRole zuerst die Tabellenbeziehung in der Datenbank festgehalten und anschließend die Klassenvariablen gesetzt. Der roleScaffold-Generator setzt dabei die Klassenvariable `roleSuperType` auf die übergeordnete Rollenklasse. Die weiteren Klassenvariablen `corrList` und `privateList` werden als leere Listen definiert. Diese können später nach belieben gefüllt werden. Ein Beispiel für das Füllen der Corresponding-List wird dabei als Kommentar beigefügt. Weiters werden die Instanzvariablen `roleOf`, `roles` und `roleOfType` deklariert, wobei die Angabe `attr_accessor` aussagt, dass diese Variablen nicht in der Datenbank hinterlegt sind. Am Ende dieser Abbildung wird noch die Getter-Methode von `roleSuperType` angezeigt. Eine derartige Getter-Methode wird für die weiteren Klassenvariablen angelegt. Die Getter-Methoden der Instanzvariablen `roleOf` und `roles` sind dabei etwas komplexer. Sie werden in der nächsten Abbildung dargestellt.

```
23   def roleOf
24     @roleOf = Person.find(self.person_id) if self.person_id
25     return @roleOf
26   end
27
28   def roles
29     @roles = Array.new
30     roleArray = Array.new
31     RoleHelper.getRoleClasses.each do |aClass|
32       begin
33         roleArray += aClass.find(:all)
34       rescue Exception
35       end
36     end
37     roleArray.each do |aRole|
38       begin
39         if aRole.attributes.include?("employee_id") &&
40            aRole.employee_id == self.id then
41           @roles.push(aRole)
42         end
43       rescue Exception
44       end
45     end
46     return @roles
47   end
```

Listing 42: Model employee.rb – Methoden roleOf und roles

Die Methode `roleOf` hat die Aufgabe, die jeweilige Objekt aus der Tabelle Person zu ermitteln, welches die ID vorweist, auf welche der Fremdschlüssel `person_id` der aktuellen `Employee`-Instanz verweist, sofern dieser gesetzt ist. Die Methode `roles` hingegen führt die Modul-Methode `getRoleClasses` vom Modul `RoleHelper` auf. Diese gibt eine Liste sämtlicher Rollenklassen zurück. Anhand dieser Rollenklassen werden alle Instanzen dieser Rollenklassen in eine Liste gespeichert und diese Instanzen auf das Attribut `employee_id` überprüft. Wenn eine Rolleninstanz dieses Attribut besitzt und dieses den Wert der ID der aktuellen `Employee`-Instanz aufweist, so wird es in die Liste `roles` eingefügt. Wurden alle Rolleninstanzen durchlaufen, wird die gesamte `roles`-Liste zurückgeliefert. Hierbei sollte noch angemerkt werden, dass die erwähnte ID von der Datenbank gesetzt wird und diese grundsätzlich nur für die Suche in den Tabellen über ActiveRecord verwendet wird.

In der nächsten Abbildung wird die `method_missing`-Methode überschrieben, wie für das Method-Dispatching zuständig ist.

```
48   def method_missing(methId, *rest, &block)
49     searchedRoleList = Array.new
50     if rest[0].instance_of?(Array) then
51       if rest[0][0].superclass == ActiveRecord::Base then
52         searchedRoleList = rest[0]
53         rest.shift
54       end
55     end
56     begin
57       super(methId, *rest, &block)
58     rescue          # method not found in actual role
59       @roleOf = Person.find(self.person_id)
60       @roles = Array.new
61       RoleHelper.getRoleClasses.each do |aClass|
62         begin
63           @roles += aClass.find(:all, :conditions =>
64                                 ["employee_id = ?", self.id])
65         rescue Exception
66         end
67       end
68       role_method_missing(methId, self, @roleOf, @roles,
69                           searchedRoleList, *rest, &block)
70     end
71   end   # end method_missing
```

Listing 43: Model employee.rb – method_missing

In dieser Methode wird zuvor überprüft, ob beim Aufruf der Methode eine Liste mit übergeben wird und die Einträge dieser Liste Rollenklassen sind. Wenn dies der Fall ist, kann davon ausgegangen werden, dass es sich dabei um die `searchedRoleList` handelt. Diese Liste enthält sämtliche Klassen, die von der `method_missing`-Methode bereits durchsucht wurde. Anschließend wird versucht, die gewünschte Methode im jeweiligen Rollenobjekt mittels der vordefinierten `method_missing`-Methode aufzurufen (dies geschieht mit dem Aufruf von super). Wenn diese Methode nicht gefunden wird, wird eine Exception geworfen, welche von der hier definierten `method_missing`-Methode wieder abgefangen wird. Nun erfolgt die Suche in der Rollenhierarchie mittels der Mix-In-Methode `role_method_missing`, welche sich im Modul `ObjectOrRole` befindet. Ihre Beschreibung befindet sich im Abschnitt 5.3.2.3. Für den Aufruf dieser Methode werden die Variable `roleOf` und `roles` benötigt. Diese werden in der `method_missing`-Methode neu gesetzt. Der Grund dafür liegt daran, dass beim Aufruf der jeweiligen Getter-Methode, welche in Listing 42 beschrieben wurden, wiederum die `method_missing`-Methode mit der jeweiligen Getter-Methode erfolgen würde. Dies würde zu einer Endlosschleife führen.

5.3.2.2 Generierte View-Dokumente

Der roleScaffold-Generator erzeugt auch die vier View-Dokumente edit.html.erb, index.html.erb, new.html.erb und show.html.erb, wobei hier wiederum nur auf die show.html.erb eingegangen wird.

Zu Beginn des Dokuments werden die Daten aus der Datenbank angezeigt, jedoch ohne den Fremdschlüsseln (siehe Listing 44). In dieser Abbildung ist ersichtlich, dass der Zugriff auf Methoden einer Rolleninstanz einfach möglich ist, wobei die Variable `employee` vom Controller an die View übergeben wird.

```
01 <p>
02   <b>Name:</b>
03   <%=h @employee.name %>
04 </p>
05 <p>
06   <b>Salary:</b>
07   <%=h @employee.salary %>
08 </p>
09 <p>
10   <b>Phoneno:</b>
11   <%=h @employee.phoneNo %>
12 </p>
```

Listing 44: Employee-View show.html.erb – Tabellendaten

In der nachfolgenden Listing werden hingegen die Rollenattribute einer Rolleninstanz von Employee dargestellt. Hier ist der Aufbau einer Ausgabe grundsätzlich überall gleich: Zu Beginn wird versucht einen Link über die Routing-Liste auf das jeweilige Objekt zu erstellen. Falls dies jedoch eine Exception wirft, weil z.B. dieses Objekt auf einem anderen Server liegt und somit kein Verweis auf deren Controller in der Routing-Liste eingetragen ist, wird ihre direkte URI zusammengestellt und diese als Link angeboten. Eine Ausnahme macht hier aber die Ausgabe des Wurzelobjektes einer Rollenhierarchie. Falls diese auf einer entfernten Quelle liegt, erfolgt die Übermittlung der Daten dieses Objektes mittels XML. Dabei wird das XML-Dokument als String übermittelt und nicht als Objekt selbst. Dies hat zur Folge, dass die Ausgabe des Wurzelobjektes der einfachen Ausgabe dieses XML-Dokuments entspricht.

```
13 <p>
14   <b>roleOf:</b>
15     <% begin %>
16      <% if @employee.roleOf %>
17        <%= link_to (@employee.roleOf.name, roleOf_url) %>
18      <% end %>
19     <% rescue Exception %>
20      <a href = "<%= @employee.roleOf.class.site %>
21       <%= @employee.roleOf.class.to_s.pluralize.underscore %>/
22       <%= @employee.roleOf.id %>"><%= @employee.roleOf.name
   %></a>
23     <% end %>
24 </p>
25
26 <p>
27   <b>root:</b>
28     <% begin %>
29      <%= link_to (@employee.root.name, roleRoot_url) %>
30     <% rescue Exception %>
31      <%=h @employee.root.to_s %>
32     <% end %>
33 </p>
34
35 <p>
36   <b>Roles: </b>
37   <% for subRole in @employee.roles %>
38     <% begin %>
39      <%= link_to subRole.name, subRole %>
40     <% rescue Exception %>
41      <a href = "<%= subRole.class.site %>
42        <%= subRole.class.to_s.pluralize.underscore %>/
43        <%= subRole.id %>"><%= subRole.name %></a>
44     <% end %>
45   <% end %>
46 </p>
```

Listing 45: Employee-View – Rollenattribute

In dieser Listing ist auch der Aufruf der Methoden `pluralize` und `underscore` interessant. Dies sind vordefinierte Funktionen, welche in diesem Beispiel den Klassennamen (z.B. „Employee“) umformen. Der Aufruf `pluralize` ändert den String indem er die Mehrzahl davon bildet (z.B. „Employees“). Bei geläufigen englischen Worten wird auch die irreguläre Mehrzahl gebildet (z.B. aus „Person“ wird „People“). Die Funktion `underscore` verändert den String, indem Großbuchstaben in Kleinbuchstaben geändert werden und diejenigen Großbuchstaben, die sich nicht am Anfang des Strings befinden werden in Kleinbuchstaben geändert und zusätzlich ein „_“ davor gesetzt (z.B. „DepartmentMgr“ wird zu „department_mgr“). Durch diese beiden Funktionen wird aus dem Klassennamen somit der Controller-Namen gebildet.

Zusätzlich zu den Attributen können auch die Rollenmethoden ausgeführt und deren Rückgabewert ausgegeben werden. Wird die URI einer `Employee`-Instanz durch eine

Query erweitert, wird versucht diese Query-Angaben als Methoden auszuführen So wird beispielsweise durch die URI „…/employees/1?existsAs=Person“ die Methode `existsAs(Person)` bei der jeweiligen `Employee`-Instanz ausgeführt.

```
47 % if !@paramsHash.empty? then %>
48   <p><b>Methods: </b></p>
49   <% @paramsHash.each do |aMeth, aParam| %>
50     <p>
51     <% begin %>
52       <b><%= aMeth %>(<%= aParam.name %>): </b>
53       <% rescue Exception %>
54         <b><%= aMeth %>(<%= aParam %>): </b>
55       <% end %>
56       <% value = @employee.send(aMeth, aParam) %>
57       <% begin %>
58         <%= link_to (value.name, value) %>
59       <% rescue Exception %>
60         <%=h value %>
61       <% end %>
62     </p>
63   <% end %>
64 <% end %>
```

Listing 46: Employee-View – Rollenmethoden

Die Query-Elemente befinden sich dabei in dem Hash `paramsHash`, wobei die Methode dem Key eines Hash-Eintrages entspricht, während der Parameter dem Value dieses Eintrages widerspiegelt. Für jede angegebene Methode wird eine Ausgabe formuliert und wiederum das jeweilige Ergebnis der Methode entweder als Link oder als einfacher Wert ausgegeben.

5.3.2.3 Generierte Controller-Klassen

Der roleScaffold-Generator erstellt drei Dateien innerhalb des Controller-Verzeichnisses einer Webapplikation. Diese sind neben dem `Employee`-Controller (employees_controller.rb) die Module `ObjectOrRole` (object_or_role.rb) und `RoleHelper` (role_helper.rb). In diesem Abschnitt wird auf nun auf diese Dateien näher eingegangen.

Zunächst werden wichtige Funktionalitäten des Employee-Controllers erläutert. Sobald eine Rolleninstanz detailliert betrachtet wird (z.B. durch die HTTP-Methode „GET employees/1“) wird die Methode show im Employee-Controller aufgerufen, welche am Ende auf die show-View wechselt. Diese Methode wird in Listing 47 dargestellt.

```
01   def show
02     @employee = Employee.find(params[:id])
03
04     @paramsHash = Hash.new
05     request.query_parameters.each do |aKey, aValue|
06       @paramsHash[aKey] = RoleHelper.getValue(aValue)
07     end
08
09     respond_to do |format|
10       format.html # show.html.erb
11       format.xml  { render :xml => employee_to_xml(@employee,
12                                    @paramsHash) }
13     end
14   end
```

Listing 47: Employee-Controller – Methode show

In dieser Listing werden zuerst die Variablen definiert. Diese Variablen sind später auch von der View zugänglich. Im nächsten Schritt wird der `paramsHash` gefüllt. Die Funktionalität in Zeile 05 sucht dabei in der aufgerufenen URI nach möglichen Queries, welche in einem Hash gespeichert werden. Dieser Hash wird durchlaufen und jeder Value – der als String abgelegt wird – anhand der Funktion `getValue` des Moduls `RoleHelper` in eine Klasse bzw. eine Rolleninstanz umgewandelt. Dieser umgewandelte Wert wird im `paramsHash` als Wert zum jeweiligen Key gespeichert. Wenn später das XML-Dokument generiert werden soll, wird dieser Hash mit übergeben. Die Generierung eines XML-Dokuments erfolgt durch die Methode employee_to_xml und wird in Listing 48 dargestellt.

```
15   def employee_to_xml(employee, *rest)
16     string = employee.to_xml(:skip_types => true) do |xml|
17       xml.roleOf(employee.roleOf.to_xml(:skip_instruct => true,
18             :skip_types => true)) if employee.roleOf
19       xml.roles do
20         employee.roles.each do |aRole|
21           xml.tag!(aRole.class.to_s, aRole.to_xml(:skip_instruct
22                       => true, :skip_types => true))
23         end
24       end
25       xml.root(employee.root.to_xml(:skip_instruct => true,
26             :skip_types => true)) if employee.root
27       if rest
28         rest[0].each do |aMeth, aParam|
29           value = employee.send(aMeth, aParam)
30           if aMeth == "as"
31             xml.as(value.to_xml(:skip_instruct => true,
32               :skip_types => true), :param => aParam)
33           elsif aMeth == "existsAs"
34             xml.existsAs(value, :param => aParam)
35           elsif aMeth == "entityEquiv"
36             xml.entityEquiv(value, :param => aParam.name)
37           end
38         end
39       end
40     end
41     doc = Document.new(string)
42     return doc
43   end
```

Listing 48: Employee-Controller – Methode employee_to_xml

Zu Beginn dieser Methode wird ein String erzeugt. Dieser erhält zuerst die standardmäßige XML-Repräsentation eines Objektes. Dazu zählen sämtliche Attribute aus der Datenbank. Zusätzlich zu diesen Attributen werden noch Tags für das übergeordnete Objekt (`roleOf`), den untergeordneten Objekten, sowie dem Wurzelobjekt angelegt, sofern diese Angaben auch existieren. Weiters können auch noch die zusätzlichen Rollenmethoden ausgeführt werden und deren Ergebnis in das XML-Dokument aufgenommen werden. Wie zuvor angesprochen wird die Variable `paramsHash` an diese Methode übergeben, wodurch auch die aufzurufenden Methoden im Hash zur Verfügung stehen. In Zeile 29 wird der Hash durchlaufen und je nachdem um welche Methode es sich handelt ein dementsprechender Tag angefügt. Abschließend wird dieser erstellte String in ein XML-Dokument – genauer gesagt in ein REXML-Dokument – umgewandelt und dieses zurückgegeben.

Im nächsten Schritt wird auf das Modul ObjectOrRole eingegangen. Dieses Modul enthält folgende Methoden, die jedem Rollenobjekt zur Verfügung stehen:

- `role_method_missing(methId, role, roleOf, subRoles, searchedRoleList)`,
- `findInPrivateList(role, methName)`,
- `getAllSubRoles`,
- `root`,
- `as(otherRoleType)`,
- `existsAs(otherRoleType)`,
- `entityEquiv(anObject)`,
- `abandon`,
- `as_of(aQualifiedRoleType, qualObj)` und
- `existsAs_of(aQualifiedRoleType, qualObj)`, sowie
- die Klasse `CorrClass`.

Die meisten dieser Methoden sind im Vergleich zu den aus Kapitel 4.3 fast unverändert und haben auch dieselben Aufgaben. Deswegen wird nun nur auf die Methode `role_method_missing` näher eingegangen. Deren Code wird in Listing 49 dargestellt.

```
01 def role_method_missing(methId, role, roleOf, subRoles,
02                          searchedRoleList, *rest, &block)
03     methName = methId.id2name
04     if !searchedRoleList.include?(role.class) then
05       searchedRoleList.push(role.class)
06
07       if role.corrList then
08         role.corrList.each do |aCorr|
09           if aCorr.varName == methName then
10             return aCorr.getCorrValue(role, *rest)
11           end
12         end
13       end
14
15       subRoles.each do |aRole|
16         if !findInPrivateList(aRole, methName) then
17           return aRole.send(methId, searchedRoleList,
18                             *rest, &block)
19         end
20       end
21     end
22
23     if roleOf then
24       if !findInPrivateList(roleOf, methName) then
25         return roleOf.send(methId, searchedRoleList,
26                            *rest, &block)
27       end
28     end
29
30     raise NoMethodError.new("Method '#{methName}' not found")
31   end
```

Listing 49: ObjectOrRole – role_method_missing

Diese Methode hat grundsätzlich die Aufgaben, welche die `method_missing`-Methode in Abschnitt 4.3.3.2 aufweist. Nur mit dem Unterschied, dass hier nicht mehr in der eigenen Klasse nach der gewünschten Methode gesucht werden muss.

Am Beginn dieser Methode wird zuvor überprüft, ob in der Klasse des jeweiligen Objektes (`role`) bereits nach der gewünschten Methode gesucht wurde. Wenn dies nicht der Fall ist, folgt die Durchsuchung der Corresponding-List des derzeitigen Rollenobjektes. Falls auch die Suche dort erfolglos bleibt wird die gewünschte Methode an die Subrolleninstanzen weitergeleitet, sofern sich kein Eintrag dieser Methode in der Private-Liste der jeweiligen Subklasse findet. Bleibt auch die Suche bei den Subrollen erfolglos wird die Methode an das übergeordnete Rollenobjekt weitergereicht, wenn auch hier kein Private-Listeneintrag besteht. Wurde die Methode auch dort nicht gefunden wird eine Exception geworfen, wodurch die Suche fehlgeschlagen ist.

Zuletzt wird in diesem Abschnitt noch das Modul `RoleHelper` vorgestellt. Dieses Modul beinhaltet Methoden, die einerseits für die Suche nach Rollenklassen verantwortlich ist und andererseits Strings in Rollenklassen bzw. Rollenobjekte umwandeln soll. In diesem Modul befinden sich insgesamt vier Methoden:

- `getRoleClasses`,
- `getRoleClass(str)`,
- `getRole(str)` und
- `getValue(str)`.

Die erste dieser Methoden wird in der folgenden Listing dargestellt. Sie wechselt zu Beginn in das Verzeichnis der Models. In diesem Verzeichnis werden alle Dateien durchlaufen, wobei jeder Datei die Erweiterung „.rb“ entfernt wird und somit ihr Name zur Verfügung steht. Dieser Name, welcher als String zur Verfügung steht, wird durch die Funktionen `camelize` und `constantize` in eine Klasse umgewandelt und diese in eine Liste gespeichert. Diese Funktionen haben dabei folgende Aufgaben. Erstere wandelt Strings der Art „department_mgr“ in „DepartmentMgr“ um, wobei letztere versucht eine Konstante zu finden, welche den Namen des Strings trägt (z.B. „Employee“ zu `Employee`).

```
01  def RoleHelper.getRoleClasses
02    roleClasses = Array.new
03    if !Dir.getwd.include?("/app/models") then
04      Dir.chdir(Dir.getwd + "/app/models")
05    end
06    Dir.foreach(Dir.getwd) {|aFile|
07      if aFile.include?(".rb") then
08        aRole = aFile.split(".")[0].camelize.constantize
09        roleClasses.push(aRole)
10      end
11    }
12    return roleClasses
13  end
```

Abbildung 20: RoleHelper – Methode getRoleClasses

Die Methoden `getRoleClass` und `getRole` haben ähnliche Aufgaben. Beide Versuchen einen String zu konvertieren (siehe Listing 50). Die Methode `getRoleClass` sucht nach einer Rollenklasse mit dem Name des übergebenen Strings, wobei `getRole` nach einer Rolleninstanz sucht.

```
14   def RoleHelper.getRoleClass(str)
15     RoleHelper.getRoleClasses.each do |aClass|
16       if aClass.name == str then
17         return aClass
18       end
19     end
20     return NilClass
21   end
22
23   def RoleHelper.getRole(str)
24     RoleHelper.getRoleClasses.each do |aClass|
25       allRoles = aClass.find(:all)
26       allRoles.each do |aRole|
27         if aRole.name == str then
28           return aRole
29         end
30       end
31     end
32     return nil
33   end
```

Listing 50: RoleHelper – Methoden getRoleClass und getRole

Die letzte Methode dieses Moduls ruft, je nach Anfangsbuchstaben des übergebenen String, entweder die Methode `getRoleClass` oder die Methode `getRole` auf. Sie wird in der folgenden Listing beschrieben.

```
34   def RoleHelper.getValue(str)
35     if str[0..1].between?('A', 'Z') then
36       return RoleHelper.getRoleClass(str)
37     else
38       return RoleHelper.getRole(str)
39     end
40   end
```

Listing 51: RoleHelper - Methode getValue

5.3.3 Einschränkungen bzgl. Ruby on Rails

Leider bestehen aufgrund von Ruby on Rails-internen Abläufen noch Einschränkungen bei der Verwendung einer mit roleScaffold generierten Webapplikation. Grundsätzlich ist es möglich, wenn Teile einer Rollenhierarchie verteilt auf mehreren Quellen gespeichert werden. Wenn beispielsweise die Rollenklasse Person auf einem Server und die Rollenklasse Employee auf einem anderen Server platziert sind. In diesem Fall kann es zu Timeouts kommen, sobald beide Server über die Existenz des anderen Servers Bescheid wissen und sie eine Verlinkung auf den jeweils anderen Server vorweisen. Wenn hier auf die untergeordnete Rolle zugegriffen werden soll, verfolgt Ruby on Rails

wiederum die `roleOf`-Beziehung dieser Rolle, wodurch wieder auf den ersten Server verwiesen wird.

In der Generierung einer Webapplikation anhand des roleScaffold-Generators besteht weiters eine Einschränkung bezüglich der Verteilung einer einzelnen Tabelle. Eine Tabelle befindet sich genau in einer Datenbank. Dies bedeutet, dass die Suche in einer Tabelle nur auf einer Quelle erfolgt. Sämtliche Elemente einer Tabelle werden also nicht verteilt abgespeichert.

Eine weitere Einschränkung besteht an der Übertragung zwischen zwei oder mehreren Servern. Die Übertragung von Objekten erfolgt über ein erzeugtes REXML-Dokument, was einerseits erwünscht ist, jedoch sind die Elemente dieses Dokumentes Strings, wodurch beispielsweise ein Zugriff auf den `root`-Tag lediglich die XML-Repräsentation dieses Objektes liefert, nicht aber das Objekt selbst.

5.4 Evaluierung

In diesem Kapitel wurden wiederum Änderungen vorgenommen um den aus Kapitel 4.3 entwickelten Ruby-Code einerseits an Ruby on Rails anzupassen und um einen Generator zu erstellen, der eine Webapplikation generiert, welche zur Verwendung des erweiterten Rollenkonzeptes eingesetzt werden kann. Dafür wurden Anforderungen analysiert und geänderte Rahmenbedingungen betrachtet. Hier stachen insbesondere die Anforderungen der Identifikation und Repräsentation von Objekten über ein dezentrales Informationssystem, sowie die Einschränkung für eine dynamische Klassenerzeugung heraus. Auf dieser Analyse aufbauend wurden konzeptionelle Lösungen gesucht um einen passenden Ruby on Rails-Code zu implementieren. Dabei zeigten sich auch einige Einschränkungen bei der Verwendung, welche durch Ruby on Rails entstanden. Für die einfache Verwendung dieser Umsetzung des erweiterten Rollenkonzeptes in Ruby on Rails wurde ein Generator entwickelt, mit deren Hilfe Webapplikationen erstellt werden können, die die Funktionalitäten des erweiterten Rollenkonzeptes beinhaltet.

Im World Wide Web werden Rollenmodelle bisher nur selten zur Verfügung gestellt. Doch die Möglichkeit zur Verwendung dieses erweiterten Rollenmodells im World

Wide Web soll neben dem Tabulator von Huemer [13] und der Webapplikation von Rossi et al. [32] ein weiterer Schritt zur Verbreitung von Rollenmodellen sein.

6 Zusammenfassung

In dieser Diplomarbeit wurden auf die unterschiedlichen Ansätze für die Modellierung und Implementierung von Rollenkonzepte in objektorientierten Systemen aus der Literatur eingegangen. Insbesondere das Rollenmodell nach Gottlob et al. [11] wurde näher beschrieben, woraufhin dieses Rollenkonzept in der Programmiersprache Ruby umgesetzt wurde. Diese Implementierung wurde daraufhin mit bereits bestehenden Umsetzungen in den Programmiersprachen Smalltalk und Java verglichen.

Aufbauend auf dieser Implementierung wurde ein erweitertes Rollenmodell entwickelt, welches für einen Einsatz in dezentralen Informationssystemen gerüstet ist. Dabei wurden insbesondere Funktionalitäten umgesetzt, die die Navigationseinschränkungen durch mögliche Zugriffsbeschränkungen mindern, die Existenz von dezentralen Teilrollenhierarchien erlaubt, die Erstellung einer Rollenhierarchie, neben Top-Down- auch mittels Bottom-Up-Ansatz gewährleistet und einen Mapping-Ansatz inkludiert, wodurch eine unterschiedliche Namensgebung für Attribute und Methoden gehandhabt werden können. Diese Funktionaltiäten beeinflussten dabei vor allem das Konzept des Method-Dispatching, welches dahingehend erweitert wurde. Weiters wurde die Möglichkeit implementiert, Attribute als privat zu kennzeichnen, wodurch ein Aufruf dieser Attribute nicht über andere Rollen möglich ist.

Damit dieses erweiterte Rollenkonzept einer breiten Masse zur Verfügung gestellt werden kann, wurde ein Generator für das Web Application Framework Ruby on Rails implementiert, welcher die Grundstruktur einer Webapplikation mit der erweiterten Rollenfunktionalität generiert. Für die Erstellung dieses Generators wurden Anpassungen an das erweiterte Rollenmodell, um Anforderungen durch das World Wide Web und Ruby on Rails zu entsprechen. REST ist dabei ein Architekturstil, der diesen Anforderungen entspricht und einen einfachen Informationsaustausch im World Wide Web gewährleistet.

7 Anhang

Im Anhang befinden sich, neben sämtlichen, für die Verwendung des roleScaffold-Generators, benötigte Installationen und Konfigurationen und einer Anleitung zur Verwendung dieses Generators, auch der gesamte Code der drei beschriebenen Teilprojekte dieser Diplomarbeit.

7.1 Installation und Konfiguration

In diesem Kapitel wird die Installation und Konfiguration für Windows-Betriebssysteme[6] beschrieben. Die angegebenen Versionen der einzelnen Komponenten entsprechen den Versionen, die bei der Implementierung verwendet wurden.

Um eine Webapplikation anhand des roleScaffold-Generators zu erstellen, sind einige wenige Installationen zu tätigen. Die Grundvoraussetzung bildet dabei die Installation von Ruby. Hierfür eignet sich der Ruby 1.8.6 One-Click Installer, welcher auf der Homepage von Ruby[7] verfügbar ist. Durch den Aufruf des „RubyGems Package Managers“ können anschließend weitere benötigte Komponenten installiert werden. Dabei sind folgende Befehle einzugeben:

1. `install rubygems` – RubyGems ist das offizielle Paketsystem von Ruby, womit zusätzliche Bibliotheken etc. installiert werden können.
2. `ruby install rake -v=0.8.4` – Rake steht für **R**uby m**ake** und ist ein Werkzeug zur Erstellung von Komponenten.
3. `ruby install rails -v=2.0.2` – Dies dient zur Installation der Ruby on Rails Komponenten.
4. `ruby install sqlite3-ruby -v=1.2.5` – Dieser Befehl installiert Komponenten für die Verwendung einer SQLite3-Datenbank. Hierfür ist auch die Installation der SQLite3-Datenbank notwendig, welche im Anschluss beschrieben wird.

[6] Installationsanleitungen für andere Betriebssysteme befinden sich auf http://www.ruby-lang.org/de/.
[7] http://www.ruby-lang.org/de/downloads/. Letzter Zugriff: 13. Dezember 2009.

Abschließend ist noch der roleScaffold-Ordner[8] im Verzeichnis „…/ruby/lib\ruby\gems\1.8\gems\rails-2.0.2\lib\rails_generator\generators\components“ zu kopieren.

Für die Installation der SQLite-Datenbank sind folgende Schritte notwendig:

1. Downloaden des komprimierten Verzeichnisses sqlite-3_6_16[9].
2. Downloaden des komprimierten Verzeichnisses sqlitedll-3_6_16.
3. Entpacken des Verzeichnisses sqlite-3_6_16.
4. Entpacken des Verzeichnisses sqlitedll-3_6_16 in den Ordner „…/ruby/bin“.

7.2 Anleitung zur Verwendung des roleScaffold-Generators

Zur Generierung einer Webapplikation anhand des roleScaffold-Generators werden im Folgenden die Befehle zur Erstellung der Webapplikation für das Beispiel aus Kapitel 1.4 angeführt. Diese Befehle sind über die Konsole des „RubyGems Package Managers“ einzugeben.

1. ***…/ruby***>rails rolesApp

   Dieser Schritt erzeugt die Grundstruktur einer rails-Webapplikation (siehe Abbildung 21)

[8] Der Ordner roleScaffold ist am Institut für Wirtschaftsinformatik – Data & Knowledge Engineering oder vom Autor erhältlich.
[9] Download erhältlich unter http://www.sqlite.org/download.html. Letzter Zugriff: 30. Juli 2009.

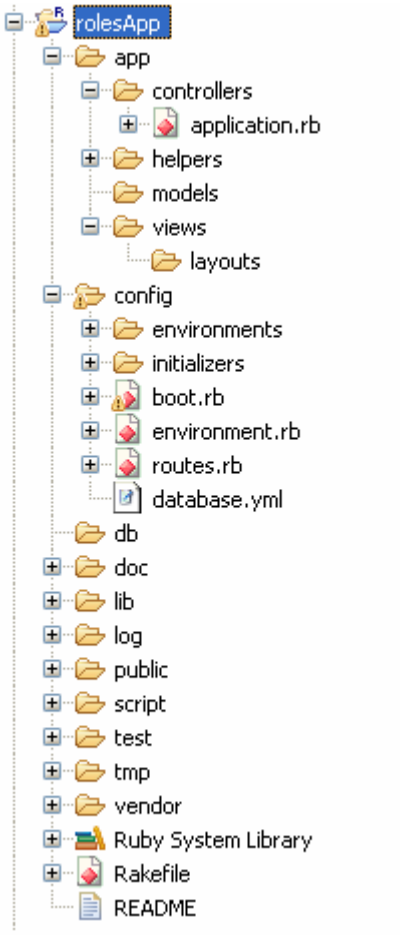

Abbildung 21: Grundstruktur einer rails-Webapplikation

2. ***…/ruby***>cd rolesApp
3. ***…/ruby/rolesApp***>ruby script/generate roleScaffold Person name:string phoneNo:string birthDate:date
4. ***…/ruby/rolesApp***>ruby script/generate roleScaffold Employee Person name:string salary:integer phoneNo:string person_id:integer
5. ***…/ruby/rolesApp***>ruby script/generate roleScaffold DepartmentMgr Employee name:string sal:integer phoneNo:string employee_id:integer
6. ***…/ruby/rolesApp***>ruby script/generate scaffold Project name:string description:text
7. ***…/ruby/rolesApp***>ruby script/generate roleScaffold q ProjectMgr Employee name:string skills:text project_id:integer employee_id:integer
8. ***…/ruby/rolesApp***>ruby script/generate roleScaffold Student Person name:string university:string sship:integer phoneNo:string person_id:integer

   Die Befehle von Schritt 3 – 8 erzeugen die jeweiligen Dateien, wodurch sich die Webapplikationsstruktur von Abbildung 21 zu jener in Abbildung 22 erweitert.

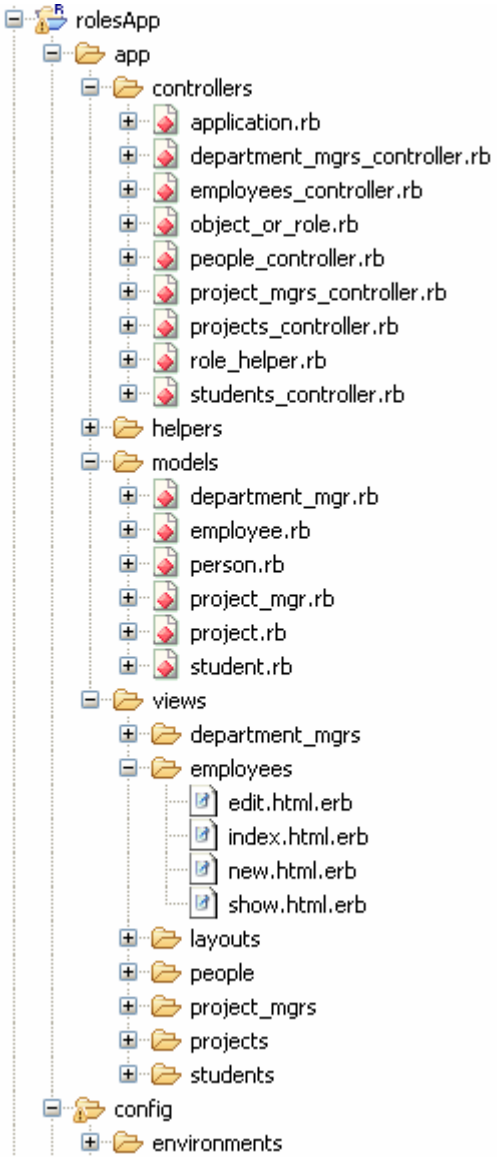

Abbildung 22: Erweiterte Struktur durch roleScaffold

9. ***…/ruby/rolesApp***>rake db:migrate

   Dieser Befehl erzeugt die Tabellen in der SQLite-Datenbank

10. ***…/ruby/rolesApp***>ruby script/server

    Dieser Befehl startet den WEBrick-Server

Über einen Webbrowser kann nun mit der URL „http://localhost:3000/“ auf den Server zugegriffen werden. Um beispielsweise auf die Ansicht aller Personen zu kommen, muss zusätzlich der String „people“ angehängt werden.

7.3 Gesamter Ruby-Code und roleScaffold-Generator-Code

7.3.1 Ruby-Code für Rollen in Ruby

Helper mix-in, which includes attributes and methods of a role

```
01 module AbstractObjectOrRole
02   attr_reader :roleOf
```

Returns super-role (roleOf)

```
03   def roleOf
04     return @roleOf
05   end
```

Returns the real world object (=root) of a role-hierarchy

```
06   def root
07     tempRole = self
08     while tempRole.roleOf != nil do
09       tempRole = tempRole.roleOf
10     end
11     return tempRole
12   end
```

Checks, if there is an object of "otherRoleType" of the same real world object as self returns the other object or nil

```
13   def as(otherRoleType)
14     if otherRoleType == self.root.class then
15       return self.root
16     end
17     self.root.getAllSubRoles.each do |tempRole|
18       if tempRole.class == otherRoleType ||
19         tempRole.class.superclass == otherRoleType then
20         return tempRole
21       end
22     end
23     return nil
24   end
```

Checks, if there is an object of "otherRoleType" of the same real world object as self returns true or false

```
25   def existsAs(otherRoleType)
26     if otherRoleType == self.root.class then
27       return true
28     end
29     self.root.getAllSubRoles.each do |tempRole|
30       if tempRole.class == otherRoleType ||
31         tempRole.class.superclass == otherRoleType then
32         return true
33       end
34     end
35     return false
36   end
```

Checks, if self and "anObject" are from the same real world object; returns true if they do, unless false

```
37   def entityEquiv(anObject)
38     if self.root == anObject.root then
39       return true
40     else
41       return false
42     end
43   end
```

Checks, if there is an object of QualifiedRoleType of self with the qualifier "qualObj" returns an instance of QualifiedRoleType if found, unless nil

```
44   def as_of(aQualifiedRoleType, qualObj)
45     self.root.getAllSubRoles.each do |tempRole|
46       if (tempRole.class == aQualifiedRoleType ||
47         tempRole.class.superclass == aQualifiedRoleType) &&
48         tempRole.qualifier == qualObj then
49          return tempRole
50       end
51     end
52     return nil
53   end
```

Checks, if there is an object of QualifiedORoleType of self with the qualifier "qualObj" returns true if found, unless false

```
54   def existsAs_of(aQualifiedRoleType, qualObj)
55     self.root.getAllSubRoles.each do |tempRole|
56       if (tempRole.class == aQualifiedRoleType ||
57         tempRole.class.superclass == aQualifiedRoleType) &&
58         tempRole.qualifier == qualObj then
59           return true
60       end
61     end
62     return false
63   end    # end existsAs_of
64 end   # end AbstractObjectOrRole
```

Class ObjectWith Roles

```
01 class ObjectWithRoles
02   include AbstractObjectOrRole
03   attr_reader :roleDictionary
```

Constructor of class ObjectWithRoles

```
04   def initialize()
05     @roleOf = nil
06     @roleDictionary = Hash.new        # initRoleDictionary
07   end
```

Searches all roles of the role-hierarchy from the roleDictionary

```
08   def getAllSubRoles
09     allRoles = Array.new
10     @roleDictionary.keys.each do |aKey|
11       if @roleDictionary[aKey].class == Array then
12         allRoles += @roleDictionary[aKey]
13       else
14         allRoles.push(@roleDictionary[aKey])
15       end
16     end
17     return allRoles
18   end
```

Adds role to the roleDictionary

```
19   def addSubRole(role)
20     if role.class.superclass == QualifiedRole then
21       tempArray = Array.new
22       if @roleDictionary[role.class] != nil then
23         tempArray = @roleDictionary[role.class]
24       end
25       tempArray.push(role)
26       @roleDictionary[role.class] = tempArray
27     else
28       @roleDictionary[role.class] = role
29     end
30   end
```

Deletes role from the roleDictionary

```
31   def deleteSubRole(role)
32     if role.class.superclass == QualifiedRole then
33       tempArray = @roleDictionary[role.class]
34       i = 0
35       tempArray.each do |aQualRole|
36         if role == aQualRole then
37           @roleDictionary[role.class].delete_at(i)
38         end
39         i += 1
40       end
41       if @roleDictionary[role.class] == nil then
42         @roleDictionary.delete(role.class)
43       end
44     else
45       @roleDictionary.delete(role.class)
46     end
47   end
48 end   # end ObjectWithRoles
```

Class Role

```
01 class Role
02   include AbstractObjectOrRole
```

Generates an object-specific class to Role and generates a role class

```
03   class << Role
04     attr :roleSuperType, true
05   end
06
07   def Role.defRoleType(roleSuperType, &body)
08     newRoleType = Class.new(Role, &body)
09
10     newRoleType.roleSuperType = roleSuperType
11     return newRoleType
12   end
```

Generates a specific role class

```
13   def Role.defSpecializedRoleType(roleClass, &body)
14     newRoleType = Class.new(roleClass, &body)
15
16     newRoleType.roleSuperType = roleClass.roleSuperType
17     return newRoleType
18   end
```

Constructor of class Role

```
19   def initialize(roleOf, *rest)     # newRoleOf
20     if self.class.roleSuperType == roleOf.class then
21       roleExists =
22 roleOf.root.roleDictionary.include?(self.class)
23       if roleExists == false || self.class.superclass ==
24 QualifiedRole then
25         @roleOf = roleOf
26         init(*rest)
27         self.root.addSubRole(self)
28       end
29     end
30   end
```

Overwrites the method_missing for method dispatching

```
31   def method_missing(methId, *rest)
32     @roleOf.send(methId, *rest)
33   end
```

Deletes the role from the role-hierarchy

```
34   def abandon
35     self.root.getAllSubRoles.each do |tempRole|
36       if tempRole.roleOf == self then
37         self.root.deleteSubRole(tempRole)
38       end
39     end
40     self.root.deleteSubRole(self)
41   end
42 end   # end Role
```

Class QualifiedRole

```
01 class QualifiedRole < Role
02   attr_reader :qualifier
```

Generates an object-specific class to QualifiedRole and generates a qualified role class

```
03   class << QualifiedRole
04     attr :classOfQualifyingObj, true
05     attr :roleSuperType, true
06   end
07
08   def QualifiedRole.defQualifiedRoleType(classOfQualifyingObj,
09       roleSuperType, &body)
10     newQualifiedRole = Class.new(QualifiedRole, &body)
11
12     newQualifiedRole.classOfQualifyingObj = classOfQualifyingObj
13     newQualifiedRole.roleSuperType = roleSuperType
14     return newQualifiedRole
15   end
```

Constructor of class QualifiedRole

```
16   def initialize(qualifier, roleOf, *rest)  # newQualifiedRoleOf
17     if self.class.roleSuperType == roleOf.class &&
18       self.class.classOfQualifyingObj == qualifier.class then
19       roleExists = false
20       if roleOf.root.roleDictionary.include?(self.class) then
21         qualRoleArray = roleOf.root.roleDictionary[self.class]
22         qualRoleArray.each do |aRole|
23           if aRole.qualifier == qualifier then
24             roleExists = true
25           end
26         end
27       end
28       if !roleExists then
29         super(roleOf, *rest)
30         @qualifier = qualifier
31       end
32     end
33   end
```

Returns the qualifier

```
34   def qualifier
35     return @qualifier
36   end
37 end   # end QualifiedRole
```

7.3.2 Ruby-Code für Rollen in dezentralen Informationssystemen

Module Helper, which includes the method findInPrivateList

```
01 module Helpers
02   # Search method in the privateList of role
03   def findInPrivateList(role, method)
04     if role.class.privateList then       # privateList
available
05       role.class.privateList.each do |aMethod|
06         if aMethod == method then
07           return true
08         end
09       end
10     end
11     return false
12   end
13 end
```

Class ObjectOrRole

```
01 class ObjectOrRole
02   include Helpers
03   attr_reader :roles        # List of all direct sub-roles
04   attr :roleOf, true        # the super-role of a role
05   attr :roleOfType, true    # type of roleOf (0..specialization,
06                             # 1..generalization, 2..annotaion)
```

Overwrites the method_missing

```
07   def method_missing(methId, *rest, &block)
08     searchedRoleList = Array.new
09     methName = methId.id2name
10
11     if rest[0].kind_of? Array then
12       searchedRoleList = rest[0]
13       rest.shift
14     end
15
```

Already searched in actual role?

```
16     if !searchedRoleList.include?(self.class) then
17       searchedRoleList.push(self.class)
18
19       # Looking for method in corrList of the actual role
20       if self.class.corrList && !findInPrivateList(self,
methName) then
21         self.class.corrList.each do |aCorr|
22           if aCorr.varName == methName then
23             return aCorr.getCorrValue(self, *rest)
24           end
25         end
26       end
```

Method not found in role and corrList => search at role.getRoles

```
27       self.getRoles.each do |aRole|
28         begin
29           return aRole.send(methId, searchedRoleList, *rest,
&block)
30         rescue ArgumentError
31           if !findInPrivateList(aRole, methName) then
32             return aRole.send(methId, *rest, &block)
33           end
34         end
35       end
36     end
```

Method not found in role, corrList and sub-roles => search at super-role

```
37     if @roleOf then
38       begin
39         return @roleOf.send(methId, searchedRoleList, *rest,
&block)
40       rescue ArgumentError
41        if !findInPrivateList(@roleOf, methName) then
42          return @roleOf.send(methId, *rest, &block)
43        end
44      end
45    end
```

Method not found/available in role-hierarchy

```
46     raise NoMethodError.new("Method '#{methName}' not found in
47                              role-hierarchy")
48   end   # end method_missing
```

Creates a singleton class of ObjectOrRole

```
49   class << ObjectOrRole
50     attr :roleSuperType, true
51     attr :corrList, true
52     attr :privateList, true
53     attr :isSingleInstRole, true
54   end
```

Creates a new ObjectOrRole class

```
55   def ObjectOrRole.defRoleType(roleSuperType, isSingleInstRole,
&body)
56     newRoleType = Class.new(ObjectOrRole, &body)
57
58     newRoleType.isSingleInstRole = isSingleInstRole
59     newRoleType.roleSuperType = roleSuperType
60     return newRoleType
61   end
```

Creates a new ObjectOrRole class as a specialized role type

```
62   def ObjectOrRole.defSpecializedRoleType(roleClass,
isSingleInstRole, &body)
63     newRoleType = Class.new(roleClass, &body)
64
65     newRoleType.isSingleInstRole = isSingleInstRole
66     newRoleType.roleSuperType = roleClass.roleSuperType
67     return newRoleType
68   end
```

Constructor of ObjectOrRole – object is specialization of

```
69   def initialize(arg)
70     @roles = Array.new
71     if arg.kind_of? ObjectOrRole then    # object is no root
72       if self.class.roleSuperType == arg.class then   # observe
hierarchy
73         roleExists = false
74         if self.class.isSingleInstRole then
75           arg.roles.each do |aRole|
76             if aRole.class == self.class then
77               roleExists = true
78             end
79           end
80         elsif self.kind_of? QualifiedObjectOrRole then
81           arg.roles.each do |aRole|
82             if aRole.class == self.class &&
83               aRole.qualifier == self.qualifier then
84               roleExists = true
85             end
86           end
87         end
88         if !roleExists then
89           super()
90           @roleOf = arg
91           @roleOfType = 0
92           arg.addSubRole(self)
93         end
94       end
```

Object is generalization of

```
95     elsif arg.kind_of? Array then
96       @roleOf = nil
97       @roleOfType = nil
98       super()
99       arg.each do |aRole|
100         if aRole.class.roleSuperType == self.class then
101           found = false
102           if aRole.class.isSingleInstRole then
103             self.getRoles.each do |actRole|
104               if actRole.class == aRole.class then
105                 found = true
106               end
107             end
108           elsif aRole.class.kind_of? QualifiedObjectOrRole then
109             self.getRoles.each do |actRole|
110               if actRole.class == aRole.class &&
111                 actRole.qualifier == aRole.qualifier then
112                 found = true
113               end
114             end
115           end
116           if !found then
117             aRole.roleOf = self
118             aRole.roleOfType = 1
119             self.addSubRole(aRole)
120           end
121         end
122       end
```

Object is root

```
123     else
124       @roleOf = nil
125       @roleOfType = nil
126       super()
127     end
128   end   # end initialize
```

Setter-Method roleOf=

```
129   def roleOf=(newRoleOf)
130     if newRoleOf.class == self.class.roleSuperType then
131       newRoleOf.addSubRole(self)
132       @roleOf = newRoleOf
133       @roleOfType = 2
134     else
135       raise NameError.new("wrong type of new super-role!")
136     end
137   end
```

Methods for navigation - Returns the roles-list

```
138   def getRoles
139     return @roles
140   end
```

Adds a role to the roles-list

```
141   def addSubRole(role)
142     @roles.push(role)
143   end
```

Returns a list of all sub-roles of the actual object or role

```
144   def getAllSubRoles
145     allSubRoles = Array.new
146     @roles.each do |aRole|
147       allSubRoles += aRole.getAllSubRoles
148       allSubRoles.push(aRole)
149     end
150     return allSubRoles
151   end
```

Clears the roles-list

```
152   def clearRoles
153     @roles.clear
154   end
155
156   # end role methods
```

Returns the real world object (=root) of a role-hierarchy

```
157   def root
158     aRole = self
159     while aRole.roleOf != nil do
160       aRole = aRole.roleOf
161     end
162     return aRole
163   end
```

Checks, if there is an object of "otherRoleType" of the same real world object as self; Returns an other object, a list of all other objects or nil

```
164   def as(otherRoleType)
165     if self.root.class == otherRoleType then
166       return self.root
167     end
168
169     self.root.getAllSubRoles.each do |aRole|
170       if aRole.class == otherRoleType ||
171         aRole.class.superclass == otherRoleType then
172         return aRole
173       end
174     end
175     return nil
176   end
```

Checks, if there is an object of "otherRoleType" of the same real world object as self; Returns true or false

```
177   def existsAs(otherRoleType)
178     if root.class == otherRoleType then
179       return true
180     else
181       root.getAllSubRoles.each do |aRole|
182         if aRole.class == otherRoleType ||
183         aRole.class.superclass == otherRoleType then
184           return true
185         end
186       end
187     end
188     return false
189   end
```

Checks, if self and "anObject" are from the same real world object; returns true if they do, unless false

```
190   def entityEquiv(anObject)
191     if self.root == anObject.root then
192       return true
193     else return false
194     end
195   end
```

Deletes the ObjectOrRole object from the role-hierarchy

```
196   def abandon
197     self.getAllSubRoles.each do |aRole|
198       aRole.clearRoles     # deletes all sub-roles of self
199     end
200     @roles.clear
201     i = 0
202     self.roleOf.getRoles.each do |aRole|
203      if aRole == self then
204         self.roleOf.getRoles.delete_at(i)
205      end
206       i += 1
207     end
208     @roleOf=nil
209   end
```

Additional methods for QualifiedRoles

checks, if there is an object of QualifiedObjectOrRole of self with the qualifier "qualObj"; returns QualifiedObjectOrRole or a list of QualifiedObjectOrRole-objects if found, unless nil

```
210   def as_of(otherQualifiedRoleType, qualObj)
211     if self.root.class == otherQualifiedRoleType &&
212        self.root.qualifier == qualObj then
213        return self.root
214     end
215
216     self.root.getAllSubRoles.each do |aRole|
217       if (aRole.class == otherQualifiedRoleType ||
218         aRole.class.superclass == otherQualifiedRoleType) &&
219         aRole.qualifier == qualObj then
220         return aRole
221       end
222     end
223     return nil
224   end
```

Checks, if there is an object of QualifiedObjectOrRole of self with the qualifier "qualObj"; returns true if found, unless false

```
225   def existsAs_of(otherQualifiedRoleType, qualObj)
226     if self.root.class == otherQualifiedRoleType &&
227       self.qualifier == qualObj then
228       return true
229     end
230     self.root.getAllSubRoles.each do |aRole|
231       if (aRole.class == otherQualifiedRoleType ||
232         aRole.class.superclass == otherQualifiedRoleType) &&
233         aRole.qualifier == qualObj then
234           return true
235       end
236     end
237     return false
238   end
239 end   # end ObjectOrRole
```

Class QualifiedObjectOrRole, inherits from ObjectOrRole

```
01 class QualifiedObjectOrRole < ObjectOrRole
02   attr_reader :qualifier
03
```

Creates a singleton class of QualifiedObjectOrRole

```
04     class << QualifiedObjectOrRole
05       attr :classOfQualifyingObj, true
06     end
```

Creates an object of QualifiedObjectOrRole

```
07   def QualifiedObjectOrRole.defQualifiedRoleType(roleSuperType,
classOfQualifyingObj, &body)
08     newQualifiedRoleType = Class.new(QualifiedObjectOrRole,
&body)
09
10     newQualifiedRoleType.isSingleInstRole = false
11     newQualifiedRoleType.roleSuperType = roleSuperType
12     newQualifiedRoleType.classOfQualifyingObj =
classOfQualifyingObj
13     return newQualifiedRoleType
14   end
```

Constructor of QualifiedObjectOrRole

```
15   def initialize(qualifier, *args)
16     @qualifier = qualifier
17     super(*args)
18   end
```

Returns the qualifier of the QualifiedObjectOrRole

```
19   def qualifier
20     return @qualifier
21   end   # end qualifier
22 end   # end QualifiedObjectOrRole
```

Corresponding class

```
01 class CorrClass
02   include Helpers
03   attr_reader :varName, :relationship, :roleArray,
:methArray, :argsArray
```

Constructor of CorrClass

```
04   def initialize(varName, relationship, *rest)
05     @roleArray = Array.new
06     @methArray = Array.new
07     @argsArray = Array.new
08     @varName = varName
09     @relationship = relationship      # Type of relationship
10
11     rest.each do |var|
12       @roleArray.push(var) if var.instance_of?(Class)
13       @methArray.push(var) if var.instance_of?(String)
14       @argsArray.push(var) if var.instance_of?(Array)
15
16       @argsArray.push(nil) if @roleArray.size - 1 >
@argsArray.size
17     end
18     @argsArray.push(nil) if @roleArray.size > @argsArray.size
19   end
```

Method getCorrValue – ID relationship

```
20   # Should be changed for special use of argsArray and/or *rest
21   def getCorrValue(role, *rest)
22     if @relationship == 0 then      # id relationship
23       i = 0
24       @roleArray.each do |aRoleClass|
25         if role.class == aRoleClass then
26           if !findInPrivateList(role, @methArray[i]) then
27             begin
28               return role.send(@methArray[i], *@argsArray[i])
29             rescue ArgumentError
30               return role.send(@methArray[i])
31             end
32           end
33         else
34           role.getAllSubRoles.each do |aRole|
35             if aRole.class == aRoleClass then
36               if !findInPrivateList(aRole, @methArray[i]) then
37                 begin
38                   return aRole.send(@methArray[i],
*@argsArray[i])
39                 rescue ArgumentError
40                   return aRole.send(@methArray[i])
41                 end
42               end
43             end
44           end
45         end
46         i += 1
47       end
```

All relationship

```
48     elsif @relationship == 1 then     # all relationship
49       valueArray = Array.new
50       i = 0
51       @roleArray.each do |aRoleClass|
52         if role.class == aRoleClass then
53           if !findInPrivateList(role, @methArray[i]) then
54             begin
55               valueArray.push(role.send(@methArray[i],
*@argsArray[i]))
56             rescue ArgumentError
57               valueArray.push(role.send(@methArray[i]))
58             end
59           end
60         else
61           role.getAllSubRoles.each do |aRole|
62             if aRole.class == aRoleClass then
63               if !findInPrivateList(aRole, @methArray[i])
then
64                 begin
65                   valueArray.push(aRole.send(@methArray[i],
*@argsArray[i]))
66                 rescue ArgumentError
67                   valueArray.push(aRole.send(@methArray[i]))
68                 end
69               end
70             end
71          end
72        end
73        i += 1
74      end
75      return valueArray
```

SUM relationship

```
76     elsif @relationship == 2 then
77       value = 0
78       i = 0
79       @roleArray.each do |aRoleClass|    # all classes of
generalization
80         if role.class == aRoleClass then
81           if !findInPrivateList(role, @methArray[i]) then
82             begin
83               number = @argsArray[i].to_s.to_i  # Caution at
argsArray!!!
84               value += role.send(@methArray[i], number)
85             rescue ArgumentError
86               value += role.send(@methArray[i])
87             end
88           end
89         else
90           role.getAllSubRoles.each do |aRole|
91             if aRole.class == aRoleClass then
92               if !findInPrivateList(aRole, @methArray[i])
then
93                 begin
94                   number = @argsArray[i].to_s.to_i   #
Caution at argsArray!!!
95                   value += aRole.send(@methArray[i], number)
96                 rescue ArgumentError
97                   value += aRole.send(@methArray[i])
98                 end
99               end
100             end
101           end
102         end
103          i += 1
104       end
105       return value
```

General relationship

```
106     else
107       return nil
108     end
109   end   # end getCorrValue
110 end   # end CorrClass
```

7.3.3 Scaffold-Generator-Code

7.3.3.1 roleScaffold

Class RoleScaffoldGenerator

```
01 class RoleScaffoldGenerator < Rails::Generator::NamedBase
02   default_options :skip_timestamps => false, :skip_migration =>
false, :skip_fixture => false
03
04   attr_reader   :controller_name,
05                 :controller_class_path,
06                 :controller_file_path,
07                 :controller_class_nesting,
08                 :controller_class_nesting_depth,
09                 :controller_class_name,
10                 :controller_underscore_name,
11                 :controller_singular_name,
12                 :controller_plural_name,
13                 :role_super_type,
14                 :role_super_type_underscore_name,
15                 :role_flag,
16                 :qualifier_type,
17                 :qualifier_type_underscore_name
18   alias_method  :controller_file_name,
:controller_underscore_name
19   alias_method  :controller_table_name, :controller_plural_name
```

Constructor RoleScaffoldGenerator

```
20   def initialize(runtime_args, runtime_options = {})
21     @role_flag = false
22     checkFlag = runtime_args[0]
23     if checkFlag.size == 1 && (checkFlag == "Q" || checkFlag
== "q")
24       @role_flag = true
25       runtime_args.shift
26     end
27
```

Check roleSuperType

```
28     checkString = runtime_args[11]
29     if checkString[0,1].between?("A","Z")
30       @role_super_type = checkString
31       @role_super_type_underscore_name =
@role_super_type.underscore
32       runtime_args[11] = runtime_args[0]
33       runtime_args.shift
34     else
35       @role_super_type = NilClass
36       @role_super_type_underscore_name = nil
37     end
38     checkAttrs = runtime_args[1..runtime_args.size]
```

Check class of qualifying object

```
39     qualFkFound = false
40     checkAttrs.each do |aAttr|
41       aAttr = aAttr.split(':')[0]
42       aAttrLength = aAttr.size
43       if aAttr[aAttrLength-3..aAttrLength] == "_id" && aAttr
!= "#{@role_super_type_underscore_name}_id"
44         qualFkFound = true
45         @qualifier_type_underscore_name =
aAttr[0..aAttrLength-4]
46         @qualifier_type =
@qualifier_type_underscore_name.capitalize
47       end
48     end
49     raise NameError.new("Error: no foreign-key for
qualifier!") if @role_flag && !qualFkFound
50
51     super
```

Initialize attributes

```
52   @controller_name = @name.pluralize
53
54     base_name, @controller_class_path, @controller_file_path,
   @controller_class_nesting, @controller_class_nesting_depth =
   extract_modules(@controller_name)
55     @controller_class_name_without_nesting,
   @controller_underscore_name, @controller_plural_name =
   inflect_names(base_name)
56     @controller_singular_name=base_name.singularize
57     if @controller_class_nesting.empty?
58       @controller_class_name =
   @controller_class_name_without_nesting
59     else
60       @controller_class_name =
   "#{@controller_class_nesting}::#{@controller_class_name_withou
   t_nesting}"
61     end
62   end
```

Method manifest

```
63   def manifest
64     record do |m|
65       # Check for class naming collisions.
66       m.class_collisions(controller_class_path,
   "#{controller_class_name}Controller",
   "#{controller_class_name}Helper")
67       m.class_collisions(class_path, "#{class_name}")
68       m.directory(File.join('app/models', class_path))
69       m.directory(File.join('app/controllers',
   controller_class_path))
70       m.directory(File.join('app/helpers',
   controller_class_path))
71       m.directory(File.join('app/views', controller_class_path,
   controller_file_name))
```

```
72        m.directory(File.join('app/views/layouts',
   controller_class_path))
73        m.directory(File.join('test/functional',
   controller_class_path))
74        m.directory(File.join('test/unit', class_path))
75
76        for action in scaffold_views
77          m.template(
78            "view_#{action}.html.erb",
79            File.join('app/views', controller_class_path,
   controller_file_name, "#{action}.html.erb")
80          )
81        end
82
83        # Layout and stylesheet.
84        m.template('layout.html.erb',
   File.join('app/views/layouts', controller_class_path,
   "#{controller_file_name}.html.erb"))
85        m.template('style.css', 'public/stylesheets/scaffold.css')
86
87        # m.dependency 'model', [name] + @args, :collision =>
   :skip
88
      # ADDED FROM MODEL_GENERATOR.RB
      # Check for class naming collisions.
89        m.class_collisions class_path, class_name,
   "#{class_name}Test"
90
91        # Model, test, and fixture directories.
92        m.directory File.join('app/models', class_path)
93        m.directory File.join('test/unit', class_path)
94        m.directory File.join('test/fixtures', class_path)
95
96        # Model class, unit test, and fixtures.
97        m.template 'model.rb',      File.join('app/models',
   class_path, "#{file_name}.rb")
98        m.template 'unit_test.rb',  File.join('test/unit',
   class_path, "#{file_name}_test.rb")
99
100         unless options[:skip_fixture]
101               m.template 'fixtures.yml',
   File.join('test/fixtures', "#{table_name}.yml")
102         end
103
104         unless options[:skip_migration]
105           m.migration_template 'migration.rb', 'db/migrate',
   :assigns => {
106             :migration_name =>
   "Create#{class_name.pluralize.gsub(/::/, '')}"
107           }, :migration_file_name =>
   "create_#{file_path.gsub(/\//, '_').pluralize}"
108         end
```

Generates role specific files

```
109        # part of role generator
110   #     m.template('model.rb', File.join('app/models',
   controller_class_path,
   "#{controller_file_name.singularize}.rb"))
111         m.template('controller.rb',
   File.join('app/controllers', controller_class_path,
   "#{controller_file_name}_controller.rb"))
112         m.template('object_or_role.rb',
   File.join('app/controllers', controller_class_path,
   "object_or_role.rb"))
113         m.template('role_helper.rb',
   File.join('app/controllers', controller_class_path,
   "role_helper.rb"))
114
115         m.template('routes.rb', File.join('config/',
   class_path, "routes.rb")) if role_super_type == NilClass
116
117         m.template('functional_test.rb',
   File.join('test/functional', controller_class_path,
   "#{controller_file_name}_controller_test.rb"))
118         m.template('helper.rb',
   File.join('app/helpers',     controller_class_path,
   "#{controller_file_name}_helper.rb"))
119
120         m.route_resources controller_file_name
121       end
122     end
```

Further methods

```
123     protected
124       # Override with your own usage banner.
125       def banner
126         "Usage: #{$0} scaffold ModelName [field:type,
   field:type]"
127       end
128
129       def add_options!(opt)
130         opt.separator ''
131         opt.separator 'Options:'
132         opt.on("--skip-timestamps",
133                "Don't add timestamps to the migration file for
   this model") { |v| options[:skip_timestamps] = v }
134         opt.on("--skip-migration",
135                "Don't generate a migration file for this
   model") { |v| options[:skip_migration] = v }
136
137         # ADDED FROM MODEL_GENERATOR.RB
138         opt.on("--skip-fixture",
139                "Don't generation a fixture file for this
   model") { |v| options[:skip_fixture] = v}
140       end
141
142       def scaffold_views
143         %w[ index show new edit ]
144       end
145
146       def model_name
147         class_name.demodulize
148       end
149   end
```

7.3.3.2 model

Generates a model-class

```
01 class <%= controller_singular_name %> < ActiveRecord::Base
02   include ObjectOrRole
03   <% if role_super_type != NilClass %>
04   belongs_to :<%= role_super_type_underscore_name %> <% end %>
05
06   @@roleSuperType = <%= role_super_type %>
07   @@corrList = []  # ADD CorrClass-objects for mapping-methods
08                  # (e.g. CorrClass.new("methName", 2, SubType,
09                    # "methName1", SubType, "methName2")
10   @@privateList = [] # ADD attribute- or method-name, if it
11                           # should not inherited
12
13   attr_accessor :roleOf         # super-role
14   attr_accessor :roles          # Array of all sub-roles
15   attr_accessor :roleOfType     # type of roleOf
   (0..specialization, 1..generalization, 2..annotation)
16   <% if role_flag %>attr_accessor :qualifier<% end %>
```

Model methods

```
17   def roleSuperType
18     return @@roleSuperType
19   end
20
21   def corrList
22     return @@corrList
23   end
24
25   def privateList
26     return @@privateList
27   end
28
29   def roleOf
30     <% if role_super_type == NilClass %>return nil<% else
   %>@roleOf = <%= role_super_type %>.find(self.<%=
   role_super_type_underscore_name %>_id) if self.<%=
   role_super_type_underscore_name %>_id
31     return @roleOf<% end %>
32   end
```

Setter of roleOf

```
33   def roleOf=(newRoleOf)
34     if @@roleSuperType == newRoleOf.class then
35       @roleOf = newRoleOf
36       @roleOfType = 2
37     end
38   end
```

Method roles

```
39   def roles
40     @roles = Array.new
41     roleArray = Array.new
42     RoleHelper.getRoleClasses.each do |aClass|
43       begin
44         roleArray += aClass.find(:all)
45       rescue Exception
46       end
47     end
48     roleArray.each do |aRole|
49       begin
50         if aRole.attributes.include?("<%= file_name %>_id") &&
51            aRole.<%= file_name %>_id == self.id then
52           @roles.push(aRole)
53         end
54       rescue Exception
55       end
56     end
57     return @roles
58   end
```

Method qualifier for qualified roles

```
59   <% if role_flag %>def qualifier
60     @qualifier = <%= qualifier_type %>.find(self.<%=
   qualifier_type_underscore_name %>_id) if self.<%=
   qualifier_type_underscore_name %>_id
61     return @qualifier
62   end <% end %>
```

Overwrites the method_missing

```
63   def method_missing(methId, *rest, &block)
64     searchedRoleList = Array.new
65     if rest[0].instance_of?(Array) then
66       if rest[0][0].superclass == ActiveRecord::Base then
67         searchedRoleList = rest[0]
68         rest.shift
69       end
70     end
71     begin
72       super(methId, *rest, &block)
73     rescue          # method not found in actual role
74       <% if role_super_type == NilClass %> @roleOf = nil <%
   else %> @roleOf = <%= role_super_type %>.find(self.<%=
   role_super_type_underscore_name %>_id)  <% end %>
75       @roles = Array.new
76       RoleHelper.getRoleClasses.each do |aClass|
77         begin
78           @roles += aClass.find(:all, :conditions => ["<%=
   file_name %>_id = ?", self.id])
79         rescue Exception
80         end
81       end
82       role_method_missing(methId, self, @roleOf, @roles,
   searchedRoleList, *rest, &block)
83     end
84   end   # end method_missing
85 end
```

7.3.3.3 views

edit.html.erb:

Generates the edit-view for a role class

```
01 <h1>Editing <%= singular_name %></h1>
02
03 <%%= error_messages_for :<%= singular_name %> %>
04
05 <%% form_for(@<%= singular_name %>) do |f| %>
06
07 <% roleOfAttr = attributes.pop if role_super_type != NilClass %>
08 <% qualifierAttr = attributes.pop if role_flag %>
09 <% for attribute in attributes -%>
10   <p>
11     <b><%= attribute.column.human_name %></b><br />
12     <%%= f.<%= attribute.field_type %> :<%= attribute.name %> %>
13   </p>
14
15 <% end %>
16 <% attributes.push(qualifierAttr) if role_flag %>
17 <% attributes.push(roleOfAttr) if role_super_type != NilClass %>
18
19   <% if role_super_type != NilClass %><p>
20     <b>roleOf</b><br />
21     <%%= f.select(:<%= role_super_type_underscore_name %>_id, <%=
   role_super_type %>.find(:all).collect{|p| [p.name, p.id]}, {
   :include_blank => true }) %>
22   </p><% end %>
23
24   <% if role_flag %><p>
25     <b>qualifier</b><br />
26     <%%= f.select(:<%= qualifier_type_underscore_name %>_id, <%=
   qualifier_type %>.find(:all).collect{|p| [p.name], p.id]}, {
   :include_blank => true }) %>
27   </p> <% end %>
28
29   <p>
30     <%%= f.submit "Update" %>
31   </p>
32 <%% end %>
33
34 <%%= link_to 'Show', @<%= singular_name %> %> |
35 <%%= link_to 'Back', <%= plural_name %>_path %>
```

index.html.erb:

Generates the index-view for a role class

```
01 <h1>Listing <%= plural_name %></h1>
02
03 <table>
04   <tr>
05 <% for attribute in attributes -%>
06     <th><%= attribute.column.human_name %></th>
07 <% end -%>
08   </tr>
09
10 <%% for <%= singular_name %> in @<%= plural_name %> %>
11   <tr>
12 <% for attribute in attributes -%>
13     <td><%%=h <%= singular_name %>.<%= attribute.name %>
   %></td>
14 <% end -%>
15     <td><%%= link_to 'Show', <%= singular_name %> %></td>
16     <td><%%= link_to 'Edit', edit_<%= singular_name %>_path(<%=
   singular_name %>) %></td>
17     <td><%%= link_to 'Destroy', <%= singular_name %>, :confirm
   => 'Are you sure?', :method => :delete %></td>
18   </tr>
19 <%% end %>
20 </table>
21
22 <br />
23
24 <%%= link_to 'New <%= singular_name %>', new_<%= singular_name
   %>_path %>
```

new.html.erb:

Generates the new-view for a role class

```
01 <h1>New <%= singular_name %></h1>
02
03 <%%= error_messages_for :<%= singular_name %> %>
04
05 <%% form_for(@<%= singular_name %>) do |f| %>
06 <% roleOfAttr = attributes.pop if role_super_type != NilClass
   %>
07 <% qualifierAttr = attributes.pop if role_flag %>
08 <% for attribute in attributes -%>
09   <p>
10     <b><%= attribute.column.human_name %></b><br />
11     <%%= f.<%= attribute.field_type %> :<%= attribute.name %>
   %>
12   </p>
13
14 <% end -%>
15 <% attributes.push(qualifierAttr) if role_flag %>
16 <% attributes.push(roleOfAttr) if role_super_type != NilClass
   %>
```

```
17 <% if role_super_type != NilClass %><p>
18     <b>roleOf</b><br />
19     <%%= f.select(:<%= role_super_type_underscore_name %>_id,
   <%= role_super_type %>.find(:all).collect{|p| [p.name, p.id]},
   { :include_blank => true }) %>
20   </p><% end %>
21
22 <% if role_flag %><p>
23     <b>qualifier</b><br />
24     <%%= f.select(:<%= qualifier_type_underscore_name %>_id,
   <%= qualifier_type %>.find(:all).collect{|p| [p.name, p.id]}, {
   :include_blank => true }) %>
25   </p><% end %>
26
27   <p>
28     <%%= f.submit "Create" %>
29   </p>
30 <%% end %>
31
32 <%%= link_to 'Back', <%= plural_name %>_path %>
```

show.html.erb:

Generates the show-view for a role class

```
01 <% roleOfAttr = attributes.pop if role_super_type != NilClass
   %>
02 <% qualifierAttr = attributes.pop if role_flag %>
03 <% for attribute in attributes -%>
04 <p>
05   <b><%= attribute.column.human_name %>:</b>
06   <%%=h @<%= singular_name %>.<%= attribute.name %> %>
07 </p><% end %>
08 <% attributes.push(qualifierAttr) if role_flag %>
09 <% attributes.push(roleOfAttr) if role_super_type != NilClass
   %>
10
11 <% if role_super_type != NilClass %><p>
12   <b>roleOf:</b>
13     <%% begin %>
14      <%%= link_to (@<%= singular_name %>.roleOf.name,
   roleOf_url) if @<%= singular_name %>.roleOf %>
15     <%% rescue Exception %>
16      <a href = "<%%= @<%= singular_name %>.roleOf.class.site
   %><%%= @<%= singular_name
   %>.roleOf.class.to_s.pluralize.underscore %>/<%%= @<%=
   singular_name %>.roleOf.id %>"><%%= @<%= singular_name
   %>.roleOf.name %></a>
17     <%% end %>
18 </p><% end %>
```

```
19 <p>
20   <b>root:</b>
21     <%% begin %>
22      <%%= link_to (@<%= singular_name %>.root.name,
   roleRoot_url) %>
23     <%% rescue Exception %>
24      <%%=h @<%= singular_name %>.root.to_s %>
25     <%% end %>
26 </p>
27
28 <% if role_flag %><p>
29   <b>qualifier:</b>
30     <%% begin %>
31       <%%=link_to (@<%= singular_name %>.qualifier.name,
   qualifier_url) if @<%= singular_name %>.qualifier %>
32     <%% rescue Exception %>
33       <a href = "<%% @<%= singular_name %>.qualifier.class.site
   %><%%= @<%= singular_name
   %>.qualifier.class.to_s.pluralize.underscore %>/<%%= @<%=
   singular_name %>.qualifier.id %>"<%%= @<%= singular_name
   %>.qualifier.name %></a>
34     <%% end %>
35 </p><% end %>


36 <p>
37   <b>Roles: </b>
38   <%% for subRole in @<%= singular_name %>.roles %>
39     <%% begin %>
40      <%%= link_to subRole.name, subRole %>
41     <%% rescue Exception %>
42      <a href = "<%%= subRole.class.site %><%%=
   subRole.class.to_s.pluralize.underscore %>/<%%= subRole.id
   %>"><%%= subRole.name %></a>
43     <%% end %>
44   <%% end %>
45 </p>
46
47 <%% if !@paramsHash.empty? then %>
48   <p><b>Methods: </b></p>
49   <%% @paramsHash.each do |aMeth, aParam| %>
50     <p>
51    <%% begin %>
52      <b><%%= aMeth %>(<%%= aParam.name %>): </b>
53       <%% rescue Exception %>
54         <b><%%= aMeth %>(<%%= aParam %>): </b>
55       <%% end %>
56       <%% value = @<%= singular_name %>.send(aMeth, aParam) %>
57       <%% begin %>
58         <%%= link_to (value.name, value) %>
59       <%% rescue Exception %>
60         <%%=h value %>
61       <%% end %>
62     </p>
63   <%% end %>
64 <%% end %>
65
66 <%%= link_to 'Edit', edit_<%= singular_name %>_path(@<%=
   singular_name %>) %> |
67 <%%= link_to 'Back', <%= plural_name %>_path %>
```

7.3.3.4 controller

Generates a controller to a role class

```
01 require 'rexml/document'
02
03 class <%= controller_class_name %>Controller  <
   ApplicationController
04   include REXML
05
06   # GET /<%= table_name %>
07   # GET /<%= table_name %>.xml
08   def index
09     @<%= table_name %> = <%= class_name %>.find(:all)
10
11     respond_to do |format|
12       format.html # index.html.erb
13       format.xml  { render :xml => @<%= table_name %> }
14     end
15   end
16
17   # GET /<%= table_name %>/1
18   # GET /<%= table_name %>/1.xml
19   def show
20     @<%= file_name %> = <%= class_name %>.find(params[:id])
21
22     @paramsHash = Hash.new
23     request.query_parameters.each do |aKey, aValue|
24       @paramsHash[aKey] = RoleHelper.getValue(aValue)
25     end
26
27     respond_to do |format|
28       format.html # show.html.erb
29       format.xml  { render :xml => <%= file_name %>_to_xml(@<%=
   file_name %>, @paramsHash) }
30     end
31   end
32
33   # GET /<%= table_name %>/new
34   # GET /<%= table_name %>/new.xml
35   def new
36     @<%= file_name %> = <%= class_name %>.new
37
38     respond_to do |format|
39       format.html # new.html.erb
40       format.xml  { render :xml => @<%= file_name %> }
41     end
42   end


43   # GET /<%= table_name %>/1/edit
44   def edit
45     @<%= file_name %> = <%= class_name %>.find(params[:id])
46   end
47
48   # POST /<%= table_name %>
49   # POST /<%= table_name %>.xml
50   def create
51     @<%= file_name %> = <%= class_name %>.new(params[:<%=
   file_name %>])
```

```
52     respond_to do |format|
53       if @<%= file_name %>.save
54         flash[:notice] = '<%= class_name %> was successfully
   created.'
55         format.html { redirect_to(@<%= file_name %>) }
56         format.xml  { render :xml => @<%= file_name %>, :status
   => :created, :location => @<%= file_name %> }
57       else
58         format.html { render :action => "new" }
59         format.xml  { render :xml => @<%= file_name %>.errors,
   :status => :unprocessable_entity }
60       end
61     end
62   end
63
64   # PUT /<%= table_name %>/1
65   # PUT /<%= table_name %>/1.xml
66   def update
67     @<%= file_name %> = <%= class_name %>.find(params[:id])
68
69     respond_to do |format|
70       if @<%= file_name %>.update_attributes(params[:<%=
   file_name %>])
71         flash[:notice] = '<%= class_name %> was successfully
   updated.'
72         format.html { redirect_to(@<%= file_name %>) }
73         format.xml  { head :ok }
74       else
75         format.html { render :action => "edit" }
76         format.xml  { render :xml => @<%= file_name %>.errors,
   :status => :unprocessable_entity }
77       end
78     end
79   end
80
81   # DELETE /<%= table_name %>/1
82   # DELETE /<%= table_name %>/1.xml
83   def destroy
84     @<%= file_name %> = <%= class_name %>.find(params[:id])
85     if @<%= file_name %>.roles.size == 0 then
86       flash[:notice] = "Deleted #{@<%= file_name %>.name}!"
87       @<%= file_name %>.destroy
88     else
89       flash[:notice] = 'Delete all sub-roles first!'
90     end
91
92     respond_to do |format|
93       format.html { redirect_to(<%= table_name %>_url) }
94       format.xml  { head :ok }
95     end
96   end
```

```
97   # GET /<%= table_name %>/1/roleOf
98   def roleOf
99     @<%= file_name %> = <%= class_name %>.find(params[:id])
100
101       respond_to do |format|
102         format.html { redirect_to(@<%= file_name %>.roleOf)   }
103         format.xml  { render :xml => @<%= file_name %>.roleOf }
104       end
105     end
106
107     # GET /<%= table_name %>/1/root
108     def root
109       @<%= file_name %> = <%= class_name %>.find(params[:id])
110
111       respond_to do |format|
112         format.html { redirect_to(@<%= file_name %>.root)   }
113         format.xml  { render :xml => @<%= file_name %>.root }
114       end
115     end
116
117     <% if role_flag %>
118     # GET /<%= table_name %>/1/qualifier
119     def qualifier
120       @<%= file_name %> = <%= class_name %>.find(params[:id])
121
122       respond_to do |format|
123         format.html { redirect_to(@<%= file_name %>.qualifier)}
124         format.xml  { render :xml => @<%= file_name
   %>.qualifier }
125       end
126     end<% end %>
```

```
127     # Rewrites xml output including role-tags
128     def <%= file_name %>_to_xml(<%= file_name %>, *rest)
129       string = <%= file_name %>.to_xml(:skip_types => true) do
   |xml|
130         <% if role_flag %> xml.qualifier(<%= file_name
   %>.qualifier.to_xml(:skip_instruct => true, :skip_types =>
   true)) <% end %>
131         xml.roleOf(<%= file_name
   %>.roleOf.to_xml(:skip_instruct => true,
132                :skip_types => true)) if <%= file_name %>.roleOf
133         xml.roles do
134           <%= file_name %>.roles.each do |aRole|
135             xml.tag!(aRole.class.to_s,
   aRole.to_xml(:skip_instruct => true,
136                :skip_types => true))
137           end
138         end
139         xml.root(<%= file_name %>.root.to_xml(:skip_instruct
   => true,
140                :skip_types => true)) if <%= file_name %>.root
```

```
141         if rest
142           rest[0].each do |aMeth, aParam|
143             value = <%= file_name %>.send(aMeth, aParam)
144             if aMeth == "as"
145               xml.as(value.to_xml(:skip_instruct => true,
146                 :skip_types => true), :param => aParam)
147             elsif aMeth == "existsAs"
148               xml.existsAs(value, :param => aParam)
149             elsif aMeth == "entityEquiv"
150               xml.entityEquiv(value, :param => aParam.name)
151             end
152           end
153         end
154       end
155       doc = Document.new(string)
156       return doc
157     end
158   end
```

7.3.3.5 object_or_role

```
01 module ObjectOrRole
02
03   # Helper-method of method_missing of all role classes
04   def role_method_missing(methId, role, roleOf, subRoles,
   searchedRoleList, *rest, &block)
05     methName = methId.id2name
06
07     # Already searched in actual role?
08     if !searchedRoleList.include?(role.class) then
09       searchedRoleList.push(role.class)
10
11       # Looking for method in corrList of the actual role
12       if role.corrList then
13         role.corrList.each do |aCorr|       # go throw
   corresponding list
14           if aCorr.varName == methName then
15             return aCorr.getCorrValue(role, *rest)
16           end
17         end
18       end
19
20       # method not found in role and corrList => search at
   role.roles
21       subRoles.each do |aRole|
22         if !findInPrivateList(aRole, methName) then
23           return aRole.send(methId, searchedRoleList, *rest,
   &block)
24         end
25       end
26     end
```

```
27     # method not found in role, corrList and sub-roles =>
   search at super-role
28     if roleOf then
29       if !findInPrivateList(roleOf, methName) then
30         return roleOf.send(methId, searchedRoleList, *rest,
   &block)
31       end
32     end
33
34     # method not found/available in role-hierarchy
35     raise NoMethodError.new("Method '#{methName}' not found in
   role-hierarchy (#{searchedRoleList})")
36   end   # end method_missing
```

Helper-method to search methName in privateList of role

```
37   def findInPrivateList(role, methName)
38     if role.privateList then
39       role.privateList.each do |aInher|
40         return true if aInher == methName
41       end
42     end
43     return false
44   end
45
```

Methods for navigation; returns a list of all sub-roles of the actual object or role

```
46   def getAllSubRoles
47     allSubRoles = Array.new
48     self.roles.each do |aRole|
49       allSubRoles += aRole.getAllSubRoles
50       allSubRoles.push(aRole)
51     end
52     return allSubRoles
53   end   # end getAllSubRoles
```

Returns the real world object (=root) of a role-hierarchy

```
54   def root
55     role = self
56     begin
57       while role.roleOf do
58         role = role.roleOf
59       end
60     rescue Exception
61     end
62     return role
63   end   # end root
```

Checks, if there is an object of "otherRoleType" of the same real world object as self; returns the other object or nil

```
64   def as(otherRoleType)
65     if self.root.class == otherRoleType then
66       return self.root
67     end
68
69     self.root.getAllSubRoles.each do |aRole|
70       if aRole.class == otherRoleType then
71         return aRole
72       end
73     end
74     return nil
75   end   # end as
```

Checks, if there is an object of "otherRoleType" of the same real world object as self; returns true or false

```
76   def existsAs(otherRoleType)
77     if self.root.class == otherRoleType then
78       return true
79     end
80
81     self.root.getAllSubRoles.each do |aRole|
82       if aRole.class == otherRoleType then
83         return true
84       end
85     end
86     return false
87   end   # end existsAs
```

Checks, if self and "anObject" are from the same real world object; returns true if they do, unless false

```
88   def entityEquiv(anObject)
89     if self.root == anObject.root then
90       return true
91     else return false
92     end
93   end   # end entityEquiv
```

Deletes the ObjectOrRole object from the role-hierarchy

```
94   def abandon
95     self.destroy
96   end  # end abandon
```

Additional methods for QualifiedRoles
Checks, if there is an object of QualifiedObjectOrRole of self with the qualifier "qualObj"; returns QualifiedObjectOrRole or a list of QualifiedObjectOrRole-objects if found, unless nil

```
97   def as_of(aQualifiedRoleType, qualObj)
98     self.root.getAllSubRoles.each do |aRole|
99       if aRole.class == aQualifiedRoleType then
100           if aRole.qualifier == qualObj then
101             return aRole
102           end
103         end
104       end
105       return nil
106     end    # end as_of
```

Checks, if there is an object of QualifiedObjectOrRole of self with the qualifier "qualObj"; returns true if found, unless false

```
107     def existsAs_of(aQualifiedRoleType, qualObj)
108       self.root.getAllSubRoles.each do |aRole|
109         if aRole.class == aQualifiedRoleType then
110           if aRole.qualifier == qualObj then
111             return true
112           end
113         end
114       end
115       return false
116     end    # end existsAs_of
```

Corresponding class - An element of a corresponding list of a role

```
117     class CorrClass
118       include ObjectOrRole
119       attr_reader :varName, :relationship, :roleArray,
   :methArray, :argsArray
120
121       def initialize(varName, relationship, *rest)
122         @roleArray = Array.new
123         @methArray = Array.new
124         @argsArray = Array.new
125         @varName = varName
126         @relationship = relationship       # Type of
   relationship (0..3)
127
128         rest.each do |variable|
129           @roleArray.push(variable) if
   variable.instance_of?(Class)
130           @methArray.push(variable) if
   variable.instance_of?(String)
131           @argsArray.push(variable) if
   variable.instance_of?(Array)
132
133           @argsArray.push(nil) if @roleArray.size - 1 >
   @argsArray.size
134         end
135         @argsArray.push(nil) if @roleArray.size >
   @argsArray.size
136       end
```

Method getCorrValue - Should be changed for special use of argsArray and/or *rest

```
137       def getCorrValue(role, *rest)
138         if @relationship == 0 then      # id relationship
139           i = 0
140           @roleArray.each do |aRoleClass|
141             if role.class == aRoleClass then
142               if !findInPrivateList(role, @methArray[i]) then
143                 begin
144                   return role.send(@methArray[i],
   *@argsArray[i])
145                 rescue ArgumentError
146                   return role.send(@methArray[i])
147                 end
148               end
149             else
150               role.getAllSubRoles.each do |aRole|
151                 if aRole.class == aRoleClass then
152                   if !findInPrivateList(aRole, @methArray[i])
   then
153                     begin
154                       return aRole.send(@methArray[i],
   *@argsArray[i])
155                     rescue ArgumentError
156                       return aRole.send(@methArray[i])
157                     end
158                   end
159                 end
160               end
161             end
162             i += 1
164         elsif @relationship == 1 then     # all relationship
165           valueArray = Array.new
166           i = 0
167           @roleArray.each do |aRoleClass|
168             if role.class == aRoleClass then
169               if !findInPrivateList(role, @methArray[i]) then
170                 begin
171                   valueArray.push(role.send(@methArray[i],
   *@argsArray[i]))
172                 rescue ArgumentError
173                   valueArray.push(role.send(@methArray[i]))
174                 end
175               end
176             else
177               role.getAllSubRoles.each do |aRole|
178                 if aRole.class == aRoleClass then
179                   if !findInPrivateList(aRole, @methArray[i])
   then
180                     begin
181                       valueArray.push(aRole.send(@methArray[i],
   *@argsArray[i]))
182                     rescue ArgumentError
183                       valueArray.push(aRole.send(@methArray[i]))
184                     end
185                   end
186                 end
187               end
188             end
189             i += 1
190           end
```

```
192         elsif @relationship == 2 then     # sum relationship
193           value = 0
194           i = 0
195           @roleArray.each do |aRoleClass| # all classes of
   generalization
196             if role.class == aRoleClass then
197               if !findInPrivateList(role, @methArray[i]) then
198                 begin
199                   number = @argsArray[i].to_s.to_i # Caution at
   argsArray!!!
200                   value += role.send(@methArray[i], number)
201                 rescue ArgumentError
202                   value += role.send(@methArray[i])
203                 end
204               end
205             else
206               role.getAllSubRoles.each do |aRole|
207                 if aRole.class == aRoleClass then
208                   if !findInPrivateList(aRole, @methArray[i])
   then
209                     begin
210                       number = @argsArray[i].to_s.to_i # Caution
   at argsArray!!!
211                       value += aRole.send(@methArray[i], number)
212                     rescue ArgumentError
213                       value += aRole.send(@methArray[i])
214                     end
215                   end
216                 end
217               end
218             end
219             i += 1
220           end
221           return value
222         else      # general relationship
223           return nil
224         end
225       end   # end getCorrValue
226     end   # end CorrClass
```

7.3.3.6 role_helper

```
01 module RoleHelper
02   def RoleHelper.getRoleClasses
03     roleClasses = Array.new
04     Dir.chdir(Dir.getwd + "/app/models") if
   !Dir.getwd.include?("/app/models")
05     Dir.foreach(Dir.getwd) {|aFile|
06       if aFile.include?(".rb") then
07         aRole = aFile.split(".")[0].camelize.constantize
08         roleClasses.push(aRole)
09       end }
10     return roleClasses
11   end
```

```
12   def RoleHelper.getRoleClass(str)
13     RoleHelper.getRoleClasses.each do |aClass|
14       if aClass.name == str then
15         return aClass
16       end
17     end
18     return NilClass
19   end
20
21   def RoleHelper.getRole(str)
22     RoleHelper.getRoleClasses.each do |aClass|
23       allRoles = aClass.find(:all)
24       allRoles.each do |aRole|
25         if aRole.name == str then
26           return aRole
27         end
28       end
29     end
30     return nil
31   end
32
33   def RoleHelper.getValue(str)
34     if str[0..1].between?('A', 'Z') then
35       return RoleHelper.getRoleClass(str)
36     else
37       return RoleHelper.getRole(str)
38     end
39   end
40 end
```

7.3.3.7 routes

```
01 ActionController::Routing::Routes.draw do |map|
02   # The priority is based upon order of creation: first created
   -> highest priority.
03
04   # Role routing
05   base = '/:controller/:id'
06   map.roleOf '/roleOf', :path_prefix => base, :controller =>
   @controller, :action => 'roleOf'
07   map.roleRoot '/root', :path_prefix => base, :controller =>
   @controller, :action => 'root'
08
09   # Only useful for QualifiedRoleTypes
10   map.qualifier '/qualifier', :path_prefix => base, :controller
   => @controller, :action => 'qualifier'
11
12   # Sample of regular route:
13   #   map.connect 'products/:id', :controller => 'catalog',
   :action => 'view'
14   # Keep in mind you can assign values other than :controller
   and :action
```

```
15   # Sample of named route:
16   #   map.purchase 'products/:id/purchase', :controller =>
   'catalog', :action => 'purchase'
17   # This route can be invoked with purchase_url(:id =>
   product.id)
18
19   # Sample resource route (maps HTTP verbs to controller
   actions automatically):
20   #   map.resources :products
21
22   # Sample resource route with options:
23   #   map.resources :products, :member => { :short => :get,
   :toggle => :post }, :collection => { :sold => :get }
24
25   # Sample resource route with sub-resources:
26   #   map.resources :products, :has_many => [ :comments, :sales
   ], :has_one => :seller
27
28   # Sample resource route within a namespace:
29   #   map.namespace :admin do |admin|
30   #     # Directs /admin/products/* to
   Admin::ProductsController
   (app/controllers/admin/products_controller.rb)
31   #     admin.resources :products
32   #   end
33
34   # You can have the root of your site routed with map.root --
   just remember to delete public/index.html.
35   # map.root :controller => "welcome"
36
37   # See how all your routes lay out with "rake routes"
38
39   # Install the default routes as the lowest priority.
40   map.connect ':controller/:action/:id'
41   map.connect ':controller/:action/:id.:format'
42 end
```

8 Literaturverzeichnis

[1] Albano A., Bergamini R., Ghelli G., Orsini R.: *An object data model with roles.* In Conference Very Large Databases, 1993.

[2] Atkinson C., Kühne T.: *Model-Driven Development: A Metamodeling Foundation.* In IEEE Software. 2003.

[3] Bayer T.: *REST Web Services – Eine Einführung.* Orientation in Objects GmbH, Deutschland. 2002.

[4] Berners-Lee T.: *Semantic Web Road map.* http://akira.ruc.dk/~jv/KIIS2004/roadmap.pdf, 1998. Letzter Zugriff: 13. Dezember 2009.

[5] Berners-Lee T., Caillian R.: *World-Wide Web.* In Computing in High Energy Physics 92, Annecy, Frankreich. September 1992.

[6] Brachman R. J., Schmolze J. G.: *An overview of the KL-ONE knowledge representation system.* In Cognitive Science, Vol. 9, No. 9, 1985.

[7] Carl D.: *Praxiswissen Ruby on Rails.* 1. Auflage, O'Reilly Verlag, Deutschland. 2007.

[8] Demsky B., Rinard M.: *Role-based exploration of object-oriented programs.* In 24th International Conference of Software Engineering, Orlando, Florida (USA), 2002.

[9] Fielding R.: *Architectural Styles and the Design of Network-based Software Architectures.* University of California, Irvine (USA). 2000.

[10] Fielding R., Taylor R. N.: *Principled design of the modern web architecture*. In ACM Transaction on Information Systems, Vol. 2, No. 3, 2002.

[11] Gottlob G., Schrefl M., Röck B.: *Extending Object-Oriented Systems with Roles.* In ACM Transaction on Information Systems, Vol. 14, No. 3, Juli 1996.

[12] Genilloud G., Wegmann A.: *A foundation for the concept of role in object modelling.* In 4th International Enterprise Distribution Object Computing Conference, Makuhari, Japan, 2000.

[13] Huemer M.: *Rollen im Web of Data – Analyse, RDF-basiertes Rollenmodell und zugehöriger Data Browser.* Institut für Wirtschaftsinformatik – Data & Knowledge Engineering, Universität Linz. September 2009.

[14] Klas W., Schrefl M.: *Metaclasses and Their Application – Data Model Tailoring and Database Integration.* Springer 1991.

[15] Kristensen B. B.: *Object-oriented modeling with roles.* In 2nd International Conference on Object-Oriented Information Systems, Dublin, Irland, 1995.

[16] Kühne T., Schreiber D.: *Can programming be liberated from the two-level style: multi-level programming with deepjava.* International Conference on Object Oriented Programming, Systems, Languages and Applications. 2007.

[17] Kühne T., Steimann F.: *Tiefe Charakterisierung*. In Rumpe B., Hesse W.(Hrsg): Modellierung. 2004.

[18] Kuncak V., Lam P., Richard M.: *Role analysis.* In 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon (USA), 2002.

[19] Lippman S. B.: *Inside the C++ object model.* Addison-Wesley, 1996.

[20] Marinschek M., Radinger W.: *Ruby on Rails – Einstieg in die effiziente Webentwicklung.* 1. Auflage, dpunkt.verlag. 2006.

[21] Milton S., Schmidt H. W.: *Dynamic Dispatch in Object-Oriented Languages.* CSIRO – Division of Information Technology, Canberra (AUS). Januar 1994.

[22] Neumayr B., Grün K., Schrefl M.: Multi-Level Domain Modeling with M-Objects and M-Relationships. APCCM 2009.

[23] Oren E., Debru R.: *ActiveRDF: object-oriented RDF in Ruby.* DERI Galway, Irland. 2006.

[24] Oren E., Delbru R., Gerke S., Haller A., Decker S.: *ActiveRDF: Object-Oriented Semantic Web Programming.* National University of Ireland. 2007.

[25] Pernici B.: *Objects with roles.* In ACM SIGOIS and IEEE CS TC-OA Conference on Office Information Systems, Vol. 11, No. 2/3, Mar. 1990.

[26] Reenskaug T., Wold P., Lehne O. A.: *Working with objects – The OOram Software Engineering Method.* Taskon Work Environments, Oslo Norwegen. März 1995.

[27] Reimer U.: *A representation construct for roles*. In Data & Knowledge Engineering, Vol. 1, No. 3, 1985.

[28] *REXML Tutorial – Home.* http://www.germane-software.com/software/rexml/docs/tutorial.html. Letzter Zugriff: 11. Dezember 2009

[29] Richardson L., Ruby S.: *Web Services mit REST*. O'Reilly. 2007.

[30] Röhrl A., Schmiedl S., Wyss C.: *Programmieren mit Ruby: Eine praxisorientierte Einführung.* 1. Auflage, dpunkt.verlag, März 2002.

[31] Roithmayr F.: *Unterlagen zu Vorlesung und Übung Management von IT-Projekten – Teil A.* Institut für Wirtschaftsinformatik – Information Engineering, Universität Linz. September 2004.

[32] Rossi G., Nanard J., Nanard M.: *Engineering Web Application Using Roles.* In Journal of Web Engineering, Vol. 6, No. 1, 2007.

[33] *Rubys Prinzipien.* http://wiki.ruby-portal.de/RubysPrinzipien. Letzter Zugriff: 4. Dezember 2009.

[34] Schrefl M., Neuhold E.: *Object class definition by generalization using upward inheritance.* In IEEE. Darmstadt 1988.

[35] Schrefl M., Thalhammer T.: *Using roles in Java.* In Software – Practice and Experience, Vol 34, 2004.

[36] Steimann F.: *On the representation of roles in object-oriented and conceptual modelling.* In Data & Knowledge Engineering, Vol. 35, 2000.

[37] Thomas D., Fowler C., Hunt A.: *Programming Ruby – The Pragmatic Programmers' Guide.* 2. Auflage. Raleigh, North Carolina (USA). 2004.

[38] *TIOBE Software: Tiobe Index.* http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html. Letzter Zugriff: 30. November 2009.

[39] *Über Ruby.* http://www.ruby-lang.org/de/about/. Letzter Zugriff: 30. November 2009.

[40] Wintschel D.: *Ruby on Rails and XML – Generate a Rails stub to manipulate an XML document.* In http://www.ibm.com/developerworks/edu/x-dw-x-rubyonrailsxml.html. April 2007. Letzter Zugriff 4.Dezember 2009.

[41] Wirdemann R., Baustert T.: *Rapid Web Development mit Ruby on Rails.* 3. Auflage, Hanser Verlag, Deutschland. 2008.

[42] Zho H., Zhou M.: *Roles in Information Systems: A Survey*. In IEEE Transactions on Systems, Man, And Cybernetics - Part C: Applications And Reviews, Vol. 38, No. 3. Mai 2008.