

Otto-von-Guericke-Universität Magdeburg



Fakultät für Informatik
Institut für Technische und Betriebliche Informationssysteme

Diplomarbeit

Sprachübergreifende Refactoring Feature Module

Verfasser:

Hagen Schink

20. August 2010

Betreuer:

Prof. Dr. rer. nat. habil. Gunter Saake,
Dipl.-Inform. Martin Kuhlemann

Universität Magdeburg
Fakultät für Informatik
Postfach 4120, D-39016 Magdeburg
Germany

Schink, Hagen:

Sprachübergreifende

Refactoring Feature Module

Diplomarbeit, Otto-von-Guericke-Universität

Magdeburg, 2010.

So schwätzt und lehrt man ungestört;
Wer will sich mit den Narrn befassen?
Gewöhnlich glaubt der Mensch, wenn er nur Worte hört,
Es müsse sich dabei doch auch was denken lassen.

Johann Wolfgang von Goethe: Faust, Der Tragödie erster Teil

Abstract

As the demand for customer specific software applications remains growing, there is a need for software developers to explore new ideas that allow an efficient description of potential customer desires and functionality in application source code. *Feature-oriented programming* (FOP) is one approach to meet that need. FOP enables the developer to automatically add features to software applications on demand.

A software application generated by the combination of features is often part of a bigger software system consisting of several different applications. Those software systems have well defined interfaces in which the new software application has to fit in. Structural but behavior preserving modification called *refactorings* can be applied to the software application, so the software application is able to communicate with the already existing interfaces used in a software system. A so called *refactoring feature module* (RFM) describes the application of refactorings on software composed of features.

Most software applications do not utilize only one programming language but rather make use of a set of different general purpose and domain specific languages respectively, e.g. Java, SQL, XML. That is what we call a *multi-language software application*. By utilizing the means of the different artifacts, software developers are able to solve problems more efficiently. But due to the co-operation between the different artifacts, the refactoring of only one artifact may already prevent interaction between the artifacts. *Multi-language refactorings* seek to solve this problem by modifying all the interacting software artifacts accordingly and by re-establishing the expected behavior of the whole software application while preserving the modifications made by the initial refactoring.

In this thesis we investigate properties and problems of multi-language refactorings by applying refactorings on different artifacts of a sample multi-language application. This refactorings include modifications of Java, SQL, the object-relational mapper Hibernate and a functional programming language called Clojure. Based on the investigation we implement a prototypical software which automatically realizes selected multi-language refactorings. This prototype is then used to realize multi-language refactorings in RFMs. We then evaluate our implementation on different software applications and case studies.

In this thesis we found that multi-language refactorings are not applicable under all conditions. A structural and behavior preserving modification on one artifact can turn out to be a behavior changing modification on an interacting artifact. Our conclusion is that one can implement multi-language refactorings under certain conditions. The conditions depend on the artifacts and the refactorings involved in the multi-language refactoring. I.e., if two artifacts of a software application are interacting with each other, one needs to know if a refactoring of one artifact can be reproduced by another refactoring on the other artifact. Otherwise, one may not be able to preserve the semantics of the software application.

Inhaltsverzeichnis

Inhaltsverzeichnis	iii
Abbildungsverzeichnis	v
Tabellenverzeichnis	vii
Verzeichnis der Abkürzungen	ix
1 Einleitung	1
1.1 Problemstellung	2
1.2 Ziel und Inhalt	2
2 Grundlagen	5
2.1 Mehrsprachige Software-Anwendungen	5
2.2 Wiederverwendbarkeit und Objekt-orientierte Programmierung	7
2.3 Refactoring	9
2.3.1 Refactoring von Datenbanken	10
2.3.2 Mehrsprachiges Refactoring	11
2.4 Software Produktlinien	14
2.5 Feature-orientierte Programmierung	15
2.6 Refactoring Feature Module	16
3 Verwandte Arbeiten	19
3.1 Analyse in mehrsprachigen Software-Anwendungen	19
3.2 Generische und sprachunabhängige Refactorings	21
3.3 Coupled Software Transformations	22
3.4 Mehrsprachige Refactorings	23

3.5	Zusammenfassung	25
4	Realisierung mehrsprachiger Refactorings	27
4.1	Beschreibung der Beispiel-Anwendung <i>HRManager</i>	27
4.1.1	Anwendungslogik	29
4.1.2	Persistenz	31
4.1.3	Objekt-relationales Mapping	33
4.1.4	Scripting	36
4.2	Probleme mehrsprachiger Refactorings	37
4.2.1	Java	38
4.2.2	SQL	49
4.2.3	Clojure	55
4.2.4	Weitere Besonderheiten	59
4.3	Zusammenfassung	60
5	Automatisierung mehrsprachiger Refactorings	65
5.1	Beschreibung fehlender Hibernate-Refactorings	66
5.1.1	Rename Target Column Refactoring	66
5.1.2	Remove Class from ORM Refactoring	66
5.2	Automatischen Umsetzung mehrsprachiger Refactorings	67
5.2.1	Implementierung der Architektur	71
5.2.2	Diskussion zur Erweiterbarkeit der Architektur	76
5.3	Integration mehrsprachiger Refactorings in den RFMComposer	76
5.4	Evaluation der mehrsprachigen Refactorings	77
5.4.1	HRManager	79
5.4.2	Seam-gen	80
5.4.3	Seam Anwendungsbeispiele	82
5.4.4	Ergebnisse der Evaluation	87
5.5	Zusammenfassung	88
6	Zusammenfassung und Ausblick	89
	Literaturverzeichnis	93

Abbildungsverzeichnis

2.1	Beispiel einer Klassenhierarchie	8
2.2	Beispiel für Instanzen	8
2.3	Beispiel EXTRACT SUPERCLASS REFACTORING	10
2.4	Beispiel SPLIT COLUMN REFACTORING	11
2.5	Beispiel mehrsprachiges Refactoring	13
4.1	Klassenhierarchie der Oberklasse <code>Employee</code>	30
4.2	Klassenhierarchie der Oberklasse <code>ExternalStaff</code>	30
4.3	Die Klassen <code>Department</code> und <code>Customer</code>	31
4.4	Repräsentation der <code>Employee</code> -Klassenhierarchie in der Datenbank	32
5.1	Extraktion der Artefakt-übergreifenden Referenzen	68
5.2	Darstellung eines automatisierten mehrsprachigen Refactorings	70
5.3	Klassen zum Aufbau eines Identifier-Repositories	72
5.4	<code>Refactor</code> -Interface und die implementierenden Klassen	75
5.5	Architektur des <code>RFMComposers</code>	77
5.6	Beschreibung des <code>RFM</code> -Schemas	79
5.7	Evaluation am <code>HRManager</code>	80
5.8	Klassen des <code>Seam Identity Managements</code>	81
5.9	Evaluation am <code>Seam-gen</code> Projekt	82
5.10	Von Hibernate verwaltete Klassen des <code>Seam DVD Store</code> Beispiels	84
5.11	Evaluation am <code>Seam DVD</code> Anwendungsbeispiel	85
5.12	Von Hibernate verwaltete Klassen des <code>Seam Space</code> Beispiels	86
5.13	Evaluation am <code>Seam Space</code> Anwendungsbeispiel	87

Tabellenverzeichnis

4.1	Zusammenfassung der Eigenschaften mehrsprachiger Refactorings	61
5.1	Die in der Implementierung verwendeten Parser	71
5.2	Objekt-Typen der Entitäten verschiedener Artefakte	73
5.3	Ergebnisse der Evaluation der mehrsprachigen Refactorings	87

Verzeichnis der Abkürzungen

API Application Programming Interface

AST Abstract Syntax Tree

DMM Dagstuhl Middle Metamodel

DSL Domain-specific language

EL Expression Language

FOP Feature-orientierte Programmierung

GPL General-purpose language

GXL Graph eXchange Language

HQL Hibernate Query Language

JNI Java Native Interface

JPA Java Persistence API

JSF JavaServer Faces

JSR Java Specification Request

OOP Objekt-orientierte Programmierung

ORM Objekt-relationales Mapping

RAD Rapid Application Development

RFM Refactoring Feature Module

RIA Rich Internet Application

SN Source Navigator

SPL Software Produktlinie

SPLE Software Produktlinien Engineering

SQL Structured Query Language

XML Extensible Markup Language

Kapitel 1

Einleitung

Die Wiederverwendung von Software und Software-Komponenten kann zu einer Steigerung der Software-Qualität, einer Verkürzung der Produkteinführungszeit sowie zur Senkung der Komplexität und des Wartungsaufwands führen [PBL05, S. 10 - 11; Bos03]. Damit verbunden ist auch eine Reduzierung der Kosten für die Software-Entwicklung [Opd92, Bos03]. Daher ist die Wiederverwendung von Software und Software-Komponenten seit langem ein wichtiger Aspekt der Software-Entwicklung [WOZS94, S. 1; Opd92, S. 6].

Um ihre Wettbewerbsfähigkeit zu sichern, sind viele namhafte Hersteller dazu übergegangen, *Software-Produktlinien* (SPL) zu implementieren [Nor02]. Eine SPL ist ein Ansatz zur Umsetzung und Nutzung von wiederverwendbaren Software-Komponenten [SB00, SKR⁺09]. Die Variabilität, die mit dem Einsatz von SPLs erreicht werden kann, bietet darüber hinaus die Möglichkeit, Software effizient für spezifische Kundenanforderungen zu entwickeln [Nor02].

Bestandteil von SPLs sind Software-Anwendungen, die eine Menge von Funktionen implementieren, die den Anforderungen eines bestimmten Markt-Segments, Anwendungsbereichs oder allgemein einer Domäne genügen. Die Software-Anwendungen werden dabei aus Basis-Komponenten in einem vordefinierten Verfahren zusammengestellt [PBL05, S. 20; Nor02].

Die Software-Anwendungen innerhalb einer SPL werden anhand der von ihnen implementierten *Features* unterschieden [SKR⁺09, AK09]. Ein Feature einer Software-Anwendung ist eine Funktionalität, die eine von der Anwendungsdomäne definierte Anforderung erfüllt. Dabei können Features optional sein und müssen demnach nicht von einer Software-Anwendung implementiert werden [AK09].

Feature-orientierte Programmierung (FOP) greift das Konzept *Feature* auf, um Software-Anwendungen einer Domäne zu beschreiben [Bat04]. In FOP werden Feature in *Feature Modulen* implementiert [BSR03]. Software-Anwendungen werden dann durch Komposition der Feature-Module erstellt [Pre97, SKS⁺08, KBA09]. Dieser Vorgang wird als *Feature-Komposition* bezeichnet [BSR03].

Die mit FOP erstellten Software-Anwendungen werden oft durch andere Programme verwendet. Um eine gegenseitige Nutzung zu ermöglichen, werden bestimmte Schnittstellen von den beteiligten Anwendungen vorausgesetzt. Sind die erwarteten Schnittstellen nicht vorhanden, können die Anwendungen nicht miteinander interagieren. Die Anwendungen werden dann als *inkompatibel* bezeichnet [KBA09].

Die Modifikation der Struktur vorhandener Software, ohne deren Semantik zu verändern, wird als *Software-Refactoring* oder nur *Refactoring* bezeichnet [Opd92, S. 2; Fow99]. Refactorings können auf unterschiedliche Software-Artefakte, wie Quellcode oder UML-Diagramme, angewendet werden [Opd92, S. 2; VSMD03]. Aktuelle Forschungsansätze versuchen die Möglichkeiten des Refactorings mit FOP zu vereinen, um Inkompatibilitäten zu beseitigen. Diese mündeten in der Beschreibung von *Refactoring Feature Modulen* (RFM) [KBA09].

1.1 Problemstellung

Moderne Software-Systeme bestehen nicht nur aus dem Quellcode einer Programmiersprache [KLW06, For08]. Bereits 1998 nutzten ein Drittel aller Software-Projekte mehr als eine Programmiersprache für die Implementierung [Jon98]. Heute wird von einer weit höheren Anwendung ausgegangen [KLW06].

In modernen Webanwendungen ist es üblich, Java, SQL und JavaScript zu verwenden [For08]. Ferner existieren daneben weitere Software-Artefakte. Dazu gehören beispielsweise XML-Konfigurationsdateien [CJ08], UML-Modelle [VSMD03, BSR03] und Design-Dokumente [BSR03]. Software-Anwendungen, die aus unterschiedlichen Software-Artefakten bestehen, werden als mehrsprachige Software-Anwendungen bezeichnet [LLMM07].

Die verschiedenen Software-Artefakte interagieren miteinander und beeinflussen sich. Die Konfiguration von bestimmten Java-Frameworks ist z.B. über XML-Konfigurationsdateien möglich [CJ08]. Darüber hinaus unterstützen aktuelle Plattformen, wie Java oder .Net, explizit die Verwendung unterschiedlicher Programmiersprachen [Lia99; O’C06; Ver08, S. 6]. Es ist z.B. über das *Java Native Interface* (JNI) möglich, nativen Code aus Java heraus aufzurufen und umgekehrt [Lia99]. Das Java-Scripting-API ermöglicht die Verwendung bestimmter Script-Sprachen aus Java heraus und den Zugriff auf die Java Standard-Bibliothek durch die unterstützen Script-Sprachen [O’C06].

Zumeist ist der Einsatz von Refactoring-Werkzeugen auf ein Software-Artefakt beschränkt. Die Interaktion zwischen unterschiedlichen Software-Artefakten wird damit nicht beachtet. Das erschwert den Einsatz von Refactorings innerhalb mehrsprachiger Software-Anwendungen [CJ08].

1.2 Ziel und Inhalt

Das Ziel dieser Arbeit liegt in der Erörterung von Problemen bei der Verwendung von Refactorings vor dem Hintergrund mehrsprachiger Software-Anwendungen. D.h., wie können Interaktionen zwischen einer großen Anzahl von unterschiedlichen Artefakten aufgedeckt und darauf aufbauend Refactorings realisiert werden. Mit Hilfe der gewonnenen Erkenntnisse soll dann eine prototypische Implementierung mehrsprachiger Refactorings für RFMs erfolgen.

Die Erläuterung von Eigenschaften mehrsprachiger Refactorings erfolgt im Bezug auf eine Beispiel-Anwendung, die mit Java, Hibernate, SQL und Clojure eine Software

zur Verwaltung von Mitarbeiter-Daten implementiert. Mit Hilfe der Beispiel-Anwendung werden ausgewählte Refactorings auf Java, SQL und Clojure angewendet und der Einfluss der Refactorings auf die Funktionalität der Beispiel-Anwendung beschrieben. Dabei werden Modifikationen erläutert, die zum einen das Resultat des jeweiligen Refactorings erhalten und zum anderen die Funktionalität der Beispiel-Anwendung wiederherstellen. Die Eigenschaften dieser Modifikationen werden verallgemeinert und zusammengefasst. Aus der Zusammenfassung werden Schlussfolgerungen für die automatisierte Umsetzung von Refactorings abgeleitet. Auf Basis dieser Schlussfolgerungen wird eine Architektur zur Umsetzung automatisierter mehrsprachiger Refactorings vorgestellt. Die prototypische Implementierung der Architektur wird in eine bestehende Software-Anwendung zur Umsetzung von RFMs integriert und evaluiert. Die Evaluierung erfolgt anhand der Beispiel-Anwendung, eines Software-Tools und zwei gewählter Anwendungsbeispiele eines Software-Frameworks.

Die Arbeit ist wie folgt gegliedert. Im Kapitel 2 werden die grundlegenden Begriffe und Konzepte eingeführt. Das Kapitel 3 stellt verwandte Arbeiten und Lösungskonzepte vor und setzt diese in Beziehung zu der vorliegenden Arbeit. Den Hauptteil bildet Kapitel 4. Es umfasst die Erarbeitung der unterschiedlichen Probleme bei der Umsetzung mehrsprachiger Refactorings sowie deren Verallgemeinerung und Zusammenfassung. In Kapitel 5 wird eine Architektur zur Umsetzung mehrsprachiger Refactorings und deren prototypische Implementierung vorgestellt. Des Weiteren wird die Integration der prototypischen Implementierung in eine bestehende Software zur Umsetzung von RFMs beschrieben und evaluiert. Den Abschluss bildet das Kapitel 6 mit einer Zusammenfassung der Arbeit sowie einem Ausblick auf weitere Forschungsfragen.

Kapitel 2

Grundlagen

In diesem Kapitel werden die für die Arbeit wichtigen Begriffe *FOP*, *Refactoring* und *mehrsprachige Software-Anwendung* genauer betrachtet. Dazu werden des Weiteren die Begriffe *Objekt-orientierte Programmierung* (OOP) und *Software-Produktlinien* (SPL) beschrieben.

Die Beschreibung des Objekt-orientierten Paradigmas erfolgt im Kontext der Wiederverwendbarkeit von Software-Komponenten, da Objekt-Orientierung zur Umsetzung von wiederverwendbaren Software-Komponenten weit verbreitet ist. Die bei der Beschreibung eingeführten Begriffe werden an einem eigens gewählten Beispiel veranschaulicht.

Dieses Kapitel ist wie folgt aufgebaut. Zunächst erfolgt die Erläuterung mehrsprachiger Software-Anwendungen und deren Verbreitung in der Software-Entwicklung. Daran schließt sich die Beschreibung von Wiederverwendbarkeit im Rahmen von OOP an. Darauf wird der Begriff Refactoring eingeführt und in den Zusammenhang mit mehrsprachigen Software-Anwendungen gebracht. Das Kapitel schließt mit den Erläuterungen zu SPL und FOP. Die Erläuterungen zu FOP werden des Weiteren durch die Einführung des Begriffs *Refactoring Feature Modul* (RFM) ergänzt.

2.1 Mehrsprachige Software-Anwendungen

In modernen Software-Applikationen ist es üblich, weit mehr als nur eine Programmiersprache einzusetzen[Jon98, LLMM07, For08, CJ08, SKL06, GPB04]. In Webanwendungen werden oft z.B. SQL und JavaScript neben einer universellen Programmiersprache (engl. General-purpose language, GPL) wie Java eingesetzt[For08].

Programmiersprachen, die für unterschiedliche Anwendungsgebiete eingesetzt werden können, werden als General-purpose Sprachen bezeichnet[SH03]. Zu diesen Programmiersprachen zählen z.B. C/C++, Java, PL/I und Ada[SH03, For08]. Im Gegensatz dazu existieren special-purpose Sprachen, die auch als Domain-spezifische Sprachen (DSL) bezeichnet werden[SH03]. Eine DSL dient nur einer spezifischen Aufgabe oder einem spezifischen Anwendungsgebiet[SH03].

Eine Software-Anwendung, deren Implementierung mit mehreren Programmiersprachen erfolgt, wird als mehrsprachige Software-Anwendung bezeichnet[LLMM07]. In [For08] wird die Entwicklung einer Software mit verschiedenen special-purpose und einer general-purpose Sprache als Polyglot Programming (engl. mehrsprachiges Programmie-

ren) definiert¹. Bini und Fields erweitern in [Fje08] diese Definition und beschreiben Polyglot Programming als generellen Einsatz mehrerer Programmiersprachen in einem Software-Projekt.

Bezug nehmend auf die genannten Definitionen, werden für diese Arbeit die Begriffe mehrsprachige Software-Anwendung und Polyglot Programming folgendermaßen definiert.

Definition Software-Anwendungen, die mit Polyglot Programming umgesetzt werden, werden als MEHRSPRACHIGE SOFTWARE-ANWENDUNGEN bezeichnet.

Die Anzahl der Systeme, die sich als mehrsprachige Software-Anwendungen bezeichnen lassen, hängt nun von der Definition des Begriffs Polyglot Programming ab. In Übereinstimmung mit der Aussage über mehrsprachige Software-Anwendungen in [LLMM07] und den Aussagen von Bini und Fields über Polyglot Programming in [Fje08] soll die folgende Definition gelten:

Definition Wenn mehrere general-purpose oder special-purpose Programmiersprachen zur Umsetzung einer Software-Anwendung eingesetzt werden, dann wird dies als POLY-GLOT PROGRAMMING bezeichnet.

Die Anwendung von Polyglot Programming ist in der Software-Entwicklung weit verbreitet[For08]. Dabei erfolgt der konkrete Einsatz von Polyglot Programming vielgestaltig. Als Beleg dafür sollen die drei folgenden Beispiele dienen.

Für den Zugriff auf Datenbanken hat sich die *Structured Query Language* (engl. Strukturierte Abfragesprache, SQL) etabliert. SQL wurde in ihren Anfängen nicht als vollständige Programmiersprache sondern als reine Abfragesprache entworfen. Zur Gewährleistung des vollen Funktionsumfangs beim Einsatz von SQL ist eine Einbettung in eine Programmiersprache vorgesehen[OP04].

Die Verwendung von *Extensible Markup Language* (engl. erweiterbare Auszeichnungssprache, XML) ist in vielen Anwendungsbereichen üblich. Vor allem ist XML als Datenaustausch-Format bekannt[OP04]. XML dient aber auch zur Beschreibung von Konfigurationsdateien[CJ08]. Allgemein wird XML zur strukturierten textuellen Beschreibung von Daten verwendet[HM02].

Die Programmiersprache Java unterstützt Polyglot Programming über entsprechende *Application Programming Interfaces* (engl. Schnittstelle zur Anwendungsprogrammierung, API). Das *Java Native Interface* (JNI) dient der Einbindung von nativen Code, der mit C/C++ entwickelt wurde[Lia99]. Über JNI ist in C/C++ auch der Aufruf von Methoden möglich, die mit Java entwickelt wurden [Lia99]. Des Weiteren spezifiziert der *Java Specification Request* (JSR) 223 ein API zur Einbettung von Script-Sprachen in Java[O’C06].

¹In [Fje08] schränkt Ford diese Definition weiter ein und sagt, dass Polyglot Programming nur dann vorliegt, wenn die verschiedenen Sprachen dieselbe Laufzeitumgebung nutzen.

2.2 Wiederverwendbarkeit und Objekt-orientierte Programmierung

Die Implementierung wiederverwendbarer Software-Komponenten ist eine klassische Forderung des Software-Engineerings[Opd92, S. 6][WOZS94]. Das Ziel ist, die Implementierung von Applikationen planbarer zu gestalten bzw. die Entwicklungszeiträume zu begrenzen, um Ressourcen und damit Kosten zu sparen[Opd92, S. 6; WOZS94].

Objekt-orientierte Programmierung (OOP) stellt Funktionen bereit, die explizit die Wiederverwendung von Software-Komponenten unterstützen[KM90, Sny86]. Insbesondere sind das die Konzepte *Objekt*, *Klasse* und *Vererbung*[KM90]. In [Opd92, S. 9] wird als weiteres Konzept der Wiederverwendbarkeit in OOP *Polymorphismus* genannt. Diese Konzepte werden wie folgt beschrieben:

Objekte stellen die Entitäten in einem Anwendungsprogramm dar[KM90]. Objekte besitzen sowohl einen Zustand als auch mit den Objekten verknüpfte Funktionen[KM90].

Klassen beschreiben Mengen von möglichen Objekten[KM90]. Damit bilden Objekte Instanzen von Klassen[Sny86]. Mit Klassen ist es möglich, Implementierungsdetails zu kapseln und zu verbergen[KM90]. Dies wird auch als Datenabstraktion bezeichnet[Sny86, Opd92]. Eine öffentliche Schnittstelle ermöglicht den Zugriff auf den Status sowie dessen Modifikation[KM90].

Die durch Instanzieren einer Klasse erstellten Objekte bilden eine Form der Wiederverwendung in OOP. Im Gegensatz zu einfachen Datentypen beinhalten diese Objekte sowohl komplexere Daten-Strukturen als auch die damit verbundenen Operationen[KM90].

Vererbung ist eine Relation zwischen Klassen, die es erlaubt, eine Klasse auf Basis der Implementierung einer bereits existierenden Klasse zu definieren[KM90; Sny86; Opd92, S. 8-9]. Dabei entstehen Abhängigkeiten zwischen den an einer Vererbung teilnehmenden Klassen[KM90]. Durch die gebildeten Abhängigkeiten stehen Veränderungen an der geerbten Klasse auch der erbenden Klasse zur Verfügung[KM90]. Andererseits können Erweiterungen vorgenommen werden, ohne die bestehende Code-Basis zu verändern[KM90]. Der neue Code wird dann in die erbenden Klassen aufgenommen. Die bestehende Code-Basis bleibt unberührt[KM90].

Eine Klasse, die von einer anderen erbt, wird als Spezialisierung bezeichnet. Dadurch entsteht eine Hierarchie zwischen der erbenden und der geerbten Klasse[Opd92, S. 8-9]. Diese Hierarchie wird im Folgenden im Einklang mit dem allgemeinen Sprachgebrauch als *Klassen-Hierarchie* bezeichnet. Gegeben sei eine Klasse X, dann werden alle Klassen, die direkt oder indirekt von X abgeleitet sind, als *Unterklasse* von X bezeichnet[Sny86]. Außerdem werden alle Klassen von denen X erbt als *Oberklasse* bezeichnet[Sny86].

Polymorphismus erlaubt den Aufruf von Funktionen auf Objekten, ohne dass die Funktionen den Typ des Objekts kennen. Dabei erwartet die Funktion eine bestimmte Schnittstelle. Damit kann die Funktion nur für Objekte wiederverwendet

werden, die die erwartete Schnittstelle implementieren [Opd92, S. 9]. Schnittstellen werden von Klassen definiert. In einigen Sprachen, wie Java und C++, wird die Klassen-Hierarchie herangezogen, um die Objekte zu identifizieren, die die erwartete Schnittstelle implementieren [KM90]. Demnach kann auf eine Funktion, die für die Klasse **X** definiert ist, auch von allen Unterklassen von **X** zugegriffen werden.

Die genannten Konzepte und deren Zusammenspiel werden durch ein Beispiel verdeutlicht, das ebenfalls in den darauf folgenden Betrachtungen genutzt wird.

Es soll eine Software zur Verwaltung von Angestellten implementiert werden. Die dazu notwendigen Daten sollen in einer eigenen Klasse **Employee** gekapselt werden. Da leitende Angestellte einen eigenen Dienstwagen erhalten, soll dieser ebenfalls in der Software vermerkt werden. Dazu wird von der Klasse **Employee** die Unterklasse **Manager** abgeleitet. Die Klasse **Manager** besitzt ein zusätzliches Attribut, das auf den Dienstwagen des entsprechenden Angestellten verweist. Die eben beschriebene Klassenhierarchie mit den Attributen der einzelnen Klassen ist in Abbildung 2.1 dargestellt.

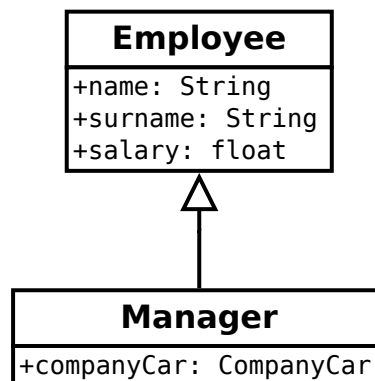
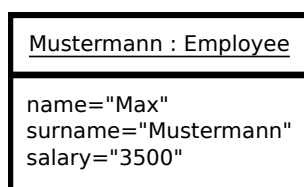
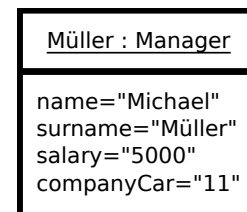


Abbildung 2.1: Die Klasse **Employee** und deren Unterklasse **Manager**. (Quelle: Eigene Darstellung)

Die ersten Mitarbeiter-Daten werden in die neue Software eingepflegt. Dazu müssen Instanzen der Klasse **Employee** bzw. **Manager** angelegt werden, die dann die konkreten Mitarbeiter repräsentieren. Abbildung 2.2 zeigt Beispiele für Instanzen der Klassen **Employee** und **Manager**.



(a) Instanz der Klasse **Employee**. (Quelle: Eigene Darstellung)



(b) Instanz der Klasse **Manager**. (Quelle: Eigene Darstellung)

Abbildung 2.2: Instanzen der Klassen **Employee** (a) und **Manager** (b).

Um Überstunden der Mitarbeiter zu protokollieren, enthält die Anwendung eine Methode zum Festhalten der zusätzlichen Arbeitszeiten. Die Methode `storeOvertime` über-

Listing 2.1: Methode zum Notieren von Überstunden

```
1 void storeOvertime(Employee employee, int minutes, Date date);
```

nimmt dazu drei Parameter: Zum einen den Mitarbeiter und zum anderen die geleistete Mehrarbeit sowie das Datum, an dem die Überstunden geleistet wurden. Die Signatur der Methode ist in Listing 2.1 dargestellt. Da die Klasse `Manager` Unterklasse von `Employee` ist, kann die Methode auch auf Instanzen der Klasse `Manager` angewendet werden. Es wird explizit der Polymorphismus der Methode `storeOvertime` ausgenutzt.

2.3 Refactoring

Neue oder veränderte Anforderungen an eine Software während oder nach deren Implementierung erfordern die Erweiterung oder Modifikation des bereits entwickelten Quellcodes[Opd92, S. 10]. Nicht immer lassen sich neue Anforderungen in die vorhandene Code-Struktur problemlos integrieren[Opd92, S. 10]. Um die Wartbarkeit des Quellcodes zu gewährleisten, sind dann u.U. strukturelle Anpassungen notwendig[Opd92, S. 10]. Je größer der Umfang dieser Modifikationen ist, desto schwerer wird es, die nötigen Veränderungen zu überblicken[Opd92, S. 8]. Damit steigt die Fehleranfälligkeit des Vorgangs und die Gefahr, dass der Code nach den Modifikationen nicht kompiliert oder ein unerwünschtes Verhalten zeigt.

Der Begriff des Refactorings bezeichnet Prozesse, die strukturelle Veränderungen am Quellcode durchführen, die nicht dessen nach außen hin sichtbares Verhalten ändern[Opd92, S. 2; Fow99, S. 53]. Das Ziel von Refactorings ist die Unterstützung des Entwicklers bei Maßnahmen zur Verbesserung der Code-Qualität[Opd92, S. 2; Fow99, S. 53]. Demnach muss ein Refactoring die Korrektheit von zuvor ebenfalls korrektem Code gewährleisten, auf den es angewendet wird[Opd92, S. 26].

Im Bezug auf die Definition von Fowler in [Fow99] wird das nach außen hin sichtbare Verhalten einer Anwendung in dieser Arbeit auch als Semantik bezeichnet.

Neben den eigentlichen Modifikationen werden zu Refactorings ebenfalls Vor- oder Nachbedingungen definiert[MT04]. Die Vorbedingungen stellen sicher, dass der zu modifizierende Sourcecode alle Voraussetzungen zur erfolgreichen Durchführung eines Refactorings erfüllt[Opd92, S. 32]. Wenn z.B. eine Methode innerhalb einer Klasse umbenannt werden soll, muss sichergestellt sein, dass nicht bereits eine andere Methode mit dem neuen Methoden-Bezeichner in der Klasse existiert[Opd92, S. 43]. Nachbedingungen überprüfen die Erhaltung der Semantik des Sourcecodes nach der Anwendung eines Refactorings[Rob99, S. 36]. Werden nach der Umbenennung einer Methode die Nachbedingungen des Refactorings geprüft, muss z.B. die Einmaligkeit des neuen Methoden-Bezeichners innerhalb der entsprechenden Klasse gelten.

Die Wirkung eines Refactorings soll im Folgenden an einem Beispiel erläutert werden. Mit Hilfe der Software-Anwendung sollen ebenfalls Verkäufer verwaltet werden. Dazu soll die Klasse `Salesperson` von der Klasse `Employee` abgeleitet werden. Wie die Klasse `Manager` enthält die Klasse `Salesperson` einen Verweis auf den Dienstwagen des jeweiligen Angestellten. Um die Wartbarkeit des Codes sicherzustellen, wird vorgeschlagen, dieses Attribut in einer gemeinsamen Oberklasse zu kapseln. In [Fow99, S. 149]

wird das **EXTRACT SUPERCLASS REFACTORING** beschrieben, das die Einführung einer gemeinsamen Oberklasse erläutert. Dieses Refactoring wird auf die in Abbildung 2.3(a) dargestellten Klassen **Manager** und **Salesperson** angewendet. Nach der Durchführung des Refactorings erhält die Klassenhierarchie die in Abbildung 2.3(b) dargestellte Struktur. Die abstrakte Klasse **PrivilegedEmployee** enthält nun den Verweis auf den Dienstwagen, den die Klassen **Manager** und **Salesperson** gemein haben.

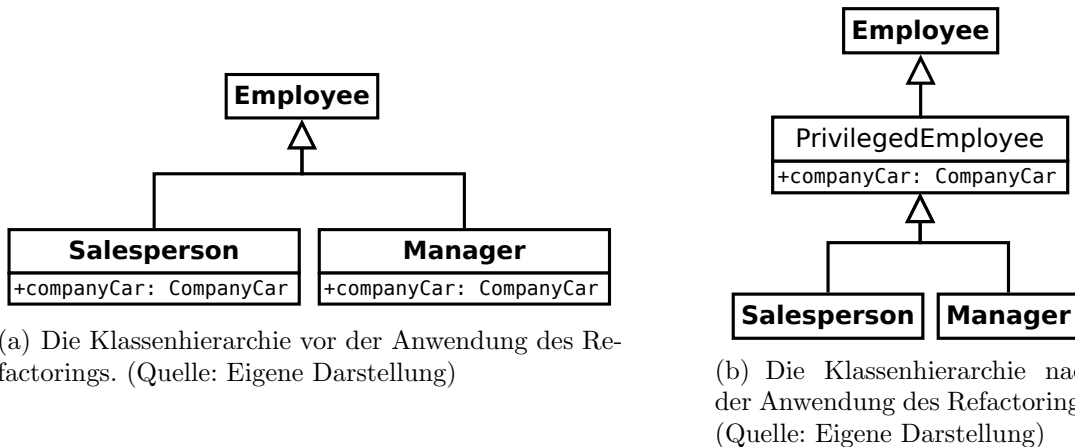


Abbildung 2.3: Die Abbildung (a) zeigt die Klassenhierarchie, nachdem die Klasse **Salesperson** eingeführt wurde. Abbildung (b) zeigt die Klassenhierarchie nach der Durchführung des **EXTRACT SUPERCLASS REFACTORINGS**.

2.3.1 Refactoring von Datenbanken

Das Refactoring von Datenbanken beschreibt Prozesse, die strukturelle Veränderungen an einem Datenbank-Schema vornehmen, ohne das äußere Verhalten der Datenbank zu verändern[Amb03, S. 177]. Ein Aspekt, der das Refactoring von Datenbanken gegenüber den zuvor beschriebenen Objekt-orientierten Refactorings unterscheidet, ist die zusätzlich geforderte Erhaltung der informationellen Semantik[Amb03, S. 178]. Das bedeutet, dass die durch das Datenbank-Schema dargestellten Informationen auch nach dem Refactoring vorhanden sein müssen[Amb03, S. 178].

Datenbank-Refactorings können zu semantischen Änderungen führen, die nach außen hin sichtbar sind[Amb03, S. 178]. Diese Datenbank-Refactorings dürfen demnach nicht als Refactoring im Sinne der in dieser Arbeit folgenden Definition bezeichnet werden. Ambler argumentiert aber, dass durch die Wahl eines anderen Blickwinkels die umgesetzten Modifikationen dennoch als Refactoring deklariert werden können. Konkret schreibt Ambler, dass eine Anwendung, die die Datenbank verwendet, im Bezug auf die geänderte Semantik des Datenbank-Schemas ebenfalls angepasst werden muss. Für den Nutzer dieser Anwendung wird die vorgenommene semantische Änderung des Datenbank-Schemas durch die zusätzliche Modifikation der Anwendung nicht sichtbar. Daher ist sowohl die Semantik wie auch das Verhalten des Datenbank-Schemas durch die vorgenommene Modifikation der Anwendung im Bezug auf den Nutzer der Anwendung erhalten worden.

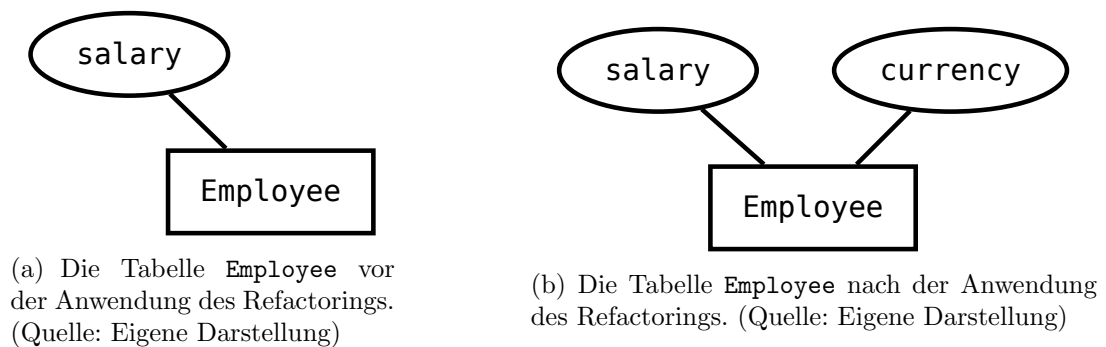


Abbildung 2.4: Die Abbildung (a) zeigt den Ausgangszustand der Tabelle `Employee`+ Abbildung (b) zeigt die Tabelle nach der Durchführung des `SPLIT COLUMN REFACTORINGS`.

Bei seiner Argumentation betrachtet Ambler nicht die direkte Nutzung der Datenbank. System-Administratoren könnten z.B. direkt Anfragen an die Datenbank stellen, ohne dass sie sich einer Anwendung bedienen. In dieser Situation wirkt sich die semantische Modifikation der Datenbank direkt auf die Nutzer aus. Ob die vorgenommene Änderung des Datenbank-Schemas als Refactoring bezeichnet werden kann, ist demnach abhängig von dem Kontext, aus dem die Modifikation des Datenbank-Schemas heraus betrachtet wird.

In [Amb03] stellt Ambler einen Katalog von Datenbank-Refactorings zur Verfügung. Es werden in dem Katalog keine Aussagen darüber gemacht, unter welchen Bedingungen die genannten Modifikationen als Refactorings oder als nicht Semantik-erhaltende Änderungen gelten.

Im Folgenden soll als Beispiel für ein Datenbank-Refactoring das `SPLIT COLUMN REFACTORING`[Amb03, S. 420] dienen. Die Software zur Verwaltung der Mitarbeiter speichert die Daten in einer Datenbank. Alle relevanten Mitarbeiter-Informationen werden dazu in der Tabelle `Employee` hinterlegt. Die Abbildung 2.4(a) zeigt einen Ausschnitt aus dem ER-Modell der Tabelle `Employee` mit dem Attribut `salary`. Das Attribut `salary` speichert sowohl die Höhe als auch die Währung des an einen Mitarbeiter ausgezahlten Gehalts. Das Attribut `salary` speichert somit zwei unterschiedliche Informationen. Diese Informationen sollen in separaten Attributen bzw. Spalten der Datenbank gespeichert werden, um die unterschiedlichen Informationen voneinander zu trennen. Dazu wird ein neues Attribut bzw. eine neue Spalte `currency` eingeführt (vgl. Abbildung 2.4(b)). Danach werden die Währungsinformationen aus der Spalte `salary` in die Spalte `currency` kopiert.

2.3.2 Mehrsprachiges Refactoring

Der in [Fow99, S. 53] definierte Begriff des Refactorings wird im Folgenden auf mehrsprachige Software-Projekte übertragen. Die Beschreibung eines Refactorings erfordert Kenntnisse über die spezifischen Eigenschaften der Programmiersprache, auf der das Refactoring angewendet werden soll[VSMD03;Tic01, S. 65]. Dies erschwert die Anwendung von Refactorings in mehrsprachigen Software-Anwendungen[CJ08]. Darüber hinaus werden in Software-Projekten nicht nur unterschiedliche Programmiersprachen verwendet.

Es existieren verschiedene weitere Dokumente, wie die Software-Dokumentation, Design-Dokumente, Anforderungskataloge und Tests[MT04]. In dieser Arbeit werden solche und andere Dokumente in Bezug auf [MT04] daher wie folgt bezeichnet:

Definition Dokumente, die zur technischen Umsetzung bzw. Implementierung einer Software-Anwendung dienen, werden als SOFTWARE-ARTEFAKTE bezeichnet.

Diese Definition umfasst ausdrücklich den Quellcode und andere Dokumente, wie sie z.B. bei mehrsprachigen Software-Projekten bzw. Polyglot Programming auftreten.

Für die Anwendung von Refactorings in mehrsprachigen Software-Anwendungen wird in dieser Arbeit der Begriff MEHRSPRACHIGES REFACTORING verwendet, der wie folgt definiert ist:

Definition Ein MEHRSPRACHIGES REFACTORING ist ein Prozess, der ein Refactoring auf ein Software-Artefakt eines Typs anwendet und ggf. weitere Refactorings auf Software-Artefakte eines anderen Typs anstößt, um die globale semantische Integrität der Software-Artefakte zu erhalten.

Mit dieser Definition sollen Refactorings, wie sie z.B. in [Opd92, Fow99] für Objekt-orientierte Software-Artefakte definiert wurden, von denen die in mehrsprachigen Software-Anwendungen eingesetzt werden können, unterschieden werden.

Für diese Arbeit ist der Begriff globale semantische Integrität wie folgt definiert:

Definition Der Begriff der GLOBALEN SEMANTISCHEN INTEGRITÄT beschreibt den Zustand in einer mehrsprachigen Software-Anwendung, der vor der Anwendung eines mehrsprachigen Refactorings innerhalb der Software-Anwendung geherrscht hat.

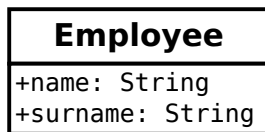
Das bedeutet, dass nach der Anwendung eines mehrsprachigen Refactorings die unterschiedlichen Software-Artefakte in der gleichen Weise miteinander interagieren, wie das vor dem Refactoring geschehen ist.

Die Definition des Begriffs mehrsprachiges Refactoring schließt explizit die Anwendung semantischer Modifikationen eines Software-Artefakts aus, um die globale semantische Integrität sicherzustellen. An dieser Stelle soll diskutiert werden, aus welchen Gründen der Ausschluss semantischer Modifikationen erfolgt. Dazu sei eine mehrsprachige Software-Anwendung angenommen, die aus einer Objekt-orientierten Anwendung A , einer Datenbank D und Unit Tests² U besteht. Die Komponenten A und D sowie D und U interagieren miteinander. Insbesondere überprüft U die Einhaltung der Software-Spezifikation³ von D . Es sei weiterhin angenommen, dass auf A ein Refactoring angewendet werden soll. Um die globale semantische Integrität der mehrsprachigen Software-Anwendung zu sichern, muss nach dem Refactoring auf A eine semantische Modifikation auf D vorgenommen werden. Da U die Semantik von D testet, muss auch U modifiziert werden. Da U die Semantik von D bezüglich einer Spezifikation überprüft,

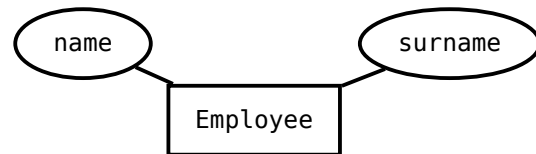
²Ein *Unit Test* überprüft einen isolierten Code-Abschnitt, genannt *Unit*, und vergleicht die Ergebnisse der Ausführung des Code-Abschnitts mit den erwarteten Ergebnissen[Ola03].

³Es wird die Definition nach [AP98, S. 6] gewählt. Danach ist eine Software-Spezifikation eine präzise Beschreibung von Objekten sowie von Methoden, die die Objekte manipulieren. Des Weiteren macht die Software-Spezifikation Aussagen über das Verhalten der Objekte, solange diese im Software-System existieren.

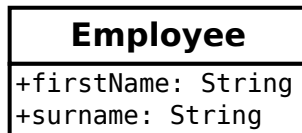
stimmen Tests und Spezifikation nach der Modifikation von U nicht mehr miteinander überein. Zur Herstellung der Konsistenz zwischen den Tests U und der Spezifikation müsste daher die Spezifikation geändert werden. Die Spezifikation bildet aber die Grundlage für die Implementierung der Software-Anwendung. Eine Änderung der Spezifikation muss daher zuvor validiert werden, um zu prüfen, ob die geänderte Spezifikation immer noch die Anforderungen an das zu entwickelnde Software-System ausdrückt [AP98, S. 7]. Refactorings dienen der strukturellen Veränderung einer Software, um z.B. Anforderungen einer geänderten Spezifikation in ein bereits bestehendes Design zu integrieren und nicht um die Spezifikation zu ändern. Die Definition mehrsprachiger Refactorings lässt daher keine semantischen Modifikationen zu, um die Integrität mit vorhandenen Spezifikationen sicherzustellen.



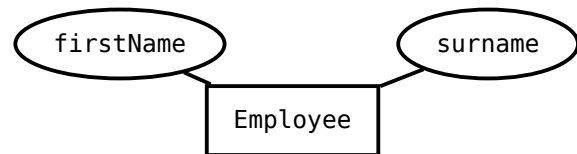
(a) Die Klasse `Employee` in ihrer ursprünglichen Definition. (Quelle: Eigene Darstellung)



(b) Die Tabelle `Employee` in ihrer ursprünglichen Definition. (Quelle: Eigene Darstellung)



(c) Die Klasse `Employee` nach dem `RENAME REFACTORING`. Das Attribut `name` wurde in `firstName` umbenannt. (Quelle: Eigene Darstellung)



(d) Die Tabelle `Employee` nach der Wiederherstellung des Mappings, durch Umbenennung des Attributs `name` in `firstName`. (Quelle: Eigene Darstellung)

Abbildung 2.5: Die Abbildung zeigt den Ablauf eines `RENAME REFACTORINGS` am Beispiel der Klasse `Employee` (a) und der dazugehörigen Datenbankrepräsentation (b). Nach der Durchführung des Refactorings muss das Attribut `name` sowohl in der Klasse `Employee` (c) wie auch in der Tabelle (d) umbenannt sein, um das Mapping zwischen der Klasse `Employee` und der Tabelle entsprechend sicherzustellen.

An einem Beispiel soll im Folgenden die Anwendung mehrsprachiger Refactorings verdeutlicht werden.

Die Software-Anwendung zur Verwaltung der Daten von Angestellten nutzt eine Datenbank, um die Daten der Mitarbeiter persistent zu speichern. Die Kommunikation mit der Datenbank erfolgt indirekt über ein *Objekt-relationales Mapping* (ORM). Ein Objekt-relationales Mapping schafft eine Verbindung zwischen den Objekt-orientierten Paradigma, wie es in OOP verwendet wird, und dem relationalen Paradigma, das in Datenbanken vorzugsweise anzutreffen ist [KS06, S. 5 ff.]. Dadurch ist es möglich, innerhalb einer Software-Anwendung Objekte zu verwenden, auch wenn die eigentlichen Daten in einer Datenbank gehalten werden. Die Abbildungen 2.5(a) und 2.5(b) zeigen die Klasse `Employee` und das Schema der zur Klasse gehörenden Tabelle in der Datenbank.

Das Attribut `name` der Klasse `Employee` soll in `firstname` umbenannt werden, um die Konsistenz in der Benennung mit dem Attribut `surname` herzustellen. Nach der erfolgten Umbenennung des Attributs besteht kein Mapping mehr zwischen der in Abbildung 2.5(c) dargestellten Klasse und dem Datenbank-Schema in Abbildung 2.5(b), da das Mapping über die Bezeichner erfolgt und nach dem Refactoring der Bezeichner des Attributs `firstname` nicht mit dem Bezeichner `name` der Tabellen-Spalte übereinstimmt. Zur Wiederherstellung des ORM ist eine äquivalente Umbenennung des Attributs `name` der Tabelle `Employee`, wie in Abbildung 2.5(d) dargestellt, notwendig. Ein mehrsprachiges Refactoring erkennt zunächst den Zusammenhang zwischen der Klasse, dem ORM und der Tabelle. Wird dann z.B. die Klasse durch ein mehrsprachiges Refactoring modifiziert, so stößt dieses Refactoring auch die Modifikation des ORM bzw. der Tabelle an. Dadurch wird die globale semantische Integrität zwischen den einzelnen Software-Artefakten gesichert.

2.4 Software Produktlinien

Eine *Software Produktlinie* (SPL) beschreibt eine Menge von Software-Anwendungen, die eine Menge von Funktionen teilen [PBL05, S. 20; Nor02; Bosch2002]. Diese Funktionen erfüllen die Anforderungen eines Markt-Segments, Anwendungsbereichs oder allgemein einer Domäne [PBL05, S. 20; Nor02].

Gemeinsame Funktionalität wird durch wiederverwendbare Software-Komponenten repräsentiert [PBL05, S. 27]. Neben wiederverwendbaren Software-Komponenten wird auch Variabilität innerhalb einer SPL beschrieben [PBL05, S. 62; BFG⁺02]. Variabilität ermöglicht die Entwicklung unterschiedlicher Software-Anwendungen aus der Menge der gemeinsamen Funktionen [PBL05, S.64]. *Variationspunkte* zeigen dabei Variabilität explizit an [PBL05, S. 62; BFG⁺02].

Zur Identifizierung der wiederverwendbaren Funktionalität und Variabilität dienen zwei Prozesse:

Domain Engineering: Dieser Prozess dient der Identifikation von wiederverwendbaren Kern-Komponenten und Variationspunkten [PBL05, S. 20].

Application Engineering: Dieser Prozess stellt Produkte aus den im Domain Engineering gefundenen Kern-Komponenten unter Zuhilfenahme der Variationspunkte zusammen [PBL05, S. 20-21].

Diese Prozesse werden auch unter dem Begriff *Software Produktlinien Engineering* (SPLE) zusammengefasst [PBL05, S. 20].

Das folgende Beispiel soll die Umsetzung und Funktion einer SPL erläutern. Die Software-Anwendung zur Verwaltung von Angestellten soll als eigenständiges Produkt anderen Unternehmen angeboten werden. Die Anforderungen unterscheiden sich hinsichtlich bestimmter Funktionen, benötigen aber grundsätzlich dieselbe Kern-Funktionalität. Um die Unterschiede und Gemeinsamkeiten in die Entwicklung der Software-Anwendung einfließen zu lassen, soll eine entsprechende SPL implementiert werden.

Die Ergebnisse des Domain-Engineering ergeben, dass alle Unternehmen eine Komponente zur allgemeinen Verwaltung von Angestellten benötigen. Die Unternehmen unterscheiden sich hinsichtlich der Art der Angestellten. Einige Unternehmen besitzen z.B.

keinen Vertrieb und müssen daher keine speziellen Daten von Verkäufern verwalten, wozu z.B. Kundenkontakte zählen. Außerdem wird in einigen Unternehmen den leitenden Angestellten kein Dienstwagen zur Verfügung gestellt.

Ausgehend von dem Domain-Engineering können verschiedenen Komponenten identifiziert werden. Dazu gehört zunächst eine Komponente zur Verwaltung von allgemeinen Daten der Angestellten. Eine weitere Komponente enthält Funktionen zur Verwaltung der Mitarbeiter des Vertriebs und ihrer Kundenkontakte. Schließlich enthält eine Komponente Funktionen, um Mitarbeiter und Dienstwagen zu verwalten. Die genannten Komponenten bilden die wiederverwendbaren Software-Komponenten der entsprechenden SPL. Die Erstellung einer Software-Anwendung kann nun durch Auswahl der benötigten Komponenten erfolgen, die zur Erfüllung der Anforderungen des jeweiligen Kunden benötigt werden.

2.5 Feature-orientierte Programmierung

Ein *Feature* ist nach [AK09] eine möglicherweise optionale Funktionalität eines Software-Systems, die eine bestimmte Anforderung erfüllt oder eine Design-Entscheidung repräsentiert. Software-Anwendungen können mit Hilfe von Features beschrieben und unterschieden werden [BSR03]. Konkrete Anwendung finden Features z.B. in SPLs, in denen sie zur Beschreibung der unterschiedlichen Software-Anwendungen einer SPL genutzt werden [AK09, RSP04, SKR⁺09].

Variabilität ist ein integraler Bestandteil innerhalb einer SPL, um unterschiedliche Software-Anwendungen aus der Menge der wiederverwendbaren Software-Komponenten beschreiben zu können [PBL05, S. 29]. In [Pre97] wird gezeigt, dass mit Hilfe von OOP Variabilität nicht befriedigend umgesetzt werden kann und Features als Lösung für diese Probleme vorgeschlagen.

In *Feature-orientierter Programmierung* (FOP) ist das Feature das zentrale Konzept zur Umsetzung von Wiederverwendbarkeit und Variabilität [Pre97, AK09]. Die Entwicklung einer Software-Anwendung erfolgt in FOP durch Zusammenstellung von Features [Pre97]. Dies wird auch als *Feature composition* (engl. Feature-Komposition) bezeichnet [BSR03].

Als Beispiel soll im Folgenden die SPL der Anwendung zur Verwaltung von Angestellten dienen. Durch das Domain-Engineering wurden zuvor bereits verschiedene Komponenten identifiziert, die Gemeinsamkeiten und Unterschiede zwischen den Anforderungen der einzelnen Unternehmen beschreiben. Diese Komponenten können auch als Feature beschrieben und mit FOP entsprechend implementiert werden.

Konkret können die Bestandteile eines Features z.B. in *Feature Modulen* implementiert werden [BSR03]. Feature Module enthalten *Refinements* (engl. Verfeinerungen) [BSR03]. Refinements beschreiben die konkreten Modifikationen und Erweiterungen, die bei der Anwendung eines Feature Modules durchgeführt werden sollen [BSR03]. Die durch ein Refinement beschriebenen Modifikationen beziehen sich dabei auf ein Basis-Artefakt [BSR03]. Ein Refinement auf ein Basis-Artefakt einer OO-Programmiersprache nutzt z.B. die Bezeichner von Klassen oder Methoden [BSR03]. Modifikationen und Erweiterungen werden dann durch Überschreiben oder Ergänzen vorhandener Entitäten realisiert [BSR03]. Bezogen auf ein Code-Artefakt bedeutet dies z.B.

die Erweiterung einer Klasse durch Hinzufügen von Attributen oder Methoden[BSR03].

2.6 Refactoring Feature Module

Die durch FOP bereitgestellten Funktionen erlauben es, ein Software-System aus Features zu konfigurieren, die den gestellten Anforderungen genügen[Bat03, AK09]. Die Möglichkeiten der Konfiguration beschränken sich dabei auf die:

Modifikation vorhandener Funktionen durch Erweitern oder Überschreiben der Entitäten eines Software-Artefakts.

Bezogen auf OOP kann dies z.B. durch Überschreiben von Methoden erreicht werden[Bat03].

Erweiterung des bestehenden Funktionsumfangs durch Hinzufügen neuer Entitäten.

In OOP ist dies gleichzusetzen mit dem Hinzufügen neuer Klassen oder Methoden[Bat03].

Zusammengefasst bedeutet dies, dass nur semantische Änderungen durch FOP ermöglicht werden.

Die Beschränkung auf reine semantische Erweiterungen beeinträchtigt die Wiederverwendung von Features bzw. Feature Modulen. In [KBA09] zeigen Kuhlemann, Batory und Apel, dass insbesondere die Unterstützung von strukturellen Modifikationen die Wiederverwendbarkeit von Features steigern kann.

Zur Darstellung des Problems wird sich in [KBA09] insbesondere auf die Anwendung von FOP zur Umsetzung von Software-Bibliotheken bezogen. Das Problem soll daher im Folgenden durch ein eigens gewähltes Beispiel verdeutlicht werden.

Die Software-Anwendung zur Verwaltung von Angestellten wurde in zwei Hauptkomponenten aufgeteilt. Eine Komponente implementiert eine Benutzeroberfläche, die der Darstellung und Bearbeitung des Datenbestandes dient. Die zweite Komponente abstrahiert den Zugriff auf den Datenbestand und sichert dessen Persistenz. Die Komponenten sind mit FOP implementiert und daher entsprechend den Anforderungen erweiterbar. Ein anderes Unternehmen möchte nun die zweite Komponente für seine eigene Software lizenzieren, um damit eine ältere Software-Bibliothek zu ersetzen, die nicht mehr den Anforderungen genügt. Eine direkte Integration der neuen Software-Bibliothek ist nicht möglich, da sich die Schnittstelle der neuen Software-Bibliothek von der Schnittstelle der alten Software-Bibliothek unterscheidet. Dies äußert sich z.B. durch unterschiedliche:

- Bezeichnungen für Klassen und Methoden,
- Methoden-Signaturen.

Der Umstieg auf die neue Software-Bibliothek ist zwingend erforderlich. Alternative Produkte mit einer möglicherweise besser zu integrierenden Schnittstelle stehen nicht zur Verfügung. Die Entwicklung einer eigenen Middleware, die die Kommunikation zwischen der Software-Anwendung und der Software-Bibliothek ermöglicht, wird aus Kostengründen ausgeschlossen. Daher sind nur noch folgende Varianten denkbar, um die Integration erfolgreich abzuschließen:

1. Änderung der Aufrufe innerhalb der bestehenden Software-Anwendung,
2. Anpassung der Schnittstelle der Software-Bibliothek an die Vorgaben der Software-Anwendung.

In beiden Varianten ist die strukturelle Modifikation einer der beteiligten Software-Komponenten notwendig.

In [KBA09] führen Kuhlemann, Batory und Apel daher *Refactoring Feature Module* (RFM) ein. RFMs enthalten Refactorings. Die Refactorings dienen der Modifikation der durch ein Feature Modul erweiterten Software-Komponente. Bezogen auf das oben genannte Beispiel bedeutet dies demnach die Modifikation der Software-Bibliothek.

Die Vorgehensweise wird wie folgt beschrieben. Zunächst wird die Basis-Software anhand der gewählten Features erweitert[KBA09]. Danach erfolgt die Anwendung der im RFM definierten Refactorings auf die durch Feature Komposition erstellte Software-Anwendung[KBA09].

Kapitel 3

Verwandte Arbeiten

In dem folgenden Kapitel werden verschiedene Arbeiten unterschiedlicher Forschungsbereiche betrachtet, die sich mit der Umsetzung mehrsprachiger Refactorings oder mit Teilproblemen der Umsetzung befassen.

Zunächst werden im Abschnitt 3.1 Arbeiten thematisiert, die den Aufbau von Referenzen zwischen verschiedenen Software-Artefakten einer mehrsprachigen Software-Anwendung betrachten. Darauf werden in Abschnitt 3.2 Arbeiten vorgestellt, die sich mit generischen und sprachunabhängigen Refactorings befassen. Coupled Software Transformations sind Modifikationen verschiedener Software-Artefakte und werden in Abschnitt 3.3 ebenfalls in Bezug zu mehrsprachigen Refactorings gesetzt. Abschließend werden in Abschnitt 3.4 Arbeiten betrachtet, die sich direkt mit der Umsetzung mehrsprachiger Refactorings befassen. Abschließend werden in Abschnitt 3.5 die Ergebnisse der vorgestellten Arbeiten zusammengefasst und in Beziehung zu den Zielen dieser Arbeit gebracht.

3.1 Programm-Verständnis und Analyse in mehrsprachigen Software-Anwendungen

Zur Umsetzung mehrsprachiger Refactorings ist es notwendig, Beziehungen zwischen verschiedenen Artefakt-Typen einer mehrsprachigen Software-Anwendung zu finden und darzustellen. In diesem Abschnitt werden Arbeiten aus den Gebieten *Program Comprehension* (engl. Program-Verständnis) und Programm-Analyse vorgestellt, die Lösungen zum Aufdecken und zur Darstellung von Beziehungen zwischen unterschiedlichen Artefakt-Typen einer mehrsprachigen Software-Anwendung aufzeigen.

Die Unterstützung des Software-Entwicklers durch die Darstellung der Abhängigkeiten zwischen den unterschiedlichen Artefakt-Typen einer Software-Anwendung führt zu einer gesteigerten Produktivität des Entwicklers[Lin95]. Die Software PolyCare findet Referenzen¹ zwischen Software-Artefakten² und stellt diese grafisch dar. Im Zusam-

¹Referenzen werden in [Lin95] auch als Programm-Abhängigkeiten bezeichnet.

²In [LBO03] wird ausgesagt, dass PolyCare nur Abhängigkeiten innerhalb von Sprachen verschiedener Paradigmen findet, wie z.B. C++ und Lisp. In [LLMM07] wird PolyCare hingegen im Zusammenhang mit X-develop genannt. X-develop ist in der Lage, Referenzen zwischen verschiedenen Software-Artefakten aufzubauen[SKL06].

menhang mit PolyCare werden die Sprachen C und Lisp genannt[Lin95]. Eine genaue Auflistung der unterstützten Programmiersprachen erfolgt nicht. Es fehlt weiterhin eine genaue Erläuterung der Architektur von PolyCare und eine Beschreibung, wie Referenzen zwischen den einzelnen Artefakten gefunden werden. Eine Recherche nach diesen Informationen blieb erfolglos.

In [KWDE98] wird ein Tool zur Wartung einer Legacy-Anwendung vorgestellt. Die Aufgabe des Tools ist es, die Abhängigkeiten zwischen den verschiedenen Artefakt-Typen der Legacy-Anwendung darzustellen. Auf Basis der gefundenen Abhängigkeiten soll der Software-Entwickler bei der Einarbeitung in das Legacy-System bzw. bei dessen Modifikation unterstützt werden. Das Software-System besteht aus verschiedenen Software-Artefakten, zu denen JCL, CSP, COBOL, PL/1, IMSDB, PSB sowie SQL DML und DDL gehören. Zunächst beschreiben die Autoren das konzeptionelle Modell, in denen die möglichen Beziehungen zwischen den unterschiedlichen Artefakt-Typen beschrieben werden. In diesem Modell ist z.B. dargestellt, dass über JCL Programme gestartet werden können. Programme können mit COBOL implementiert werden. Daraus ergibt sich, dass über JCL COBOL-Programme gestartet werden können. Über einen vier Schritte umfassenden Prozess werden aus dem Sourcecode der Software-Anwendung die Informationen entnommen, die für den Aufbau von Relationen zwischen den Code-Artefakten notwendig sind. Dazu gehören dann z.B. die aus JCL-Skripten heraus aufgerufenen und die per COBOL definierten Programme. Diese Informationen werden in einem Repository gespeichert. Mit Hilfe der speziellen Abfragesprache GReQL ist es möglich, spezifische Informationen aus den im Repository gespeicherten Daten zu extrahieren. Es können z.B. alle Programme abgefragt werden, die von JCL-Skripten gestartet werden. In [KWDE98] wird ein verkleinertes, als *Macro* bezeichnetes Modell gezeigt. Das Makro-Modell dient dazu einen Überblick über die Artefakte der Legacy-Anwendung zu gewinnen und besteht aus über 30 Entitäten und über 35 Beziehungen zwischen den Entitäten. Bei einer allgemeinen Betrachtung mehrsprachiger Refactorings, wie sie in dieser Arbeit erfolgt, müssen mehr als die im Makro-Modell dargestellten sieben Artefakt-Typen (SQL DML und DDL zusammengefasst) und deren spezifischen Beziehungen zueinander betrachtet werden. Daher ist eine allgemeine Darstellung der Beziehungen zwischen unterschiedlichen Artefakt-Typen durch ein Modell, wie es in [KWDE98] beschrieben wird, nicht zweckmäßig. Denn ein Modell, das alle möglichen Artefakt-Typen und deren Beziehungen zueinander beschreibt, würde weitaus mehr Entitäten und Beziehungen aufweisen als das in [KWDE98] vorgestellte konzeptionelle Modell. Ein entsprechendes Modell wäre daher zu komplex, um der Forderung nach Wartbarkeit und Erweiterbarkeit Rechnung zu tragen.

Die folgenden Ansätze beschreiben allgemeine Modelle, die von den Sprachspezifika einzelner Programmiersprachen abstrahieren und die Sicht auf die einzelnen Software-Artefakte verallgemeinert darstellen.

Moise und Wong nutzen das Open-Source-Tool *Source Navigator* (SN), um Referenzen zwischen verschiedenen Software-Artefakten aufzubauen[MW05]. Die Ergebnisse werden in der *Graph eXchange Language* (GXL) gespeichert, die auch als Austauschformat dient, um die Weiterverarbeitung der Ergebnisse durch andere Tools zu ermöglichen. Die Analyse des Sourcecode erfolgt für jede Sprache durch einen spezifischen *Extractor*. Die eigentliche Extraktion der Referenzen erfolgt in drei Schritten:

1. Extraktion und Speicherung der im Sourcecode gefundenen Fakten

2. Auffinden der Referenzen zwischen Fakten der selben Programmiersprache,
3. Auffinden der Referenzen zwischen Fakten unterschiedlicher Programmiersprachen.

Das beschriebene Vorgehen kann dazu genutzt werden, um Beziehungen zwischen unterschiedlichen Artefakt-Typen zu finden. Die Erweiterung des SN durch Moise und Wong kann nicht direkt für die Realisierung mehrsprachiger Refactorings genutzt werden, da über die gefundenen Referenzen keine direkte Modifikation des vom SN geparsten Sourcecodes möglich ist. Der Sourcecode der verschiedenen Artefakte muss erneut geparst werden, um Modifikationen zum Beispiel über einen *Abstract Syntax Tree* (engl. Abstrakter Syntax Baum, AST) zu ermöglichen.

In [SKL06] werden Referenzen zwischen unterschiedlichen Software-Artefakten durch ein *allgemeines Meta-Modell* dargestellt. Sprachspezifische Front-Ends extrahieren alle relevanten Entitäten aus den gegebenen Sourcecode. Relevant sind dabei alle Entitäten, die syntaktische und semantische Relationen beinhalten, die für die weitere Analyse von Bedeutung sind. Diese sprachspezifischen Entitäten werden über ein ebenso sprachspezifisches Mapping in die Entitäten des allgemeinen Modells überführt. Auch die relevanten Relationen zwischen den sprachspezifischen Entitäten werden auf das allgemeine Meta-Modell übertragen. Über das allgemeine Meta-Modell erfolgt dann die Verknüpfung von Entitäten unterschiedlicher Artefakt-Typen. Das in [SKL06] vorgestellte allgemeine Modell basiert auf dem *Dagstuhl Middle Metamodel* (DMM)[SLLL07]. DMM ist ein Meta-Modell zur Repräsentation Objekt-orientierter und nicht Objekt-orientierter Programmiersprachen[Let04]. Eine Modifikation des AST von Sourcecode über das DMM ist nicht möglich[Let04]. Des Weiteren fehlen entsprechende Tools, um unterschiedliche Artefakt-Typen in das DMM zu überführen. Daher wurde das DMM nicht weiter betrachtet.

3.2 Generische und sprachunabhängige Refactorings

Lämmel definiert in [Lä02] den Begriff *Generic Refactorings* (engl. generische Refactorings). Im Gegensatz zu Refactorings beschreiben Generic Refactorings allgemeine strukturelle, Semantik-erhaltende Modifikationen, die auf unterschiedlichen Programmiersprachen-Artefakten ausgeführt werden können. Als Beispiel für ein Generic Refactoring nennt Lämmel das ABSTRACT EXTRACTION REFACTORING. Das ABSTRACT EXTRACTION REFACTORING soll an einem konkreten Beispiel erläutert werden. In Listing 3.1 ist die Methode `addOneToList` dargestellt. Die einzige Aufgabe der Methode ist es, die Elemente der als Parameter übergebenen Liste `list` in Zeile 6 zu inkrementieren. Vor und nach dem Inkrementieren werden alle Listen-Elemente einmal ausgegeben (Zeile 2 bzw. 10). Der Code zur Ausgabe der Listen-Elemente kann nicht wiederverwendet werden, da der Code Bestandteil der Methode `addOneToList` ist. Nach [Lä02] ist der Code *anonym*. Damit der Code wiederverwendet kann, muss dieser in eine Methode ausgelagert werden. Das Ergebnis der Extraktion zeigt Listing 3.2. In den Zeilen 2 bzw. 8 wird die in Zeile 11 definierte Methode `printListElements` aufgerufen. Die Methode `printListElements` implementiert nur den zuvor anonymen Code. Da laut Lämmel das ABSTRACT EXTRACTION REFACTORING auf *jede Programmiersprache* anwendbar ist, handelt es sich um ein generisches Refactoring. Die Anwendung von

Listing 3.1: Inkrementieren von Listen-Elementen vor dem Refactoring

```

1  private static void addOneToList(int
2     [] list) {
3     for (int i : list) {
4         System.out.println(i);
5     }
6     for (int x = 0; x < list.length; x
7         ++){
8         ++list[x];
9     }
10    for (int i : list) {
11        System.out.println(i);
12    }
13 }

```

Listing 3.2: Inkrementieren von Listen-Elementen nach dem Refactoring

```

1  private static void addOneToList(int
2     [] list) {
3     printListElements(list);
4     for (int x = 0; x < list.length; x
5         ++){
6         ++list[x];
7     }
8     printListElements(list);
9 }
10
11 private static void printListElements
12     (int[] list) {
13     for (int i : list) {
14         System.out.println(i);
15     }

```

generischen Refactorings wird von Lämmel explizit auf Programmiersprachen-Artefakte beschränkt. Daher ist eine Nutzung von generischen Refactorings in mehrsprachigen Software-Projekten im Zusammenhang mit mehrsprachigen Refactorings nicht möglich. Denn mehrsprachige Refactorings betrachten alle Software-Artefakte einer mehrsprachigen Software-Anwendung (vgl. Abschnitt 2.3.2).

Weitere Ansätze zur allgemeinen Realisierung von Refactorings beschreiben die im Folgenden vorgestellten Arbeiten. In [DLT00] wird das Refactoring-Framework MOOSE beschrieben. MOOSE nutzt zur abstrakten Darstellung der unterschiedlichen Software-Artefakte FAMIX. FAMIX ist ein Meta-Modell zur abstrakten Darstellung Objekt-orientierter Sprachen[Tic01]. Basierend auf FAMIX können Refactorings beschrieben werden. Über sprachspezifische Front-Ends werden Refactorings am eigentlichen Sourcecode umgesetzt. Die Idee, Refactorings unabhängig von der spezifischen Programmiersprache zu beschreiben, wird auch in den Arbeiten von López et. al. in [NSCF06] und Marticorena in [Mar05] auf Basis des Meta-Modells MOON beschrieben. Sowohl FAMIX wie auch MOON stellen Abstraktionen für Objekt-orientierte Programmiersprachen dar[Tic01, NSCF06]. In der vorliegenden Arbeit wurde festgestellt, dass mehrsprachige Software-Anwendungen weder zwingend aus Artefakten Objekt-orientierter Programmiersprachen bestehen noch wird das Vorhandensein Objekt-orientierter Code-Artefakte vorausgesetzt (vgl. Abschnitt 2.1). Eine Erweiterung der gegebenen Modelle wurde aufgrund der expliziten Ausrichtung auf Objekt-orientierte Programmiersprachen-Artefakte nicht in Betracht gezogen. Des Weiteren muss die Zweckmäßigkeit einer solchen Erweiterung in Bezug auf die in Abschnitt 3.1 diskutierten Ergebnisse aus [KWDE98] hinterfragt werden, denn die Erweiterung der Modelle würde deren Komplexität steigern.

3.3 Coupled Software Transformations

Coupled Software Transformations (engl. Gekoppelte Software Transformationen) betrachten die Konsistenz-Erhaltung zwischen unterschiedlichen Software-Artefakten[Lä04]. Coupled Transformations schließen im Gegensatz zu mehrsprachigen

Refactorings semantische Modifikationen nicht aus[Lä04]. In [BCPV07] wird die Anwendung von Coupled Transformations zur Modifikation von XML- und Datenbank-Schemas sowie der entsprechenden XML-Dokumente und Datenbanken gezeigt. Die Transformationen werden auf *Typen* vorgenommen. Ein Typ stellt entweder einen konkreten Datentyp, wie Ganzzahl, oder eine Datenstruktur dar[COV06]. Über spezifische Frontends werden die konkreten XML- und SQL-Schemas und -Daten in die Typ-Darstellung überführt und wieder aus der Typ-Darstellung zurückgeschrieben[BCPV07]. Anhand eines Beispiels wurde die Transformation von XML- und SQL-Schemas und der entsprechend Daten beschrieben. Dabei wird das SQL-Schema aus dem transformierten XML-Schema abgeleitet[BCPV07]. D.h., es wird eine Transformation auf das XML-Schema angewendet und dann das resultierende XML-Schema in ein SQL-Schema umgewandelt. Es wird also keine Transformation auf das SQL-Schema direkt angewendet. Dies ist möglich, da ein direktes Mapping zwischen XML und SQL hergestellt werden kann. In der vorliegenden Arbeit wird gezeigt, dass beispielsweise zwischen Java-Klassen und Datenbank-Tabellen, kein direktes Mapping angenommen werden kann. Des Weiteren werden die in [COV06] vorgestellten Typen ausschließlich zur Darstellung von Schemas oder Datenstrukturen aus SQL- und XML-Artefakten verwendet. In mehrsprachigen Software-Anwendungen müssen aber weitere Artefakt-Typen, wie Sourcecode und Design-Dokumente, betrachtet werden.

In [CHH05] wird beschrieben, wie die gekoppelte Transformation eines Datenbank-Schemas, der im Datenbank-Schema dargestellten Daten und einer Software-Anwendung mit Hilfe der Tools DB-Main und ASF+SDF realisiert wurde. Aus der Beschreibung kann nicht entnommen werden, wie sich die Transformation des Datenbank-Schemas auf den Sourcecode der Software-Anwendung auswirkt. Außerdem wird nicht beschrieben, wie Artefakte der Software-Anwendung mit den Artefakten des Datenbank-Schemas in Beziehung gebracht werden.

Die semantische Modifikation einzelner Artefakte, wie sie durch Coupled Software Transformations möglich sind, kann zu Konflikten mit vorhandenen Software-Spezifikationen führen. Daher werden semantische Modifikationen im Zusammenhang mit mehrsprachigen Refactorings ausgeschlossen (vgl. Abschnitt 2.3.2). Deshalb wird das Konzept der Coupled Software Transformations für die Umsetzung mehrsprachiger Refactorings nicht weiter in dieser Arbeit betrachtet.

3.4 Mehrsprachige Refactorings

Das Refactoring von UML-Diagrammen ist Inhalt der Arbeit [SPTJ01]. Dabei werden Refactorings für Klassen- und Zustandsdiagramme definiert und deren gegenseitiger Einfluss beschrieben. Die Beziehung zwischen den einzelnen Diagramm-Typen ist dabei explizit durch das UML-Meta-Modell gegeben. Es wird allerdings gezeigt, dass die durch das Meta-Modell beschriebenen Beziehungen nicht ausreichen, um bestimmte Refactorings zu realisieren. In [VSMD03] wird gezeigt, wie UML derart erweitert werden kann, um das konsistente Refactoring von UML und den durch UML beschriebenen Sourcecode zu ermöglichen. Dabei bleibt der Fokus auf dem Refactoring Objekt-orientierter Programmiersprachen. Eine konkrete Implementierung oder eine Beschreibung, wie eine Beziehung zwischen UML und Sourcecode dargestellt wird, liegt nicht im Fokus der Arbeit. Des Weiteren ist nicht ersichtlich, wie sich z.B. Änderungen an einem UML-

Zustandsdiagramm auf die damit beschriebenen Sourcecode-Artefakte auswirken.

In [SKL06] wird die IDE X-develop³ vorgestellt, die der Analyse mehrsprachiger Software-Anwendungen und Umsetzung der mehrsprachiger Refactorings dient. X-develop unterstützt z.B. C#, Java, XML, ASP und Javascript. Neben dem RENAME METHOD REFACTORING [Fow99, S. 273] unterstützt X-develop das Refactoring von Methoden-Signaturen und das MOVE CLASS REFACTORING⁴. Die Refactorings sind auf Objekt-orientierte Aspekte der unterstützten Software-Artefakte definiert. Die Autoren beschreiben konkret die Anwendung des RENAME METHOD REFACTORINGS anhand des allgemeinen Modells, beschreiben dabei aber nicht die Modifikationen an den unterschiedlichen Software-Artefakten. Des Weiteren werden keine Refactorings von nicht Objekt-orientierten Artefakten erläutert, wie z.B. XML. In [SKL06] wird die Erweiterbarkeit des allgemeinen Modells hervorgehoben. Im Abschnitt 3.1 wurde im Zusammenhang mit der Arbeit [KWDE98] argumentiert, dass eine Erweiterung der Darstellung der spezifischen Beziehungen zwischen unterschiedlichen Artefakt-Typen für die Realisierung einer allgemeinen Umsetzung mehrsprachiger Refactorings nicht unbedingt zweckmäßig ist.

Verschiedene Arbeiten zeigen die Umsetzung mehrsprachiger Software-Refactorings auf Basis der Eclipse IDE. Chen und Johnson stellen in [CJ08] die Umsetzung eines RENAME CLASS REFACTORINGS zwischen Java, XML und JSP vor. Die XML-Artefakte bestehen konkret aus Konfigurationsdateien der Frameworks Struts, Hibernate und Spring. Das bedeutet, dass das RENAME CLASS REFACTORING konkret zwischen Java und den Frameworks Struts, Hibernate und Spring automatisiert umgesetzt wurde. Die Umbenennung einer Entität im Java-Artefakt wird erkannt und automatisiert auf die XML- und JSP-Dateien übertragen, die die umbenannte Java-Entität referenzieren. Chen und Johnson fassen dieses Prinzip als *detection* (engl. Erkennen) und *propagation* (engl. Weiterleiten) zusammen. Es werden also Änderungen an einem Software-Artefakt erkannt und an andere Software-Artefakte weitergeleitet. In [KKKS08] wird die Umsetzung eines RENAME REFACTORINGS zwischen Java- und Groovy-Artefakten vorgestellt. Die Arbeit [TTS⁺08] zeigt die Realisierung der RENAME CLASS und RENAME FIELD REFACTORINGS zwischen Java und JPA-Abfragen (Java Persistence API). Mit JPA können Mappings zwischen Java-Klassen und Datenbank-Elementen hergestellt werden. Mit Hilfe des Mappings liefern Anfragen über JPA direkt Instanzen von Java-Klassen. Tatlock et. al. zeigen, wie nach einem auf dem Java-Artefakt durchgeführten Refactoring die JPA-Abfrage analysiert und angepasst werden kann, um das Refactoring des Java-Artefakts entsprechend in der JPA-Abfrage abzubilden. [CJ08], [KKKS08] und [TTS⁺08] zeigen die konkrete Umsetzung von mehrsprachigen Refactorings zwischen zwei bzw. drei Software-Artefakten. Dabei wird insbesondere nur der Fall betrachtet bzw. realisiert, bei dem ein Refactoring auf ein Java-Element gestartet und die globale semantische Integrität durch eine Modifikation in weiteren Artefakten wiederhergestellt wird. Eine Realisierung von Refactorings in umgekehrter Reihenfolge wurde nicht betrachtet [TTS⁺08] oder aufgrund technischer Restriktionen von Eclipse nicht realisiert [CJ08, KKKS08].

³X-develop ist von der Firma Omnicore unter dem Namen CodeGuide verfügbar. <http://www.omnicore.com/de/codeguide.htm>

⁴Das in X-develop umgesetzte MOVE CLASS REFACTORING modifiziert `package-` bzw. `namespace-` Deklarationen in Java bzw. C#. Dieses Refactoring ist daher nicht zu verwechseln mit dem gleichnamigen, von Opdyke in [Opd92, S. 53] definierten Refactoring.

3.5 Zusammenfassung

Die vorgestellten Arbeiten und Tools zeigen insbesondere zwei Probleme auf, die der allgemeinen Realisierung automatischer Refactorings entgegen stehen. Zum einen sind die bisherigen Meta-Modelle nicht für die Darstellung von Beziehungen zwischen unterschiedlichen Artefakt-Typen geeignet, da die Meta-Modelle:

1. explizit auf die Darstellung von Artefakten eines Typs, wie Sourcecode Objekt-orientierte Programmiersprachen, ausgerichtet sind,
2. durch Erweiterung der unterstützten Artefakt-Typen in ihrer Komplexität zunehmen und damit ihre Zweckmäßigkeit zur Darstellung von Beziehung zwischen Artefakten einer mehrsprachigen Software-Anwendung hinterfragt werden muss.

Es ist daher sinnvoll ein Meta-Modell zu entwickeln, das:

1. zur Darstellung von Beziehungen zwischen unterschiedlichen Artefakt-Typen geeignet ist,
2. das erweiterbar ist,
3. eine beschränkte Komplexität besitzt.

In Kapitel 5 wird ein entsprechendes Meta-Modell vorgestellt, das zur prototypischen Realisierung von mehrsprachigen Refactorings angewendet wird.

Zum anderen konzentrieren sich bisherige Ansätze zur Realisierung mehrsprachiger Refactorings auf die Darstellung von Objekt-orientierten Refactorings bzw. von Refactorings auf Objekt-orientierten Artefakten unterschiedlichen Typs, wie zwei unterschiedliche Objekt-orientierte Programmiersprachen. Spezifische Refactorings anderer Artefakte, wie XML, Groovy oder ASP, werden unterdessen nicht betrachtet. Allgemein fehlt es an einer Auflistung möglicher Eigenheiten bezüglich der Wiederherstellung der globalen semantischen Integrität einer mehrsprachigen Software-Anwendung, nachdem ein Refactoring eines Artefakts durchgeführt wurde. Daher werden im Kapitel 4 zunächst Eigenschaften und Probleme bei der händischen Wiederherstellung der globalen semantischen Integrität nach dem Refactoring eines Artefakts betrachtet. Dabei werden insbesondere Artefakt-Typen in die Betrachtung einbezogen, die sich hinsichtlich der verwendeten Paradigmen voneinander unterscheiden. Damit wird insbesondere der Diversität mehrsprachiger Software-Anwendungen Rechnung getragen.

Kapitel 4

Realisierung mehrsprachiger Refactorings

Dieses Kapitel beschreibt die Umsetzung mehrsprachiger Refactorings anhand einer Beispiel-Anwendung. Dabei sollen Eigenschaften mehrsprachiger Refactorings gefunden werden, die bei der Realisierung automatischer mehrsprachiger Refactorings zu beachten sind. Die Objekt-orientierten Refactorings wurden aus [Opd92] bzw. [Fow99] gewählt. Die Datenbank-Refactorings für die SQL-Artefakte wurden aus [Amb03] entnommen. Das Clojure-Artefakt wurde mit Refactorings aus [Li06] modifiziert. Die Auswahlkriterien werden in den entsprechenden Abschnitten beschrieben. Alle in diesem Kapitel genannten Refactorings wurden manuell umgesetzt. Im Kapitel 5 wird basierend auf den Ergebnissen dieses Kapitels die automatisierte Umsetzung mehrsprachiger Refactorings erläutert.

Dieses Kapitel ist wie folgt gegliedert. Zunächst erfolgt in Abschnitt 4.1 die Beschreibung der Beispiel-Anwendung. Darauf folgt in Abschnitt 4.2 die Beschreibung der Probleme, die während der Umsetzung mehrsprachiger Refactorings bzgl. der Beispiel-Anwendung auftreten. Den Abschluss bildet die Zusammenfassung der Ergebnisse dieses Kapitels in Abschnitt 4.3.

4.1 Beschreibung der Beispiel-Anwendung *HRManager*

Die Beispiel-Anwendung *HRManager* ist eine eigens für die vorliegende Arbeit entwickelte Software-Applikation. Die Anwendung dient ausschließlich der Umsetzung mehrsprachiger Refactorings. Daher wurde bei der Implementierung der Anwendung insbesondere darauf geachtet, dass die Code-Basis der Anwendung möglichst klein und verständlich ist, damit die Umsetzung der Refactorings nachvollziehbar bleibt. Außerdem werden unterschiedliche Software-Artefakte innerhalb der Beispiel-Anwendung eingesetzt, um Bedingungen einer mehrsprachigen Software-Anwendung nachzubilden.

Insgesamt werden in der Anwendung vier unterschiedliche Software-Artefakte verwendet:

- Java,

- Hibernate,
- SQL sowie
- Clojure.

Ohne die benötigten Bibliotheken mit einzuberechnen, besteht die Anwendung insgesamt aus 10 Java-Klassen (558 LOC), 25 SQL-Dateien (100 LOC), einer Clojure-Datei (49 LOC) und einer XML-Konfigurationsdatei (21 LOC) für Hibernate¹.

Im Folgenden soll erläutert werden, unter welche Gesichtspunkten die Auswahl der Software-Artefakte erfolgte. Eine genaue Beschreibung wichtiger Funktionen erfolgt in den entsprechenden Abschnitten. Java wurde für die Umsetzung der Beispiel-Anwendung aus verschiedenen Gründen gewählt. Zum einen zählt Java zu den etablierten Objektorientierten Programmiersprachen, auf deren Basis eine Vielzahl von Bibliotheken und Tools zur Verfügung stehen. Zum anderen finden sich unter den für Java zur Verfügung stehenden Tools auch technisch ausgereifte ASTs wie JastAddJ[EH07], über die in Kapitel 5 die Realisierung der Java-Refactorings erfolgt. Datenbanken sind Bestandteil einer Vielzahl von Anwendungen. Als Standard-Sprache für die Kommunikation mit Datenbanken hat sich SQL etabliert[OP04], weshalb SQL ebenfalls in der Beispiel-Anwendung genutzt wird. Die Kommunikation zwischen Java und der Datenbank erfolgt über Hibernate². Hibernate ist Bestandteil verschiedener Software-Produkte, zu denen der JBoss Application Server³, das Rich Internet Application (RIA) Framework Seam⁴ und das Application Framework Spring⁵ gehören. Zur dynamischen Anpassung der Anwendungslogik des HRManagers wird Clojure⁶ eingesetzt. Mit Clojure steht eine LISP-basierte, rein funktionale Programmiersprache zur Verfügung, die sich sowohl in Java integrieren lässt als auch auf in Java definierte Funktionalität zugreifen kann[Van10, S. 1-3]. Mit der Verwendung von Clojure wird der steigenden Popularität funktionaler Programmiersprachen Rechnung getragen[Fis10, SG10].

Die Anwendung implementiert Funktionen zur Verwaltung der Belegschaft eines Unternehmens. Die Anwendung ist dazu in vier Bereiche unterteilt, die jeweils einen anderen Aspekt der Applikation kapseln. Die folgende Auflistung enthält die Bereiche sowie die Artefakte, mit denen die Bereiche umgesetzt wurden:

- Anwendungslogik (Java),
- Persistenz (SQL),
- Objekt-relationales Mapping (Hibernate),
- Scripting (Clojure).

Im Folgenden sollen die vier Bereiche und die damit verbundenen Artefakte der Software-Anwendung im Detail erläutert werden.

¹Die Ergebnisse für Java, SQL und Clojure wurden mit dem Tool `sloccount` ermittelt. <http://www.dwheeler.com/sloccount/>

²<http://www.hibernate.org/>

³<http://www.jboss.org/jbossas>

⁴<http://seamframework.org/>

⁵<http://www.springsource.org/>

⁶<http://clojure.org/>

4.1.1 Anwendungslogik

Die Umsetzung der Anwendungslogik erfolgt mit Hilfe der Objekt-orientierten Programmiersprache Java. Die Entitäten, die in der Anwendungslogik Verwendung finden, werden durch sieben Klassen repräsentiert. Die Klassen und die dazugehörigen Entitäten der realen Welt sind in folgender Auflistung aufgeführt:

- **Customer** (Kunde)
- **Department** (Abteilung)
- **Employee** (Angestellter)
- **ExternalStaff** (Externer Angestellter)
- **Janitor** (Hausmeister)
- **Manager** (leitender Angestellter)
- **Salesperson** (Verkäufer)

Zur Abstraktion der Belegschaft dient eine Klassenhierarchie, die die folgenden Klassen enthält:

- **Employee**
- **Manager**
- **Salesperson**

Die Klasse **Employee** abstrahiert die Informationen, die allen Angestellten eigen sind. Dazu zählen der Vor- und Nachname, das Gehalt und die Abteilung des Angestellten, die durch die privaten Attribute **name**, **surname**, **salary** und **department** repräsentiert werden. Darüber hinaus besitzt die Klasse **Employee** das private Attribut **id**, das der eindeutigen Identifizierung eines Angestellten dient. In den Klassen wird der Zugriff auf private Attribute über Getter- und Setter-Methoden geregelt.

Die Klasse **Employee** dient des Weiteren als Ausgangspunkt zur Ableitung der spezialisierten Klassen **Manager** und **Salesperson**. Die Klasse **Manager** enthält die Attribute **account**, **companyCarLicensePlate** und **boss**. Das Attribut **account** speichert den Namen der Kostenstelle, die einem leitenden Angestellte zugeordnet ist. Im Attribut **companyCarLicensePlate** wird das Kennzeichen des Dienstwagens gespeichert, der dem leitenden Angestellten zur Verfügung gestellt wird. Ist der leitende Angestellte einem Vorgesetzten unterstellt, wird eine Referenz auf diesen Vorgesetzten im Attribut **boss** gespeichert. Die von der Klasse **Manager** abgeleitete Klasse **Salesperson** definiert das Attribut **takesCareOf**, das Referenzen auf Datensätze von Kunden enthält. Die Referenzen stellen eine Beziehung zwischen einem Verkäufer und den von dem Verkäufer betreuten Kunden her. Kunden werden durch die Klasse **Customer** repräsentiert. Die Abbildung 4.1 zeigt die von der Klasse **Employee** ausgehende Klassenhierarchie, sowie die Attribute und Methoden der einzelnen Klassen.

In einer weiteren Klassenhierarchie werden externe Mitarbeiter abstrahiert. Den Ausgangspunkt bildet dabei die abstrakte Klasse **ExternalStaff**, die alle gemeinsamen Informationen externer Mitarbeiter kapselt. Die Klasse **ExternalStaff** enthält bis auf **department** die gleichen Attribute wie die Klasse **Employee**, die sich auch hinsichtlich ihrer Bedeutung nicht unterscheiden. Zusätzlich definiert **ExternalStaff** das Attribut

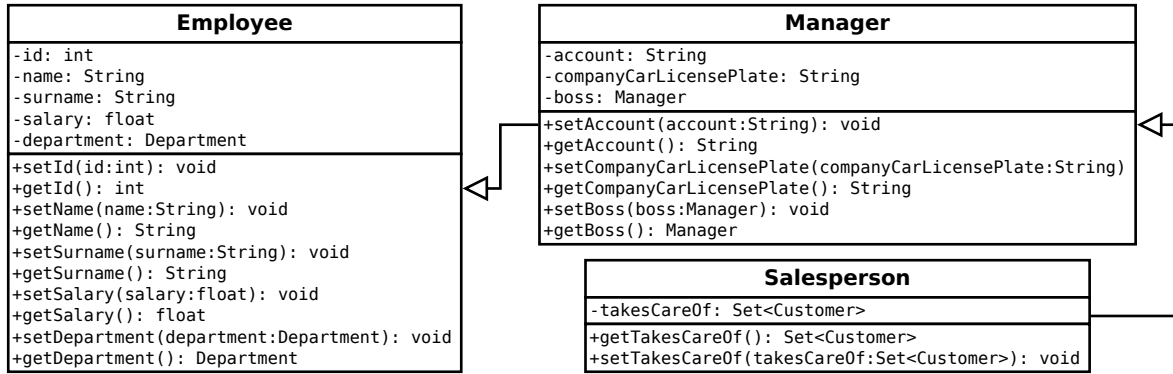


Abbildung 4.1: Klassenhierarchie ausgehend von der Oberklasse `Employee` (Quelle: Eigene Darstellung)

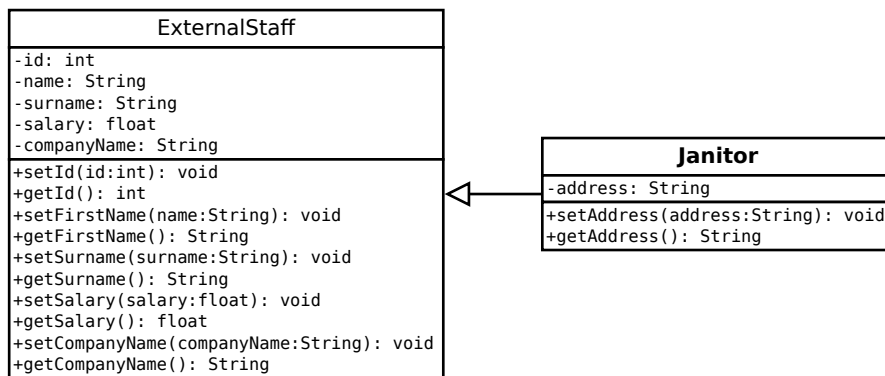
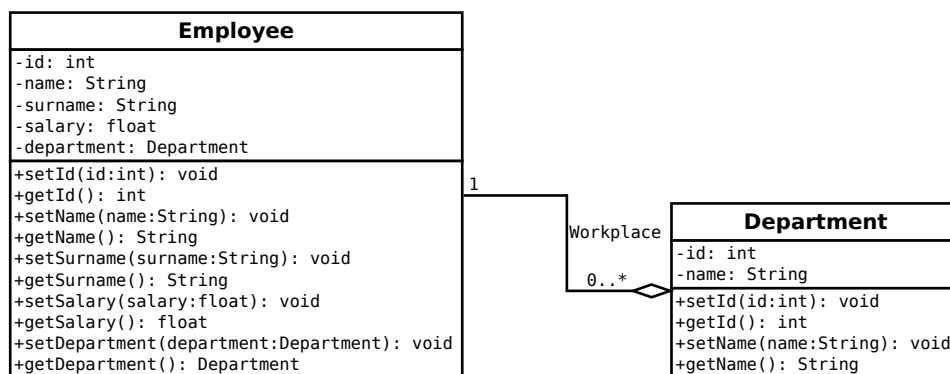


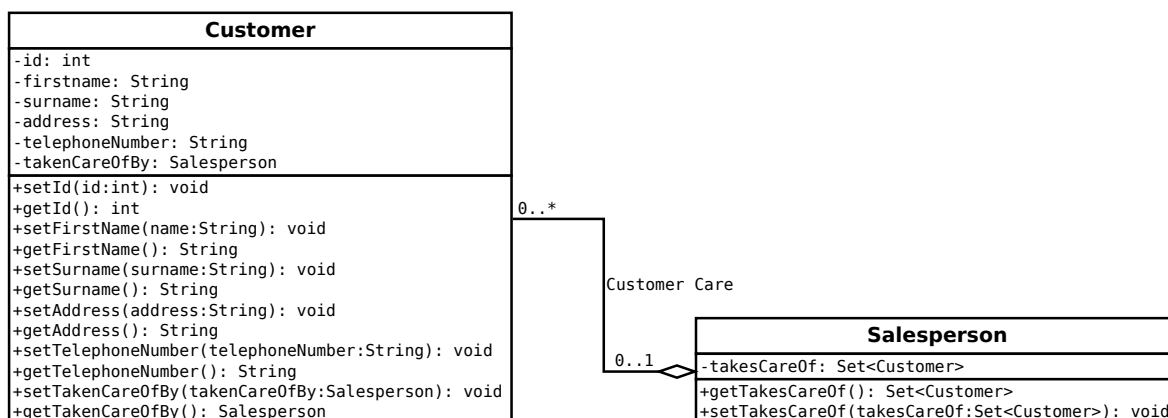
Abbildung 4.2: Klassenhierarchie ausgehend von der Oberklasse `ExternalStaff` (Quelle: Eigene Darstellung)

`companyName`, das den Namen der Firma enthält, von dem der externe Mitarbeiter entliehen ist. Es existiert in der Beispiel-Anwendung die Klasse `Janitor` als konkrete Implementierung der Klasse `ExternalStaff`. Die Klasse `Janitor` definiert das Attribut `address`, das die Adresse des Gebäudes enthält, das von dem jeweiligen Hausmeister betreut wird. Die Abbildung 4.2 stellt die Klassen `ExternalStaff` und `Janitor`, deren Beziehung zueinander sowie die definierten Attribute und Methoden dar.

Des Weiteren existieren die Klassen `Department` und `Customer`. Die Klasse `Department` kapselt den Namen einer Abteilung im Attribut `name`, die im `HRManager` als einzig relevante Informationen benötigt wird. `Department` wird von der Klasse `Employee` verwendet, um einen Angestellten mit einer konkreten Abteilung in Verbindung zu bringen. Die Abbildung 4.3(a) stellt die Klasse `Department` mit den dort definierten Attributen und Methoden und die Beziehung zur Klasse `Employee` grafisch dar. Die Klasse `Customer` enthält alle wichtigen Informationen bzgl. eines Kunden des Unternehmens. Darüber hinaus wird in der Klasse `Customer` im Attribut `takenCareOfBy` vermerkt, welcher Verkäufer den jeweiligen Kunden betreut. Ein Verkäufer kann mehrere oder gar keinen Kunden betreuen. Ein Kunde kann nur von höchstens einem Verkäufer betreut werden. In 4.3(b) werden die Klassen `Customer` und `Salesperson` mit ihren Attributen und Methoden sowie die Beziehung zwischen den Klassen grafisch dargestellt.



(a) Die Klasse `Department` und ihre Relation zur Klasse `Employee`
(Quelle: Eigene Darstellung)



(b) Die Klasse `Customer` und ihre Relation zur Klasse `Salesperson` (Quelle: Eigene Darstellung)

Abbildung 4.3: Die Abbildungen (a) und (b) zeigen die Relationen zwischen den Klassen `Employee` und `Department` sowie `Salesperson` und `Customer`, wie sie in der Beispiel-Anwendung umgesetzt sind.

4.1.2 Persistenz

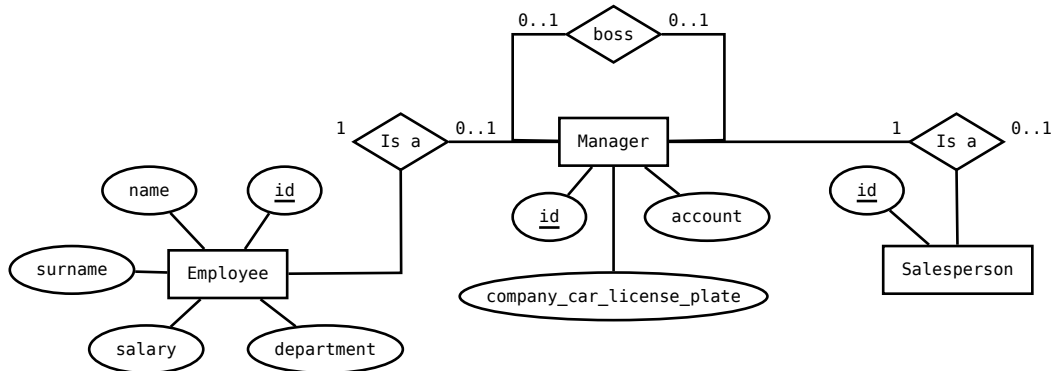
Zur Sicherung aller wichtigen Informationen wird eine SQL-Datenbank eingesetzt⁷. Der Aufbau der benötigten Tabellen erfolgt über `CREATE TABLE`-Statements.

Grundsätzlich existiert für jede im Abschnitt *Anwendungslogik* definierte Klasse eine eigene Tabelle. Zur Repräsentation der Klassenhierarchien werden jedoch zwei unterschiedliche Definitionen der dazugehörigen Tabellen verwendet. Diese unterschiedlichen Repräsentation werden betrachtet, um ihren Einfluß auf die Umsetzung mehrsprachiger Refactorings zu untersuchen.

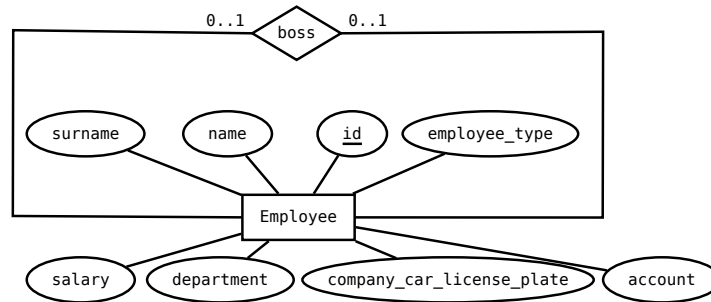
Die Repräsentation der Klassenhierarchien unterscheiden sich hinsichtlich der Darstellung der Unterklassen, wobei entweder eine Tabelle pro:

1. Unterklasse [KS06, S. 304] oder
2. Klassenhierarchie [KS06, S. 307]

⁷Als SQL-Datenbank wird konkret SQLite eingesetzt. SQLite hat den Vorteil gegenüber Datenbanken wie MySQL oder MSSQL, dass keine separate Datenbank-Installation erfolgen muss, womit sich der Aufwand zur Realisierung der Anwendung einschränken lässt. www.sqlite.org



(a) Repräsentation der Klassenhierarchie durch mehrere Tabellen pro Unterklasse.
(Quelle: Eigene Darstellung)



(b) Repräsentation der Klassenhierarchie durch eine einzelnen Tabelle. Die Unterscheidung welche Zeile welche konkrete Klasse repräsentiert, erfolgt durch die Spalte `employee_type`.
(Quelle: Eigene Darstellung)

Abbildung 4.4: Die ER-Modelle (a) und (b) zeigen unterschiedliche Realisierungen der `Employee`-Klassenhierarchie aus Abbildung 4.1 als Datenbank-Schema.

verwendet wird⁸. Der Vorteil der ersten Darstellungsweise, wie sie die Abbildung 4.4(a) dargestellt ist, liegt in der Vermeidung von Daten-Redundanz. Durch die Verwendung einer Tabelle pro Unterklasse ist es außerdem einfacher, die Klassenhierarchie in der Tabellen-Struktur wieder zu erkennen. Es werden allerdings u.U. mehr *Joins* zwischen den Tabellen benötigt, um zusammenhängende Informationen miteinander zu verknüpfen. Bei der zweiten Herangehensweise, wie sie die Abbildung 4.4(b) zeigt, werden alle Informationen einer Klassenhierarchie in einer Tabelle gespeichert. Dadurch sind keine *Joins* notwendig, um an die Informationen spezieller Unterklassen zu gelangen. Da alle Attribute der Klassenhierarchie in einer Tabelle vorliegen, ist nicht sofort ersichtlich, welches Attribut welcher Unterklasse zuzuordnen ist.

Des Weiteren werden `INSERT`- und `UPDATE`-Statements verwendet, um die Datenbank mit Test-Informationen zu füllen. Es soll untersucht werden, welchen Einfluß mehrsprachige Software-Refactorings auf die Definition dieser Statements ausüben. Das SQL-Statement in Listing 4.1 fügt einen Eintrag mit der Bezeichnung `IT` in die Tabelle `departments` ein. Das Listing 4.2 zeigt, wie eine Referenz vom Datensatz des Managers *Müller* auf den Datensatz der Managerin *Langenhahn* aufgebaut wird. Dazu wird

⁸In [KS06] werden auf Seite 251 und 253 weitere Mapping-Strategien beschrieben. Auf diese Strategien wird im weiteren Verlauf der Arbeit nicht eingegangen.

Listing 4.1: Anlegen der Abteilung IT in der Tabelle departments

```
1 INSERT INTO departments (name)
2   VALUES('IT');
```

Listing 4.2: Herstellen einer Referenz zwischen den Datensätzen der Managerin Langenhahn und des Managers Müller

```
1 UPDATE managers
2   SET boss = (SELECT id
3               FROM employees
4               WHERE surname = 'Langenhahn')
5 WHERE (SELECT id
6        FROM employees
7        WHERE employees.surname = 'Müller'
8        AND employees.id = managers.id);
```

über das `SELECT`-Statement in Zeile 5 der Datensatz des Managers *Müller* gewählt. Mit dem `SELECT`-Statement in Zeile 2 wird der Wert der Spalte `id` des Datensatzes der Managerin *Langenhahn* gewählt. Der Wert der Spalte `id` wird dann an die Spalte `boss` des Datensatzes *Müller* übergeben.

4.1.3 Objekt-relationales Mapping

Im HRManager wird ein ORM über das *Java Persistence API* (JPA) realisiert. Die Spezifikation des JPA liegt aktuell in der Version 2 vor [KS09, S. 13]. Über JPA werden in der Beispiel-Anwendung die Klassen der Anwendungsebene mit den Tabellen der Persistenzebene verknüpft. Mit diesem Mapping zwischen Klassen und Tabellen ist es möglich, Objekte anstatt Relationen aus der Datenbank abzufragen. Für die Beschreibung des Mappings werden Annotation verwendet. Annotationen stehen auch in den zu Beginn des Kapitels vorgestellten Anwendungen und Frameworks zur Verfügung bzw. werden dort genutzt (vgl. Abschnitt 4.1 Seite 28). Des Weiteren wird die Verwendung von Annotationen im praktischen Einsatz von Hibernate empfohlen [Tee07].

Zur Beschreibung des ORM sind die Java-Klassen, wie in Listing 4.3 für die Klasse `Department` dargestellt, zu annotieren. Zunächst muss die Klasse, wie in Zeile 1 dargestellt, mit `@Entity` annotiert werden. Dadurch übernimmt das ORM die Verwaltung der Klasse und alle damit zusammenhängenden Aufgaben [KS06, S. 21], von denen in dieser Arbeit nur die Abfrage von Objekten der markierten Klasse aus der Datenbank von Belang ist. Darauf folgt in Zeile 2 die `@Table`-Annotation, mit der die Tabelle angegeben wird, in der Objekte vom Typ `Department` gespeichert werden. Erfolgt diese Annotation nicht, versucht das ORM eine Tabelle mit dem Namen *Department* zu finden, wobei Groß- und Kleinschreibung nicht beachtet wird [KS06, S. 74]. Eine solche Tabelle existiert in der Datenbank nicht, stattdessen werden alle Objekte vom Typ `Department` in der Tabelle `departments` verwaltet. Des Weiteren muss ein Attribut mit der `@Id`-Annotation versehen werden, das dann vom ORM zur eindeutigen Identifikation jedes Objekts der Klasse `Department` verwendet wird [KS06, S. 21]. Die Attributierung erfolgt an der Getter-Methode des Attributs `id` (siehe Zeile 11). Durch die Attributierung der Getter-Methode wird das Attribut `id` nicht direkt vom ORM beschrieben. Stattdessen erfolgt der Zugriff immer über die Getter-Methode [KS06, S. 73]. Attribute ohne Anno-

Listing 4.3: Mapping der Klasse `Department` auf die Tabelle `departments`

```

1  @Entity
2  @Table(name="departments")
3  public class Department implements Serializable {
4      private int id;
5      private String name;
6
7      public void setId(int id) {
8          this.id = id;
9      }
10
11     @Id
12     public int getId() {
13         return id;
14     }
15
16     public void setName(String name) {
17         this.name = name;
18     }
19
20     public String getName() {
21         return name;
22     }
23 }

```

Listing 4.4: Definition der Klasse `ExternalStaff` mit `Single-Table-Strategie`

```

1  @Entity
2  @Table(name="external_staff")
3  @Inheritance(strategy = InheritanceType.SINGLE_TABLE)
4  @DiscriminatorColumn(
5      name="business",
6      discriminatorType=DiscriminatorType.STRING
7  )
8  public abstract class ExternalStaff implements Serializable {
9      /* Definition der Klasse ExternalStaff */
10 }

```

tation werden standardmäßig über die dazugehörigen Getter- und Setter-Methoden in das ORM mit aufgenommen [KS06, S. 73 und 76]. Dies wird bei dem Attribut `name` ausgenutzt, das in Listing 4.3 keine explizite Annotation aufweist. Das ORM erkennt die Methoden `setName` (Zeile 16) und `getName` (Zeile 20) als Getter und Setter und nimmt die Methoden in das Mapping mit auf. Über das *Default Mapping* bringt das ORM die Methoden mit der Spalte `name` der Tabelle `departments` in Verbindung [KS06, S. 73]. Wird der Wert der Spalte `name` gelesen oder geschrieben, nutzt das ORM die Methoden `getName` bzw. `setName`. Dadurch wird der Wert des Attributs `name` in Zeile 21 gelesen bzw. in Zeile 17 geschrieben. Das Attribut `name` selbst wird vom ORM ignoriert [KS06, S. 73].

Das ORM dient des Weiteren der Abbildung der Vererbungshierarchie der Java-

Listing 4.5: Definition der Klasse `Janitor` mit *Single-Table-Strategie*

```

1  @Entity
2  @DiscriminatorValue("janitor")
3  public class Janitor extends ExternalStaff {
4      /* Definition der Klasse Janitor */
5  }

```

Listing 4.6: Definition der Klasse `Employee` mit *Joined*-Strategie

```
1 @Entity
2 @Table(name="employees")
3 @Inheritance(strategy = InheritanceType.JOINED)
4 public class Employee implements Serializable {
5     /* Definition der Klasse Employee */
6 }
```

Listing 4.7: Definition der Klasse `Manager` mit *Joined*-Strategie

```
1 @Entity
2 @Table(name = "managers")
3 public class Manager extends Employee {
4     /* Definition der Klasse Manager */
5 }
```

Klassen auf Tabellen in der Datenbank. In der Beispiel-Anwendung werden die Vererbungsstrategien *Single-Table*- und *Joined*-Strategie verwendet. Die Strategien entsprechen der in Abschnitt 4.1.2 vorgestellten Darstellung der Klassenhierarchie durch eine einzige Tabelle bzw. durch eine Tabelle für jede Unterklasse. Das Listing 4.4 zeigt einen Ausschnitt der Klasse `ExternalStaff`. In Zeile 3 wird mit dem Attribut `strategy` der `@Inheritance`-Annotation die *Single-Table*-Strategie eingestellt. Mit der `@DiscriminatorColumn`-Annotation in Zeile 4 wird die Tabellen-Spalte bestimmt, anhand der die Unterscheidung der verschiedenen Klassen erfolgt, die durch die Tabelle `external_staff` repräsentiert werden. Dazu wird der Name der Spalte über das Attribut `name` und der in der Spalte gespeicherte Datentyp in dem Attribut `discriminatorType` angegeben. In Listing 4.5 ist ein Ausschnitt aus der Klasse `Janitor` dargestellt, die von der Klasse `ExternalStaff` erbt. Um die Klasse `Janitor` korrekt mit der Darstellung in der Datenbank zu verbinden, ist die `@Discriminator`-Annotation in Zeile 2 notwendig. Die Annotation gibt den Wert an, anhand Datensätze der Klasse `Janitor` von anderen Datensätzen in der Tabelle unterschieden werden können. Die Angabe einer Tabelle ist nicht notwendig, da über die `@Table`-Annotation der Oberklasse `ExternalStaff` bekannt ist, in welcher Tabelle Instanzen der Unterklassen von `ExternalStaff` gespeichert werden sollen. Listing 4.6 zeigt einen Ausschnitt aus der Definition der Klasse `Employee`, die über die `@Inheritance`-Annotation in Zeile 3 eine *Joined*-Strategie einstellt. Schließlich zeigt das Listing 4.7 die Unterklasse `Manager` der Klasse `Employee`. Die Klasse `Manager` muss über die `@Table`-Annotation die Tabelle angeben, in der Instanzen vom Typ `Manager` gespeichert werden sollen. Eine Vererbungsstrategie muss nicht mit angegeben, da diese über die Oberklasse `Employee` bekannt ist.

Als konkrete Implementierung des JPA wird *Hibernate*⁹ verwendet. Über eine Konfigurationsdatei muss Hibernate bekannt gemacht werden, welche Klassen in das ORM mit aufgenommen werden sollen. Die Konfigurationsdatei speichert das Mapping in einer XML-Struktur, wie sie das Listing 4.8 für den `HRManager` zeigt. In der Zeile 3 wird das Java-Package angegeben, aus dem die folgenden Klassen referenziert werden. In den Zeilen 4 bis 8 werden die Java-Klassen angegeben, die durch das ORM verwaltet werden sollen.

⁹Die Anbindung von Hibernate an die SQLite-Datenbank erfolgt über das *hibernate-sqlite*-Projekt. <http://code.google.com/p/hibernate-sqlite/>

Listing 4.8: Mapping der Java-Klassen in der Hibernate-Konfiguration

```

1 <hibernate-configuration>
2   <session-factory>
3     <mapping package="hrm" />
4     <mapping class="hrm.Department" />
5     <mapping class="hrm.Employee" />
6     <mapping class="hrm.Manager" /></mapping>
7     <!-- weitere Mappings -->
8     <mapping class="hrm.Janitor" /></mapping>
9   </session-factory>
10 </hibernate-configuration>

```

Listing 4.9: Funktion zur Summierung aller Gehälter der in der Liste `list` enthaltenen Instanzen der Klassen `Employee` und `ExternalStaff`

```

1 (def sumSalary (fn [list] (if (not (instance? java.util.List list))
2   (throw
3     (new java.lang.IllegalArgumentException "list is not a list")
4   )
5   (if (and
6     (not (empty? list))
7     (and
8       (not (instance? hrm.ExternalStaff (first list)))
9       (not (instance? hrm.Employee (first list)))
10    )
11   )
12   (throw
13     (new java.lang.IllegalArgumentException)
14   (if (empty? list) 0
15     (+ (. (first list) getSalary) (sumSalary (rest list))))))))))

```

4.1.4 Scripting

Bestandteile der Anwendungslogik sind nicht direkt in Java implementiert. Stattdessen befinden sich diese in separaten Skripten. Durch die Auslagerung ist eine Modifikation dieser Bestandteile möglich, ohne direkt den Java-Quellcode verändern und neu compilieren zu müssen.

Als Skript-Sprache wird Clojure verwendet. Clojure ist eine von Lisp abgeleitete funktionale Programmiersprache, die für den Einsatz auf der Java-Plattform entwickelt wurde [Van10, S. 1-2].

Als Beispiel für die Verwendung von Clojure in der Beispiel-Anwendung soll eine in Clojure definierte Funktion und deren Aufruf von Java aus im Folgenden erläutert werden. Das Listing 4.9 zeigt die Funktion `sumSalary`. Die Funktion dient der Berechnung der Summe aller Gehälter der in der Liste `list` enthaltenen Objekte. Dabei müssen die Objekte Instanzen der Klassen `Employee` bzw. `ExternalStaff` sein. In Zeile 1 wird dazu geprüft, ob es sich bei `list` um eine Liste vom Typ `java.util.List` oder einer davon abgeleiteten Klasse handelt. In den zwei auf Zeile 6 folgenden Zeilen wird die Zugehörigkeit des ersten Listen-Elements zu den genannten Klassen `Employee` und `ExternalStaff` geprüft. Sollten die Liste oder die Objekte in der Liste nicht den Anforderungen genügen, wird in Zeile 12 eine Ausnahme geworfen. Schließlich erfolgt in der letzten und vorletzten Zeile die eigentliche Berechnung der Summe der Gehälter.

Der in Listing 4.9 gezeigte Code kann aus Java heraus aufgerufen werden. Zunächst muss dazu die entsprechende Skript-Datei geladen und die gewünschte Funktion refe-

Listing 4.10: Berechnung der Summe aller Gehälter der in der Liste `managers` enthaltenen Instanzen der Klassen `Employee` und `ExternalStaff`

```
1 RT.loadResourceScript("scripting/Scripting.clj");
2 Var sumSalary = RT.var("scripting", "sumSalary");
3
4 float sum = (Float)sumSalary.invoke(managers);
5 System.out.println("Salary␣managers:␣" + sum);
```

renziert werden. Listing 4.10 zeigt in der Zeile 1 das Laden der Skript-Datei und in der darauf folgenden Zeile 2 die Erstellung einer Referenz auf die Funktion `sumSalary`. Die letzten beiden Zeilen 4 und 5 zeigen zum einen den Aufruf der referenzierten Funktion¹⁰ mit der Liste `managers` und zum anderen die abschließende Ausgabe des Ergebnisses¹¹.

4.2 Probleme bei der Realisierung von mehrsprachigen Refactorings

Dieser Abschnitt beschreibt die Durchführung verschiedener Refactorings anhand der Beispiel-Anwendung. Ziel der Durchführung ist die Erhaltung der globalen semantischen Integrität zwischen den Software-Artefakten innerhalb des in Abschnitt 4.1 beschriebenen HRManagers. Die globale semantische Integrität wird aus Sicht des Nutzers der Beispiel-Anwendung betrachtet.

Die Erläuterung der notwendigen Refactorings und Modifikationen zur Umsetzung des jeweiligen mehrsprachigen Refactorings erfolgt nach dem folgenden Schema:

- n. Software-Artefakt
 - nn. Refactorings und Modifikationen
 - oder*
 - nn. Art der Repräsentation der Klassenhierarchie in der Datenbank
 - nnn. Refactorings und Modifikationen

Die Betrachtung des Refactorings beginnt immer mit dem Software-Artefakt, auf dem das initiale Refactoring durchgeführt wird. In den Unterpunkten werden die Modifikationen beschrieben, die zur Umsetzung des initialen Refactorings notwendig sind. Wird durch das initiale Refactoring die globale semantische Integrität verletzt, werden bis zu n weitere Software-Artefakte betrachtet, die von der nicht mehr vorhandenen globalen semantischen Integrität betroffen sind. Dazu werden in den jeweiligen Unterpunkten die Modifikationen beschrieben, die zur Wiederherstellung der globalen semantischen Integrität aus Sicht des betrachteten Software-Artefakts notwendig sind. Das bedeutet nicht, dass sich diese Modifikationen ausschließlich auf das betrachtete Software-Artefakt beziehen. Die Modifikationen können auch Änderungen an anderen Software-Artefakten

¹⁰Auf eine Fehlerbehandlung wurde in diesem Beispiel verzichtet.

¹¹Eine Beschreibung der Klassen `RT` und `Var` und der dazugehörigen Methoden `loadResourceScript` bzw. `var` und `invoke` ist in [Van10, S. 149 - 150] zu finden.

beschreiben, wenn diese aus der Sicht des aktuell betrachteten Software-Artefakts notwendig sind.

Die auf den im Folgenden beschriebenen Software-Artefakten ausgeführten Refactorings werden in den jeweiligen Abschnitten kurz erläutert und deren Verwendung und Nutzen dargestellt. Grundsätzlich wird bei den Beispielen zunächst davon ausgegangen, dass alle Vor- und Nachbedingungen des jeweiligen Refactorings erfüllt werden. Des Weiteren wird angenommen, dass bekannt ist, in welcher Beziehung die Entitäten in den unterschiedlichen Artefakten zueinander stehen. Wenn z.B. ein Refactoring auf der Java-Klasse `Employee` durchgeführt wird, so ist bekannt, dass dabei auch das Hibernate-Artefakt und die Definition der Tabelle `employees` im SQL-Artefakt mit betrachtet werden müssen. Eine ausführliche Betrachtung und Diskussion der genannten Annahmen und der damit verbundenen Probleme erfolgt nach der Durchführung der Refactorings im Abschnitt 4.2.4.

Die Gliederung dieses Abschnitts erfolgt anhand der Software-Artefakte, von denen das jeweilige Refactoring ausgegangen ist. In Abschnitt 4.2.1 werden alle Refactorings erläutert, die durch Anwendung auf das Java-Artefakt gestartet wurden. Darauf folgt in Abschnitt 4.2.2 die Erläuterung der Refactorings, die ausgehend von dem SQL-Artefakt umgesetzt wurden. Der Abschnitt 4.2.3 beschreibt schließlich die Refactorings, die am Clojure-Artefakt durchgeführt wurden. Für ORMs wurden keine Refactorings in der Literatur gefunden, daher werden im Folgenden keine Refactorings ausgehend vom ORM umgesetzt. Im Abschnitt 4.2.4 werden weitere Besonderheiten gesondert betrachtet, auf die bei den einzelnen Refactorings nicht eingegangen wird.

4.2.1 Java

Ausgehend von den Java-Artefakten wurden folgende Refactorings angewandt:

- `RENAME METHOD` nach [Fow99, S. 273],
- `PULL UP METHOD` nach [Fow99, S. 322],
- `MOVE CLASS` nach [Opd92, S. 53].

In der Beispiel-Anwendung werden Methoden durch das ORM mit Spalten in Datenbank-Tabellen verknüpft. Das `RENAME METHOD REFACTORING` wurde gewählt, da mit dem `RENAME COLUMN REFACTORING` [Amb03, S. 414] ein entsprechendes Äquivalent für Datenbanken existiert. Das `PULL UP METHOD` und `MOVE CLASS REFACTORING` wurden als Vertreter von Refactorings gewählt, die Veränderungen an der Struktur von Klassen bzw. Klassen-Hierarchien vornehmen. Es soll betrachtet werden, wie sich diese Veränderungen eines spezifischen Objekt-orientierten Merkmals auf andere Software-Artefakte auswirkt, die, wie SQL, dieses Merkmal nicht unterstützen.

Das `PULL UP METHOD` und `MOVE CLASS REFACTORING` werden jeweils an den zwei unterschiedlichen Datenbank-Schemas dargestellt, die in Abschnitt 4.1.2 zur Darstellung von Klassenhierarchien beschrieben wurden.

Rename Method

Im HRManager ist die Bedeutung des Rückgabewerts der Methode `getName` der Klasse `Employee` nicht eindeutig ersichtlich, da nicht klar wird ob der gesamte Name oder nur der Vorname des Angestellten zurückgegeben wird. Daher soll die Methode `getName` mit Hilfe des RENAME METHOD REFACTORINGS in `getFirstName` umbenannt werden, da der Rückgabewert der Vorname des Angestellten ist.

Das RENAME METHOD REFACTORING wird durch Umbenennen der Methode `getName` in `getFirstName` gestartet. Folgende Schritte sind zur Anwendung des RENAME METHOD REFACTORINGS und zur Herstellung der globalen semantischen Integrität zwischen den Software-Artefakten notwendig:

1. Java

Umbenennen der Methode `getName` in `getFirstName` und Änderung aller Verweise auf die Methode `getName` in allen Java-Artefakten

2. Hibernate

Umbenennen der Methode `setName` in `setFirstName` und Änderung aller Verweise auf die Methode `setName` in allen Java-Artefakten

3. SQL

- (a) Durchführung eines RENAME COLUMN REFACTORINGS[Amb03, S. 414], also Umbenennen der Spalte `name` zu `firstname`¹² in der Tabelle `employees` und Änderung aller Verweise auf die Spalte `name` der Tabelle `employee` in allen SQL-Artefakten

oder

- (b) Hinzufügen einer `@Column`-Annotation zur Methode `getFirstName`, über die die Spalte `name` in der Datenbank referenziert wird[KS06, S. 76]

Insbesondere der zweite Schritt kann zu Problemen bei der Automatisierung des Refactorings führen. Nach der Umbenennung der Methode `getName` in `getFirstName` geht der Zusammenhang zu der Methode `setName` im ORM verloren, so dass der alte Name der Methode `getName` gespeichert werden muss, damit das mehrsprachige Refactoring den Zusammenhang zur Methode `setName` herstellen kann.

Die in den Schritten 3a und 3b genannten Alternativen stellen an sich kein Hindernis für die Automatisierung des Refactorings dar. Im Allgemeinen ist der Schritt 3b dem Schritt 3a aus den folgenden Gründen vorzuziehen. Zunächst werden Modifikationen im SQL-Artefakt vermieden, die zu weiteren Konflikten mit existierenden Bezeichnern oder Schlüsselwörtern führen können. Der Aufwand zur Umsetzung der in Schritt 3b beschriebenen Modifikation ist begrenzt. Im schlechtesten Fall muss entweder eine bestehende `@Column`-Annotation modifiziert oder eine neue hinzugefügt werden. In der Variante 3a

¹²An dieser Stelle wäre z.B. auch `firstName` als Spalten-Name zulässig, da das ORM im HRManager Groß- und Kleinschreibung nicht beachtet. Dies ist allerdings abhängig von der verwendeten Datenbank[KS09].

muss hingegen eine unbekannte Anzahl von Statements modifiziert werden. Für die automatische wie händische Umsetzung ist daher das in Schritt 3b beschriebenen Vorgehen vorzuziehen.

Rename Method Refactoring im Zusammenhang mit Clojure Da im Clojure-Artefakt weder die Methode `getName` noch `setName` verwendet werden, musste bei der zuvor beschriebenen Umsetzung des RENAME METHOD REFACTORINGS keine Umbenennung im Clojure-Artefakt umgesetzt werden. Dennoch sollen an dieser Stelle einige Probleme diskutiert werden, die bei der Durchführung des RENAME METHOD REFACTORINGS im Zusammenhang mit Clojure auftreten können.

Bei Clojure handelt es sich um eine dynamische Sprache, es muss also nicht explizit der Datentyp einer Variable angegeben werden [Van10, S. 51]. Welche Auswirkungen dies auf die Durchführung eines RENAME METHOD REFACTORINGS hat, soll anhand der in Listing 4.9 definierten Funktion `sumSalary` erläutert werden. In Zeile 14 erfolgt auf das erste Element der Liste `list` der Aufruf der Methode `getSalary`. Clojure nutzt in diesem Fall *Reflection*, um die Methode `getSalary` des ersten Listenelements aufzurufen [Van10, S. 191]. D.h., dass Clojure über das von Java bereitgestellte Reflection-API dynamisch auf Funktionen eines Objekts zugreifen kann, ohne statische Typ-Informationen über dieses Objekt besitzen zu müssen. Angenommen es soll ein RENAME METHOD REFACTORING auf die Methode `getSalary` der Klasse `Employee` durchgeführt und die Methode in `getMonthlySalary` umbenannt werden. Durch die fehlenden Typ-Informationen in der Funktion `sumSalary` kann nicht entschieden werden, ob der Aufruf der Methode `getSalary` auf ein Objekt des Typs `Employee` oder auf irgendeinen anderen Typ mit einer gleichnamigen Methode erfolgt. Mit sogenannten *Type Hints* lässt sich Reflexion innerhalb von Clojure vermeiden [Van10, S. 191]. *Type Hints* wirken wie statische Typ-Information, geben also den Typ einer Variable explizit an. Mit Hilfe von *Type Hints* kann somit eine Umsetzung des RENAME METHOD REFACTORINGS innerhalb von Clojure erfolgen, da dann die benötigten Typ-Informationen zur Verfügung stehen. Dabei muss aber bedacht werden, dass durch die Nutzung von *Type Hints* auf die Möglichkeiten dynamischer Typisierung verzichtet wird.

In [RBJ97] wird ein weiterer Lösungsansatz für die ebenfalls dynamisch getypte Sprache Smalltalk über sogenannte *method wrappers* beschrieben. Dazu wird nach dem Umbenennen der Methode `getSalary` in `getMonthlySalary` eine weitere Methode mit dem alten Namen `getSalary` zum Sourcecode hinzugefügt. Die neue Methode `getSalary` ist in der Lage die Ausführung des Programms zu stoppen und zur Laufzeit Modifikationen am Quellcode durchzuführen. Nach dem Hinzufügen wird das Programm ausgeführt. Sobald ein Methodenaufruf den Method Wrapper `getSalary` der umbenannten Methode `getMonthlySalary` aufruft, benennt der Method Wrapper auf Basis der Informationen des Call-Stacks den Methodenaufruf um. Roberts et al. weisen darauf hin, dass mit Method Wrappers die vollständige Umsetzung des RENAME METHOD REFACTORINGS nicht garantiert werden kann, da möglicherweise nicht alle relevanten Methodenaufrufe im Sourcecode besucht werden.

Pull Up Method

Das PULL UP METHOD REFACTORING dient dazu, Methoden, die in verschiedenen Klassen genutzt werden, aber die gleiche Aufgabe erfüllen, in eine gemeinsame Oberklasse zu verschieben.

In der Beispiel-Anwendung besitzt die Klasse `Manager` die Methoden `getBoss` und `setBoss`, um leitenden Angestellten einen Vorgesetzten zuzuweisen. Allerdings besitzen auch einfache Angestellte, wie sie durch die Klasse `Employee` repräsentiert werden, einen Vorgesetzten. Daher sollen die beiden Methoden von der Unterklasse `Manager` in die Oberklasse `Employee` verschoben werden.

Zur Durchführung des PULL UP METHOD REFACTORINGS und zum Erreichen der globalen semantischen Integrität der Software-Anwendung sind folgenden Schritte durchzuführen:

1. Java

- (a) Verschieben der Methode `setBoss` aus der Klasse `Manager` in die Klasse `Employee`
- (b) Verschieben des privaten Attributs `boss` aus der Klasse `Manager` in die Klasse `Employee`, da die in Schritt 1a verschobene Methode `setBoss` aus der Oberklasse `Employee` nicht auf das Attribut `boss` zugreifen kann (siehe dazu Abbildung 4.1)

2. Hibernate

Verschieben der Methode `getBoss` aus der Klasse `Manager` in die Klasse `Employee`

3. SQL

- (a) Eine Tabelle pro Unterklasse
 - i. Verschieben der Spalte `boss` aus der Tabelle `managers` in die Tabelle `employees`
 - ii. Änderung aller Verweise auf die Spalte `boss` in der Tabelle `managers` auf die Spalte in der Tabelle `employees`
- (b) Eine Tabelle pro Klassenhierarchie
Keine Modifikationen notwendig

Für die Durchführung des PULL UP METHOD REFACTORINGS ergeben sich folgende Besonderheiten:

- Hibernate erwartet zu der annotierten Setter-Methode `setBoss` die entsprechende Getter-Methode `getBoss`. Die Methode `setBoss` wurde im Schritt 1a in die Oberklasse `Employee` verschoben. Von der Klasse `Employee` ist die Methode `getBoss` in der Unterklasse `Manager` nicht sichtbar. Die Methode `getBoss` muss, wie in Schritt 2 beschrieben, von der `Manager` in die Klasse `Employee` verschoben werden, damit das ORM zum Setter `setBoss` den Getter `getBoss` findet. Erst dann funktioniert das ORM wieder korrekt.

Listing 4.11: Füllen der Datenbank mit Manager-Daten

```

1  — Bosses
2  INSERT INTO employees (firstname, surname)
3     VALUES ('Michael', 'Müller');
4
5  INSERT INTO managers (id)
6     SELECT id
7     FROM employees
8     WHERE surname = 'Müller';
9
10 INSERT INTO employees (firstname, surname)
11    VALUES ('Linda', 'Langenhahn');
12
13 INSERT INTO managers (id, account)
14    SELECT id
15    FROM employees
16    WHERE surname = 'Langenhahn';
17
18 — Relations between managers
19
20 UPDATE managers
21    SET boss = (SELECT id
22                FROM employees
23                WHERE surname = 'Langenhahn')
24    WHERE (SELECT id
25            FROM employees
26            WHERE employees.surname = 'Müller'
27            AND employees.id = managers.id);

```

- Die Schritte 3a und 3b unterscheiden sich deutlich hinsichtlich der beschriebenen Modifikationen. Die Unterschiede werden durch die unterschiedliche Darstellung der Klassenhierarchie in der Datenbank hervorgerufen.

Wie in Punkt 2 beschrieben, unterscheiden sich die durchgeführten Maßnahmen zur Erhaltung der globalen semantischen Integrität deutlich zwischen den Darstellungen der Klassenhierarchie in der Datenbank. Bei der Darstellung der Klassenhierarchie in einer Tabelle sind keine weiteren Modifikationen notwendig. Daher eignet sich diese Darstellung besser für die Anwendung des durchgeführten Refactorings.

Es wurde zuvor darauf hingewiesen, dass die Datenbank per SQL mit Test-Daten gefüllt wird. Das Eintragen dieser Daten erfolgt durch SQL-Statements. Listing 4.11 zeigt, wie Daten von leitenden Angestellten in die Datenbank eingepflegt werden. Dazu werden zunächst in den Zeilen 2 und 10 die Angestellten *Müller* und *Langenhahn* in der Tabelle `employee` angelegt. In den Zeilen 5 und 13 werden Referenzen von der Tabelle `managers` auf die Datensätze der Angestellten *Müller*, respektive *Langenhahn* in der Tabelle `employee` hergestellt. Damit sind die beiden Datensätze *Müller* und *Langenhahn* als leitende Angestellte in der Anwendung bekannt. Durch das abschließende UPDATE-Statement in Zeile 20 wird eine Beziehung zwischen den Datensätzen der Vorgesetzten und dem ihr unterstellten Mitarbeiter realisiert. Konkret wird die leitende Angestellte *Langenhahn* Vorgesetzte des leitenden Angestellten *Müller*. Nach der Durchführung des Refactorings besitzt das UPDATE-Statement die in Listing 4.12 dargestellte Form. Insbesondere das WHERE-Statement in Zeile 5 unterscheidet sich von seiner Ausgangsform in Listing 4.11 Zeile 24 darin, dass eine SELECT-Anweisung durch eine einzelne Bedingung ersetzt wurde. Des Weiteren verweist das Update-Statement in Listing 4.12 ausschließlich auf die Tabelle `employees`. Im Gegensatz dazu enthält das Update-Statement in Listing 4.11 Verweise auf die Tabellen `employees` und `managers`. Die Modifikation der

Listing 4.12: Einfügen einer Referenz zwischen zwei `Manager`-Datensätzen

```
1 UPDATE employee
2   SET boss = (SELECT id
3               FROM employee
4               WHERE surname = 'Langenhahn')
5   WHERE employee.surname = 'Müller';
```

referenzierten Tabellen sowie der Bedingungen im `WHERE`-Statement wird in dieser Arbeit als Änderung des internen Verhaltens zwischen den `UPDATE`-Statements in Listing 4.11 und 4.12 verstanden. Daher handelt es sich in diesem Fall nicht nur um eine strukturelle, sondern um eine semantische Modifikation des Statements. Die durchgeführten Anpassungen können daher nicht als Refactoring bezeichnet werden.

Eine semantische Modifikation eines Ausdrucks erschwert die Automatisierung der Modifikation, da neben einer strukturellen auch eine semantische Analyse durchgeführt werden muss. Die semantische Analyse ermittelt, welches Ergebnis die Ausführung eines Ausdrucks liefert. Ohne das Ergebnis der semantischen Analyse kann nicht entschieden werden, ob zwei Ausdrücke zum gleichen Ergebnis kommen. Bezogen auf die zuvor durchgeführten Modifikationen kann ohne eine semantische Analyse nicht entschieden werden, ob das `UPDATE`-Statement in Listing 4.12 semantisch äquivalent zu dem `UPDATE`-Statement in Listing 4.11 ist. Eine Aussage darüber, ob die durchgeführten Modifikationen die semantische Integrität wahren, ist dann ebenfalls nicht möglich.

Move Class

Das `MOVE CLASS REFACTORING` dient dazu, die Oberklasse der jeweiligen Klasse zu ändern. Opdyke beschreibt in [Opd92] das `MOVE CLASS REFACTORING` sowohl für das Verschieben innerhalb derselben Klassenhierarchie, wie auch zwischen unterschiedlichen Klassenhierarchien¹³. Im Folgenden werden diese Möglichkeiten getrennt betrachtet.

Bewegen einer Klasse innerhalb einer Klassenhierarchie Im `HRManager` ist die Klasse `Salesperson` von der Klasse `Manager` abgeleitet. Diese Ableitung wurde gewählt, da leitenden Angestellten, wie auch Verkäufern, ein Dienstwagen zur Verfügung gestellt wird. Diese Ableitung spiegelt aber nicht die Realität in den Unternehmen wieder, in der Verkäufer nicht gleichzeitig leitende Angestellte sind. Um diesen Sachverhalt auch in der Klassenhierarchie wiederzufinden, wird ein `MOVE CLASS REFACTORING` auf der Klasse `Salesperson` geplant. Das Ziel des Refactorings ist es, die Oberklasse von `Salesperson` von der Klasse `Manager` auf die Klasse `Employee` zu ändern (vgl. Abb. 4.1).

Zur Umsetzung des `MOVE CLASS REFACTORINGS` sind folgende Schritte zur Sicherung der globalen semantischen Integrität notwendig:

1. Java
 - (a) Kopieren der Felder `account`, `companyCarLicensePlate` und der dazugehörigen Getter- und Setter-Methoden aus der Klasse `Manager` in die Klas-

¹³Beim Verschieben innerhalb derselben Klassenhierarchie ist die neue Oberklasse also bereits Ober- oder Unterklasse der Klasse, die verschoben wird. Das ist beim Verschieben zwischen unterschiedlichen Klassenhierarchien nicht der Fall.[Opd92].

se `Salesperson`, da diese von der neuen Oberklasse `Employee` nicht geerbt werden können

- (b) Änderung der Oberklasse von `Salesperson` zu `Employee`

2. SQL

- (a) Eine Tabelle pro Unterklasse
 - i. Kopieren der Spalten-Definitionen `account` und `company_car_license_plate` aus der Tabelle `managers` in die Tabelle `salespersons` (vgl. Abb. 4.4)
 - ii. Änderung der Referenz der Tabelle `salespersons` von `managers` auf `employees`
 - iii. Änderung von Statements, die Spalten in der Tabelle `managers` ansprechen, sich aber auf Datensätze von Verkäufern beziehen
- (b) Eine Tabelle pro Klassenhierarchie
Keine Modifikationen notwendig

Die notwendigen Änderungen am Datenbank-Schema bei der Verwendung einer Tabelle pro Klassenhierarchie sollen an den folgenden SQL-Statements verdeutlicht werden. Zunächst werden in Zeile 2 des Listing 4.13 die Daten des Angestellten *Becker* zur Datenbank hinzugefügt. Da es sich bei dem Angestellten um einen Verkäufer handelt und Verkäufer im HRManager als Spezialisierung von Managern behandelt werden, wird in Zeile 5 ein weiterer Datensatz in die Tabelle `managers` hinzugefügt, der eine Referenz auf den Datensatz des Angestellten *Becker* in Tabelle `employees` enthält. Dabei fügt das `INSERT`-Statement den Wert *Project_Y* in die Spalte `account` ein. Schließlich wird in Zeile 10 ein neuer Eintrag zur Tabelle `salesperson` hinzugefügt, damit der Datensatz *Becker* korrekt als Verkäufer in der Datenbank abgebildet wird. Dieser Eintrag enthält eine Referenz auf den Eintrag des Angestellten *Becker* in der Tabelle `employees`, der auch in der Tabelle `managers` referenziert wird. Dies wird durch das `WHERE`-Statement in Zeile 13 beschrieben. Durch die `INSERT`-Statements in den Zeilen 2, 5 und 10 wird somit die gesamte Vererbungshierarchie der Java-Klasse `Salesperson` beschrieben. Das `INSERT`-Statement in Zeile 16 nutzt diese Nachbildung der Vererbungshierarchie im `WHERE`-Statement in Zeile 19 aus, um in der Tabelle `customers` eine Referenz auf den richtigen Eintrag in der Tabelle `salespersons` anzulegen¹⁴. Die `INSERT`-Statements in den Zeilen 5 und 16 gehören somit zu den in Schritt 2(a)iii beschriebenen Statements, denn sie beziehen sich auf die Vererbungsbeziehung zwischen den Java-Klassen `Manager` und `Salesperson`. Die Statements in Zeile 5 und 10 bauen die Vererbungshierarchie auf, die im `WHERE`-Statement in Zeile 19 genutzt wird, um den richtigen Datensatz zu referenzieren. Nach der Ausführung des Refactorings wird das in Listing 4.13 Zeile 5 dargestellte `INSERT`-Statement nicht mehr benötigt, da eine Referenz zwischen den Java-Klassen `Manager` und `Salesperson` und damit auch zwischen den Tabellen `managers` und `salespersons` nicht mehr existiert. Die anderen Statements haben die in Listing 4.14 dargestellte Form.

¹⁴Um die Korrektheit dieses `INSERT`-Statements sicherzustellen, ist es in diesem Fall nicht zwingend notwendig die Tabelle `managers` im Listing 4.13 Zeile 19 zu referenzieren. Um zu garantieren, dass nur der Verkäufer und nicht der Angestellte mit dem Nachnamen *Becker* selektiert wird, genügt die Referenz der Tabellen `employees` und `salespersons`. Werden aber alle Daten des Verkäufers *Becker* benötigt, so ist auch die Tabelle `managers` zu referenzieren. Dieser Fall wird in dem Beispiel dargestellt.

Listing 4.13: Füllen der Datenbank mit Salesperson- und Customer-Daten

```
1  — Salespersons
2  INSERT INTO employees (firstname, surname)
3    VALUES ('Bernd', 'Becker');
4
5  INSERT INTO managers (id, account)
6    SELECT id, 'Project_Y'
7    FROM employees
8    WHERE surname = 'Becker';
9
10 INSERT INTO salespersons (id)
11    SELECT id
12    FROM employees, managers
13    WHERE employees.id = managers.id AND surname = 'Becker';
14
15 — Customers
16 INSERT INTO customers ('firstname', 'surname', 'taken_care_of_by')
17    SELECT 'Erhard', 'Erlich', salespersons.id
18    FROM employees, managers, salespersons
19    WHERE employees.id = managers.id AND managers.id = salespersons.id
20    AND surname = 'Becker';
```

Listing 4.14: Füllen der Datenbank mit Salesperson- und Customer-Daten nach dem Refactoring

```
1  — Salespersons
2  INSERT INTO employees (firstname, surname)
3    VALUES ('Bernd', 'Becker');
4
5  INSERT INTO salespersons (id, account)
6    SELECT id, 'Project_Y'
7    FROM employees
8    WHERE surname = 'Becker';
9
10 — Customers
11 INSERT INTO customers ('firstname', 'surname', 'taken_care_of_by')
12    SELECT 'Erhard', 'Erlich', salespersons.id
13    FROM employees, salespersons
14    WHERE employees.id = salespersons.id
15    AND surname = 'Becker';
```

Durch das Refactoring ist es erforderlich, die Referenzen zur Tabelle `managers` zu entfernen. In den `WHERE`-Statements Zeile 8 und Zeile 14 sind die Referenzen auf die Tabelle `managers` entfernt worden. Des Weiteren muss die Information über die Kostenstelle, die in Zeile 6 des Listings 4.13 angegeben wird, erhalten werden. Durch den Schritt 2(a) sind alle Spalten in der Tabelle `salespersons` vorhanden, die vor dem Refactoring nur in der Tabelle `managers` zur Verfügung standen. Dadurch ist es möglich, die Information über die Kostenstelle zur Tabelle `salespersons` hinzuzufügen, wie es in Listing 4.14 Zeile 5 dargestellt wird.

An den Veränderungen der `INSERT`-Statements in Listing 4.13 soll verdeutlicht werden, dass es sich bei den Änderungen um semantische Modifikationen handelt. Zunächst wird das `INSERT`-Statements in Zeile 5 entfernt. Das Statement in Zeile 10 ist semantisch so verändert, dass die Daten aus dem weggefallenen Statement nicht verloren gehen und die informationelle Semantik erhalten wird. Listing 4.14 Zeile 5 zeigt das Ergebnis dieser Modifikation. Die Menge der Ziel-Spalten (Zeile 5) und der zu lesenden Werte (Zeile 6) sind erweitert. Schließlich müssen auch Referenzen auf die nicht mehr vorhandene Relation zwischen den Tabellen `employees`, `managers` und `salespersons` entfernt

werden. Im `WHERE`-Statement in Listing 4.13 Zeile 19 wird auf diese Referenz verwiesen. Nach der Modifikation ist die Referenz auf die Tabelle `managers` sowohl in dem `FROM`-, wie auch `WHERE`-Statement in der Zeile 13 bzw. 14 des Listings 4.14 nicht mehr vorhanden. Die Modifikation des `WHERE`-Statements wird als Änderung des internen Verhaltens des `INSERT`-Statements verstanden und ist somit keine strukturelle, Semantik-erhaltende Modifikation.

Die Durchführung des betrachteten mehrsprachigen Refactorings zeigt, dass das `MOVE CLASS REFACTORING` des Java-Artefakts zu einer nicht Semantik-erhaltende Modifikation in dem SQL-Artefakt führte. In SQL stehen, ohne Objekt-orientierte Erweiterungen zu nutzen, keine Mittel zur Verfügung, die Klassenhierarchie im Java-Artefakt strukturell nachzubilden. Über Fremdschlüssel-Beziehungen können Klassenhierarchien in SQL semantisch nachgebildet werden. Zur Darstellung einer Vererbungshierarchie wird dazu mit Fremdschlüsseln eine *Ist-ein*-Beziehung abgebildet. Im `HRManager` ist z.B. definiert, dass die Klasse `Manager` auch vom Typ `Employee` ist, da die Klasse `Manager` von der Klasse `Employee` abgeleitet wird. Diese Beziehung wird in der Tabelle `managers` über die Definition eines Fremdschlüssels zur Tabelle `employees` dargestellt. Es werden über Fremdschlüssel aber auch *Hat-ein*-Beziehungen dargestellt. Diese besteht z.B. zwischen den Tabellen `employees` und `departments`, da die Java-Klasse `Employee` einen Verweis auf ein Objekt von Typ `Department` besitzt. Anhand einer Fremdschlüssel-Definition an sich ist nicht erkennbar, welche Art der Beziehung durch den Fremdschlüssel dargestellt werden soll. Da kein strukturelles Äquivalent zur Darstellung von *Ist-ein*- und *Hat-ein*-Beziehungen in SQL existiert, können z.B. in SQL dargestellte Vererbungshierarchien nicht strukturell modifiziert werden.

Bewegen einer Klasse zwischen unterschiedlichen Klassenhierarchien Im `HR-Manager` werden externe Mitarbeiter in einer eigenen Klassenhierarchie verwaltet. Die abstrakte Klasse `ExternalStaff` ist die Oberklasse dieser Hierarchie. In der Beispiel-Anwendung werden Hausmeister in der Klasse `Janitor` abstrahiert, die Unterklasse von `ExternalStaff` ist. Da Unternehmen existieren, die eigene Hausmeister anstellen, soll eine Version der Beispiel-Anwendung hergestellt werden, die diesen Sachverhalt widerspiegelt. Dazu soll ein `MOVE CLASS REFACTORING` realisiert werden, dass die Oberklasse von `Janitor` von `ExternalStaff` auf die Klasse `Employee` ändert.

Die Repräsentation der Hierarchien der Klassen `Employee` und `ExternalStaff` erfolgt mit unterschiedlichen Ansätzen. Das Datenbank-Schema der Klassenhierarchie von `Employee` sieht die Nutzung unterschiedlicher Tabellen pro Unterklasse vor. Die Abbildung der Klassenhierarchie von `ExternalStaff` erfolgt in einer einzelnen Tabelle. Die Umsetzung der unterschiedlichen Klassenhierarchien mit SQL ist in Listing 4.15 angedeutet. In Zeile 1 und 10 werden die Tabellen für die Klassen `Employee` und deren Unterklasse `Manager` definiert. In Zeile 15 erfolgt die Definition der Tabelle für die gesamte Klassenhierarchie der Klasse `ExternalStaff`. Die Zeile 26 zeigt ein `INSERT`-Statement zum Einfügen eines `Janitor`-Datensatzes.

Zur Durchführung des mehrsprachigen Refactorings und zur Herstellung der globalen semantischen Integrität sind folgende Schritte notwendig:

1. Java

Kopieren des Attributs `companyName` und der dazugehörigen Getter- und Setter

Listing 4.15: Definition der Tabellen für die Klassenhierarchien der Klassen `Employee` und `ExternalStaff`

```

1 CREATE TABLE employees(
2   id INTEGER PRIMARY KEY AUTOINCREMENT,
3   firstname varchar(255),
4   surname varchar(255),
5   salary real,
6   department integer REFERENCES departments(id)
7   boss integer REFERENCES employees(id)
8 );
9
10 CREATE TABLE managers(
11   id integer REFERENCES employees(id) PRIMARY KEY,
12   account varchar(255),
13 );
14
15 CREATE TABLE external_staff(
16   id INTEGER PRIMARY KEY AUTOINCREMENT,
17   firstname varchar(255),
18   surname varchar(255),
19   salary real,
20   business varchar(255), — 'janitor', etc.
21   company_name varchar(255),
22   address varchar(255)
23 );
24
25 — Janitors
26 INSERT INTO external_staff (firstname, surname, salary, business,
27   company_name, address)
28 VALUES ('Gerald', 'Gemrich', 3850, 'janitor',
29   'JanitorExpress', 'Unternehmensstraße 10a');

```

von `ExternalStaff` in die Klasse `Janitor`

2. Hibernate

Anpassen der Mapping-Strategie der Klasse `Janitor` von einer Tabelle pro Klassenhierarchie zu einer Tabelle pro Unterklasse, so dass die Mapping-Strategie mit der Mapping-Strategie der Klasse `Employee` übereinstimmt. Diese Änderung ist notwendig, da die Klasse `Janitor` in der Datenbank nach dem Refactoring nicht mehr durch die Tabelle `external_staff` repräsentiert wird. Stattdessen muss für die Klasse `Janitor` eine eigene Tabelle in der Datenbank definiert werden, damit die Klasse in die Vererbungshierarchie der Klasse `Employee` aufgenommen werden kann (vgl. Schritt 3).

3. SQL

- (a) Erstellen einer neuen Tabelle `janitors` mit einer Referenz auf die Tabelle `employees`
- (b) Kopieren der Spalten `company_name` und `address` in die Tabelle `janitors`
- (c) Anpassen aller SQL-Statements, die auf die Tabelle `external_staff` verweisen und einen Datensatz vom Typ `Janitor` ansprechen, so dass z.B. `INSERT`- und `UPDATE`-Statements die Tabellen `employees` und `janitors` modifizieren

Die Klasse `Janitor` besaß vor dem Refactoring keine eigene Tabelle zur Repräsentation, da deren Daten zuvor in der Tabelle `external_staff` gespeichert wurden. Diese Tabelle

Listing 4.16: Definition der Tabellen für die Klassenhierarchien der Klassen `Employee`, `ExternalStaff` und `Janitor`

```

1 CREATE TABLE employees(
2   id INTEGER PRIMARY KEY AUTOINCREMENT,
3   firstname varchar(255),
4   surname varchar(255),
5   salary real,
6   department integer REFERENCES departments(id)
7 );
8
9 CREATE TABLE external_staff(
10  id INTEGER PRIMARY KEY AUTOINCREMENT,
11  firstname varchar(255),
12  surname varchar(255),
13  salary real,
14  business varchar(255), — external jobs different from 'janitor'
15  company_name varchar(255),
16  address varchar(255)
17 );
18
19 CREATE TABLE janitors(
20  id integer REFERENCES employees(id) PRIMARY KEY,
21  company_name varchar(255),
22  address varchar(255)
23 );

```

Listing 4.17: Das `INSERT`-Statement zum Anlegen eines `Janitor`-Datensatzes nach dem Refactoring

```

1 — Janitors
2 INSERT INTO employees (firstname, surname, salary)
3   VALUES ('Gerald', 'Gemrich', 3850);
4
5 INSERT INTO janitors (id, company_name, address)
6   SELECT id, 'JanitorExpress', 'CompanyWay_10a'
7   FROM employees
8   WHERE surname = 'Gemrich';

```

muss daher zunächst in Schritt 3a definiert werden, da das Hibernate-Artefakt die Existenz einer eigenen Tabelle für jeden von der Klasse `Employee` abgeleiteten Typ vorsieht. Das Hinzufügen einer Tabelle ist eine Erweiterung des Datenbank-Schemas und daher kein Datenbank-Refactoring[Amb03, S. 178].

Durch die Einführung der zusätzlichen Tabelle `janitors` muss das `INSERT`-Statement in Listing 4.15 Zeile 26 auf die zwei in Listing 4.17 dargestellten Statements aufgeteilt werden. Um die Relation zwischen den Daten, die in Zeile 2 und in Zeile 5 eingefügt werden, herzustellen, wird im zweiten `INSERT`-Statement ein `SELECT` benötigt. Mit dem `SELECT`-Statement werden die in Zeile 2 eingefügten Daten referenziert, der dazugehörige eindeutige Identifikator `id` abgefragt und in die Tabelle `janitors` eingefügt. Dadurch wird die semantische Relation zwischen den Daten in Tabelle `employees` und `janitors` hergestellt.

Über die Analyse der Definitionen der Tabellen `employees`, `external_staff` und `janitors` ist es möglich, das in Listing 4.15 Zeile 26 dargestellte `INSERT`-Statement automatisiert in die in Listing 4.17 dargestellte Form zu überführen. Außerdem muss bekannt sein, dass sich die Tabellen `employees` und `janitors` in einer *Ist-ein*-Beziehung zueinander befinden. Erst durch die Berücksichtigung der *Ist-ein*-Beziehung kann über

die INSERT-Statements die Fremdschlüssel-Beziehung korrekt aufgebaut werden. Die Information darüber, dass die INSERT-Statements in Relation stehen und die Daten desselben Objekts manipulieren, ist in dem SQL-Artefakt nicht direkt nachvollziehbar. Wenn z.B. das eben beschriebene mehrsprachige Refactoring durch ein weiteres Refactoring rückgängig gemacht werden soll, ist eine Auswertung des Verhaltens der INSERT-Statements in Listing 4.17 notwendig. Erst dann kann eine Beziehung zwischen den beiden Statements aufgedeckt werden, denn Strukturell kann diese Beziehung in SQL nicht dargestellt werden.

Das MOVE CLASS REFACTORING kann in Bezug auf die Definition nicht als mehrsprachiges Refactoring angesehen werden, da das Hinzufügen einer neuen Tabelle zu einer Datenbank explizit als Refactoring ausgeschlossen wird.

4.2.2 SQL

Ausgehend von den SQL-Artefakten wurden folgende Refactorings umgesetzt:

- INTRODUCE DEFAULT VALUE[Amb03, S. 408],
- INTRODUCE REDUNDANT COLUMN[Amb03, S. 409],
- REMOVE TABLE[Amb03, S. 413].

Diese Refactorings wurden zufällig aus dem Katalog von Ambler in [Amb03] gewählt. Der Katalog in [Amb03] nennt zu den dort aufgelisteten Refactorings keine Vor- oder Nachbedingungen. Daher wird im Folgenden gefordert, dass die Vor- und Nachbedingungen der Refactorings *a priori* erkennbar und überprüfbar sein müssen.

Introduce Default Value

Das INTRODUCE DEFAULT VALUE REFACTORING dient der Einführung eines Standard-Wertes für eine Tabellen-Spalte. Dieser Wert wird immer dann verwendet, wenn ein neuer Datensatz in die Tabelle eingefügt wird, ohne dass für die entsprechende Spalte explizit ein Wert angegeben wurde. Durch dieses Refactoring können unterschiedliche Standard-Werte für eine Spalte, die in verschiedenen Anwendungen verwendet werden, durch einen Standard-Wert ersetzt werden[Amb03, S. 408]. Dazu muss sich auf einen für alle Anwendungen gültigen Standard-Wert verständigt werden.

In den Unternehmen werden leitende Angestellte grundsätzlich erst einmal einer allgemein gültigen Kostenstelle zugeordnet. Um diese Praxis auch in den Daten der Datenbank widerzuspiegeln, soll der Spalte `account` der Tabelle `managers` der Name der Kostenstelle als Standard-Wert zugeordnet werden.

Zur Umsetzung dieses Refactorings sind folgenden Schritte notwendig:

1. SQL
 - (a) Hinzufügen des Standard-Werts zur Definition der Spalte `account` der Tabelle `managers`
 - (b) Hinzufügen des Standard-Werts zur Definition des Attributs `account` in der Klasse `Manager`

Listing 4.18: Definition der Tabelle `managers` nach dem INTRODUCE DEFAULT VALUE REFACTORING

```

1 CREATE TABLE managers (
2   id integer references employees(id) primary key,
3   account varchar(255) CONSTRAINT std_account DEFAULT 'Akquise'
4 );

```

Listing 4.19: Definition des Attribut `account` der Klasse `Manager` nach dem INTRODUCE DEFAULT VALUE REFACTORING

```

1 private String account = "Akquise";

```

Listing 4.18 und 4.19 zeigen die Definition der Spalte bzw. des Attributs `account` in der Tabelle `managers` bzw. Klasse `Manager`. Zu den beiden Definitionen wurden jeweils Standard-Werte angegeben. Die Angabe des Standard-Wertes in der Java-Klasse muss erfolgen, da ohne die Angabe beim Instanzieren eines Objekts vom Typ `Manager` das Attribut `account` mit dem Wert `null` initialisiert wird. Dieser Wert wird dann vom ORM in die Datenbank geschrieben. Der in der Tabelle `managers` definierte Standard-Wert wird dabei überschrieben.

Bei der in Listing 4.19 dargestellten Initialisierung der Variable `account` handelt es sich um eine semantische Modifikation, da möglicherweise die Initialisierung der Variable `account` mit `null` von bestimmten Funktionen erwartet wird und eine Initialisierung mit dem Wert `Akquise` das erwartete Verhalten ändern würde.

In der Beispiel-Anwendung erfolgt der Zugriff auf die Attribute der vom ORM verwalteten Klassen über Getter- und Setter-Methoden. Das ORM hat keine Kenntnis über die Klassen-Attribute, die durch die Getter und Setter modifiziert werden. Dadurch ist es nicht möglich, einen direkten Zusammenhang zwischen Tabellen-Spalten und Klassen-Attributen über das ORM herzustellen. Im Fall des gezeigten Refactorings ist es also nicht möglich, direkt eine Verbindung zwischen der Spalte `account` der Datenbank-Tabelle `managers` und dem Attribut `account` der Klasse `Manager` herzustellen. Stattdessen müssen zunächst die entsprechenden Getter und Setter `getAccount` und `setAccount` gefunden und in Bezug zur Spalte `account` gebracht werden. Nach einer Analyse der Getter und Setter ist es schließlich möglich, das Attribut `account` der Klasse `Manager` in Verbindung mit der Spalte `account` der Tabelle `managers` zu bringen und den Standard-Wert zu setzen. Dazu muss über die Analyse ein Attribut gefunden werden, dass im Setter geschrieben und im Getter gelesen wird. Wenn die Getter und Setter über das einfache Lesen bzw. Schreiben eines Attributs hinausgehende Modifikationen an den gelesenen bzw. zu schreibenden Werten durchführen, ist zusätzlich eine semantische Analyse der Getter- und Setter-Methoden notwendig, um die semantisch korrekte Ausführung des Codes sicherzustellen.

Das folgende Beispiel soll das zuvor beschriebene Problem erläutern. Dazu wird angenommen, dass im `HRManager` neben dem eigentlichen Gehalt auch die Währung gespeichert wird, in dem das Gehalt ausgezahlt wird. Erhält ein Angestellter z.B. 3500 Euro, dann wird in der Datenbank die Zeichenfolge `3500EUR` gespeichert. In dem Setter `setSalary` der Klasse `Employee` werden alle Zeichenfolgen dieser Form getrennt und die Informationen in den Attributen `salary` für das Gehalt und `currency` für die

Listing 4.20: Methode `setSalary` zur Verarbeitung einer Zeichenkette mit Gehalts- und Währungsinformationen

```
1 public void setSalary(String salary) {
2     int length = salary.length();
3
4     this.salary = Float.parseFloat(salary.substring(0, length - 3));
5     this.currency = salary.substring(length - 3, length);
6 }
```

Listing 4.21: Methode `getSalary` zur Ausgabe einer Zeichenkette mit Gehalts- und Währungsinformationen

```
1 public String getSalary() {
2     return this.salary + this.currency;
3 }
```

Währung gespeichert. Das Listing 4.20 zeigt eine mögliche Implementierung der Methode `setSalary` zu Verarbeitung von Zeichenketten der Form `3500EUR`. Dazu wird in Zeile 4 der erste Teil der Zeichenkette zu einer Fließkommazahl umgewandelt. In Zeile 5 werden die letzten drei Symbole als Währungsinformation in einem eigenen Attribut gespeichert. Listing 4.21 zeigt den Getter zu der in Listing 4.20 dargestellten Setter-Methode. Die Methode in Listing 4.21 verknüpft in Zeile 2 die Gehalts- und Währungsinformation der Attribute `salary` und `currency` und gibt die dadurch entstandene Zeichenkette zurück. Es soll nun ein `INTRODUCE DEFAULT VALUE REFACTORING` auf diese Anwendung ausgeführt werden, um allen Angestellten standardmäßig ein Gehalt von 2500 Euro zuzuweisen. Dazu wird in der Datenbank die Zeichenkette `2500EUR` als Standard-Wert für die Spalte `salary` der Tabelle `managers` gesetzt. Um die globale semantische Integrität sicherzustellen, müssen die Attribute `salary` und `currency` die Werte 2500 respektive `EUR` erhalten. Eine Automatisierung dieser Modifikation erfordert die Analyse des Verhaltens der Methode `setSalary`. Ohne diese Analyse ist nicht ersichtlich, wie der neue Standard-Wert `2500EUR` verändert werden muss und welche Attribut-Initialisierungen angepasst werden müssen. Denn ein Trennen der Zeichenkette `2500EUR` in `25` und `00EUR` wäre u.U. auch möglich, aber semantisch nicht korrekt.

Introduce Redundant Column

Mit Hilfe des `INTRODUCE REDUNDANT COLUMN REFACTORINGS` wird eine Kopie einer festgelegten Spalte einer Quell-Tabelle erzeugt und einer Ziel-Tabelle hinzugefügt. Dieses Refactoring ist besonders dann von Nutzen, wenn auf die Spalte der Quell-Tabelle immer dann zugegriffen wird, wenn auch ein Zugriff auf die Ziel-Tabelle erfolgt. Durch die Kopie der Spalte kann direkt auf die entsprechenden Werte durch die Ziel-Tabelle zugegriffen werden. Zusätzliche Joins zwischen Quell- und Ziel-Tabelle werden eingespart und die Performance der Datenbank-Abfrage dadurch gesteigert.

Es wurde festgestellt, dass im `HRManager` ein Zugriff auf Daten der Angestellten auch immer mit der Abfrage der Abteilung einhergehen, in der die Angestellten tätig sind. Für die Abfrage ist nur der Name der Abteilung von Bedeutung. Daraus ergibt sich zwischen den Tabellen `employees` und `departments` die Möglichkeit, durch Hinzufügen einer zusätzlichen Spalte zu `employees` das `INTRODUCE REDUNDANT COLUMN RE-`

FACTORING nutzbringend anzuwenden. Dazu soll in der Tabelle `employees` die Spalte `departments_name` definiert werden, die eine Kopie des Wertes der Spalte `name` der Tabelle `departments` enthält. Der Name `departments_name` wurde gewählt, um leichter erkennbar zu machen, auf welchen Inhalt die neue Spalte verweist. Weiterhin wird bei der Beschreibung der Folgenden Modifikationen davon ausgegangen, dass sich die Verbesserung der Performance-Steigerung ebenfalls auf die Java-Artefakte auswirken soll.

Zur Umsetzung dieses Refactorings sind folgende Änderungen an der Beispiel-Anwendung notwendig:

1. SQL

- (a) Hinzufügen einer neuen Spalte `departments_name` zur Tabelle `employees` vom Typ der Spalte `name` der Tabelle `departments`
- (b) Hinzufügen von *Triggern* zur Erhaltung der Konsistenz zwischen den Spalten `departments_name` und `name` der Tabellen `employees` respektive `departments`. Listing 4.22 zeigt die Trigger, die zur Erhaltung der Daten-Konsistenz in der Datenbank verwendet werden. Der Trigger `populate_new_departments_name` (Zeile 1) aktualisiert den Abteilungsnamen in allen Mitarbeiter-Datensätzen, die den Datensatz einer Abteilung referenzieren, deren Namen geändert wurde. Der Trigger `add_departments_name_to_new_employee` (Zeile 11) schreibt den Abteilungsnamen in einen neu erstellten Mitarbeiter-Datensatz. Der letzte Trigger `change_departments_name_when_employee_updates` (Zeile 24) aktualisiert den Abteilungsnamen eines Mitarbeiter-Datensatzes, wenn sich die Referenz auf die Abteilung im Mitarbeiter-Datensatz ändert. Damit die genannten Trigger zur Erhaltung der Datenbank-Konsistenz ausreichen, werden weiterhin folgende Annahmen gemacht. Änderungen an der Spalte `departments_name` der Tabelle `employees` sind nur über die Trigger möglich. D.h., dass sowohl Nutzer, wie auch Anwendungsprogramme keine Veränderungen an der Spalte `departments_name` vornehmen dürfen, da sonst die Konsistenz der Daten nicht gewährleistet werden kann. Die Spalte `department` der Tabelle `employees` referenziert immer mindestens einen Datensatz aus der Tabelle `departments`. Damit werden Fehler bei der Ausführung der Trigger ausgeschlossen, die aufgrund einer fehlenden Referenz auf einen Datensatz in der Tabelle `departments` auftreten könnten.

2. Hibernate

- (a) Hinzufügen eines Attributs `departmentName` zur Klasse `Employee`
- (b) Hinzufügen einer Getter- und einer Setter-Methode für das Attribut `departmentName`
- (c) Annotieren der Getter-Methode, um die Verknüpfung zwischen dem Attribut `departmentName` der Klasse `Employee` und der Spalte `departments_name` der Tabelle `employees` herzustellen. Das Listing 4.23 zeigt den Getter `getDepartmentName` (Zeile 2) und den Setter `setDepartment` (Zeile 2). Die Methode `getDepartmentName` ist mit einer `@Column`-Annotation annotiert, die auf die Spalte `departments_name` in der Tabelle `employees` verweist.

Listing 4.22: Definition der *Trigger*, die zur Erhaltung der Datenbank-Konsistenz dienen

```
1 CREATE TRIGGER populate_new_department_name
2 AFTER
3 UPDATE OF name
4 ON departments
5 FOR EACH ROW
6 BEGIN
7 UPDATE employees SET departments_name = NEW.name
8 WHERE department = NEW.id;
9 END;
10
11 CREATE TRIGGER add_department_name_to_new_employee
12 AFTER
13 INSERT
14 ON employees
15 WHEN NEW.department IS NOT NULL
16 BEGIN
17 UPDATE employees
18 SET departments_name = (SELECT name
19 FROM departments
20 WHERE id = NEW.department)
21 WHERE employees.id = NEW.id;
22 END;
23
24 CREATE TRIGGER change_department_name_when_employee_updates
25 AFTER
26 UPDATE OF department
27 ON employees
28 FOR EACH ROW
29 BEGIN
30 UPDATE employees
31 SET departments_name = (SELECT name
32 FROM departments
33 WHERE id = employees.department)
34 WHERE employees.id = NEW.id;
35 END;
```

- (d) Erweiterung der Methode `setDepartment` der Klasse `Employee`, so dass eine Änderung des Attributs `department` auch eine Änderung des Attributs `departmentName` zur Folge hat. In Listing 4.23 Zeile 8 ist dargestellt, wie nach der Änderung des Attributs `department` der Name des Attributs `departments_name` aktualisiert wird.
- (e) Implementierung des Observer-Patterns [GHJV95, S. 293] zur Sicherstellung der Konsistenz zwischen Objekten der Klassen `Department` und `Employee`, so dass eine Änderung des Namens eines `Department`-Objekts auch zur Änderung des Attributs `departmentName` in den mit dem `Department`-Objekt verknüpften `Employee`-Objekten führt.

Eine Referenz auf die neue Tabellen-Spalte `departments_name` aus der Klasse `Employee` heraus ist nicht zwingend erforderlich. Der `HRManager` ist nach dem Hinzufügen der Spalte `departments_name` und der entsprechenden *Trigger* weiterhin nutzbar. Der durch das `INTRODUCE REDUNDANT COLUMN REFACTORING` erzielten Performance-Gewinn ist dann aber im Java-Artefakt nicht nutzbar.

Um von dem Performance-Gewinn dieses Refactorings in der Beispiel-Anwendung zu profitieren, muss die Klasse `Employee` ein Attribut zur Verfügung stellen, das die Spalte `departments_name` referenziert. Dadurch wird allerdings die Kapselung der Daten aufgeweicht, da die Klasse `Employee` dann sowohl die Daten des Angestellten als

Listing 4.23: Definition des Getters `getDepartment` und des Setters `setDepartment` in der Klasse `Employee`

```

1 @Column(name="departments_name")
2 public String getDepartmentName() {
3     return departmentName;
4 }
5
6 public void setDepartment(Department department) {
7     this.department = department;
8     this.departmentName = department.getName();
9 }

```

auch Teile der Daten der Klasse `Department` enthält. Darüber hinaus sind Fragen der Konsistenz-Erhaltung im Java-Artefakt bzgl. der Daten zu beantworten. In der Datenbank erfolgt die Konsistenz-Erhaltung über *Trigger*. Eine Änderung des Namens einer Abteilung in der Tabelle `departments` kann damit auch auf die entsprechenden Angestellten in der Tabelle `employees` übertragen werden. Wie in Schritt 2e beschrieben, ist im Java-Artefakt dazu die Implementierung neuer Strategien erforderlich.

Es ist möglich die globale semantische Integrität und Funktionalität der Anwendung zu erhalten, ohne die redundante Spalte in das ORM aufzunehmen. Als Problem der Automatisierung des `INTRODUCE REDUDANT COLUMN REFACTORINGS` verbleibt daher die Erstellung der entsprechenden *Trigger* zu Erhaltung der Daten-Konsistenz in der Datenbank.

Remove Table

Das `REMOVE TABLE REFACTORING` dient dem Entfernen einer nicht mehr benötigten Tabelle aus dem Datenbank-Schema.

Nach der Durchführung des `MOVE CLASS REFACTORINGS`, dass in Abschnitt 4.2.1 auf Seite 43 beschrieben wurde, wird die Tabelle `external_staff` nicht weiter verwendet. Daher soll die Tabelle aus dem Datenbank-Schema entfernt werden.

Zur Umsetzung dieses Refactorings sind folgende Schritte durchzuführen:

1. SQL

Entfernen der Definition der Tabelle `external_staff` aus dem SQL-Artefakt

2. Hibernate

- (a) Durchführung des `DELETE UNREFERENCED CLASS REFACTORINGS`[Opd92, S. 41], also entfernen der Definition der Klasse `ExternalStaff` aus dem Java-Artefakt¹⁵
- (b) Entfernen des Eintrags der Klasse `ExternalStaff` aus der Konfigurationsdatei des Hibernate-Mappings

¹⁵Tatsächlich wird in der Beispiel-Anwendung die Klasse `ExternalStaff` von der Funktion `sumSalary` (siehe Listing 4.9) referenziert. Da zu Beginn dieses Kapitels angenommen wurde, dass alle nötigen Vor- und Nachbedingungen erfüllt sind, wird diese Referenz als nicht vorhanden betrachtet.

Da die Klasse `ExternalStaff` Bestandteil des Hibernate-Mappings ist, die dazugehörige Tabelle `external_staff` aber entfernt wurde, muss auch die Klasse, wie in Schritt 2a beschrieben, entfernt werden. Da die Klasse dann nicht mehr existiert, muss auch die Referenz auf die Klasse aus der Konfiguration von Hibernate, wie in Schritt 2b beschrieben, gelöscht werden.

Da die Tabelle `external_staff` und die Klasse `ExternalStaff` an keiner weiteren Stelle im `HRManager` referenziert werden, sind keine weiteren Modifikationen notwendig. Mögliche Folgen einer dennoch bestehenden Referenz werden zu einem späteren Zeitpunkt genau diskutiert.

4.2.3 Clojure

Die folgenden Refactorings sind aus [Li06] entnommen und für die Sprache Haskell definiert. Da es sich bei Haskell ebenfalls um eine funktionale Programmiersprache handelt, wurden die Refactorings für die Bearbeitung des Clojure-Artefakts übernommen. Am Clojure-Artefakt wurden folgende Refactorings durchgeführt:

- `INTRODUCE NEW DEFINITION`[Li06, S. 18],
- `PROMOTE DEFINITION`[Li06, S. 16],
- `MOVE DEFINITION`[Li06, S. 20].

In [Li06] werden die Refactorings für Haskell in drei Kategorien eingeteilt:

- Structural (engl. Strukturell),
- Modul,
- Data-oriented (engl. Daten-orientiert).

Sowohl `INTRODUCE NEW DEFINITION` als auch `PROMOTE DEFINITION` gehören zu den strukturellen Refactorings. `Move Definition` gehört der Kategorie Modul Refactoring an. Da im Clojure-Artefakt der Beispiel-Anwendung keine eigenen Datentypen verwendet werden, wurde keine Betrachtung von Daten-orientierten Refactorings vorgenommen.

Die Refactorings werden zunächst an einem von der Anwendung unabhängigen Beispiel-Funktion `incr` betrachtet. Der kleine Umfang der Funktion `incr` dient dabei der leichteren Erläuterung der Refactorings. Die Funktion `incr` wird im `INTRODUCE NEW DEFINITION REFACTORING` eingeführt.

Introduce New Definition

Mit dem `INTRODUCE NEW DEFINITION REFACTORING` wird ein beliebiger Ausdruck in eine benannte Definition überführt. Im Folgenden soll dieses Refactoring an der in Listing 4.24 definierten Funktion `addOne` erläutert werden. In Zeile 3 befindet sich der Ausdruck `(+ (first x) 1)`, der zum ersten Element der Liste `x` 1 addiert. Dieser Ausdruck soll durch ein `INTRODUCE NEW DEFINITION REFACTORING` in den benannten Ausdruck `incr` überführt werden. Das Ergebnis zeigt Listing 4.25. In Zeile 4 des Listings

Listing 4.24: Die Funktion `addOne` addiert 1 auf alle Elemente der Liste `x` und gibt das Ergebnis als neue Liste zurück

```

1 (def addOne (fn [x] (if (empty? x)
2   nil
3   (cons (+ (first x) 1) (addOne (rest x))))))

```

Listing 4.25: Die Funktion `addOne` nach dem Hinzufügen der neuen Definition `incr`

```

1 (def addOne (fn [x] (if (empty? x)
2   nil
3   (cons
4     (letfn [(incr [x] (+ x 1))]
5       (incr (first x)))
6     (addOne (rest x))))))

```

4.25 wird die Funktion `incr` mit Hilfe des `letfn`-Statements definiert. Alle in einem `letfn`-Statement definierten Funktionen stehen in den Ausdrücken zur Verfügung, die das `letfn`-Statement umschließt. Im Beispiel besteht der Körper des in Zeile 4 angegebenen `letfn`-Statements gerade aus dem Ausdruck `(incr (first x))` in Zeile 5.

Das beschriebene Refactoring soll auf den Ausdruck `(instance? hrm.ExternalStaff x)` der in Listing 4.9 dargestellten Funktion `sumSalary` angewendet werden, um die Wiederverwendung des Ausdrucks in anderen Funktionen zu ermöglichen. Der Name der Funktion soll `isExternalStaff?` lauten. Um dieses Refactoring durchzuführen, sind folgende Schritte notwendig:

1. Clojure

- (a) Umschließen des Ausdrucks `(instance? hrm.ExternalStaff x)` mit einem `letfn`-Statement
- (b) Definition der Funktion `isExternalStaff?` mit dem gewählten Ausdruck in dem zuvor erstellten `letfn`-Statement
- (c) Ersetzen des vom `letfn`-Statement umschlossenen Ausdrucks `(instance? hrm.ExternalStaff x)` durch die neu definierte Funktion `isExternalStaff?`

Das Ergebnis des Refactorings zeigt das Listing 4.26. Die Zeile 7 zeigt das neu hinzugefügte `letfn`-Statement. Die darauf folgende Zeile enthält den Körper des Statements.

Die Funktion `isExternalStaff?` ist durch das `INTRODUCE NEW DEFINITION REFACTORING` nur für die vom `letfn`-Statement umschlossenen Ausdrücke sichtbar. Die Funktion konnte daher vorher nicht von anderen Artefakten referenziert werden. Damit werden keine Referenzen zu anderen Artefakten berührt und es sind keine weiteren Modifikationen notwendig.

Promote Definition

Das `PROMOTE DEFINITION REFACTORING` dient der Erhöhung der Sichtbarkeit einer Definition. Dadurch kann die Definition auch in anderen Bereichen des Codes verwendet

Listing 4.26: Die Funktion `sumSalary` nach der Durchführung des `INTRODUCE NEW DEFINITION REFACTORINGS`

```

1 (def sumSalary (fn [x] (if (not (instance? java.util.List x))
2   (throw (new java.lang.IllegalArgumentException "x is not a list
3     (if (and
4       (not (empty? x))
5       (and
6         (not
7           (letfn [(isExternalStaff? [x] (instance? hrm.
8             ExternalStaff x))]
9             (isExternalStaff? (first x))
10          )
11         (not (instance? hrm.Employee (first x)))
12        )
13       )
14      (throw (new java.lang.IllegalArgumentException))
15      (if (empty? x) 0
16          (+ (. (first x) getSalary) (sumSalary (rest x)))))))

```

Listing 4.27: Nach der Anwendung des `PROMOTE DEFINITION REFACTORINGS` steht die Funktion `incr` im höchsten Sichtbarkeitsbereich zur Verfügung

```

1 (def incr (fn [x] (+ x 1)))
2
3 (def addOne (fn [x] (if (empty? x)
4   nil
5   (cons
6     (incr (first x))
7     (addOne (rest x))))))

```

werden. Zur Erläuterung dieses Refactorings soll die in Listing 4.25 definierte Funktion `incr` dienen. Diese Funktion ist nur in den vom `letfn`-Statement umschlossenen Ausdrücken sichtbar. Um die Funktion `incr` auch in anderen Funktionen verwenden zu können, muss die Sichtbarkeit von `incr` erhöht werden. Durch die Anwendung des `PROMOTE DEFINITION REFACTORINGS` kann die Funktion `incr` in den Bereich höchster Sichtbarkeit verschoben werden. Das Ergebnis zeigt das Listing 4.27. In Zeile 1 wird die Funktion `incr` im globalen Skopus definiert und kann von anderen Funktionen aufgerufen werden. Das `letfn`-Statement wurde aus der Funktion `addOne` in Zeile 3 entfernt.

Im `HRManager` soll die zuvor durch das `INTRODUCE DEFINITION REFACTORING` in Listing 4.26 definierte Funktion `isExternalStaff?` in anderen Funktionen wiederverwendet werden. Dazu ist es nötig, dass die Sichtbarkeit der Definition von `isExternalStaff?` erhöht wird. Zur Umsetzung des Refactorings sind folgenden Schritte durchzuführen:

1. Clojure

- (a) Einfügen einer neuen Definition der Funktion `isExternalStaff?` in den Bereich höchster Sichtbarkeit
- (b) Verschieben des Körpers der im `letfn`-Statement definierten Funktion `isExternalStaff?` in die zuvor neu hinzugefügte Funktionsdefinition
- (c) Entfernen des `letfn`-Statements aus der Funktion `sumSalary`

Listing 4.28: Die Funktionen `isExternalStaff?` und `sumSalary` nach der Durchführung des PROMOTE DEFINITION REFACTORING

```

1 (def isExternalStaff? (fn [x] (instance? hrm.ExternalStaff x)))
2
3 (def sumSalary (fn [x] (if (not (instance? java.util.List x))
4     (throw (new java.lang.IllegalArgumentException "x is not a list
5         ")
6         (if (and
7             (not (empty? x))
8             (and
9                 (not (isExternalStaff? (first x)))
10                (not (instance? hrm.Employee (first x)))
11            )
12            )
13            (throw (new java.lang.IllegalArgumentException))
14            (if (empty? x) 0
                (+ (. (first x) getSalary) (sumSalary (rest x))))))))))

```

Listing 4.29: Definition der Funktionen `addOne` im Namespace `Compute`

```

1 (ns Compute)
2
3 (def addOne (fn [x] (if (empty? x)
4     nil
5     (cons
6         (+ (first x) 1)
7         (addOne (rest x))))))

```

Im Listing 4.28 ist das Ergebnis des PROMOTE DEFINITION REFACTORINGS aufgeführt. In Zeile 1 wird die Funktion `isExternalStaff?` im globalen Sichtbarkeitsbereich definiert. Die Funktion `sumSalary` in Zeile 3 kann nun auf die global definierte Funktion `isExternalStaff?` zugreifen. Eine eigene Definition ist nicht mehr notwendig, daher wurde das entsprechende `letfn`-Statement aus der Funktion `sumSalary` entfernt.

Darüber hinaus sind keine weiteren Modifikationen an der Beispiel-Anwendung notwendig, da das durchgeführte Refactoring andere Artefakte nicht berührt.

Move Definition

In Clojure können Funktionen in sogenannten *Namespaces* zusammengefasst werden [Van10, S. 24]. Durch Namespaces ist es möglich, Funktionen und Variablen-Definitionen zusammenzufassen. Außerdem können die gleichen Bezeichner in unterschiedlichen Namespaces verwendet werden, ohne durch die mehrfache Vergabe des Bezeichners in Konflikt zu stehen. Das Listing 4.29 zeigt die Funktion `addOne`, die im Namespace `Compute` definiert wird. Die Definition des Namespaces erfolgt in Zeile 1 durch Verwendung der Funktion `ns`.

Nicht immer steht eine Funktion im logischen Zusammenhang mit dem Namespace in dem sie definiert wurde. Durch Verschieben der Funktion in einen anderen Namespace kann die Struktur des Programms verbessert werden, wenn dadurch Funktionen zusammengeführt werden, die Aufgaben im selben Problembereich lösen. Das die Semantik erhaltende Verschieben von Funktionen wird durch das MOVE DEFINITION REFACTORING beschrieben.

Im HRManager ist die Funktion `managersWithBoss` im Namespace `salary` defi-

Listing 4.30: Erstellen der Referenz auf die Funktionen `managersWithBoss` im Namespace `salary`

```
1 Var managersWithBoss = RT.var("salary", "managersWithBoss");
```

Listing 4.31: Erstellen der Referenz auf die Funktionen `managersWithBoss` nach dem Verschieben in den Namespace `management`

```
1 Var managersWithBoss = RT.var("management", "managersWithBoss");
```

niert wurden. Der Namespace `salary` umfasst Funktionen zur Berechnung des Gehalts von Angestellten. Die Funktion `managersWithBoss` ermittelt aus einer Liste mit Instanzen der Klasse `Manager` diejenigen, die einen Vorgesetzten besitzen. Die Funktion `managersWithBoss` steht an sich nicht mit der Berechnung des Gehalts in Verbindung und soll daher von `salary` in den Namespace `management` verschoben werden, der Funktionen der Art von `managersWithBoss` zusammenfasst. Dazu sind folgende Veränderungen an der Beispiel-Anwendung notwendig:

1. Clojure

- (a) Kopieren der Funktionsdefinition `managersWithBoss` aus dem Namespace `salary`
- (b) Einfügen der Funktionsdefinition `managersWithBoss` in den Namespace `management`
- (c) Löschen der Funktionsdefinition `managersWithBoss` aus dem Namespace `salary`

2. Java

- (a) Änderung des Namespaces auf `management` bei der Referenzierung der Funktion `managersWithBoss`

Unter Schritt 1 sind keine weiteren Modifikationen mehr notwendig, da `managersWithBoss` nicht referenziert wird. Listing 4.30 zeigt die Referenzierung der Funktion `managersWithBoss` aus dem Java-Artefakt heraus vor der Durchführung des Schritts 2a. Der erste Parameter der Methode `var` ist der Namespace, in dem die durch den zweiten Parameter angegebene Funktion definiert wird. Das Listing 4.31 zeigt die Erstellung der Referenz auf die Funktion `managersWithBoss` nach der Durchführung des Schritts 2a. Der erste Parameter wurde entsprechend angepasst und zeigt auf den neuen Namespace `management`.

Die zur Umsetzung des Refactorings notwendigen Schritte weisen keine Besonderheiten auf, die einer Automatisierung des MOVE DEFINITION REFACTORINGS entgegen stehen.

4.2.4 Weitere Besonderheiten

In der Beispiel-Anwendung werden unterschiedliche Bezeichner in den Java- und SQL-Artefakten verwendet. Das ORM besitzt die notwendigen Informationen, um Relationen

zwischen den Tabellen in der Datenbank und den jeweiligen Java-Klassen herzustellen. Das bedeutet, dass aus dem Java- und SQL-Code alleine diese Referenzen nicht gefunden werden können. Erst über das Hibernate-Artefakt ist eine sichere Aussage über die Relationen zwischen Java-Klassen und Datenbank-Tabellen möglich, da bekannt ist, dass Hibernate Java- und SQL-Artefakte in Zusammenhang bringt. Als Beispiel seien die Klasse `Department` und die Tabelle `departments` genannt. Da die Bezeichner sich in gewisser Hinsicht gleichen, könnte daraus eine Relation zwischen der Klasse und der Tabelle abgeleitet werden. Tatsächlich existiert diese direkte Relation nicht. Erst über die `@Table`-Annotation des Hibernate-Artefakts werden die beiden Artefakte in Relation gebracht.

Die Durchführung eines Refactorings auf ein Artefakt einer mehrsprachigen Software-Anwendung kann die Modifikation weiterer Artefakte erfordern. Dabei kann nicht grundsätzlich angenommen werden, dass die zur Durchführung der Modifikationen notwendigen Vor- oder Nachbedingungen von allen betroffenen Software-Artefakten erfüllt werden. Die Erhaltung der globalen semantischen Integrität kann nicht garantiert werden, wenn Modifikationen nur an den Artefakten vorgenommen werden, die sowohl die Vor- wie auch Nachbedingungen erfüllen. Als Beispiel sei hier das `REMOVE TABLE REFACTORING` genannt, das in Abschnitt 4.2.2 auf Seite 54 auf die Tabelle `external_staff` angewendet wurde. Um die globale semantische Integrität zu erhalten, wurde zur Umsetzung des Refactorings die Java-Klasse `ExternalStaff` aus dem ORM entfernt und die Klasse selbst gelöscht. Das ist möglich, da die Klasse `ExternalStaff` vor der Durchführung des Refactorings in dem Java-Artefakt nicht direkt verwendet wurde. Es gelten also die nötigen Vor- und Nachbedingungen, um die Klasse `ExternalStaff` aus dem Java-Artefakt zu entfernen. Gelten die Vor- oder Nachbedingungen zum Entfernen der Klasse `ExternalStaff` nicht, darf die Klasse nicht aus dem Java-Artefakt entfernt werden. Unter diesen Bedingungen ist dann auch ein Entfernen der Klasse `ExternalStaff` aus dem ORM sowie ein Löschen der Tabellen-Definition von `external_staff` ebenfalls nicht möglich. Denn wird z.B. eine Instanz der Klasse `ExternalStaff` innerhalb der Anwendung aus der Datenbank abgerufen, kann der Objekt-relationale Mapper ohne die entsprechenden Informationen im ORM keine Referenz zu einer Tabelle herstellen. Es treten dann Laufzeitfehler auf. Daher ist die Erhaltung der globalen semantischen Integrität beim Entfernen der Tabelle `external_staff` nur möglich im Zusammenhang mit dem Entfernen der Klasse `ExternalStaff` aus den Java-Artefakten und dem ORM.

4.3 Zusammenfassung

Im Folgenden werden die Besonderheiten zusammengefasst und verallgemeinert, die bei der Durchführung der Refactorings in Abschnitt 4.2 gefunden wurden. Aus diesen Besonderheiten lassen sich die folgenden allgemeinen Aussagen ableiten:

1. Die durch ein Refactoring in einem Software-Artefakt angestoßene Modifikation eines anderen Software-Artefakts ist nicht in jedem Fall durch ein Refactoring beschreibbar.
 - (a) Ein Refactoring in einem Software-Artefakt kann zu semantischen Modifikationen in einem anderen Software-Artefakt führen.

- (b) Um eine Modifikation korrekt umzusetzen, werden möglicherweise Informationen über die Semantik von Software-Artefakten benötigt.
2. Ein Software-Artefakt kann weitere Refactorings an bereits modifizierten Artefakten anstoßen.
 3. Es sind u.U. Modifikationen an Tool-spezifischen Artefakten, wie Konfigurationsdateien, notwendig.
 4. Die Umsetzung eines mehrsprachigen Refactorings in einem Artefakt kann durch unterschiedliche Modifikationen realisiert werden. Die Modifikationen können sich voneinander in der Anzahl der Änderungen und der betroffenen Software-Artefakte unterscheiden.
 5. Ob zwischen unterschiedlichen Software-Artefakten Beziehungen bestehen, kann nicht ausschließlich aus den Bezeichnern geschlossen werden. Eine qualifizierte Aussage über die Beziehung zwischen zwei Artefakten kann erst durch die zusätzliche Betrachtung der jeweiligen Artefakt-Typen getroffen werden.
 6. Wenn ein Artefakt die Vor- oder Nachbedingungen eines Refactorings erfüllt, kann nicht geschlussfolgert werden, dass die Bedingungen für ein Refactoring eines in Beziehung stehenden Artefakt-Typs ebenfalls erfüllt werden.

Die Tabelle 4.1 bringt die allgemeinen Aussagen mit den jeweiligen Refactorings in Abschnitt 4.2 in Beziehung, so dass nachvollzogen werden kann, aus welchen konkreten Beispielen die Aussagen entwickelt wurden.

Aussage	RM	PUM	MC	IDV	IRC	RT	IND	PD	MD
	39	41	43	49	51	54	55	56	58
1	×	×	×		×				
1a		×	×	×	×				
1b		×	×	×					
2	×	×	×		×				
3						×			
4	×				×				
5	vgl. Abschnitt 4.2.4 Seite 59								
6	vgl. Abschnitt 4.2.4 Seite 60								

Tabelle 4.1: Die allgemeinen Aussagen verknüpft mit den jeweiligen Refactorings in Abschnitt 4.2. Unter den Abkürzungen der Refactorings sind die entsprechenden Seitenzahlen zum Nachschlagen angegeben.

Drei der allgemeinen Aussagen sollen im Folgenden ausführlicher diskutiert werden. In Punkt 1 wird zusammengefasst, dass ein Refactoring eines Software-Artefakts nicht immer durch ein weiteres Refactoring in einem anderen Software-Artefakt nachvollzogen werden kann. Es existiert also nicht zu jedem auf ein Software-Artefakt definierten

Refactoring ein entsprechendes Refactoring in einem anderen Software-Artefakt. In Abschnitt 4.2.1 auf Seite 39 wurde gezeigt, dass ein `RENAME METHOD REFACTORING` auf dem Java-Artefakt durch ein `RENAME COLUMN REFACTORING` im SQL-Artefakt nachvollzogen werden kann. Im Gegensatz dazu existiert nach besten Wissen zu dem in Abschnitt 4.2.1 auf Seite 41 durchgeführten `PULL UP METHOD REFACTORING` kein entsprechendes Datenbank-Refactoring, das eine entsprechende Modifikation für SQL-Artefakte beschreibt. Damit ist eine generische Umsetzung dieses Refactorings im Sinne des Begriffs Generic Refactoring nicht möglich (vgl. Abschnitt 3.2).

Der Punkt 1a widerspricht der Definition von Refactorings, die nur strukturelle Modifikationen an einem Artefakt gestatten. Das bedeutet, dass nicht unter allen Bedingungen die globale semantische Integrität durch ausschließliche Anwendung von Refactorings bzw. strukturelle Modifikationen auf die betroffenen Software-Artefakte gesichert werden kann. Bei der Durchführung der Refactorings in Abschnitt 4.2 waren semantische Modifikationen am SQL-Artefakt z.B. beim Pull Up Method, wie auch `MOVE CLASS REFACTORING` notwendig (vgl. Seite 41 und 43). Am Java-Artefakt wurden semantische Modifikationen z.B. bei der Umsetzung des `INTRODUCE DEFAULT VALUE` und `INTRODUCE REDUNDANT COLUMN REFACTORINGS` durchgeführt (vgl. Seite 49 und 51). Das zeigt, dass nicht nur der Artefakt-Typ bestimmt, ob semantische Modifikationen vorgenommen werden müssen. Wichtig dabei ist auch die Betrachtung des jeweiligen Refactorings, das umgesetzt werden soll.

Im Punkt 3 wird beschrieben, dass es zur Wiederherstellung der globalen semantischen Integrität möglicherweise notwendig ist, Tool-spezifische Artefakte zu modifizieren. Dieses Problem tritt bei der Durchführung des `REMOVE TABLE REFACTORINGS` in Abschnitt 4.2.2 auf, bei der die Modifikation der Hibernate-spezifischen Konfigurationsdatei notwendig ist. In [CJ08] wurde dieses Problem bereits für das `RENAME CLASS REFACTORING` erkannt und eine Lösung für die Frameworks Struts, Hibernate und Spring umgesetzt. Für die Implementierung automatisierter mehrsprachiger Refactorings bedeutet dies, dass spezifische Eigenschaften unterschiedlicher Software-Tools ebenfalls betrachtet werden müssen. Damit ist eine allgemeine Realisierung mehrsprachiger Refactorings nicht möglich, es müssen die Eigenheiten unterschiedlicher Software-Tools bei der Realisierung mit betrachtet werden.

Sowohl das `PULL UP METHOD` wie auch `MOVE CLASS REFACTORING` sind in SQL nicht durch strukturelle, Semantik-erhaltende Modifikationen darstellbar, da SQL keine Sprachmittel zur expliziten Darstellung Objekt-orientierter Vererbungsbeziehungen besitzt¹⁶. Das gleiche Problem besteht für das `INTRODUCE REDUNDANT COLUMN`, das in SQL über Trigger dargestellt werden kann. In Java existiert kein Sprachkonstrukt mit der Funktionalität von Triggern, wodurch diese durch funktionale Erweiterungen nachgebildet werden muss. Diese Beobachtungen legen die Vermutung nahe, dass immer dann semantische Modifikationen notwendig sind, wenn grundsätzliche Unterschiede hinsichtlich der Paradigmen oder zumindest der verwendeten Sprachkonstrukte der betroffenen Artefakt-Typen bestehen. So kennt SQL kein Äquivalent zur strukturellen Beschreibung von Klassenhierarchien, wie Java keine strukturelle Entsprechung für Datenbank-Trigger kennt. Inwiefern die genannte Vermutung zutreffend ist, kann im Rahmen der vorliegenden Arbeit nicht beantwortet werden und wird daher weiterführenden Untersuchungen

¹⁶Objekt-relationale Spracherweiterung, wie sie mit SQL:1999[OP04] eingeführt wurden, werden dabei nicht betrachtet.

als Fragestellung überlassen.

Eine Umsetzung mehrsprachiger Refactorings im Sinne sukzessiver Anwendung von Refactorings auf verschiedene, interagierende Software-Artefakte (vgl. Definition mehrsprachiges Refactoring in Abschnitt 2.3.2) ist, wie gezeigt wurde, im Allgemeinen nicht möglich. Eine Umsetzung mehrsprachiger Refactorings kann daher nur im Hinblick auf das zu realisierende Refactoring und der daran beteiligten Artefakt-Typen betrachtet werden.

Kapitel 5

Automatisierung mehrsprachiger Refactorings

Im vorherigen Kapitel wurden Refactorings auf ein Artefakt einer mehrsprachigen Software-Anwendung umgesetzt und durch weitere Modifikationen weiterer Software-Artefakte die globale semantische Integrität wiederhergestellt. Von den in Abschnitt 4.2 durchgeführten Modifikationen entsprechen nur das `METHOD RENAME REFACTORING` und das `REMOVE TABLE REFACTORING` den durch die Definition¹ in Abschnitt 2.3.2 gegebenen Anforderungen. Die Anforderungen besagen, dass ein mehrsprachiges Refactoring aus sukzessiv durchgeführten Refactorings bestehen muss. Dabei entsteht durch das Fehlen von Refactorings für Hibernate kein Widerspruch, denn die beim `METHOD RENAME` und `REMOVE TABLE REFACTORING` durchgeführten Modifikationen an den Hibernate-Artefakten können als Refactorings beschrieben werden. Eine Erläuterung dieser Refactorings erfolgt in diesem Kapitel.

Die Umsetzung der mehrsprachigen Refactorings ist Bestandteil einer Software-Anwendung zur automatisierten Anwendung von RFMs, genannt *RFMComposer*. Der *RFMComposer* wird im Rahmen der Dissertation von Kuhleemann entwickelt. Eine genaue Erläuterung der Architektur des *RFMComposer* erfolgt im Zusammenhang mit der Beschreibung der Integration der mehrsprachigen Refactorings.

Dieses Kapitel ist wie folgt gegliedert. Zunächst werden in Abschnitt 5.1 fehlende Hibernate-Refactorings definiert, die zur Umsetzung des `RENAME METHOD` und des `REMOVE TABLE REFACTORINGS` notwendig sind. In Abschnitt 5.2 erfolgt die Beschreibung der Software-Architektur, mit der das `RENAME METHOD` und `REMOVE TABLE REFACTORING` umgesetzt wurden. Abschnitt 5.3 beschreibt die Architektur des *RFMComposer*. Dazu wird ebenfalls die Integration der prototypisch implementierten mehrsprachigen Refactorings erläutert. Darauf folgt in Abschnitt 5.4 die Evaluation der implementierten und in den *RFMComposer* integrierten mehrsprachigen Refactorings. Abschließend werden die Ergebnisse dieses Kapitels im Abschnitt 5.5 zusammengefasst.

¹Ein `MEHRSPRACHIGES REFACTORING` ist ein Prozess, der ein Refactoring auf ein Software-Artefakt eines Typs anwendet und ggf. weitere Refactorings auf Software-Artefakte eines anderen Typs anstößt, um die globale semantische Integrität der Software-Artefakte zu erhalten.

5.1 Beschreibung fehlender Hibernate-Refactorings

Bei der Durchführung des `RENAME METHOD` und `REMOVE TABLE REFACTORINGS` in Abschnitt 4.2.1 Seite 39 bzw. Abschnitt 4.2.2 Seite 54 wurden Modifikationen am Hibernate-Artefakt vorgenommen. Diese Modifikationen haben die Semantik des HRManagers im Rahmen der durchgeführten mehrsprachigen Refactorings erhalten. Im Folgenden sollen die strukturellen Modifikationen beschrieben werden, um die Modifikationen am Hibernate-Artefakt als Refactorings einzuordnen.

Bei der Beschreibung der Refactorings wird angenommen, dass die Beschreibung des Objekt-relationalen Mappings, wie im Abschnitt 4.1.3 beschrieben, über Annotationen erfolgt.

5.1.1 Rename Target Column Refactoring

Das `RENAME TARGET COLUMN REFACTORING` findet Anwendung auf eine Methode, wenn:

- die Methode zu einer mit `@Entity` annotierten Klasse gehört,
- die Methode nicht mit `@Transient` annotiert ist, da die Methode dann nicht durch das ORM verwaltet wird [KS06, S. 82],
- die Methode oder die durch die Methode referenzierte Tabellen-Spalte umbenannt wurde.

Sind alle genannten Bedingungen erfüllt, kann das `RENAME TARGET COLUMN REFACTORING` angewendet werden. Das Refactoring wird durch die folgenden Modifikationen beschrieben:

optional:

Wenn die Methode noch keine `@Column`-Annotation besitzt, füge eine `@Column`-Annotation mit einem leeren `name`-Attribut in der Form `@Column(name="")` hinzu.

1. Setze den Wert des `name`-Attributs der `@Column`-Annotation auf den Namen der referenzierten Tabellen-Spalte.

5.1.2 Remove Class from ORM Refactoring

Das `REMOVE CLASS FROM ORM REFACTORING` wird angewendet, wenn:

- eine Tabelle durch ein `REMOVE TABLE REFACTORING` aus dem Datenbank-Schema entfernt wurde,
- die Tabelle über ein ORM durch eine Klasse referenziert wird,
- die referenzierende Klasse nicht mehr im Zusammenhang mit Funktionen des ORM genutzt wird.

Wenn eine Klasse über das ORM verwaltet wird und Funktionen des ORM nutzt, wie z.B. das Speichern einer Instanz der Klasse in der Datenbank, dann führt das Entfernen der Klasse aus dem Objekt-relationalen Mapping zu Laufzeitfehlern. In der Beschreibung des REMOVE TABLE REFACTORINGS in [Amb03, S. 413] wird explizit die Überarbeitung aller Anwendungen gefordert, die die Datenbank nutzen. Da in dieser Arbeit solche Modifikationen externer Anwendungen ausgeschlossen werden (vgl. Diskussion Abschnitt 2.3.1), wird bzgl. des REMOVE CLASS FROM ORM REFACTORINGS die Wiederherstellung der entfernten Datenbank-Tabelle gefordert. Erst wenn die Klasse keine vom ORM bereitgestellten Funktionen mehr nutzt, kann die semantisch und funktional korrekte Durchführung dieses Refactorings garantiert werden.

Sind alle Voraussetzungen erfüllt, ist die folgende Modifikation zur Umsetzung des Refactorings notwendig:

Entfernen des `<mapping></mapping>`-Eintrags der zu entfernenden Klasse aus der Hibernate-Konfigurationsdatei.

5.2 Architektur zur automatischen Umsetzung mehrsprachiger Refactorings

Zur Realisierung automatischer Refactorings müssen die folgenden Aufgaben umgesetzt werden:

1. Finden der Referenzen zwischen verschiedenen Artefakt-Typen einer mehrsprachigen Software-Anwendung,
2. Sukzessive Ausführung von Refactorings auf der mehrsprachigen Software-Anwendung.

Im Folgenden wird eine Architektur zur Umsetzung mehrsprachiger Refactorings beschrieben, die die genannten Punkte realisiert.

Zunächst müssen Referenzen zwischen den einzelnen Artefakt-Typen einer zu modifizierenden Software-Anwendung gefunden werden. Der Aufbau der Referenzen erfolgt auf Basis des von Moise und Wong in [MW05] vorgestellten 3-stufigen Prozesses (vgl. Abschnitt 3.1 Seite 20). Der Prozess wurde um die Nutzung von ASTs erweitert und enthält die folgenden Schritte:

1. Aufbau eines AST für jeden unterstützten Artefakt-Typ,
2. Extraktion der Entitäten aus den ASTs, die für den Aufbau von Relationen zwischen unterschiedlichen Artefakt-Typen von Bedeutung sind,
3. Herstellung einer Beziehung zwischen Entitäten verschiedener Artefakt-Typen, die miteinander in Relation stehen.

Dieses Vorgehen stellt die Abbildung 5.1 vereinfacht dar. Auf der linken Seite der Abbildung sind die unterschiedlichen Artefakt-Typen dargestellt. Mit einem Artefakt-spezifischen Parser wird der AST für den jeweiligen Artefakt-Typ aufgebaut. Für jeden

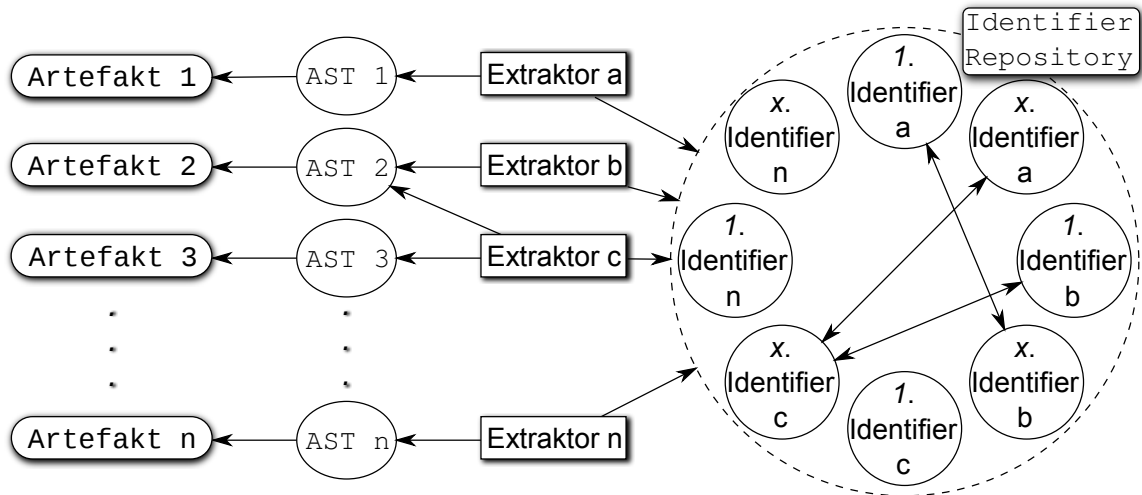


Abbildung 5.1: Extraktion der Referenzen zwischen Artefakten durch *Extraktoren*.
(Quelle: Eigene Darstellung)

unterstützten Artefakt-Typ steht ein sogenannter *Extraktor* zu Verfügung. Dieser extrahiert aus dem AST alle Entitäten, die durch andere Artefakt-Typen referenziert werden können. Aus dem Java-AST werden z.B. Klassen, Methoden und Attribute extrahiert, aus dem SQL-AST Tabellen und Spalten. In der dargestellten Architektur werden Entitäten durch *Identifier* gekapselt. Die Abbildung 5.1 stellt jeweils den *1.* bzw. *x-ten* Identifier jedes Artefakt-Typs dar. Spezifische Identifier existieren für jeden Extraktor. Identifier sind der Ausgangspunkt für die Umsetzung eines mehrsprachigen Refactorings und enthalten die folgenden Informationen:

1. Referenz auf die gekapselte Entität im jeweiligen AST,
2. Referenzen zu anderen Identifiern,
3. spezifische Informationen, die für den Aufbau einer Relation mit anderen Identifiern und zur Umsetzung mehrsprachiger Refactorings notwendig sind.

Identifier können miteinander verglichen werden. Über den Vergleich von zwei Identifiern wird geprüft, ob die Identifier miteinander in Beziehung stehen. Ist der Vergleich positiv, wird eine *bidirektionale* Relation zwischen den beiden Identifiern aufgebaut. Der Algorithmus 5.1 beschreibt das Vorgehen schematisch. Die in Zeile 3 genutzte Funktion `is related to` überprüft, ob zwei gegebene Identifier miteinander in Beziehung stehen. Die Implementierung der Funktion `is related to` erfolgt spezifisch für die Identifier jedes Artefakt-Typen. Der Vergleich erfolgt über die in den Identifiern enthaltenen Informationen. Eine allgemeine Implementierung ist nicht möglich, da verschiedene Artefakt-Typen auf unterschiedlichen Wegen miteinander interagieren. Es wird angenommen, dass nicht jeder Identifier Informationen darüber enthält, in welcher Weise eine Beziehung zu Identifiern anderer Artefakt-Typen besteht. Bezug nehmend auf die Abbildung 5.1 bedeutet dies beispielsweise, dass Identifier *1.* des Extraktors *b* keine Informationen darüber enthält, ob es mit Identifiern des Extraktors *c* in Beziehung stehen kann. Es genügt, wenn Identifier des Extraktors *c* Informationen über die Beziehung zu Identifiern des Typs *b* besitzen. Aus diesem Grund erfolgt die Prüfung, ob eine Relation

Algorithmus 5.1: Aufbau von Relationen zwischen den gefundenen Identifiern

```
1: for all Identifizier i1 in allIdentifiziers do
2:   for all Identifizier i2 in allIdentifiziers do
3:     if  $i1 \neq i2 \wedge \text{artefactOf}(i1) \neq \text{artefactOf}(i2)$ 
        $\wedge (i1 \text{ is related to } i2 \vee i2 \text{ is related to } i1)$  then
4:       create references between i1 and i2 if not already done
5:     end if
6:   end for
7: end for
```

zwischen zwei Identifiern besteht, in Zeile 3 des Algorithmus 5.1 jeweils aus der Sicht des Identifiers *i1* und *i2*. Des Weiteren wird in Zeile 3 mit Hilfe der Funktion `artefactOf` geprüft, ob zwei Identifier nicht aus dem gleichen Artefakt-Typ stammen. Da Identifier nur zur Darstellung der Beziehung zwischen unterschiedlichen Artefakt-Typen dienen, werden Referenzen zwischen gleichen Artefakt-Typen nicht aufgebaut. Für die Darstellung der Beziehungen innerhalb eines Artefakt-Typs werden die entsprechenden ASTs verwendet. Nachdem alle Identifier über die Funktion `is related to` in Zeile 3 miteinander verglichen und bei positiven Vergleich die entsprechenden Relationen in Zeile 4 aufgebaut wurden, steht ein Repository von miteinander in Beziehung stehenden Identifiern zur Verfügung.

Die Idee für den Aufbau eines Identifier-Repositories wurde aus [SKL06] entliehen, wo dazu allerdings ein allgemeines Meta-Modell beschrieben wird. Im Gegensatz zu der in [SKL06] beschriebenen Darstellung werden zwischen Identifiern keine typisierten Relationen verwendet, sprich die Relation zwischen zwei Identifiern sagt nichts über die Art der Relation aus (z.B. Hibernate-Annotation annotiert Java-Klasse). Dadurch ist eine Ergänzung weiterer Identifier möglich, ohne ein zugrunde liegendes Modell um gegebenenfalls neue Sprachelemente und Relationen erweitern zu müssen. Außerdem ist aus den Ergebnissen in [KWDE98] ersichtlich, dass schon die Unterstützung weniger Artefakt-Typen mit den spezifischen Relationen zu aufwendigen Meta-Modellen führen kann.

In der Architektur-Beschreibung wird von der Nutzung verschiedener, Artefakt-spezifischer ASTs ausgegangen. Für die Nutzung Artefakt-spezifischer ASTs spricht, dass für verschiedene Software-Artefakte bereits ausreichend getestete und bewährte ASTs zur Verfügung stehen. Wenn der AST zusätzlich Modifikationen an sich selbst zulässt, ist darüber die Beschreibung von Refactorings möglich. Durch die Verwendung eines AST wird eine aufwendige und fehleranfällige direkte Modifikation z.B. von Sourcecode vermieden. Gegen die Verwendung eines spezifischen AST spricht deren möglicherweise hohe Anzahl, da dadurch der Aufwand für Einarbeitung und Wartung erheblich steigt. Da eine Entwicklung eines eigenen Parsers inklusive eines allgemeinen AST für alle betrachteten Software-Artefakte den Rahmen dieser Arbeit verlassen würde, wurde sich für die Verwendung Artefakt-spezifische Parser mit den jeweiligen ASTs entschieden.

Die Idee des mehrsprachigen Refactorings besteht darin, sukzessiv Refactorings auf einzelne Artefakte anzuwenden. In der Architektur wird dazu eine abgewandelte Form des *detection-* und *propagation-*Konzepts umgesetzt[CJ08]. In der Architektur wird eine explizite Angabe des initialen Refactorings durch den Nutzer erwartet. Ein Erkennen des

initialen Refactorings (detection) muss somit nicht erfolgen.

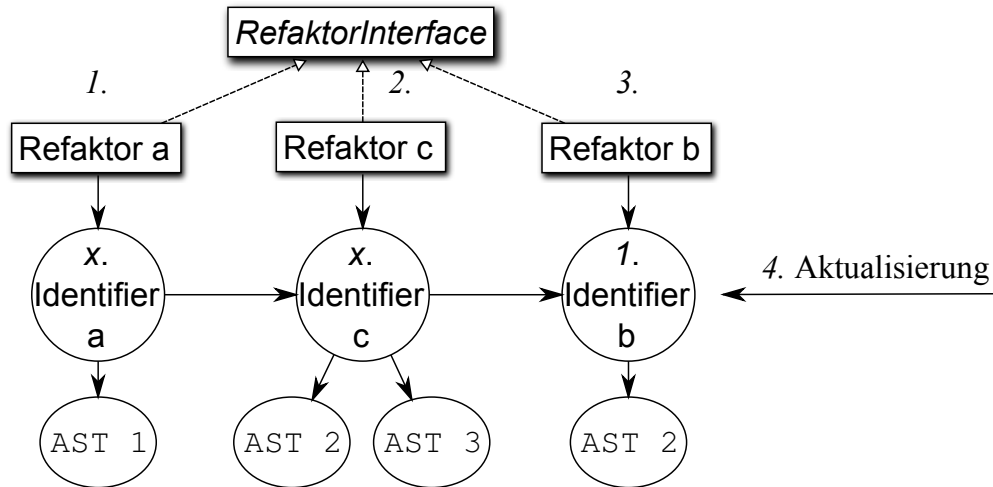


Abbildung 5.2: Schema der Umsetzung eines mehrsprachigen Refactorings durch *Refaktoren*. (Quelle: Eigene Darstellung)

Die Abbildung 5.2 stellt das Vorgehen bei der Durchführung eines mehrsprachigen Refactorings grafisch dar. Über das initiale Refactoring am Identifier *x.* des Artefakts *a* (1.) werden weitere Refactorings in anderen Artefakt-Typen angestoßen. Die Umsetzung der Refactorings erfolgt über sogenannte *Refaktoren*. Dazu greift der Refaktor über den Identifier auf den AST des Artefakts zu, auf dem die Modifikation durchgeführt wird. Ein Refaktor implementiert die im *RefaktorInterface* deklarierten Refactorings für einen spezifischen Artefakt-Typ. Durch den Refaktor wird außerdem entschieden, welche der vom Identifier referenzierten Entitäten ebenfalls modifiziert werden müssen. Durch die Möglichkeit der Auswahl der nächsten zu modifizierenden Identifier, kann der Refaktor unnötige Refactorings weiterer Entitäten verhindern. In der Abbildung erfolgt in den Schritten *zwei* und *drei* bei *Refaktor c* bzw. *Refaktor b* die Modifikation der mit dem initialen Identifier direkt oder indirekt in Beziehung stehenden Artefakte. Im *vierten* Schritt werden die in den Identifiern enthaltenen Informationen aktualisiert, da diese Informationen durch die Umsetzung des mehrsprachigen Refactorings möglicherweise nicht mehr mit den in den ASTs enthaltenen Informationen übereinstimmen.

Der Algorithmus 5.2 zeigt die algorithmische Idee zur Umsetzung des mehrsprachigen Refactorings. Zunächst wird in Zeile 1 über die Funktion `getInitialIdentifier` der initiale Identifier der Variable *i* zugewiesen. Die Funktion `refactor` (siehe Zeile 2 und 7) übernimmt die Aufgabe, den spezifischen Refaktor des Identifiers *i* zu finden. Daneben repräsentiert die Funktion `refactor` auch das Refactoring, das umgesetzt werden soll. Die Funktion `refactor` muss demnach für jeden Artefakt-Typ spezifisch implementiert werden, daher wurde die Definition der Funktion `refactor` im Algorithmus 5.2 nicht aufgeführt. Der Funktion `refactor` wird neben dem zu modifizierenden Identifier auch eine Liste der bereits besuchten Identifier `visitedIdentifier` übergeben. Über die Liste erhält das Refactoring Kontext-Informationen des mehrsprachigen Refactorings. Mit Hilfe der Kontext-Informationen der Liste `visitedIdentifier` ist in der Funktion `refactor` ersichtlich, welche Identifier bereits besucht wurden. Innerhalb der Funktion kann dann mit Hilfe der Liste entschieden werden, welche Identifier noch modifiziert werden müssen. Da zu Beginn keine Identifier modifiziert wurden, wird in Zeile 2 eine leere Menge überge-

Algorithmus 5.2: Algorithmische Umsetzung des mehrsprachigen Refactorings

```

1: Identifizier  $i \leftarrow \text{getInitialIdentifizier}()$ 
2: referencedIdentifiziers  $\leftarrow \text{refactor}(i, \emptyset)$ 
3:  $\text{push}(\text{identifizierQueue}, \text{referencedIdentifiziers})$ 
4:  $\text{add}(\text{visitedIdentifiziers}, i)$ 
5: while identifizierQueue  $\neq \emptyset$  do
6:    $i \leftarrow \text{pop}(\text{identifizierQueue})$ 
7:   referencedIdentifiziers  $\leftarrow \text{refactor}(i, \text{visitedIdentifiziers})$ 
8:    $\text{push}(\text{identifizierQueue}, \text{referencedIdentifiziers})$ 
9:    $\text{add}(\text{visitedIdentifiziers}, i)$ 
10: end while
11:  $\text{refresh}(\text{visitedIdentifiziers})$ 

```

ben. Die Refactorings werden somit nur weitergeleitet (*propagation*) und nicht erkannt (*detection*). Die Funktion `refactor` gibt an die Variable `referencedIdentifiziers` eine Liste der Identifier zurück, die noch besucht werden müssen. Die zuvor in der Liste `referencedIdentifiziers` referenzierten Identifier werden dabei gelöscht.

5.2.1 Implementierung der Architektur

In Abschnitt 4.2 wurden die Modifikationen an Java-, Hibernate-, SQL- und Clojure-Artefakten vorgenommen. Bei der Implementierung des METHOD RENAME und REMOVE TABLE REFACTORINGS wird das Clojure-Artefakt aufgrund des Fehlens eines entsprechenden AST nicht mit betrachtet. Für die weiteren Artefakte wurden die in Tabelle 5.1 aufgelisteten Parser verwendet. Aus der Tabelle 5.1 ist ersichtlich, dass für das Hibernate-Artefakt sowohl ein Java-Parser wie auch ein XML-Parser verwendet werden. Die Verwendung der zwei Parser ist notwendig, da sowohl die Annotationen im Java-Artefakt wie auch die Hibernate-Konfigurationsdatei möglicherweise bei einem Refactoring angepasst werden müssen.

Artefakt	Parser
Java	JastAddJ ¹
Hibernate	JastAddJ und Xerces2 ² (XML)
SQL	JSqlParser ³

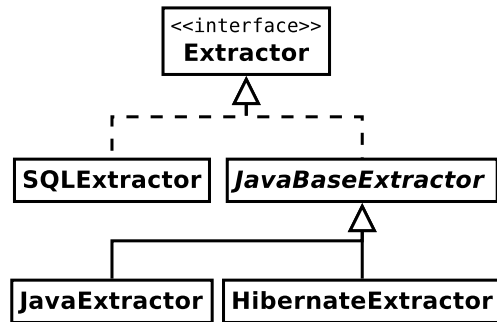
¹ <http://jastadd.org/>

² <http://xerces.apache.org/xerces2-j/>

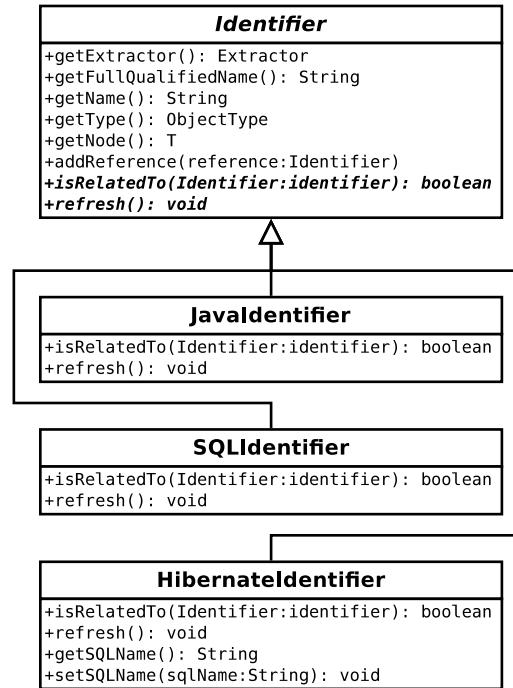
³ <http://jsqlparser.sourceforge.net/>

Tabelle 5.1: Die in der Implementierung verwendeten Parser

Zunächst soll die Implementierung der Extraktoren und der Identifier sowie der Aufbau des Identifier-Repositories erläutert werden. Zur Extraktion der spezifischen Identifier stehen die Klassen `JavaExtractor`, `HibernateExtractor` und `SQLExtractor` zur Verfügung. Diese Klassen implementieren das `Extractor`-Interface, um einen einheitlichen Zugriff auf die verschiedenen Parser zu ermöglichen. Die Abbildung 5.3(a) zeigt das Interface `Extractor` sowie die implementierenden Klassen. In der Abbildung ist ebenfalls



(a) Das Interface `Extractor` und die das Interface implementierenden Klassen. (Quelle: Eigene Darstellung)



(b) Die abstrakte Klasse `Identifier` und die davon abgeleiteten Klassen. (Quelle: Eigene Darstellung)

Abbildung 5.3: Klassen zum Aufbau eines Identifier-Repositories

eine abstrakte Klasse `JavaBaseExtractor` aufgeführt. Die Klasse `JavaBaseExtractor` vereinheitlicht den Aufbau des Java-AST, da sowohl die Klasse `JavaExtractor` wie auch `HibernateExtractor` auf dem selben AST arbeiten.

Jeder Extraktor erstellt mit Hilfe des AST Identifier für die zur Bildung von Artefakt-übergreifenden Relationen benötigten Entitäten. Jeder Extraktor definiert dazu einen eigenen Identifier, der von der abstrakten Klasse `Identifier` abgeleitet werden muss. Ein Identifier definiert verschiedene Methoden, die für den Aufbau des Identifier-Repositories und für die Durchführung von Refactorings von Bedeutung sind:

getExtractor: Gibt den Extraktor zurück, der den Identifier erstellt hat.

getFullQualifiedName: Gibt den voll-qualifizierten Namen der referenzierten Entität zurück. Der voll-qualifizierte Name enthält z.B. den Namen eines Namespaces oder Packages.

getName: Gibt den Namen der referenzierten Entität zurück, ohne Informationen eines Namespaces oder Packages.

getType: Gibt den Typen der Entität zurück, die durch den Identifier gekapselt wird. Typen dienen der Klassifikation der Entität des jeweiligen Identifiers. Die Klassifikation dient im Zusammenhang der Methode `getExtractor` dazu, ein Mindestmaß an Typ-Information zu Verfügung zu stellen.

getNode: Gibt eine Referenz auf die Entität im jeweiligen AST zurück.

<i>Objekt-Typ</i>	Java	Hibernate	SQL
<i>Object</i>	Klasse	Annotiert eine Klasse	Tabelle
<i>Method</i>	Methode	Annotiert eine Methode	
<i>Attribute</i>	Attribut	Annotiert ein Attribut	Spalte

Tabelle 5.2: Objekt-Typen der Entitäten verschiedener Artefakte

relatedTo: Vergleicht zwei Identifier miteinander und überprüft, ob diese miteinander in Beziehung stehen. Die Methode `relatedTo` ist abstrakt definiert, damit eine spezifische Implementierung bezüglich des jeweiligen Artefakts erfolgen muss.

refresh: Synchronisiert die Informationen im Identifier mit den Eigenschaften der gekapselten Entität. Die Methode `refresh` ist abstrakt definiert, da zur Aktualisierung der Daten der Zugriff auf den Artefakt-spezifischen AST notwendig ist.

Darüber hinaus definiert die Klasse `HibernateIdentifier` die Methoden:

setSQL und getSQL: Setzen oder geben den Namen der Tabelle oder Spalte zurück, die über das Hibernate-Artefakt referenziert wird. Die Spalten- oder Tabellennamen sind notwendig, um die Referenz zwischen Java- und SQL-Artefakt herzustellen. Das Java-Artefakt wird bereits in der Klasse `HibernateIdentifier` über die zuvor beschriebenen Methoden referenziert, daher sind die zusätzlichen Methoden `setSQL` und `getSQL` notwendig.

Die durch die Methoden zurückgegebenen Informationen sind notwendig, da in der Architektur verschiedene Parser und ASTs verwendet werden. Ein Nachteil bei der Verwendung verschiedener Parser bzw. ASTs ist, dass ein Identifier eines Artefakt-Typs nicht ohne Kenntnis der im Identifier eines anderen Artefakt-Typs verwendeten Parser und ASTs auf die Informationen im AST des anderen Identifier zugreifen kann. Ohne diese Informationen, wie z.B. Methoden oder Klassennamen, ist es nicht möglich zu entscheiden, ob zwei Identifier unterschiedlicher Extraktoren zueinander in Beziehung stehen. Daher werden von den Extraktoren die dazu nötigen Informationen im Identifier hinterlegt. Neben dem Namen und dem Artefakt-Typ wird im Identifier auch ein sogenannter *Objekt-Typ* gespeichert. Der Objekt-Typ gibt die Art der vom Identifier gekapselten Entität an. Die Tabelle 5.2 fasst die für die prototypische Umsetzung der mehrsprachigen Refactorings benötigten Objekt-Typen und die durch den Objekt-Typ im jeweiligen Artefakt repräsentierte Entität zusammen.

Die Realisierung der Methode `relatedTo` soll an dem Vergleich zwischen `Hibernate`- und `SQLIdentifier` beispielhaft erläutert werden. Das Listing 5.1 zeigt einen Ausschnitt aus der Methode `relatedTo` der Klasse `HibernateIdentifier`. Zunächst wird in der `if`-Anweisung in Zeile 2 geprüft, ob es sich bei dem übergebenen Identifier `identifier` um einen `SQLIdentifier` handelt. Des Weiteren wird mit `this.getType() == ObjectType.Method` bzw. `identifier.getType() == ObjectType.Attribute` überprüft, ob der `HibernateIdentifier` `this` eine Methode annotiert bzw. der `SQLIdentifier` `identifier` eine Tabellen-Spalte repräsentiert (vgl. Tabelle 5.2). Werden in der `if`-Anweisung alle Bedingungen erfüllt, wird in den Zeilen 6 bis 8 der Tabellen- und Spalten-Name aus dem `SQLIdentifier` extrahiert. Zur Realisierung eines ORM, muss zunächst die entsprechende Java-Klasse

Listing 5.1: Ausschnitt aus der Implementierung der Methode `relatedTo` am Beispiel des Hibernate-Identifiers

```

1 private boolean relatedTo(Identifier<?> identifier) {
2     if (identifier.getExtractor().getID().equals("SQL")
3         && this.getType() == ObjectType.Method
4         && identifier.getType() == ObjectType.Attribute) {
5
6         String fullQualifiedName2 = identifier.getFullQualifiedName();
7         String tableName2 = fullQualifiedName2.substring(0, fullQualifiedName2.lastIndexOf(
8             '.'));
9         String columnName2 = identifier.getName();
10
11        if (((JPAIdentifier)this.getParent()).sqlName.equals(tableName2)
12            && this.sqlName.equals(columnName2)) {
13            return true;
14        }
15    }
16    return false;
17 }

```

mit einer `@Entity`-Annotation versehen sein (vgl. Abschnitt 4.1.3). In der letzten `if`-Anweisung in Zeile 10 wird überprüft, ob die Klasse über das Default Mapping oder über eine `@Table`-Annotation auf die Tabelle verweist, die die Tabellen-Spalte definiert, die durch `identifier` repräsentiert wird². Des Weiteren wird überprüft, ob die Bezeichnung der Methode mit der des Spalten-Namens übereinstimmt. Sind alle Bedingungen der letzten `if`-Anweisung erfüllt, besteht eine Beziehung zwischen dem Hibernate- und SQLIdentifier.

Die Umsetzung des RENAME METHOD und REMOVE TABLE REFACTORINGS erfolgt durch die Klassen `JavaRefactoring`, `HibernateRefactoring` und `SQLRefactoring`. Diese Klassen repräsentieren die Refaktoren, die bei der Beschreibung der Architektur vorgestellt wurden. Die Refaktoren implementieren das `MLRRefactoring`-Interface. Das Interface definiert zwei Methoden, `rename` und `remove`. An die Methoden wird der zu modifizierende Identifier `identifier` und eine Menge der bereits modifizierten Identifier `alreadyVisited` übergeben. Zusätzlich wird an die Methode `rename` der neue Name übergeben, den die zu modifizierende Entität nach dem Refactoring tragen soll. Abbildung 5.4 stellt das Interface `MLRRefactoring` und die implementierenden Klassen grafisch dar.

Die Implementierung des eigentlichen Refactorings soll anhand des RENAME METHOD REFACTORINGS erläutert werden. Listing 5.2 zeigt die Umsetzung des Algorithmus 5.2. In Zeile 3 wird zunächst über die Klasse `RefactoringFactory` der spezifische Refaktor zum Artefakt-Typ des `initialIdentifiers` erstellt. In Zeile 6 erfolgt das initiale Refactoring über den Aufruf der Methode `rename`. Als letzter Parameter wird an `rename` `null` übergeben, da bis zu diesem Zeitpunkt kein Identifier modifiziert wurde. Die Referenzen zu weiteren zu modifizierenden Identifiern wird durch die Methode `rename` als Rückgabewert ebenfalls in Zeile 6 an `list` übergeben. In Zeile 10 wird der erste modifizierte Identifier als besucht vermerkt. Ab Zeile 12 wird das eben beschrie-

²Die in Listing 5.1 verwendete Methode `getParent` gibt den Identifier der übergeordneten Entität zurück. In SQL ist die übergeordnete Entität einer Spalte eine Tabelle. In Java ist eine Klasse die übergeordnete Entität einer Methode. Diese Strukturierung wird auch bei dem `HibernateIdentifier` genutzt.

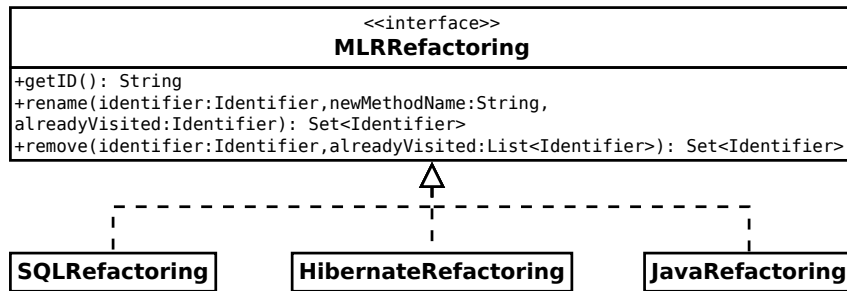


Abbildung 5.4: Das Refactor-Interface mit den implementierenden Klassen. (Quelle: Eigene Darstellung)

Listing 5.2: Implementierung des Algorithmus 5.2 zur Durchführung des RENAME METHOD REFACTORING

```

1 LinkedList<Identifier<?>> visited = new LinkedList<Identifier<?>>();
2
3 Refactoring<?> refactoring = RefactoringFactory.getRefactoring(initialIdentifier);
4
5 Queue<Identifier<?>> list
6   = new LinkedList<Identifier<?>>(refactoring.rename(initialIdentifier,
7                                           newMethodName,
8                                           null));
9
10 visited.add(initialIdentifier);
11
12 while(!list.isEmpty()) {
13     Identifier<?> i = list.poll();
14
15     if (visited.contains(i))
16         continue;
17
18     refactoring = RefactoringFactory.getRefactoring(i);
19     list.addAll(new LinkedList<Identifier<?>>(refactoring.rename(i,
20                                                         newMethodName,
21                                                         visited)));
22
23     visited.addFirst(i);
24 }
  
```

bene Vorgehen für alle Elemente in `list` wiederholt. Da die Implementierungen der im `MLRRefactoring`-Interface deklarierten Methoden alle Identifier zurückgeben dürfen, mit denen der modifizierte Identifier in Beziehung steht, können aufgrund der bidirektionalen Beziehungen zwischen den Identifiern Zyklen entstehen, die in einer unendlichen langen Programmausführung münden. Daher werden in Zeile 15 bereits modifizierte Identifier übersprungen. Zusätzliche Refactorings sind über die Liste `visited` der bereits modifizierten Identifier möglich. Die Liste `list` wird in Zeile 19 an die Methode `rename` übergeben.

Aus dem in Listing 5.2 dargestellten Sourcecode ist die konkrete Umsetzung für ein RENAME METHOD REFACTORING auf eine Java-Methode nicht direkt ersichtlich. Die Umsetzung erfolgt durch die Angabe der zu modifizierenden Java-Methode als initialen Identifier.

In der Implementierung werden JPA-Artefakte mit den Hibernate-Artefakten zusammen im Hibernate-Extraktor bzw. -Refaktor betrachtet. Im Bezug auf den HRManager ist die gemeinsame Betrachtung zweckmäßig, da für die prototypische Implemen-

tierung keine anderen Objekt-relationalen Mapper außer Hibernate betrachtet werden. Für die Unterstützung weiterer Objekt-relationaler Mapper ist es aber von Vorteil, JPA-spezifische Artefakte von denen spezifischer Objekt-relationaler Mapper, wie Hibernate, zu trennen. Durch die Trennung wird eine bessere Erweiterbarkeit der unterstützten Artefakte erreicht.

5.2.2 Diskussion zur Erweiterbarkeit der Architektur

Auf Basis der vorgestellten Architektur wurden das `RENAME METHOD` sowie auch das `REMOVE TABLE REFACTORING` implementiert. In diesem Abschnitt wird die Erweiterbarkeit der Architektur hinsichtlich der Ergänzung weiterer Artefakte und Refactorings diskutiert.

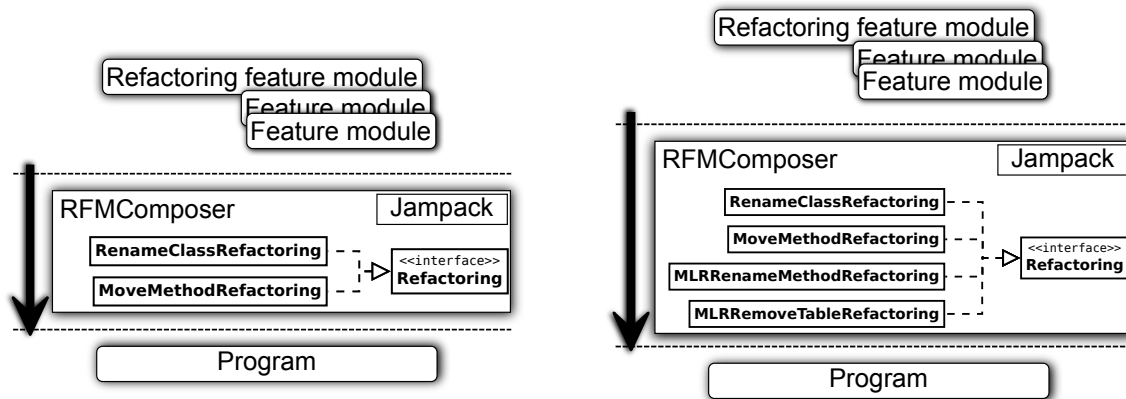
Durch ein neues Software-Artefakt werden möglicherweise weitere Sprachkonstrukte eingeführt, die für den Aufbau von Beziehungen zwischen verschiedenen Artefakt-Typen relevant sind. Für die Unterstützung der neuen Sprachkonstrukte ist dann die Erweiterung der in Tabelle 5.2 definierten Objekt-Typen notwendig. Mit der Einführung neuer Objekt-Typen müssen möglicherweise auch bereits bestehende Extraktoren angepasst werden, um bisher nicht extrahierte, aber für den Aufbau von Beziehungen zwischen unterschiedlichen Artefakt-Typen notwendige Informationen zur Verfügung zu stellen. Ein Beispiel hierfür ist die Interaktion zwischen Clojure und Java. Mit Hilfe des Aufrufs der Methode `var sumSalary = RT.var("scripting", "sumSalary")` wird aus Java heraus die in Clojure definierte Funktion `sumSalary` referenziert. (vgl. Listing 4.10 Zeile 2). Der Aufbau einer Referenz zwischen dem Java- und Clojure-Artefakt ist damit nur über die Kenntnis des Methodenaufrufs `RT.var()` möglich. Es müssten daher die in Tabelle 5.2 genannten Objekt-Typen durch einen weiteren Objekt-Typ *MethodCall* erweitert werden.

5.3 Integration mehrsprachiger Refactorings in den RFMComposer

In [KBA09] wird das Konzept des Refactoring Feature Modules (RFM) vorgestellt (vgl. Abschnitt 2.6). Neben dem eigentlichen Konzept wird auch eine Implementierung zur Komposition von RFMs beschrieben, die im weiteren als *RFMComposer* bezeichnet wird. Der RFMComposer implementiert RFMs auf Basis von *Jak*. *Jak* ist eine Erweiterung der Programmiersprache Java, um Feature-Komposition (vgl. Abschnitt 2.5) innerhalb von Java zu unterstützen [BSR03]. Mit Hilfe des Tools *jampack* aus der AHEAD Tool Suite³ setzt der RFMComposer Feature zusammen.

Die Abbildung 5.5(a) stellt die Struktur des RFMComposer grafisch dar. Das Kernstück des RFMComposers ist der *Composer*, der mit Hilfe von *jampack* die Feature Module komponiert. Trifft der Composer auf ein RFM, werden nach der Zusammensetzung der Features die im RFM definierten Refactorings auf die bereits komponierten Artefakte angewendet. Dazu lädt der Composer über ein Plug-In-System die vom RFM definierten Refactorings. Die Refactorings implementieren ein gemeinsames Interface *Refactoring*.

³<http://userweb.cs.utexas.edu/users/schwartz/ATS.html>



(a) Architektur des RFMComposers (Quelle: Eigene Darstellung auf Basis von [KBA09])

(b) Architektur des RFMComposers nach der Einbindung der mehrsprachigen Refactorings. (Quelle: Eigene Darstellung)

Abbildung 5.5: Die ursprüngliche Architektur des RFMComposers (5.5(a)) und nach der Einbindung der mehrsprachigen Refactorings (5.5(b)).

Die Integration der mehrsprachigen Refactorings erfolgt über das Plug-In-System des RFMComposers. Die Klassen `MLRRenameMethodRefactoring` und `MLRRemoveTableRefactoring` implementieren das `Refactoring`-Interface des RFMComposers. Die Klassen implementieren jeweils den Aufbau des Identifier-Repositories und die Durchführung des eigentlichen mehrsprachigen Refactorings.

Der RFMComposer wendet die in den RFMs definierten Refactorings nicht auf die Feature Module direkt an. Stattdessen werden die Refactorings durch den RFMComposer auf das Ergebnis der Komposition der Feature Module angewendet. Ein Vorteil der Anwendung der in den RFMs beschriebenen Refactorings auf komponierte Features liegt darin, dass keine spezielle Betrachtung von Refactorings über Features erfolgen muss. Damit können mehrsprachige Refactorings im Kontext von RFMs unabhängig von den durch FOP eingeführten Konzepten betrachtet werden.

5.4 Evaluation der mehrsprachigen Refactorings

Die Evaluation der im RFMComposer integrierten mehrsprachigen Refactorings erfolgt im Folgenden anhand des HRManagers, des Tools Seam-gen und verschiedener Anwendungsbeispiele aus dem RIA Framework JBoss Seam. Seam-gen sowie die Anwendungsbeispiele nutzen das JPA bzw. Hibernate zur Realisierung des ORM und eignen sich damit zur Evaluierung der implementierten mehrsprachigen Refactorings. Neben dem HRManager wurde im Rahmen dieser Arbeit keine Software-Anwendungen oder Fallbeispiele zur weiteren Evaluierung des REMOVE TABLE REFACTORINGS gefunden. Die dazu nötigen CREATE TABLE-Statements werden bei Seam-gen bzw. in den Anwendungsbeispielen in Seam nicht verwendet. Eine Evaluierung des REMOVE TABLE REFACTORINGS ist somit nicht möglich.

Zur Ausführung des mit Seam-gen erstellten Projekts bzw. der bei Seam mitgelieferten Anwendungsbeispiele wird ein Java Application Server benötigt. Alle folgenden Betrachtungen beziehen sich dabei auf den JBoss AS 5.1 CR1, der zur Ausführung der

Listing 5.3: Beispiele für von JastAddJ fehlerhaft geschriebene Annotationen und deren korrekter Form

```

1 // fehlerhafte Ausgabe der @Table-Annotation durch JastAddJ
2 @Table(uniqueConstraints = @UniqueConstraint(columnNames = {"membername", }))
3 // korrekte Form der @Table-Annotation
4 @Table(uniqueConstraints = @UniqueConstraint(columnNames = "membername"))
5
6 // fehlerhafte Ausgabe der @Observer-Annotation durch JastAddJ
7 @Observer(value = {JpaIdentityStore.EVENT_USER_CREATED, })
8 // korrekte Form der @Observer-Annotation
9 @Observer(JpaIdentityStore.EVENT_USER_CREATED)

```

Anwendungen verwendet wurde. JBoss Seam wurde in der Version 2.2.1.CR1 eingesetzt.

Da das REMOVE TABLE REFACTORING bzgl. der gewählten Beispiele nicht näher evaluiert werden kann, wurde stattdessen das von Fowler in [Fow99, S. 328] definierte PUSH DOWN METHOD REFACTORING für Seam-gen bzw. die Anwendungsbeispiele in Seam evaluiert. Das PUSH DOWN METHOD REFACTORING beschreibt das Verschieben einer Methode aus einer Klasse in alle direkten Unterklassen. Dadurch ist es möglich, die Methode aus bestimmten Unterklassen zu entfernen, in denen die Methode keinen sinnvollen Zweck erfüllt. Dazu wurde das in Abschnitt 5.2 beschriebene Interface `Refactoring` um die Methode `pushDown` erweitert. Des Weiteren wurde zum RFMComposer die Klasse `MLRPushDownMethodRefactoring` hinzugefügt (vgl. Abschnitt 5.3).

Bei der Durchführung des PULL UP METHOD REFACTORINGS auf den HRManager in Abschnitt 4.2.1 auf Seite 41 wurde gezeigt, dass in der Kombination mit Hibernate- und SQL-Artefakten keine Wiederherstellung der globalen semantischen Integrität durch Refactorings möglich ist. Daher wird das PUSH DOWN METHOD REFACTORING nicht für den HRManager betrachtet. Bei Seam-gen bzw. den Anwendungsbeispielen in Seam fehlen hingegen die im HRManager vorhandenen CREATE TABLE-Statements, da die benötigten Tabellen automatisiert über das ORM erstellt werden. Es wird daher betrachtet, ob sich das PUSH DOWN METHOD REFACTORING unter diesen Bedingungen Semantik-erhaltend umsetzen lässt.

Mit JPA bzw. Hibernate ist neben dem Objekt-relationalen Mapping von Klassen hin zu einem bestehenden Datenbank-Schema auch die automatische Generierung eines Datenbank-Schemas möglich. Die Erzeugung des Datenbank-Schemas erfolgt ausgehend von den annotierten Klassen und Methoden bzw. Attributen [ML05, S. 10]. Bei Seam-gen wie auch den Anwendungsbeispielen wird diese Funktion genutzt, um die für die Anwendungen notwendigen Tabellen automatisiert zu erstellen. Daher befinden sich unter den Ressourcen keine SQL-Artefakte mit CREATE TABLE-Statements.

Bei der Durchführung der Refactorings auf Seam-gen und die bei Seam implementierten Anwendungsbeispiele wurden Probleme bei der Verarbeitung bestimmter Java-Annotationen durch den verwendeten Java-Parser JastAddJ festgestellt. Das Listing 5.3 zeigt zwei Beispiele für die von JastAddJ fehlerhaft ausgegebenen Annotationen. Jeweils direkt unter den fehlerhaften Annotationen ist die korrekte Form angegeben. Die fehlerhaften Annotationen wurden manuell korrigiert.

Für die Umsetzung der folgenden Refactorings mit Hilfe des RFMComposers ist die Definition von Feature Modulen und RFMs notwendig. Die RFMs enthalten die auf die Feature anzuwendenden Refactorings. Da keine Software-Anwendungen evaluiert wer-

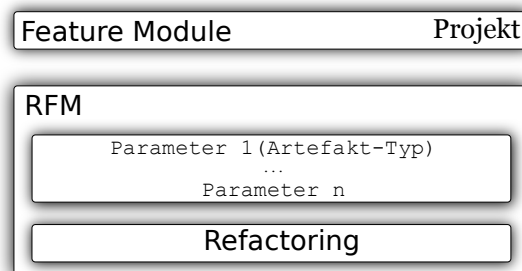


Abbildung 5.6: Schema zur Beschreibung der Feature Module und RFMs, die zur Umsetzung der Refactorings mit Hilfe des RFMComposers verwendet wurden. (Quelle: Eigene Darstellung)

Listing 5.4: Ersatz für das durch den SQL-Parser nicht verarbeitbare UPDATE-Statement aus Listing 4.12

```

1 INSERT INTO employees (firstname, surname, salary, department, boss)
2   VALUES('Michael', 'Müller', 5000, 1, 4);
  
```

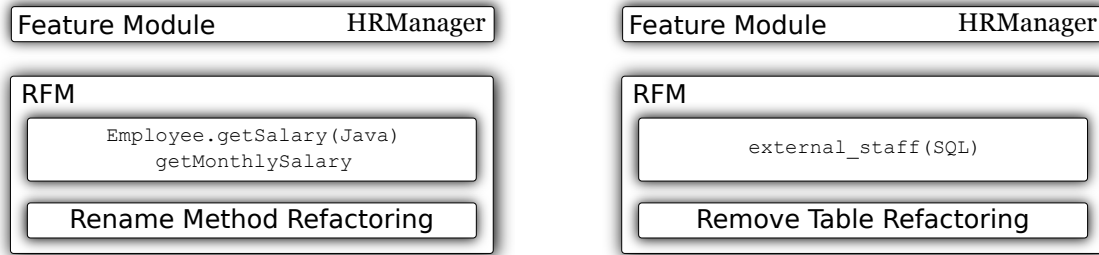
den, die Feature-orientiert entwickelt wurden, werden keine Feature sondern vollständige Software-Anwendungen als Ausgangspunkt für die Refactorings angegeben. Die Abbildung 5.6 zeigt das Schema, über das im Folgenden die verwendeten RFMs beschrieben werden. Über das mit *Feature Module* überschriebene Rechteck wird das *Projekt* angegeben, auf das ein Refactoring durchgeführt werden soll. Im mit *RFM* überschriebenen Rechteck werden die für das Refactoring notwendigen *Parameter* und das durchzuführende *Refactoring* selbst angegeben. Wenn es sich bei einem der Parameter um den qualifizierten Namen eines Identifiers handelt, ist zusätzlich der *Artefakt-Typ* anzugeben.

5.4.1 HRManager

Die Implementierung der mehrsprachigen Refactorings wurden zunächst anhand des HRManagers getestet und evaluiert. Zur Durchführung des RENAME METHOD und REMOVE TABLE REFACTORINGS musste aufgrund technischer Restriktionen des verwendeten SQL-Parsers das in Listing 4.12 dargestellte UPDATE-Statement angepasst werden. Der SQL-Parser ist nicht in der Lage, die in dem Listing dargestellten SELECT-Statements korrekt zu parsen. Das UPDATE-Statement wurde für die Evaluation durch ein semantisch äquivalentes Statement ersetzt, das die Referenz zwischen den zwei Datensätzen durch einen direkten Verweis auf den entsprechenden Schlüssel aufbaut (vgl. Listing 5.4). Darüber hinaus waren keine weiteren Modifikationen des HRManagers notwendig.

Da die Beispiel-Anwendung sowie die durchgeführten Refactorings in Abschnitt 4.1 bzw. 4.2 bereits beschrieben wurden, werden im Folgenden nicht die einzelnen Modifikationen im Detail aufgeführt. Nach der Durchführung der genannten Refactorings konnte die Anwendung weiterhin kompiliert und ausgeführt werden. Auch die erwarteten Ergebnisse wurden ausgegeben.

Rename Method Refactoring Das RENAME METHOD REFACTORING wurde anhand der Methode `getSalary` der Klasse `Employee` (vgl. Abbildung 4.1) getestet. Dazu



(a) Feature Modul und RFM zur Durchführung des `RENAME METHOD REFACTORINGS` am HR-Manager. (Quelle: Eigene Darstellung)

(b) Feature Modul und RFM zur Durchführung des `REMOVE TABLE REFACTORINGS` am HR-Manager. (Quelle: Eigene Darstellung)

Abbildung 5.7: Die Abbildungen stellen die Feature Module und RFMs dar, die zur Durchführung der Refactorings am HRManager durch den RFMComposer verarbeitet wurden.

wurde die Methode `getSalary` in `getMonthlySalary` umbenannt. Die Abbildung 5.7(a) zeigt das an den RFMComposer übergebene RFM mit den entsprechenden Parametern. Das `RENAME METHOD REFACTORING` wurde durch Hinzufügen bzw. Modifikation einer `@Column`-Annotation umgesetzt, wie es für die Methode `getName` in Abschnitt 4.2.1 Seite 39 beschrieben wird.

Remove Table Refactoring Der Test des `REMOVE TABLE REFACTORINGS` erfolgte anhand der Tabelle `external_staff`. Die Abbildung 5.7(b) zeigt das an den RFMComposer übergebene RFM mit den entsprechenden Parametern. Die Umsetzung des `REMOVE TABLE REFACTORINGS` erfolgte wie in Abschnitt 4.2.2 Seite 54 beschrieben.

5.4.2 Seam-gen

Das Rapid Application Development (RAD) Tool Seam-gen ist Bestandteil des RIA Frameworks JBoss Seam. Seam-gen generiert ein Grundgerüst für eine Seam-basierte Anwendung und entlastet damit Entwickler vom Schreiben Seam-spezifischen Codes, der zum Betrieb des Frameworks notwendig ist[OA10].

Seam-gen stellt verschiedene Kommandozeilen-Parameter zur Verfügung, über die sich ein Seam-Projekt erstellen, erweitern und verwalten lässt. Dazu gehört die Möglichkeit, Funktionen zur Nutzerverwaltung in ein bestehendes oder neues Seam-Projekt zu integrieren[OA10]. Über den Kommandozeilenaufruf `seam add-identity-management` werden die dazu notwendigen Dateien in das Seam-Projekt integriert und die entsprechenden Einstellungen vorgenommen. Zu den notwendigen Dateien gehören die in Abbildung 5.8 dargestellten Java-Klassen `UserAccount`, `UserPermission` und `UserRole`. Die Klassen und die darin definierten Methoden sind mit JPA-Annotationen versehen und damit Bestandteil eines ORM (vgl. Abschnitt 4.1.3).

Neben den genannten Klassen fügt Seam-gen die Datei

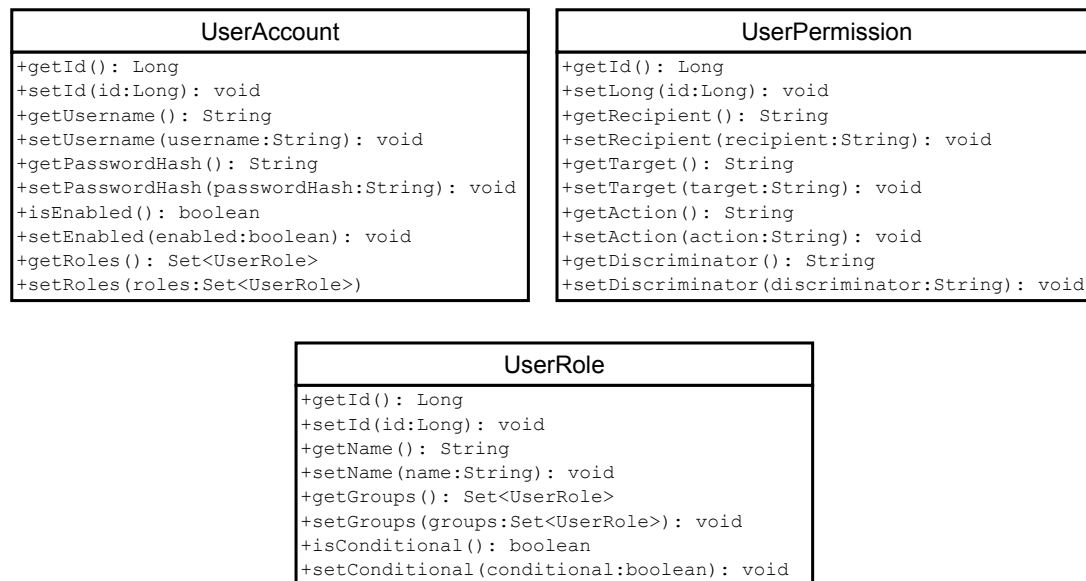


Abbildung 5.8: Die durch Seam-gen zum Seam-Projekt hinzugefügten Klassen der Nutzerverwaltung und öffentlichen Methoden. (Quelle: Eigene Darstellung)

Listing 5.5: Durch Seam-gen hinzugefügte initiale Nutzerdaten

```

1 — admin password is blank
2 insert into user_account (id, username, password_hash, enabled) values (1, 'admin', 'Ss/
  jICpf9c9GeJj8WKqx1hUC1EE=', 1);
3 insert into user_role (id, name, conditional) values (1, 'admin', false);
4 insert into user_role (id, name, conditional) values (2, 'member', false);
5 insert into user_role (id, name, conditional) values (3, 'guest', true);
6 insert into user_account_role (account_id, member_of_role) values (1, 1);
7 insert into user_role_group (role_id, member_of_role) values (1, 2);
```

`import-identity-management.sql` zum Seam-Projekt hinzu⁴. Den Inhalt der Datei zeigt das Listing 5.5. Zunächst wird in Zeile 2 ein aktiver Nutzer mit dem Namen *admin* und leerem Passwort in die Tabelle `user_account` eingetragen. In den Zeilen 3 bis 5 werden die Nutzer-Rollen *admin*, *member* und *guest* zur Tabelle `user_role` hinzugefügt. Über das Statement in Zeile 6 wird der Nutzer *admin* zur gleichnamigen Nutzergruppe hinzugefügt. Abschließend wird in Zeile 7 die Nutzergruppe *admin* der Nutzergruppe *member* zugeordnet. Die in Listing 5.5 dargestellten Statements können in dieser Form nicht vom verwendeten SQL-Parser geparkt werden. Dazu müssten alle Statements jeweils in einer eigenen Datei vorliegen. Da das `RENAME METHOD REFACTORING` mit Hilfe der `@Column`-Annotation umgesetzt wird, ist eine Modifikation der SQL-Statements nicht notwendig (vgl. Schritt 3b Seite 39). Daher wird auf das Anpassen der SQL-Statements im Rahmen dieser Evaluation verzichtet.

Die betrachteten Klassen sind nicht Bestandteil einer Klassen-Hierarchie, daher bietet sich die Durchführung des `PUSH DOWN METHOD REFACTORINGS` nicht an.

⁴Es werden weitere Dateien mit der Endung `.sql` hinzugefügt. Diese Dateien enthalten allerdings nur Kommentare, die vom verwendeten SQL-Parser nicht korrekt verarbeitet werden können und daher gelöscht wurden.

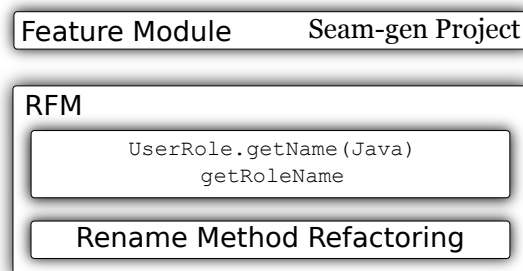


Abbildung 5.9: Das an den RFMComposer übergebene Feature Module und RFM zur Realisierung des RENAME METHOD REFACTORINGS am Seam-gen Projekt. (Quelle: Eigene Darstellung)

Rename Method Refactoring Für die Evaluation des RENAME METHOD REFACTORINGS wurde die Methode `getName` der Klasse `UserRole` mit Hilfe des RFMComposers automatisiert in `getRoleName` umbenannt, um die Aufgabe der Methode eindeutiger herauszustellen. Die Abbildung 5.9 stellt das an den RFMComposer übergebene RFM grafisch dar. Danach erfolgte der Test des modifizierten Seam-Projekts, indem es mit `seam explode` an den Application Server übergeben wurde. Nach der Übergabe erfolgte der Test der modifizierten Funktionalität, durch Aufrufen, Hinzufügen und Bearbeiten von Nutzer-Rollen.

Weitere zusätzliche Modifikationen an dem Seam-Projekt waren nach der Durchführung des Refactorings nicht notwendig. Dies ist wie folgt begründet. Die Methode `getName` der Klasse `UserRole` ist mit `@RoleName` annotiert. Durch die Annotation wird dem Seam-Framework bekannt gemacht, dass die annotierte Methode den Namen der Nutzer-Rolle zurückliefert⁵. Dadurch sind weitere Modifikationen nicht notwendig, denn durch die `@RoleName`-Annotation sind dem Framework die notwendigen Informationen bekannt.

5.4.3 Seam Anwendungsbeispiele

Das Seam-Framework enthält neben den notwendigen Bibliotheken und Seam-gen über 30 Anwendungsbeispiele, die die Anwendung des Seam-Frameworks beschreiben. Innerhalb von 17 Anwendungsbeispielen werden Annotationen der JPA verwendet. Auf zwei der 17 Anwendungsbeispiele wurde zur Evaluation das mehrsprachige RENAME METHOD und auf ein Anwendungsbeispiel das PUSH DOWN METHOD REFACTORING angewendet. Die gewählten Anwendungsbeispiele besitzen im Vergleich zu den anderen Anwendungsbeispielen eine höhere Anzahl von mit JPA-Annotationen versehenen Klassen. Die gewählten Beispiele implementieren ein rudimentäres Soziales Netzwerk (Seam Space) und einen DVD-Verleih (DVD Store).

Die gewählten Anwendungsbeispiele lassen sich ohne weitere Modifikationen auf dem Application Server starten. Dazu stehen die folgenden Kommandozeilenparameter zur Verfügung:

ant test: Führt die für das Fallbeispiel definierten Unit Tests durch.

⁵ <http://docs.jboss.org/seam/2.2.1.CR1/reference/en-US/html/security.html>

ant deploy: Startet das Beispiel auf dem Application Server. Auf die Anwendung kann über `http://localhost:8080/ExampleName` zugegriffen werden. Dabei ist *ExampleName* durch `seam-booking` bzw. `seam-dvdstore` zu ersetzen.

ant undeploy: Entfernt das Beispiel vom Application Server.

Die Evaluation mit Hilfe der gewählten Anwendungsbeispiele erfolgte nach dem folgenden Schema:

1. Refactoring des Fallbeispiels,
2. Aufruf der definierten Unit Tests,
3. Starten des Fallbeispiels auf dem Application Server und Test der Funktionalität der modifizierten Klassen.

Für die Darstellung von Webseiten verwendet Seam JavaServer Faces (JSF)[Far07, Seite 1-2]. JSF ist ein standardisiertes Framework zur Implementierung Web-basierter Nutzerschnittstellen[Ber04, Seite 1-2]. Des Weiteren werden in den Projekten HQL-Statements (engl. Hibernate Abfragesprache, HQL) verwendet. HQL-Statements dienen dazu, aus einer Datenbank konkrete Java-Instanzen anstatt einzelne Datensätze abzufragen. Bei der Evaluation werden sowohl JSF- wie auch HQL-Artefakte manuell modifiziert, wenn dies zur Herstellung der globalen semantischen Integrität notwendig ist. Die Modifikationen werden an entsprechender Stelle genannt und als Refactorings evaluiert. Somit stehen diese Modifikationen im Einklang mit der in dieser Arbeit getroffenen Definition mehrsprachiger Refactorings.

Seam Anwendungsbeispiel: DVD Store

Das Seam DVD Store Beispiel besteht aus insgesamt 39 Klassen, die einen Online DVD Verleih implementieren. Von den 39 Klassen werden neun über das ORM verwaltet. Die vom ORM verwalteten Klassen werden in Abbildung 5.10 dargestellt. Hierbei sind insbesondere die Klassen `Inventory` und `User` von Bedeutung. Die Klasse `Inventory` kapselt alle Informationen bezüglich des Bestands einer konkreten Instanz der Klasse `Product`. Instanzen der Klasse `Product` sind DVD-Filme, die über den DVD-Verleih geordert werden können. Die Klasse `Inventory` speichert dazu, wieviele Exemplare einer DVD noch vorhanden (`Quantity`) sind und wieviele davon insgesamt verliehen (`Sales`) wurden. Über die Methode `order` können DVDs aus dem Bestand entnommen werden. Die Klasse `User` kapselt alle für die Nutzerverwaltung relevanten Informationen, wie Name, Passwort etc., und stellt die Informationen den Unterklassen `Customer` und `Admin` zur Verfügung.

Rename Method Refactoring Die Aufgabe der Methode `getQuantity` der Klasse `Inventory` kann durch die Wahl eines eindeutigeren Bezeichners herausgestellt werden. Dazu wurde die Methode `getQuantity` in `getNumberOfProducts` umbenannt. Die Abbildung 5.11(a) zeigt, dass dazu an den RFMComposer übergebene RFM. Die folgenden Modifikationen wurden automatisiert vorgenommen:

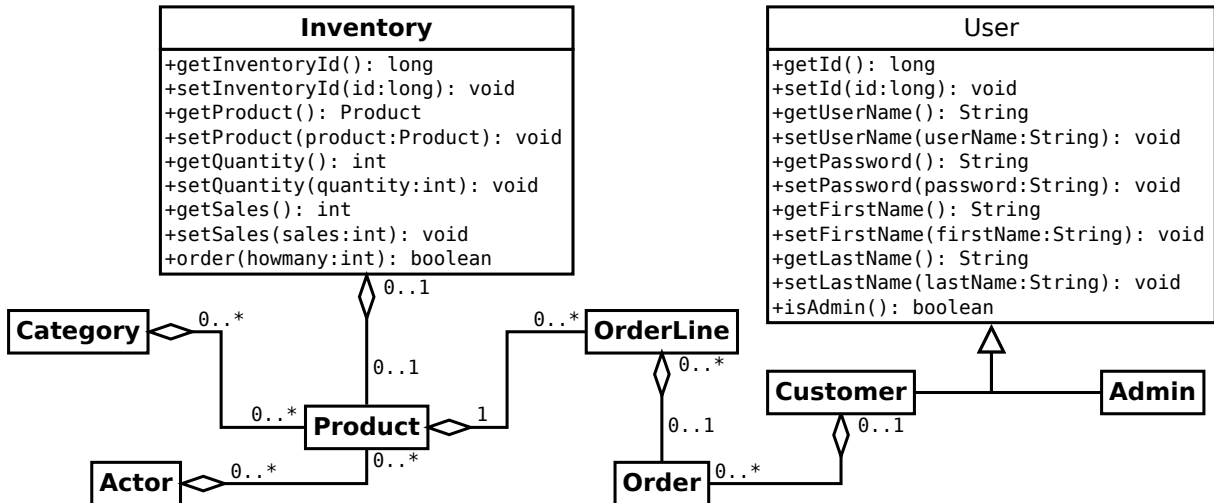


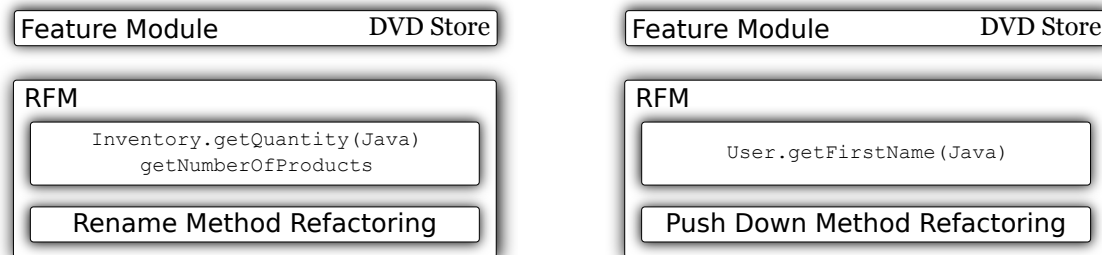
Abbildung 5.10: Die im Seam DVD Store Beispiel von Hibernate verwalteten Klassen. Die zu modifizierenden Klassen `Inventory` und `User` werden mit den in den Klassen definierten öffentlichen Methoden dargestellt. (Quelle: Eigene Darstellung)

1. Umbenennen des Bezeichners der Methode `getQuantity` und der auf die Methode gerichteten Referenzen in `getNumberOfProducts`,
2. Umbenennen der Methode `setQuantity` und der auf die Methode gerichteten Referenzen in `setNumberOfProducts`.

Es musste keine `@Column`-Annotation hinzugefügt werden, da die Methode bereits entsprechend annotiert ist. Des Weiteren musste das HQL-Statement `select sum(i.quantity) from Inventory i` in der Klasse `StoreManagerBean` per Hand angepasst werden. Das HQL-Statement gibt die Anzahl der Produkte im Inventar zurück. Durch Änderung des HQL-Statements in `select sum(i.numberofProducts) from Inventory i` wird die Funktionalität wiederhergestellt. Bei dieser Modifikation handelt es sich um ein Refactoring [TTS+08]. Darüber hinausgehende Modifikationen waren nicht notwendig, um die Unit Tests erfolgreich zu durchlaufen und die Anwendung auf dem Application Server zu starten.

Push Down Method Refactoring Neben dem `RENAME METHOD REFACTORING` bietet sich auch die Anwendung des `PUSH DOWN METHOD REFACTORINGS` in diesem Anwendungsbeispiel an. In der Klasse `User` sind die Methoden `getFirstName` und `setFirstName` definiert. Diese Methoden werden nur im Zusammenhang mit der Klasse `Customer` verwendet, daher ist es sinnvoll diese Methoden in die Klasse `Customer` zu verschieben.

Die Abbildung 5.11(b) zeigt das an den `RFMComposer` übergebene RFM. Zur Realisierung des `PUSH DOWN METHOD REFACTORINGS` ist nur die Angabe der zu verschiebenden Methode notwendig, denn zur korrekten Umsetzung des Refactorings wird die gewählte Methode immer nur genau eine Hierarchie-Ebene tiefer verschoben. Die Angabe einer Hierarchie-Ebene ist somit nicht notwendig. Zur Umsetzung des Refactorings wurden die folgenden Modifikationen automatisiert vorgenommen:



(a) Feature Module und RFM zur Durchführung des `RENAME METHOD REFACTORINGS` am DVD Store Anwendungsbeispiel. (Quelle: Eigene Darstellung)

(b) Feature Module und RFM zur Durchführung des `PUSH DOWN METHOD REFACTORINGS` am DVD Store Anwendungsbeispiel. (Quelle: Eigene Darstellung)

Abbildung 5.11: Die Abbildungen stellen die Feature Module und RFMs dar, die zur Durchführung der Refactorings am Seam DVD Store Anwendungsbeispiel durch den RFMComposer verarbeitet wurden.

1. Verschieben der Methode `getFirstName` in die Klassen `Customer` und `Admin`,
2. Setzen des Zugriffsmodifizierers des Feldes `firstName` der Klasse `User` von `private` auf `protected`,
3. Verschieben der Methode `setFirstName` in die Klassen `Customer` und `Admin`.

Die Klassen-Hierarchie der Klassen `User`, `Customer` und `Admin` wird über das ORM auf eine einzelne Datenbank-Tabelle abgebildet. Durch die Verwendung der Single-Table-Strategie ist es möglich, annotierte Methoden innerhalb der Klassenhierarchie zu verschieben, ohne eine nicht Semantik-erhaltende Anpassung der entsprechenden Datenbank-Tabelle vornehmen zu müssen (vgl. Abschnitt 4.2.1 Seite 41). Des Weiteren wird innerhalb der Java- und JSF-Artefakte auf die Methoden `getFirstName` und `setFirstName` nur über Instanzen der Klasse `Customer` zugegriffen. Daher sind nach der Durchführung des `PUSH DOWN METHOD REFACTORINGS` keine weiteren Modifikationen an den JSF-Artefakten erforderlich.

Seam Fallbeispiel: Seam Space

Das Seam Space Beispiel besteht aus insgesamt 24 Klassen, von denen neun über das ORM verwaltet werden. Die Klassen implementieren ein funktional unvollständiges soziales Netzwerk. Die Objekt-relational verwendeten Klassen werden in der Abbildung 5.12 dargestellt. Wie aus der Abbildung ersichtlich wird, stellt die Klasse `Member` die zentrale Entität des Anwendungsbeispiels dar, da die Klasse die meisten Entitäten miteinander in Beziehung bringt. Die Klasse `Member` speichert alle relevanten Informationen der Mitglieder des sozialen Netzwerks Seam Space. Dazu werden z.B. der Name, die Email-Adresse und die Freunde des Mitglieds gespeichert.

Die betrachteten Klassen sind nicht Bestandteil einer Klassen-Hierarchie, daher bietet sich die Durchführung des `PUSH DOWN METHOD REFACTORINGS` nicht an.

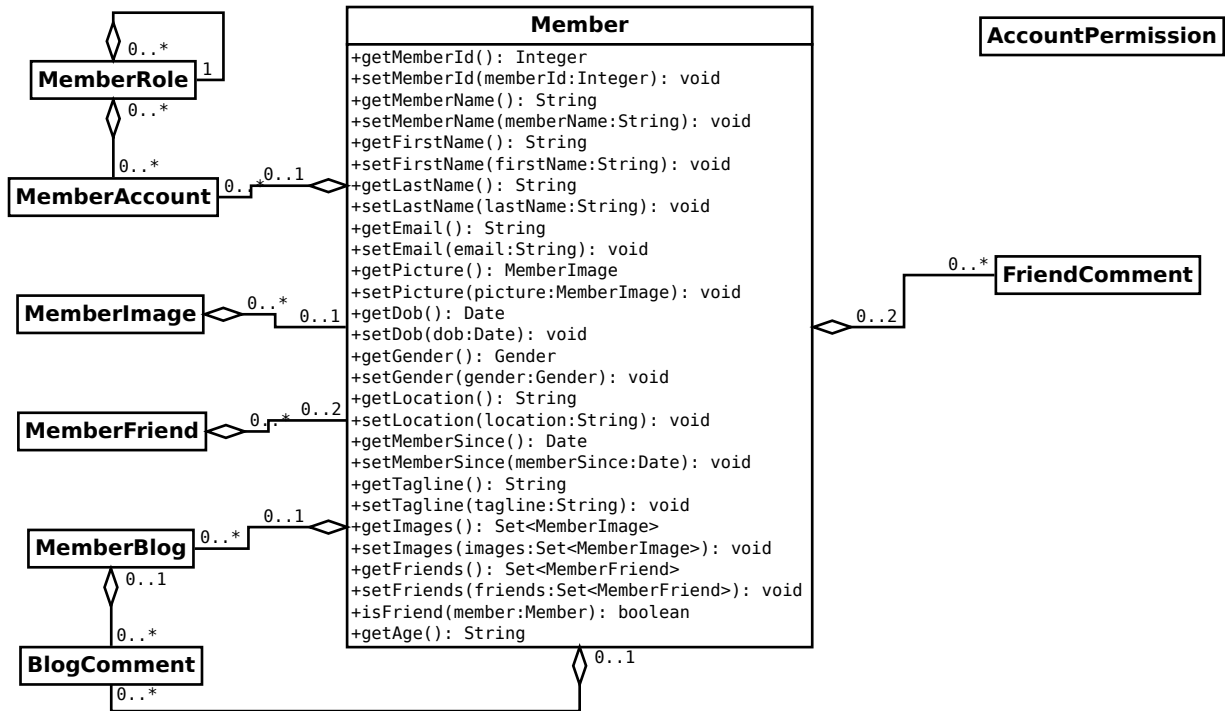


Abbildung 5.12: Die im Seam Space Beispiel von Hibernate verwalteten Klassen. Die zu modifizierende Klasse `Member` wird mit Getter- und Setter Methoden dargestellt. (Quelle: Eigene Darstellung)

Rename Method Refactoring Als Ausgangspunkt für das Refactoring dient die Methode `setDob` der Klasse `Member`. Aus der Parameter-Liste ist ersichtlich, dass die Methode ein Datum vom Typ `Date` erwartet. Aus dem Methoden-Bezeichner wird nicht eindeutig erkennbar, welche Information der Wert `Dob` darstellt. Aus der Betrachtung der Methode `getAge` der Klasse `Member` kann abgeleitet werden, dass es sich bei `Dob` um die Abkürzung für *Date Of Birth* handelt. Um direkt die Bedeutung der Methode `setDob` sichtbar zu machen, soll diese in `setDateOfBirth` umbenannt werden. Die Abbildung 5.13 visualisiert das dazu an den RfmComposer übergebene RFM. Zur Umbenennung sind die folgenden automatischen Schritte durchgeführt worden:

1. Umbenennen der Methode `setDob` und der auf die Methode gerichteten Referenzen in `setDateOfBirth`,
2. Umbenennen der Methode `getDob` und der auf die Methode gerichteten Referenzen in `getDateOfBirth`,
3. Hinzufügen einer `@Column`-Annotation mit dem `name`-Attribut `dob`.

Des Weiteren war die manuelle Anpassung des Unit Tests `testRegister` in der Klasse `RegisterTest` und der Datei `register.xhtml` notwendig. In der Klasse bzw. Datei musste der Aufruf `register.member.dob` durch `register.member.dateOfBirth` ersetzt werden. Dabei handelt es sich um Ausdrücke der JSF Expression Language (EL)[Ber04, S. 102ff]. Mit EL kann über JSF auf Attribute bestimmter, für die Verwaltung der Daten verantwortlicher Klassen zugegriffen werden. Vereinfacht dargestellt, könnte der EL-Ausdruck

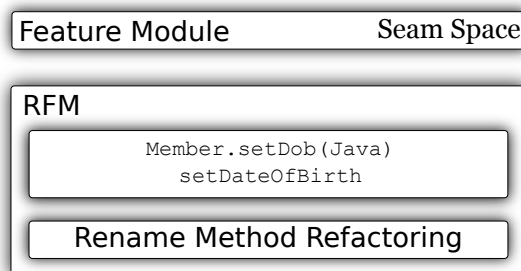


Abbildung 5.13: Das an den RFMComposer übergebene Feature Module und RFM zur Realisierung des RENAME METHOD REFACTORINGS am Seam Space Anwendungsbeispiel. (Quelle: Eigene Darstellung)

	Refactorings					
	REMOVE TABLE		RENAME METHOD		PUSH DOWN	
Anwendung / Fallbeispiel	<i>A</i>	<i>M</i>	<i>A</i>	<i>M</i>	<i>A</i>	<i>M</i>
<i>HRManager</i>	×		×		n.m.	
<i>Seam-gen</i>	n.m.		×		n.m.	
<i>DVD Store</i>	n.m.		×	×	×	
<i>Seam Space</i>	n.m.		×	×	n.m.	

(A) (teil-)automatisiert; (M) manuelle Modifikationen; (n.m.) nicht möglich

Tabelle 5.3: Ergebnisse der Evaluation der mehrsprachigen Refactorings

als Methoden-Aufruf aus (X)HTML heraus beschrieben werden. Weitere Modifikationen waren zur Herstellung der Funktionalität nach der Anwendung des RENAME METHOD REFACTORINGS nicht notwendig.

5.4.4 Ergebnisse der Evaluation

Die Ergebnisse der Evaluation werden in der Tabelle 5.3 zusammengefasst. In der Tabelle wird unterschieden, ob die durchgeführten Refactorings (teil-)automatisiert durchgeführt werden konnten und ob zur Herstellung der globalen semantischen Integrität zusätzlich manuelle Modifikationen notwendig waren. Nicht durchgeführte bzw. nicht mögliche Refactorings werden ebenfalls in der Tabelle ausgewiesen.

Die Evaluation hat gezeigt, dass das RENAME METHOD und PUSH DOWN METHOD REFACTORING über Java- und Hibernate-Artefakte automatisiert durchgeführt werden kann. An den unterstützten Artefakten waren nach der automatischen Umsetzung der Refactorings keine manuellen Modifikationen mehr notwendig.

Die erfolgreiche Umsetzung der Refactorings ist dabei unter der Beachtung der spezifischen Gegebenheiten, wie beteiligte Artefakte und Verwendung der definierten Refactorings, zu betrachten. So war die Umsetzung des RENAME METHOD REFACTORINGS innerhalb von Seam-gen ohne manuelle Modifikationen nur möglich, da eine deklarative Annotation die Modifikation weiterer Artefakte unnötig machte. Durch die spezifische Darstellung der Klassenhierarchie der Nutzerverwaltung des DVD Stores in einer einzel-

nen Tabelle war die Realisierung des PUSH DOWN METHOD REFACTORINGS überhaupt erst möglich. Es ist daher von Bedeutung, die spezifische Struktur einer mehrsprachigen Anwendung zu kennen, um erfolgreich mehrsprachige Refactorings anwenden zu können.

5.5 Zusammenfassung

In diesem Kapitel wurde die Architektur eines Software-Systems zur prototypischen Umsetzung mehrsprachiger Software-Anwendungen vorgestellt. Des Weiteren wurde die Integration der prototypischen Implementierung in den RFMComposer, einem Tool zur automatischen Komposition von RFMs, beschrieben. Mit der Integration in den RFMComposer ist ein erster Schritt hin zu sprachübergreifenden Refactoring Feature Modulen umgesetzt worden.

Es wurde gezeigt, dass sich sowohl das RENAME METHOD, wie auch das REMOVE TABLE REFACTORING, in einem Prozess umsetzen lassen, der die Durchführung eines mehrsprachigen Refactorings allgemein als sukzessive Anwendung von Refactorings auf einzelne Artefakte beschreibt. Dazu ist zunächst der Aufbau eines Identifier-Repositories durch Extraktoren notwendig. In dem Identifier-Repository sind Referenzen zwischen verschiedenen Software-Artefakten dargestellt. Auf Basis des Repositories erfolgt die sukzessive Ausführung von Refactorings durch Sprach-spezifische Refaktoren.

Die Evaluation des RENAME METHOD und des PUSH DOWN METHOD REFACTORINGS haben die Funktionstüchtigkeit der implementierten mehrsprachigen Refactorings gezeigt. Darüber hinaus ist anhand der gewählten Beispiele die Notwendigkeit zur Unterstützung weiterer Artefakt-Typen deutlich geworden.

Kapitel 6

Zusammenfassung und Ausblick

Für die vorliegende Arbeit wurden die folgenden Ziele formuliert. Zunächst sollten die Eigenschaften von Refactorings im Kontext von Software Anwendungen erläutert werden, bei deren Implementierung unterschiedliche, miteinander agierende Artefakt-Typen, wie z.B. Sourcecode, UML-Modelle oder Spezifikationen, verwendet werden. Aufbauend auf den gefundenen Eigenschaften sollten prototypisch Refactorings implementiert werden, die die Struktur unterschiedlicher Artefakt-Typen modifizieren, dabei aber die Funktionalität der Artefakte in ihrer Gesamtheit nicht verändern. Die Umsetzung der Ziele wird im Folgenden genauer erläutert.

In der vorliegenden Arbeit wurde zunächst der Begriff der mehrsprachigen Software-Anwendung definiert, als Software zu deren Umsetzung mehrere general- oder special-purpose Programmiersprachen verwendet werden (siehe Abschnitt 2.1). Zu einer mehrsprachigen Software-Anwendung gehören neben dem Sourcecode der verwendeten Programmiersprachen weitere Dokumente, die zur Realisierung der mehrsprachigen Software-Anwendung benötigt werden. Diese Dokumente wurden in der vorliegenden Arbeit als Software-Artefakte bezeichnet. Schließlich wurde definiert, dass ein mehrsprachiges Refactoring einen Prozess bezeichnet, der nach der Anwendung eines initialen Refactorings auf einen Artefakt-Typ sukzessiv weitere Refactorings verschiedener Artefakt-Typen anstößt, bis die globale semantische Integrität wiederhergestellt ist. Die globale semantische Integrität beschreibt den Zustand einer mehrsprachigen Software-Anwendung vor der Anwendung eines mehrsprachigen Refactorings. Dabei wurde argumentiert, dass zur Wiederherstellung der globalen semantischen Integrität kein einzelnes Artefakt semantisch modifiziert werden darf, da dies der Definition des Begriffs Refactoring widerspricht. Darüber hinaus kann die Verwendung semantischer Modifikationen zu Veränderungen von Software-Artefakten, wie Spezifikationen, führen, die gerade die Funktionalität einer mehrsprachigen Software-Anwendung beschreiben und als Grundlage für die Implementierung nicht verändert werden dürfen (vgl. Abschnitt 2.3.2).

Bei der Betrachtung verwandter Arbeiten in Kapitel 3 wurde gezeigt, dass die bisherigen Modelle und Ansätze zumeist die Beziehung zwischen Objekt-orientierten Artefakt-Typen betrachten. Die Erweiterung dieser Modelle um weitere Artefakt-Typen würde die Komplexität der Modelle steigern, so dass Kriterien wie Handhabbarkeit und Wartbarkeit nicht mehr gesichert werden können. Des Weiteren wurden Refactorings nur ausgehend vom Objekt-orientierten Artefakt-Typ betrachtet. Der Einfluss von nicht Objekt-orientierten Refactorings auf Objekt-orientierte Artefakte wurde nicht untersucht.

Zur Analyse der Eigenschaften mehrsprachiger Refactorings wurde zunächst eine mehrsprachige Beispiel-Anwendung HRManager entwickelt. Der HRManager realisiert eine rudimentäre Anwendung zur Verwaltung von Mitarbeiter-Daten. Für die Implementierung des HRManagers wurden Java, SQL, das Persistence Framework Hibernate sowie die funktionale Programmiersprache Clojure verwendet. Auf den HRManager wurden dann per Hand verschiedene Objekt-orientierte, funktionale sowie Datenbank-Refactorings angewendet und deren Einfluß auf die einzelnen Artefakte und Artefakt-Typen beschrieben. Dabei wurde festgestellt, dass die Anwendung eines Refactorings auf einen Artefakt-Typ in einem anderen Artefakt-Typ zu nicht Semantik-erhaltenden Modifikationen führen kann. Des Weiteren können Artefakte weitere Refactorings an einem bereits durch ein Refactoring modifizierten Artefakt anstoßen. Das Auftreten dieser Effekte ist dabei abhängig vom Refactoring und von den daran beteiligten Software-Artefakten. So konnte das RENAME METHOD REFACTORING Semantik-erhaltend über Java-, Hibernate- und SQL-Artefakte durchgeführt werden. Das MOVE CLASS REFACTORING wiederum konnte nur unter bestimmten Bedingungen Semantik-erhaltend umgesetzt werden. Aus der Betrachtung der mehrsprachigen Beispiel-Anwendung HRManager und der unterschiedlichen Refactorings konnten nur das RENAME METHOD und REMOVE TABLE REFACTORING als geeignet für die Realisierung als automatisiertes mehrsprachiges Refactoring über Java-, Hibernate-, SQL- sowie Clojure-Artefakte angesehen werden (vgl. Kapitel 4).

Darauf folgte die Implementierung des RENAME METHOD und REMOVE TABLE REFACTORINGS. Die Grundlage für die Implementierung bildet eine Architektur zur Darstellung der Beziehungen zwischen einzelnen Artefakt-Typen einer mehrsprachigen Software-Anwendung. Zu den wichtigsten Bestandteilen der Architektur gehören Extraktoren, Identifier und Refaktoren. Die Extraktoren erstellen zu jeder relevanten Information, wie Klassen-, Methoden- oder Tabellen- und Spalten-Definitionen, einen Identifier. Die Identifier werden dann miteinander in Beziehung gebracht, indem sie miteinander verglichen werden. Refaktoren können dann über die Identifier Refactorings an den durch die Identifier repräsentierten Artefakten umsetzen. Die implementierte Architektur wurde anhand unterschiedlicher Anwendungen bzw. Anwendungsbeispiele aus einem Rich Internet Application Toolkit evaluiert. Aufgrund der vom HRManager verschiedenen Struktur der Anwendungen bzw. Anwendungsbeispiele wurde ebenfalls das PUSH DOWN METHOD REFACTORING bei der Evaluation mit betrachtet. Dabei zeigte sich, dass die implementierten Refactorings auf den unterstützten Java-, Hibernate- und SQL-Artefakten korrekte Modifikationen vornahmen. Ein automatisches Refactoring von Clojure-Artefakten wurde aufgrund des Fehlens eines geeigneten AST nicht umgesetzt.

Bei der Durchführung der Evaluation mussten Artefakte, die durch die Architektur nicht unterstützt wurden, manuell modifiziert werden. Das bedeutet, dass die bisher Unterstützten Java-, Hibernate- und SQL-Artefakte nicht ausreichen, um mehrsprachige Refactorings vollständig automatisiert auf einer gegebenen mehrsprachigen Anwendung durchzuführen. Dazu muss zukünftig die Unterstützung weiterer Artefakt-Typen in der vorgestellten Architektur erfolgen.

In der vorliegenden Arbeit wurden das RENAME METHOD und REMOVE TABLE REFACTORING über Java-, Hibernate- und SQL-Artefakte sowie das PUSH DOWN METHOD REFACTORING über Java- und Hibernate-Artefakte als Semantik-erhaltend identifiziert. In Bezug auf diese Artefakte könnten weitere mehrsprachige Refactorings ausgehend vom

CHANGE CLASS NAME bzw. CHANGE VARIABLE NAME REFACTORING [Opd92, S. 42] gefunden und realisiert werden. Das CHANGE CLASS NAME und CHANGE VARIABLE NAME REFACTORING modifizieren die Bezeichner von Klassen bzw. Klassen-Attributen. Klassen und Klassen-Attribute bzw. deren Bezeichner werden von Hibernate über ähnliche oder gleiche Java-Annotationen wie Klassen-Methoden bzw. Methoden-Bezeichner behandelt. Daher gelten für die Umbenennung der Bezeichner von Klassen bzw. Klassen-Variablen möglicherweise ähnliche Modifikationen, wie sie im Zusammenhang mit dem RENAME METHOD REFACTORING umgesetzt wurden.

Allgemein können durch zukünftige Betrachtungen weiterer Artefakt-Typen weitere Refactorings gefunden werden, die im Kontext einer mehrsprachigen Software-Anwendung Semantik-erhaltend und automatisiert umgesetzt werden können. Das bedeutet auch, dass sich diese Betrachtungen auf konkrete Software-Artefakte beziehen müssen. Denn sowohl die Art der Interaktion zwischen zwei Software-Artefakten, wie auch der Einfluss eines Refactorings auf die Artefakte ist immer abhängig von den betrachteten Artefakt-Typen. Eine allgemeine Betrachtung unterschiedlicher Artefakt-Typen und Refactorings ist somit sehr schwierig zu realisieren, wenn überhaupt möglich.

Bisher gibt es keine Untersuchungen darüber, welche Artefakt-Typen in praktischen mehrsprachigen Software-Projekten gemeinsam verwendet werden. Auf Basis der Ergebnisse solcher Untersuchungen wäre eine zielgerichtete Suche nach interagierenden Artefakt-Typen und der damit verbundenen, praktisch relevanten mehrsprachigen Refactorings möglich. Damit stellt die Aufstellung entsprechender Untersuchungen ebenfalls ein Ziel zukünftiger Betrachtungen dar.

Literaturverzeichnis

- [AK09] Apel, S.; Kästner, C.: An overview of feature-oriented software development. *Journal of Object Technology (JOT)*, Band 8, Nr. 5, S. 49–84, 2009.
- [Amb03] Ambler, S.: *Agile Database Techniques: Effective Strategies for the Agile Software Developer*. John Wiley & Sons, Inc., New York, NY, USA, 2003.
- [AP98] Alagar, V. S.; Periyasamy, K.: *Specification of Software Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1998.
- [Bat03] Batory, D.: A tutorial on feature oriented programming and product-lines. *Proceedings of the 25th International Conference on Software Engineering, May 3-10, 2003, Portland, Oregon, USA*, S. 753–754, 2003.
- [Bat04] Batory, D.: Feature-oriented programming and the AHEAD tool suite. In *26th International Conference on Software Engineering (ICSE 2004), 23-28 May 2004, Edinburgh, United Kingdom*, S. 702–703. IEEE Computer Society, 2004.
- [BCPV07] Berdaguer, P.; Cunha, A.; Pacheco, H.; Visser, J.: Coupled schema transformation and data conversion for XML and SQL. *Practical Aspects of Declarative Languages, 9th International Symposium, PADL 2007, Nice, France, January 14-15, 2007*, Band 4354, S. 290–304, 2007.
- [Ber04] Bergsten, H.: *JavaServer Faces*. O’Reilly & Associates, Inc., Sebastopol, CA, USA, 2004.
- [BFG⁺02] Bosch, J.; Florijn, G.; Greefhorst, D.; Kuusela, J.; Obbink, J.; Pohl, K.: Variability issues in software product lines. *Software Product-Family Engineering, 4th International Workshop, PFE 2001, Bilbao, Spain, October 3-5, 2001, Revised Papers*, Band 2290, S. 13–21, 2002.
- [Bos03] Bosch, J.: Maturity and Evolution in Software Product Lines: Approaches, Artefacts and Organization. *Software Product Lines, Second International Conference, SPLC 2, San Diego, CA, USA, August 19-22, 2002, Proceedings*, Band 2379, S. 257–271, 2003.
- [BSR03] Batory, D.; Sarvela, J.; Rauschmayer, A.: Scaling step-wise refinement. In *Proceedings of the 25th international conference on Software engineering*, S. 187–197. IEEE Computer Society Washington, DC, USA, 2003.

- [CHH05] Cleve, A.; Henrard, J.; Hainaut, J.: Co-transformations in information system reengineering. *Electronic Notes in Theoretical Computer Science*, Band 137, Nr. 3, S. 5–15, 2005.
- [CJ08] Chen, N.; Johnson, R.: Toward refactoring in a polyglot world: extending automated refactoring support across Java and XML. *WRT '08: Proceedings of the 2nd Workshop on Refactoring Tools*, S. 1–4, 2008.
- [COV06] Cunha, A.; Oliveira, J.; Visser, J.: Type-safe two-level data transformation. *FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings*, Band 4085, S. 284–299, 2006.
- [DLT00] Ducasse, S.; Lanza, M.; Tichelaar, S.: MOOSE: an Extensible Language-Independent Environment for Reengineering Object-Oriented Systems. *In Proceedings of CoSET '00 (2nd International Symposium on Constructing Software Engineering Tools)*, S. 24–30, 2000.
- [EH07] Ekman, T.; Hedin, G.: The jastadd extensible java compiler. *Proceedings of the 22nd annual ACM SIGPLAN*, Band C, Nr. 2, S. 4–7, 2007.
- [Far07] Farley, J.: *Practical JBoss® Seam Projects*. Apress, Berkely, CA, USA, 2007.
- [Fis10] Fischbach, R.: Comeback - Renaissance funktionaler Programmierung. *iX Special - Programmieren heute*, S. 61–65, 2010.
- [Fje08] Fjeldberg, H.-C.: *Polyglot programming*. Dissertation, Norwegian University of Science and Technology, 2008.
- [For08] Ford, N.: *The productive programmer*. O'Reilly, 2008.
- [Fow99] Fowler, M.: *Refactoring: Improving the Design of existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
- [GHJV95] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
- [GPB04] Grechanik, M.; Perry, D. E.; Batory, D.: Reengineering Large-Scale Polylingual Systems. *International Workshop on Incorporating COTS into Software Systems: Tools and Techniques (IWICSS) co-located with the 3rd International Conference on COTS-Based Software Systems (ICCBSS), Redondo Beach, CA, February 2004*, S. 22–32, 2004.
- [HM02] Harold, E. R.; Means, W. S.: *XML in a nutshell*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.
- [Jon98] Jones, T. C.: *Estimating software costs*. McGraw-Hill, Inc., Hightstown, NJ, USA, 1998.

- [KBA09] Kuhlemann, M.; Batory, D.; Apel, S.: Refactoring feature modules. *Formal Foundations of Reuse and Domain Engineering, 11th International Conference on Software Reuse, ICSR 2009, Falls Church, VA, USA, September 27-30, 2009. Proceedings*, Band 5791, S. 1–10, 2009.
- [KKKS08] Kempf, M.; Kleeb, R.; Klenk, M.; Sommerlad, P.: Cross language refactoring for Eclipse plug-ins. In *Companion to the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-13, 2007, Nashville, TN, USA*, S. 1–4. ACM, New York, New York, USA, 2008.
- [KLW06] Kontogiannis, K.; Linos, P.; Wong, K.: Comprehension and Maintenance of Large Scale Multi-Language Software. In *22nd IEEE International Conference on Software Maintenance (ICSM 2006), 24-27 September 2006, Philadelphia, Pennsylvania, USA*, S. 497–500. IEEE Computer Society, Philadelphia, 2006.
- [KM90] Korson, T.; McGregor, J. D.: Understanding object-oriented: a unifying paradigm. *Communications of the ACM*, Band 33, Nr. 9, S. 40–60, 1990.
- [KS06] Keith, M.; Schincariol, M.: *Pro EJB 3: Java Persistence API (Pro)*. Apress, Berkely, CA, USA, 2006.
- [KS09] Keith, M.; Schincariol, M.: *Pro JPA 2: Mastering the Java Persistence API*. Apress, Berkely, CA, USA, 2009.
- [KWDE98] Kullbach, B.; Winter, A.; Dahm, P.; Ebert, J.: Program comprehension in multi-language systems. In *Fifth Working Conference on Reverse Engineering*, S. 135–143. Citeseer, 1998.
- [Lä02] Lämmel, R.: Towards generic refactoring. *Proceedings of the 2002 ACM SIGPLAN Workshop on Rule-Based Programming, Pittsburgh, Pennsylvania, USA, 2002*, S. 15–28, 2002.
- [Lä04] Lämmel, R.: Coupled software transformations. In *Proc. SET 2004, First Int. Workshop on Software Evolution Transformations*, S. 31–35, 2004.
- [LBO03] Linos, P.; Berrier, S.; O’Rourke, B.: A tool for understanding multi-language program dependencies. *11th International Workshop on Program Comprehension (IWPC 2003), May 10-11, 2003, Portland, Oregon, USA*, S. 64–72, 2003.
- [Let04] Lethbridge, T.: The Dagstuhl Middle Metamodel: A Schema For Reverse Engineering. *Electronic Notes in Theoretical Computer Science*, Band 94, Nr. 94, S. 7–18, Mai 2004.
- [Li06] Li, H.: *Refactoring Haskell Programs*. Phd thesis, University of Kent, Canterbury, Kent, UK, 2006.
- [Lia99] Liang, S.: *Java Native Interface: Programmer’s Guide and Reference, 1st edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

- [Lin95] Linos, P. K.: PolyCARE: A Tool for Re-engineering Program Integrations Multi-language. *1st IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '95), November 6-10, 1995, Fort Lauderdale, Florida, USA*, S. 338–341, 1995.
- [LLMM07] Linos, P. K.; Lucas, W.; Myers, S.; Maier, E.: A Metrics Tool for Multi-Language Software. *Proceedings of the 11th IASTED International Conference on Software Engineering and Applications. Cambridge, Massachusetts. Nov. 2007*, S. 324–329, 2007.
- [Mar05] Marticorena, R.: Analysis and definition of a language independent refactoring catalog. *17th Conference on Advanced Information Systems Engineering (CAiSE 05). Doctoral Consortium, Porto, Portugal*, S. 8–16, 2005.
- [ML05] Minter, D.; Linwood, J.: *Pro Hibernate 3 (Expert's Voice)*. Apress, Berkely, CA, USA, 2005.
- [MT04] Mens, T.; Tourwé, T.: A survey of software refactoring. *IEEE Transactions on software engineering*, Band 30, Nr. 2, S. 126–139, 2004.
- [MW05] Moise, D. L.; Wong, K.: Extracting and Representing Cross-Language Dependencies in Diverse Software Systems. In *WCRE '05: Proceedings of the 12th Working Conference on Reverse Engineering*, S. 209–218. IEEE Computer Society, Washington, DC, USA, 2005.
- [Nor02] Northrop, L.: SEI's software product line tenets. *IEEE Software*, Band 19, Nr. 4, S. 32–40, Juli 2002.
- [NSCF06] Nozal, C. L.; Sánchez, R. M.; Crespo, Y.; Francisco Javier Pérez: Towards a language independent refactoring framework. *ICSOFTE 2006, First International Conference on Software and Data Technologies, Setúbal, Portugal, September 11-14, 2006*, S. 165–170, 2006.
- [OA10] Orshalick, J.; Assar, N.: *JBoss Seam : Agile RIA Development Framework*, 2010.
- [O'C06] O'Connor, J.: Scripting for the Java Platform. <http://java.sun.com/developer/technicalArticles/J2SE/Desktop/scripting/>, 2006.
- [Ola03] Olan, M.: Unit testing: test early, test often. *Journal of Computing Sciences in Colleges*, Band Mic, Nr. 08240, S. 319–328, 2003.
- [OP04] Osterhold, A.; Pistor, P.: *Datenbanksprache - Die SQL-Normen - Normen der Reihe ISO/IEC 9075*, 2004.
- [Opd92] Opdyke, W.: *Refactoring object-oriented frameworks*. Dissertation, University of Illinois at Urbana-Champaign, 1992.
- [PBL05] Pohl, K.; Böckle, G.; Linden, F. J. V. D.: *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

- [Pre97] Prehofer, C.: Feature-oriented programming: A fresh look at objects. *ECOOP*, Band 1241, S. 419–443, 1997.
- [RBJ97] Roberts, D.; Brant, J.; Johnson, R.: A refactoring tool for Smalltalk. *Theory and Practice of*, Band D, Nr. 1304, S. 1–15, 1997.
- [Rob99] Roberts, D. B.: *Practical Analysis for Refactoring*. Phd thesis, University of Illinois at Urbana-Champaign, 1999.
- [RSP04] Riebisch, M.; Streitferdt, D.; Pashov, I.: Modeling variability for object-oriented product lines. *Object-Oriented Technology*, S. 165–178, 2004.
- [SB00] Svahnberg, M.; Bosch, J.: Issues concerning variability in software product lines. *Software Architectures for Product Families, International Workshop IW-SAPF-3, Las Palmas de Gran Canaria, Spain, March 15-17, 2000, Proceedings*, Band 1951, S. 146–157, 2000.
- [SG10] Stal, M.; Gleim, U.: Parallelwelten - Entwicklung für Multi-Core-Systeme. *iX Special - Programmieren heute*, S. 105–110, 2010.
- [SH03] Sammet, J. E.; Hemmendinger, D.: *Programming languages*, S. 1470—1475. John Wiley and Sons Ltd., Chichester, UK, April 2003.
- [SKL06] Strein, D.; Kratz, H.; Lowe, W.: Cross-Language Program Analysis and Refactoring. *Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2006), 27-29 September 2006, Philadelphia, Pennsylvania, USA*, S. 207–216, September 2006.
- [SKR⁺09] Siegmund, N.; Kästner, C.; Rosenmüller, M.; Heidenreich, F.; Apel, S.; Saake, G.: Bridging the Gap Between Variability in Client Application and Database Schema. *Datenbanksysteme in Business, Technologie und Web (BTW 2009), 13. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme“ (DBIS), Proceedings, 2.-6. März 2009, Münster*, Band 144, S. 297–306, 2009.
- [SKS⁺08] Sunkle, S.; Kuhlemann, M.; Siegmund, N.; Rosenmüller, M.; Saake, G.: Generating highly customizable SQL parsers. *EDBT'08 Workshop on Software Engineering for Tailor-made Data Management, Proceedings, Nantes, France, March 29, 2008*, S. 29–34, 2008.
- [SLLL07] Strein, D.; Lincke, R.; Lundberg, J.; Löwe, W.: An extensible meta-model for program analysis. *IEEE Transactions on Software Engineering*, Band 33, Nr. 9, S. 592–607, September 2007.
- [Sny86] Snyder, A.: Encapsulation and inheritance in object-oriented programming languages. *ACM SIGPLAN Notices*, Band 21, Nr. 11, S. 38–45, November 1986.
- [SPTJ01] Sunyé, G.; Pollet, D.; Traon, Y. L.; Jézéquel, J.: Refactoring UML models. *UML 2001 - The Unified Modeling Language, Modeling Languages, Concepts, and Tools, 4th International Conference, Toronto, Canada, October 1-5, 2001, Proceedings*, Band 2185, S. 134–148, 2001.

- [Tee07] Teese, B.: Real-World Experiences With Hibernate. <http://www.shinetech.com/thoughts/thought-articles/63-real-world-experiences-with-hibernate>, S. 11, 2007.
- [Tic01] Tichelaar, S.: *Modeling object-oriented software for reverse engineering and refactoring*. Phd thesis, University of Berne, Switzerland, 2001.
- [TTS⁺08] Tatlock, Z.; Tucker, C.; Shuffelton, D.; Jhala, R.; Lerner, S.: Deep type-checking and refactoring. *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-23, 2008, Nashville, TN, USA*, S. 37–52, Oktober 2008.
- [Van10] VanderHart, L.: *Practical Clojure*. Apress, 2010.
- [Ver08] Verbaere, M.: *A language to script refactoring transformations*. Phd thesis, University of Oxford, 2008.
- [VSMD03] Van Gorp, P.; Stenten, H.; Mens, T.; Demeyer, S.: Towards automating source-consistent UML refactorings. *UML 2003-the unified modeling language: modeling languages and applications: 6th international conference, San Francisco, CA, USA, October 20-24, 2003: proceedings*, Band 2863, S. 144–158, 2003.
- [WOZS94] Weide, B. W.; Ogden, W. F.; Zweben, S. H.; Science, I.: Reusable software components. *ACM SIGAda Ada Letters*, Band XIV, Nr. 2, S. 24–49, März 1994.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur mit erlaubten Hilfsmitteln angefertigt habe.

Magdeburg, den 20. August 2010

Hagen Schink

