UNIVERSITÄT KOBLENZ · LANDAU
Fachbereich 4: Informatik

Untersuchung des Themengebietes „Aspekt-orientierte Programmierung“ anhand des Projektes „101companies“

Bachelorarbeit

zur Erlangung des Grades eines Bachelor of Science
im Studiengang Informatik

vorgelegt von
Dmitri Nikonov

Erstgutachter: Ralf Lämmel
Institut für Informatik

Zweitgutachter: Andrei Varanovich
Institut für Informatik

Koblenz, im August 2014

Erklärung

Hiermit bestätige ich, dass die vorliegende Arbeit von mir selbständig verfasst wurde und ich keine anderen als die angegebenen Hilfsmittel – ins besondere keine im Quellenverzeichnis nicht benannten Internet–Quellen – benutzt habe und die Arbeit von mir vorher nicht in einem anderen Prfungsverfahren eingereicht wurde. Die eingereichte schriftliche Fassung entspricht der auf dem elektronischen Speichermedium (CD-Rom).

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

. . . . . .

(Ort, Datum) (Unterschrift)

Abstract

Aspect-oriented programming (AOP) is a programming paradigm which extends object orientation. This thesis examines this paradigm, its language constructs and its power. Several existing tools and languages that cover aspect-orientation will be demonstrated and analyzed whether they are suitable for a custom implementation. The 101companies platform will be extended with a customly designed implementation of this matter. The previously presented languages will be used, provided those are suited for the usage. Prior to this there will be features chosen, which are going to be used for the implementation. Furthermore existing implementations of AOP will be presented and analysed.

Zusammenfassung

Aspekt-orientierte Programmierung (AOP) ist ein Paradigma, welches die Objekt-Orientierung erweitert. Die vorliegende Arbeit untersucht dieses Paradigma auf seine Sprachkonstrukte und seine Mächtigkeit. Dabei werden verschiedene existierende Werkzeuge und Sprachen, welche die Aspekt-Orientierung beinhalten, demonstriert und auf die Möglichkeit einer Implementation mit Hilfe dessen analysiert. Die 101companies Plattform soll mit Hilfe von eigens angefertigten Implementationen dieses Themas erweitert werden. Dabei werden die zuvor präsentierten Sprachen eingesetzt, vorausgesetzt sie sind für eine Nutzung geeignet. Zuvor werden geeignete Features der 101companies Plattform für eine Implementierung ausgesucht. Außerdem werden existierende Implementationen im Kontext der Aspekt-Orientierung vorgestellt und analysiert.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Die Softwaresprachen sind ein weit umfassendes Wissensgebiet und Teildisziplin der Informatik. Hiervon werden die verschiedenen Sprachen zur Entwicklung von Anwendungen, aber auch die dazugehörigen Technologien und darauf aufbauenden Frameworks umfasst. Ein großer Teil davon sind sogenannte Objekt-orienterte Sprachen. Unter den zehn beliebtesten und meist genutzten Programmiersprachen finden sich acht, welche Objekt-orientierte Konzepte unterstützen [Jul]. Der Vorteil der Nutzung von Objekt-orientierten Sprachen ergibt sich aus der Möglichkeit, komplexe Systeme mittels softwaretechnischer Prinzipien in überschaubare Modulbausteine zu zerlegen, und diese somit beherrschbar zu machen. Allerdings finden sich in der Welt der Softwareentwicklung oftmals Anforderungen, die ein Softwaresystem durchziehen. Diese lassen sich mit klassisch Objekt-orientierten Konstrukten nicht hinreichend bewältigen. Diese sogenannten Cross-Cutting-Concerns verletzen das Prinzip der Modularität innerhalb eines Softwaresystems, da sie nicht ausreichend gekapselt werden können.

Eine Erweiterung der Objekt-Orientierung zur besseren Modularisierung von Cross-Cutting-Concerns stellt die Aspekt-orientierte Programmierung dar. Dieses Sprachparadigma unterstützt die Objekt-Orientierung durch neue Sprachkonstrukte, die die Modularisierung von sogenannten Aspekten erlauben.

Ziel dieser Arbeit ist es die Aspekt-orientierte Programmierung hinsichtlich ihrer Mächtigkeit anhand des Projektes “101companies” [101a] zu untersuchen. Zunächst müssen die wesentlichen Konzepte der Aspekt-Orientierung herausgearbeitet werden. Anhand geeigneter existierender AOP-Technologien und Frameworks wird gezeigt, inwiefern das Projekt 101companies hinsichtlich der Aspekt-Orientierung erweitert und bereichert werden kann. Anhand des Featuremodells von 101companies sind die Möglichkeiten ein bestehendes Softwareprojekt durch AOP zu erweitern, zu diskutieren.

1.2 Aufbau der Arbeit

Zu Beginn werden dem Leser in Kapitel 2 elementare Grundlagen hinsichtlich des Kontextes dieser Arbeit präsentiert. Im Kontext der Erläuterung der Objekt-orientierten Programmierung erfolgt ebenfalls der Nachweis, dass diese keine hinreichenden Konstrukte für die Cross-Cutting-Concerns aufweist, welche ebenfalls definiert werden. Anschließend werden die Aspekt-orientierte Programmierung sowie deren wesentliche Eigenschaften vorgestellt.
Kapitel 3 beschäftigt sich mit der Plattform "101companies". Die Eigenschaften und der Funktionsumfang dieser Plattform werden aufgezeigt. Anschließend wird dem Leser das sogenannte Featuremodell präsentiert.Darüber hinaus werden die einzelnen Features des Modells näher beleuchtet.
Das 4. Kapitel stellt verschiedene existierende Technologien und Frameworks für die Aspekt-Orientierung vor. Dabei werden zunächst Kriterien festgesetzt, nach denen die verschiedenen Lösungen dem Leser präsentiert werden. Für jede AOP-Lösung werden kurze Beispiele der Anwendung gezeigt. Außerdem wird die Potenz der jeweiligen AOP-Lösung erläutert. Den Abschluss des Kapitels bildet ein tabellarischer Vergleich zwischen den vorgestellten AOP-Lösungen.
In Kapitel 5 werden die Grundlagen für den praktischen Teil dieser Arbeit gelegt. Zunächst werden die in Kapitel 3 vorgestellten Features aussortiert, woraufhin die Eignung der Features für die aspektorientierte Implementationen diskutiert wird. Darüber hinaus wird eine Auswahl für die späteren Implementationen der in Kapitel 4 vorgestellten AOP-Lösungen getroffen.
Kapitel 6 präsentiert bereits auf 101companies existierende Implementationen, welche den Kontext der Aspekt-Orientierung einführen. Dabei werden deren elemtare Bestandteile analysiert und diskutiert.
Im 7. Kapitel erfolgt die Vorstellung der eigens angefertigten Implementationen. Dabei werden die jeweils umgesetzten Features präsentiert und deren Umsetzung, beziehungsweise deren Nutzen diskutiert. Hierbei wird auch verdeutlicht, inwiefern die Konzepte aus Kapitel 4 realisiert werden können.
Das abschließende 8. Kapitel fasst die Ergebnisse der Arbeit zusammen, wobei die speziellen Resultate aus Kapitel 7 diskutiert und zusammenfassend wiedergegeben werden. Eine abschließende Meinung zur Tauglichkeit von AOP innerhalb von Softwareprojekten soll aufgezeigt werden.

Kapitel 2

Grundlagen

Dieses Kapitel beschreibt grundlegende Begriffe, die für das Verständnis dieser Arbeit essentiell sind. Zunächst wird die Objekt-orientierte Programmierung und das Problem der Cross-Cutting-Concerns erläutert. Daraufhin erfolgt die Einbringung des Begriffs der Aspekt-orientierten Programmierung sowie die Präsentation der wesentlichen Sprachkonstrukte.

2.1 Objekt-orientierte Programmierung

Die Objekt-orientierte Programmierung, kurz "OOP" genannt, stellt ein Programmiersprachenparadigma dar, wie auch die prozedurale oder die funktionale Programmierung. Das Grundprinzip dieses Paradigmas baut, wie der Name es schon verrät, auf Objekten auf. Diese bestehen aus einem internen Zustand sowie einer definierten Menge an Opertationen, die auf diesem Objekt ausgeführt werden können [Boo86]. Übergreifend kennzeichnet man auch die Objekt-orientierte Entwicklung. Das Ziel hierbei ist ein System unter Anwendung softwaretechnischer Prinzipien in kleinstmögliche Module aufzugliedern. Hierdurch erhält man eine saubere, klar definierte und abgekapselte Struktur [Boo86]. Um die Anforderungsdomäne an die Software möglichst genau zu spezifizieren, erstellt man anhand dieser Domäne Klassen, welche die Objekte der "echten Welt" weitestgehend abbilden. Kapselung, Abstraktion und Veerbung sind elementare Prinzipien von OOP. Ein Objekt soll also ein für sich abgeschlossenes Modul bilden. Mittels "divide-and-conquer" soll doppelter Aufwand vermieden werden, sprich jedes abgeschlossene Modul und jede darauf definierte Operation existiert nur genau ein Mal.

2.2 Cross-Cutting-Concerns

Zuvor wurde festgestellt, dass in OOP abgeschlossene Module mit darauf definierten Operationen nur ein einziges Mal existieren sollen. Somit soll der Wartungsaufwand minimiert und die allgemeine Wartbarkeit und Lesbarkeit des Quelltextes einer Software

verbessert werden. Cross-Cutting-Concerns, kurz “CCC” genannt, verstoßen gegen dieses Prinzip. CCCs sind Programmeigenschaften, die über mehr als ein Modul hinweg eingesetzt werden müssen [Kic+97]. Die Objekt-orientierte Programmierung bietet keine hinreichenden Möglichkeiten solche CCCs zu abstrahieren, sodass kein Verstoß mehr gegen dessen Prinzipien vorliegt. CCCs können ebenfalls in prozeduralen Sprachparadigmen auftreten, jedoch geht diese Arbeit nicht näher auf prozedurale Programmierung ein, weshalb auf eine weitere Ërläuterung verzichtet wird.

Ein System der Rechteabfrage in einer Objekt-orientierten Software stellt ein Beispiel eines CCCs dar. In diesem Szenario wird dem ausführenden Benutzer eine Rechtematrix für die Software intern bereitgestellt und verarbeitet. Dabei sollen Methodenaufrufe, für die der Benutzer keine Rechte besitzt, verhindert werden. Im klassischen Sinne der OOP muss der Entwickler der Software in jeder Methode eine Rechteabfrage einbetten. Dies setzt die Wartbarkeit der Software enorm ab, denn der Zwang die Logik der Rechteabfrage modifizieren zu müssen, zieht gegebenenfalls die Anpassung in jeder Methode nach sich. Somit sind das Prinzip der Kapselung sowie das der Abstraktion verletzt. Eine Möglichkeit CCCs weitreichend zu abstrahieren bietet die Aspekt-orientierte Programmierung, welche im folgenden Kapitel vorgestellt wird.

2.3 Aspekt-orienterte Programmierung

Die Programmeigenschaften einer Objekt-orientierten Software lassen sich in zwei Thematiken aufgliedern. Lassen diese sich sauber in eigene Einheiten, im Kontext der OOP beispielsweise Klassen kapseln, so nennt man diese Eigenschaften “Komponente” [Kic+97]. Entsprechend gibt es, wie bereits beschrieben, auch Programmeigenschaften, welche sich nicht in eine eigene Einheit abkapseln lassen. Kiczales et al. sprechen hierbei von sogenannten “Aspekten”. Ziel der AOP ist es also einen Mechanismus zur sauberen Kapselung und Abstraktion der in Kapitel 2.2 genannten CCCs bereitzustellen.

2.3.1 Sprachkonstrukte

Das Paradigma der Objekt-Orientierung kennt keinen Mechanismus zur Kapselung von Aspekten. Ein Compiler oder ein Interpreter einer Objekt-orientierten Sprache wüsste nicht, wie er mit entsprechenden syntaktischen Ausdrücken verfahren sollte. Aus diesem Grund erweitert die Aspekt-Orientierung die Objekt-Orientierung um weitere Sprachkonstrukte, die in diesem Kapitel vorgestellt werden. Im wesentlichen bringt die AOP drei Sprachkonstrukte mit, nämlich Joinpoints, Pointcuts und Advices [Kic+01]. Im Folgenden werden diese drei Sprachkonstrukte erläutert.

Joinpoint

Ein Joinpoint ist ein wohldefinierter Punkt im Programmablauf [Kic+01]. Wohldefinierte Punkte sind beispielsweise Methodenaufrufe wie das Lesen oder Beschrei-

ben von Programmvariablen. Diese sind somit ein elementares Sprachkonstrukt der Aspekt-Orientierung, da mit ihnen das Erweitern klassicher Programmlogik erst möglich gemacht wird.

Pointcut

"A pointcut is a set of join points, plus, optionally, some of the values in the execution context of those join points." [Kic+01] Ein Pointcut fasst also eine bestimmte Menge an Joinpoints zusammen. Zusätzlich lassen sich Variablen und Objekte aus dem Programmkontext des eingetretenen Joinpoints innerhalb des Pointcuts binden.

Advice

Advices sind eine bestimmte Art von Funktionen oder Methoden [Kic+01]. Diese beschreiben den zusätzlichen Aufruf von Programmlogik beim Auftritt jedes Joinpoints eines definierten Pointcuts. Ein Advice wird zu einem festgelegten Eintrittspunkt in der Programmlogik ausgeführt, beispielsweise vor oder nach dem Auftritt eines Joinpoints.

Ein Aspekt, wie er in diesem Kapitel bereits definiert und beschrieben wurde, setzt sich aus diesen drei Sprachkonstrukten zusammen. Zusätzlich ist es erlaubt, in Aspekten jede mögliche Definition von Programmlogik vorzunehmen, die auch in einer Klasse der jeweiligen Programmiersprache erlaubt ist [Kic+01].

Die bloße Definition eines Aspektes genügt jedoch nicht, denn der in Form von Aspekten definierte erweiternde Programmablauf muss noch in den bereits existierenden Code eingebunden werden. Dies geschieht durch das sogenannte "Weaving", welches im Folgenden vorgestellt wird.

2.3.2 Weaving

Wie bereits erläutert, muss die erweiterte Programmlogik in den bestehenden Programmfluss nahtlos eingebunden werden, damit die AOP ihre Wirkung entfalten kann. Dies geschieht durch den in Kapitel 2.3 beschriebenen Weaver. Dieser ist verantwortlich für das Verarbeiten aller hinzugefügten Aspekte und deren Einarbeitung in die bestehende Programmlogik. [PZ03; PGA02] Dabei gibt es verschiedene Wege, die Aspekte in den existierenden Code einzugliedern. Das Eingliedern der Aspekte kann entweder zur ***Compiletime***, zur ***Loadtime*** oder zur ***Runtime*** geschehen. Die drei genannten Möglichkeiten werden nun näher erläutert:

Compiletime

Beim Weaving zur Compiletime werden die verarbeiteten Aspekte gemeinsam mit der existierenden Programmlogik zusammengeführt und die neue Programmlogik als Ergebnis ausgegeben. Dabei kann auf Preprocessing zurückgegriffen werden.

Alternativ muss die gelieferte Technologie einen eigenen Compiler mitbringen oder gegebenenfalls den existierenden erweitern.

Loadtime

Der Ansatz des Weavings während der Loadtime bindet die verarbeiteten Aspekte während der Ladezeit der Klassen eines Programmes ein. Dieses Verfahren bietet die Möglichkeit die Programmlogik zu erweitern, die nicht in Form von Quelltext verfügbar ist, sondern bereits kompiliert bereitgestellt wird. Hierzu muss der Classloader einer Laufzeitumgebung mit dem Ziel, das Einbinden von Aspektcode zu ermöglichen, erweitert oder ersetzt werden.

Runtime

Das Einbinden von Aspekten zur Laufzeit erfordert einen Mechanismus, der Aufrufe von Aspekten während dem eigentlichen Programmaufruf validiert und injiziert. Um dies zu erreichen, bieten viele Objekt-orientierte Programmiersprachen Bibliotheken zur Metaprogrammierung, wie beispielsweise das Reflection-Programming im JDK [Ref]. Unabhängig davon muss in der Programmlogik eine Art Ökosystem eingegliedert sein, das die Aspekte zur Laufzeit einbindet. Ein Beispiel für eine solche Umsetzung bietet der Mechanismus des Interceptor in der Java Enterprise Edition, kurz JavaEE [Int].

Kapitel 3

101companies

In Kapitel 2 wurden elementare Grundlagen vorgestellt. Zum einen wurde OOP sowie dessen Mangel an Modularisierungsmechanismen für die CCCs erläutert, zum anderen mit AOP ein eben dafür geschaffener Mechanismus präsentiert. Darüber hinaus wurden dessen theoretische Grundkonzepte vorgestellt, um einen hinreichenden Einblick in die Thematik zu erhalten. In diesem Kapitel beschäftigt sich diese Arbeit mit `101companies` [101a], die Plattform auf deren Basis im Verlauf dieser Arbeit Implementationen ausgearbeitet werden. Zunächst wird die Plattform vorgestellt, woraufhin aufgezeigt wird, inwiefern die Plattform als Basis und Hilfsmittel für die Ausarbeitung verschiedener Implementationen nutzbar ist.

3.1 Vorstellung der Plattform

"*The 101companies Project [..] is an open knowledge resource covering software technologies, technological spaces, software languages, and software concepts. 101 targets programmers, software engineers, teachers, learners, and technologists; they can leverage 101 and they are encouraged to contribute to 101.*" [101a]

Das Projekt 101companies bietet also eine Plattform, auf der verschiedene Softwaresprachen, Technologien und Konzepte untersucht und präsentiert werden. Interessenten wird die Möglichkeit geboten, aktiv an dem Projekt mitzuwirken. Das Grundkonzept basiert auf dem sogenannten 101system [101c], welches auf dem Entwurf einer fiktiven Verwaltungssoftware für Unternehmensstrukturen beruht. Ein Unternehmen besitzt beliebig viele Abteilungen. Jede Abteilung kann beliebig viele Unterabteilungen haben und jeder Abteilung sind `0..n` Mitarbeiter zugewiesen. Ein Mitarbeiter ist auch in der Lage der Leiter einer Abteilung zu sein. Für Unternehmen, Abteilungen und Mitarbeiter werden Stammdaten wie Namen, Adressen und Gehälter der Mitarbeiter erfasst.

Dieses Grundkonzept wird durch die Features erweitert [Fea]. Eine bestimmte Menge von Anforderungen an das System wird im Namespace "Features" festgelegt. Modulari-

sierung spielt hierbei eine kennzeichnende Rolle, denn jedes Feature muss eigenständig implementiert sowie integriert werden können. Die Features stellen hierbei mögliche Erweiterungen des Grundkonzept von 101system dar. Auf die verschiedenen Features geht diese Arbeit im Detail in Kapitel 3.2 ein. Weitere nennenswerte Bestandteile der Plattform sind das 101wiki [101a] sowie das 101repo [101b].

Das 101wiki stellt ein Dokumentationswerkzeug für die 101companies Plattform dar, mit dessen Hilfe alle relevanten Themen übersichtlich zusammengefasst und zur Verfügung gestellt werden. Beispielsweise finden sich hier Backlinks zu den relevanten Werkzeugen von 101companies, die Vorstellung und Erläuterung der Implementationen sowie die Präsentation des Featuremodells.

Das 101repo ist, wie der Name schon sagt, ein Repository innerhalb der 101companies Plattform. Hier liegen bereitgestellte Implementationen, wichtige werden mittels einem separaten Register indiziert. Das 101repo ist ein zentraler Bestandteil der Plattform, da es als Quelle für alle Demonstrationszwecke dient. Jeglicher Sourcecode wird hier zur Präsentation bereitgestellt und kann von hier aus geklont werden. Das Repository fasst mehrere unterschiedliche physikalisch getrennte Repositories intern zu logischen Einheiten zusammen.

Die Plattform des 101companies Projektes beinhaltet noch weitere Werkzeuge, wie beispielsweise den 101worker [101d]. Da dieser und weitere Werkzeuge jedoch für den folgenden Verlauf dieser Arbeit keine Relevanz haben, geht dieses Kapitel nicht näher darauf ein. Stattdessen beschäftigt sich der nächste Abschnitt mit dem Featuremodell von 101companies.

3.2 Featuremodell

Das Featuremodell von 101companies bildet einen zentralen Bestandteil der Plattform. Es beinhaltet teils optionale Features, die auf das 101system angewendet werden können, um Konzepte und Techniken von Softwaresprachen und Werkzeugen demonstrieren zu können. [Fea] Alle Features haben einen motivierenden Aspekt, der beschreibt, inwiefern die Umsetzung eines Features relevant oder gar interessant im Kontext der Analyse von Softwaretechniken und -werkzeugen sein kann.

Das Featuremodell unterscheidet zwischen drei Typen von Anforderungen, nämlich Anforderungen an das Datenmodell, funktionale und nicht-funktionale Anforderungen. Zusätzlich existieren noch Anforderungen an die grafische Schnittstelle, worauf jedoch in dieser Arbeit kein weiterer Bezug genommen wird. Im Folgenden werden die Features einzeln kurz präsentiert und erläutert. Die Features und deren Anforderungen sind der Beschreibung des Featuremodells des 101wiki entnommen [Fea].

3.2.1 Anforderungen an das Datenmodell

Die Anforderung oder Features an das Datenmodell beschreiben speziell, inwiefern gewisse Datenstrukturen innerhalb des Softwarekomplexes existieren können. [Req] Zwar lassen sich diese Anforderungen als funktional eingliedern, jedoch wurden diese aus dem Grund in eine eigene Anforderungskategorie ausgelagert, da es sich hier um spezielle Features handelt, die explizit das Datenmodell von 101system beeinflussen und mitgestalten. Im Folgenden werden die Features, welche an diese Anforderungsgruppe geknüpft sind, vorgestellt.

Company

Dieses Feature beschreibt den Grundbaustein von 101system. Es existiert ein Unternehmen, welches beliebig viele Abteilungen mit jeweils beliebig vielen Arbeitnehmern hat. Die Abteilungen können wahlweise hierarchisch oder ohne Hierarchiezuordnung existieren. Alle Entitäten speichern Informationen als Objekteigenschaften, wie beispielsweise Name, Gehälter und Adressen.

Conflicts of Interest

Mitarbeiter haben die Möglichkeit, andere Mitarbeiter in ihr Umfeld der Interessenskonflikte aufzunehmen. Hierbei sollen die Abteilungsleiter einen Überblick über die Konfliktfelder ihrer jeweiligen Mitarbeiter haben.

Mentoring

Jeder Mitarbeiter hat Anspruch auf einen Betreuer. Mitarbeiter von Unternehmen können sowohl Betreuer sein als auch gleichzeitig betreut werden.

Ranking

Innerhalb eines Unternehmens gelten gewisse Grundsätze. Die Implementierung dieser Grundsätze ist dem Autor freigestellt. Beispielsweise darf das Gehalt jedes Departments nur um 10% vom Unternehmensdurchschnitt abweichen. Bei jeder Operation, die - in diesem Fall die Gehälter - Daten von Unternehmensinternen verändert, muss das Unternehmen auf die Einhaltung der Grundsätze überprüft werden.

Singleton

Hierbei geht es um die Möglichkeit, die Software auf ein einziges Unternehmen zu beschränken. Diesbezüglich muss sichergestellt werden, dass eine Objektinstanz eines Unternehmens nur maximal einmal pro Programminstanz instanziiert werden darf.

History

Änderungen an Entitäten vom Unternehmen müssen erfasst und zur Erhebung verfügbar gemacht werden. Beispielsweise sollen alle Lohnänderungen von Mitarbeitern protokolliert werden. Diese können anschließend weiterverarbeitet werden. Die Weiterverarbeitung ist nicht näher spezifiziert.

3.2.2 Funktionale Anforderungen

Funktionale Anforderungen beschreiben jene Anforderungen an das System, welche, wie deren Name bereits andeutet, direkten Einfluss auf die Funktionalität des Softwaresystems haben. Diese haben jedoch keinen Einfluss auf das Datenmodell und stehen somit in dieser Liste. Das Featuremodell weist folgende funktionalen Anforderungen auf:

Total

Dieses Feature bietet die Möglichkeit, die Gehälter aller Mitarbeiter eines Unternehmens aufzusummieren.

Cut

Hierbei werden die Gehälter innerhalb eines Unternehmens halbiert. Dies kann entweder nur den Mitarbeiter, eine oder mehrere Abteilungen sowie das ganze Unternehmen betreffen.

Median

Der Unternehmensleitung soll neben anderen Daten auch der Median von allen Mitarbeitern im Unternehmen verfügbar gemacht werden.

Logging

Änderungen an Unternehmensdaten sollen protokolliert werden. Die Protokolle sollen dazu genutzt werden können, um daraus entsprechende Daten, wie den Median berechnen zu können.

Depth

Bei Unternehmen mit hierarchisch angeordneten Abteilungen soll die jeweilige Tiefe durch die Software berechnet und bereitgestellt werden.

Parsing

Zwecks Austausch von Unternehmensdaten soll es die Möglichkeit geben, diese Daten in einem bestimmten Format einlesen zu können.

Unparsing

Dieses Feature bildet das Gegenstück zum Parsing Feature. Hierbei sollen Unternehmensdaten in einem bestimmten Format ausgegeben werden.

3.2.3 Nicht-funktionale Anforderungen

Dieser Abschnitt stellt die nicht-funktionalen Anforderungen an das System vor. Hierbei geht es um Qualitätsanforderungen, die den Funktionsumfang des Softwaresystems nicht oder höchstens indirekt erweitern. Die folgenden Features gehören in die Reihe der nicht-funktionalen Anforderungen.

Serialization

Jegliche Unternehmensdaten müssen zu jedem beliebigen Zeitpunkt in serialisierter Form verfügbar sein. Dies dient beispielsweise der Datenbeständigkeit zwischen Abstürzen des Systems. Das Format der Serialisierung ist nicht spezifiziert und kann vom Autor der Implementation frei gewählt werden.

Persistence

Das Persistence Feature beinhaltet die Anforderung Unternehmensdaten jederzeit speicher- und wiederabrufbar zu machen. Hierbei dreht es sich zusätzlich um Anforderungen an die Skalierbarkeit des Softwaresystems. Daten müssen in großen Mengen abrufbar und peristierbar sein.

Mapping

Dieses Feature beinhaltet die Anforderung, Unternehmensdaten auf verschiedene Technologien anzuwenden. Beispielsweise sollen diese Daten vom System zu XML oder SQL-Datensätzen verarbeitet werden können. Hierzu muss eine Schnittstelle existieren, die dafür Sorge trägt.

Distribution

Das System soll bei Umsetzung dieses Features verteilte Berechnungen als Client-Server-Architektur unterstützen können.

Parallelism

Hierbei ist das Szenario interessant, in dem die Software Unternehmen mit sehr großen Datenmengen beinhaltet. Die Abarbeitung von Methodenaufrufen in serieller Form wird hierbei zum Flaschenhals. Das Feature fordert die Möglichkeit Daten der Unternehmen nebenläufig oder parallel verarbeiten zu können.

In diesem Kapitel erfolgte die Diskussion bezüglich des Featuremodells sowie die Beschreibung der drei Anforderungskategorien. Allerdings kann hier noch keine Auswahl der für AOP relevanten Features gemacht werden. Dieser Schritt wird in Kapitel 5.1 umgesetzt.

Kapitel 4

Aspekt-orientierte Technologien

Im vergangenen Kapitel wurde die Plattform 101companies und dessen Featuremodell vorgestellt. Für die Demonstration und Anwendung von AOP auf das Featuremodell sind zunächst Vorkenntnisse über AOP-relevante Technologien notwendig. In Kapitel 2 wurden die theoretischen Grundlagen der Aspekt-Orientierung behandelt. Hier werden nun spezielle Aspekt-orienterte Technologien, Sprachen und Frameworks ausgewählt und beschrieben. Anschließend werden die gezeigten Technologien tabellarisch miteinander verglichen. Eine Auswahl konkreter Technologien geschieht in Kapitel 5.

4.1 Kriterien zur Auswahl

Zunächst befasst sich dieses Kapitel mit der Auswahl geeigneter AOP-Sprachen, oder auch Frameworks. Diese werden anhand der hier erläuterten Kriterien ausgewählt und deren Funktionsumfang beziehungsweise deren Leistungsfähigkeit vorgestellt.
Im Folgenden werden Kriterien aufgelistet, die eine AOP-Sprache oder ein Framework nach Möglichkeit erfüllen muss.

1. Das Framework beziehungsweise die Sprache muss in der Lage sein dem Entwickler das Umsetzen von Aspekten zu ermöglichen.

2. Das Framework bietet eine weitgehend breite Unterstützung von Sprachmitteln für Pointcuts.

3. Das Framework wurde bereits innerhalb der 101companies Plattform in jedweder Form untersucht.

4. Die Implementationen der Aspekte sollen auf der klassischen Sprachplattform ausführbar sein, sprich eine Erweiterung durch AOP innerhalb von Java muss auf der normalen JVM ausführbar sein.

5. Eine Objekt-orientierte Sprache erfährt ein breites Nutzerspektrum und wird hinreichend durch ein entsprechendes AOP-Framework erweitert.

6. Die Nutzung des Frameworks muss sich möglichst natürlich anfühlen.

Anhand der Liste für Entwicklerwerkzeuge der AOSD [Dev] sowie einer Plattform für das Ranking von Programmiersprachen werden anschließend vier verschiedene AOP-Technologien beziehungsweise Werkzeuge vorgestellt. Dabei wird die Auswahl folgend gestaltet:

1. Vorstellung von statisch sowie dynamisch typisierten Sprachen
2. Die Sprache ist Objekt-orientiert
3. Die Sprache hat eine breite Nutzerbasis
4. Das Werkzeug der Aspekterstellung ist leistungsstark, sprich, es bietet eine breite Palette an Werkzeugen und Sprachmitteln

4.2 AspectJ

AspectJ [Asp] wurde 2001 am Xerox Palo Alto Research Center durch Gregor Kiczales et al. [Kic+01] entwickelt und stellt eine AOP-Lösung für die Programmiersprache Java dar, welche mitunter die beliebteste Objekt-orientierte Programmiersprache ist [Jul]. Die Entwickler dieser Lösung sind für erste Überlegungen im Themengebiet der Aspektorientierung verantwortlich (vgl. Kapitel 2.3).

"*AspectJ extends Java with support for two kinds of crosscutting implementation. The first makes it possible to define additional implementation to run at certain well- defined points in the execution of the program. We call this the dynamic crosscutting mechanism. The second makes it possible to define new operations on existing types. We call this static crosscutting because it affects the static type signature of the program.*" [Kic+01]

Mit AspectJ lassen sich also sowohl Operationen auf Objekten erweitern als auch zusätzlich neue Operationen definieren. Damit wird dem Entwickler ein mächtiges Werkzeug zur Verfügung gestellt.

Beim Design von AspectJ wurden einige Kriterien festgesetzt, die es mit einer AOP-Lösung zu erfüllen gilt. Diese sind wie folgt definiert: [Kic+01]

- **Aufwärtskompatibilität** Jedes AspectJ Programm ist ein gültiges Java-Programm.
- **Plattformkompatibilität** Jedes AspectJ Programm muss in der Lage sein in einer normalen Java Laufzeitumgebung ausführbar zu sein.

- **Werkzeugkompatibilität** Es muss die Möglichkeit bestehen, existierende Java-Werkzeuge hinsichtlich der Aspekt-Orientierung und AspectJ zu erweitern.
- **Programmierkompatibilität** Bei Vertrautheit mit Java muss die Vertrautheit mit AspectJ möglichst schnell hergestellt werden. Das Programmieren mit AspectJ muss sich also möglichst natürlich anfühlen, also weitestgehend an Java angelehnt sein.

Neben der Tatsache, dass AspectJ eine flexible und weit entwickelte AOP-Lösung für Java darstellt, wird es im Allgemeinen auch als eine Art empirisches Experiment gehandhabt. Ziel war es anhand einer großen Entwicklergemeinschaft zu untersuchen, wie das Werkzeug genutzt wird und welche Richtlinien sich dabei aufzeigen lassen.

AspectJ verwendet für die Darstellung von Aspekten klassenähnliche Strukturen und erfüllt damit das Ziel der Programmierkompatibilität. Im Folgenden wird ein klassisches Beispiel eines CCCs anhand von AspectJ aufgezeigt.

Listing 4.1: Ein Aspekt zum Persistieren von Bankdaten in AspectJ [Wam07]

```
1 aspect PersistenceAspect {
2   pointcut stateChange(BankAccount ba):
3     (call(void BankAccount.deposit(*)) || call(void BankAccount.withdraw(*))) &&
          target(ba);
4
5   after (BankAccount ba) returning : stateChange(ba) {
6     // persist ba.getBalance() value
7   }
8 }
```

Hier sieht man einen Aspekt zum Persistieren von Bankdaten. Im klassischen Objekt-orientierten Stil ist es die Aufgabe des Entwicklers, jede Methode, in der etwas persistiert werden soll, in die Geschäftslogik dieses Vorgangs einzubinden. Diese durchstreift verschiedene Stellen der eigentlichen Programmierlogik. Ist im Nachhinein etwas an dieser zu ändern, muss im schlimmsten Fall der ganze Programmcode angepasst werden. AspectJ wirkt hier entgegen. Ein Aspekt wird zu Beginn einer Datei als “aspect” deklartiert (Zeile 1). Hierbei wird die Ähnlichkeit zur Java-basierten Klasse deutlich.

Innerhalb des Aspektes werden beliebig viele Pointcuts sowie Advices definiert. Im Beispiel wird ein Pointcut “stateChange” erstellt. Dieser definiert, dass ein Pointcut beim Aufruf der Methoden “deposit” sowie “withdraw” mit beliebig vielen Argumenten innerhalb der Objektinstanz eines BankAccounts auftritt (Zeile 2-3).
Weiterhin wird ein After-Advice definiert. Dieser wird aufgerufen, nachdem ein Methodenaufruf innerhalb des oben definierten Pointcuts seinen Rückgabewert zurückgeliefert

hat. Die dort eingeflochtene Geschäftslogik ist hierbei nur mit einem Kommentar versehen. Innerhalb dieses Methodenrumpfs wird die Logik für das Persistieren von des BankAccounts eingebunden (Zeile 5-7).

Das Einpflegen von Aspekten in die vorhandene Programmlogik muss laut den oben genannten Kriterien möglichst nahtlos passieren. Deswegen hat das Entwicklerteam von AspectJ einen Compiler bereitgestellt, der das Weaving zur Compiletime übernimmt (vgl. Kapitel 2.3.2). Komponenten die von den definierten Aspekten nicht betroffen sind, werden von einer klassischen JVM kompiliert [Kic+01].

4.3 Spring AOP

Spring AOP ist ein Framework für Aspekt-orientiertes Programmieren innerhalb der Spring-Plattform [Spr]. Das Spring Framework wird unter Anderem zur Erstellung von Webapplikationen genutzt und ist unter den vorhandenen Java Frameworks für Webapplikationen das beliebteste [The]. Spring AOP ist somit ebenfalls eine AOP-Lösung für die Programmiersprache Java, allerdings beschränkt sich hierbei die Nutzung auf die eben genannte Plattform - eigenständig lässt sich diese nicht verwenden.

Spring AOP unterstützt eine Definition von Aspekten mittels Spring XML Konfiguration [Sch] sowie dem AspectJ-Stil mittels Java-Annotationen [Sup]. Der zweite Stil wurde im vorherigen Kapitel nicht näher erläutert, da sich speziell dieses Kapitel mit dem Thema des AspectJ-Annotations-Stils während der Vorstellung von Spring AOP befasst.

Im Folgenden wird ein Beispiel einer Aspekt-Deklaration mittels XML gezeigt.

Listing 4.2: Ein Beispielaspekt mittels XML in Spring AOP [Sch]

```
1  <aop:config>
2    <aop:aspect id="myAspect" ref="aBean">
3      <aop:pointcut id="businessService"
4          expression="execution(* com.xyz.myapp.service.*.*(..))"/>
5    ...
6    </aop:aspect>
7  </aop:config>
8
9  <bean id="aBean" class="...">
10   ...
11 </bean>
```

An dieser Stelle wird ein Beispielaspekt präsentiert, der im gegebenen Kontext ohne Funktion agiert, da seine nähere Implementation mittels eines POJOs nicht weiter beschrieben wird, da es hier lediglich um die Deklaration von Aspekten geht. Zeile 1-7 stellen den XML-Tag für die Konfiguration von Spring AOP dar. Dieser ist verpflichtend.

Innerhalb dieses Tags werden alle Aspekte innerhalb des Programmes definiert. Innerhalb der “aop:config” können beliebig viele Aspekte definiert werden. Im Bereich eines Aspektes können nun wiederrum beliebig viele Pointcuts sowie Advices definiert werden, ähnlich der Pointcut-Sprache von AspectJ (vgl. Kapitel 4.2).

Weiterhin lässt sich, wie schon kurz zuvor angedeutet, ein Aspekt in Spring AOP mittels der Annotationssprache von AspectJ definieren. Spring AOP nutzt nämlich die AspectJ Definition von Aspekten sowie deren Pointcut-Sprache [Spr]. Allerdings werden die Aspekte innerhalb der Spring Plattform nicht während der Compiletime in die Programmlogik eingewoben. Hierbei wird der Ansatz des Runtime-Weaving verwendet. Mittels sogenannter Proxies [Pat] wird die Funktionalität der klassischen Programmteile zur Laufzeit dynamisch erweitert.

Spring AOP stellt in diesem Zusammenhang durch das beschriebene sogenannte “Proxying” einen Mechanismus zum Erweitern der Methodenfunktionalität für eine Klasse dar. Das Erweitern der Zugriffe auf Objektfelder ist aufgrund der Limitierung des Proxy-Mechanismus nicht möglich [Spr].

Im Folgenden wird ein Beispielaspekt mittels Definition durch AspectJ-Annotationen gezeigt.

Listing 4.3: Ein Beispielaspekt mittels AspectJ-Annotationen in Spring AOP [Sup]

```
1  @Aspect
2  public class BeforeExample {
3
4    @Pointcut("execution(* com.xyz.someapp.dao.*.*(..))")
5    public void dataAccessOperation() {}
6
7    @Before("dataAccessOperation()")
8    public void doAccessCheck() {
9      // modifying code here
10   }
11
12 }
```

Der Unterschied zum klassischen AspectJ-Stil (vgl. Kapitel 4.2) ist, dass hierbei simpele POJOs zur Deklaration von Aspekten verwendet werden. Es gibt kein syntaktisches Konstrukt für einen Aspekt. Die Java-Klasse muss jedoch mit einer Annotation versehen werden, damit während der Laufzeit festgestellt werden kann, dass es sich hierbei um einen Aspekt handelt (Zeile 1). Pointcuts und Advices sind normale Methoden einer Java-Klasse - hier existiert ebenfalls kein Sprachkonstrukt zur diesbezüglichen Definition. (Zeile 5+8). Ein Pointcut wird mit einer *@Pointcut*, ein Advice mit der enstprechenden Advice-Annotation versehen (Zeile 4+7). In diesem Fall handelt es sich um einen

Before-Advice, was bedeutet, dass der Code der Methode vor dem Aufruf des jeweiligen JoinPoints ausgeführt wird.

4.4 YAPAF

In diesem Kapitel wird das YAPAF-Framework vorgestellt. YAPAF ist eine Abkürzung und steht für "Yet another PHP Aspect Framework". Es wurde 2012 im Rahmen einer Diplomarbeit von Markus Schulte konzipiert und realisiert [Sch12].

Das Framework basiert auf dem Grundsatz, die Plattformkompatibilität zu wahren. Dies bedeutet, dass jedes YAPAF Programm ein gültiges PHP Programm darstellt. Somit ist dieses mit einem klassischen PHP-Interpreter ausführbar. Das Konzept basiert auf dem Einlesen der PHP-Klassen mittels eines Streamers. Funktionen werden nach Aspekt-Definitionen analysiert und, falls notwendig, in die Aspektlogik eingewoben. Entsprechend geschieht das Einweben von Aspekten hierbei ebenso zur Laufzeit wie bei Spring AOP [Sch12].

Listing 4.4: Ein Beispielaspekt realisiert mit YAPAF (eigene Darstellung)

```
1  <?php
2
3    class SimpleClass {
4
5      /**
6      * @Call(class="SomeClass", method="someMethod")
7      */
8      public function aspectMethod() {
9        // do something here
10     }
11 }
```

Das Listing zeigt einen Beispielaspekt, der in YAPAF realisiert wurde. Innerhalb einer normalen PHP-Klasse existiert eine beliebige Methode, welche in bestimmte JoinPoints eingewoben werden soll (Zeile 8-10). Auf jeder Methode befinden sich beliebig viele Annotationen, die sogenannten "ClassPropertyJoinpoints" [Sch12]. Dies sind angefertigte Annotationen, welche Pointcuts darstellen sollen. Innerhalb dieser Annotation wird ein Feld "class" sowioe "method" definiert. Diese Felder beschreiben, welche Methode einer definierten Klasse um Funktionalität erweitert werden soll. Eine Aspektmethode kann mit beliebig vielen Annotationen versehen werden.

4.5 Aquarium

Aquarium ist eine AOP-Erweiterung für die Programmiersprache Ruby [Lan]. Entwickelt wurde sie von Dean Wampler im Jahre 2008. Das Ziel dieses Frameworks ist es eine mächtige Erweiterung für eine dynamisch typisierte Sprache wie "Ruby" bereitzustellen

[Wam08]. Ruby ist eine sehr beliebte Objekt-orientierte Programmiersprache, welche in einer Beliebtheitsskala unter den ersten Zehn liegt [Jul]. Das Aquarium-Projekt liegt momentan auf GitHub und wird vom Begründer aktuell nicht weiterentwickelt. Jedoch bietet Wampler jedem die freie Weiterentwicklung dieses Projektes an. Im folgenden werden die Ziele dieses Projektes erläutert:

- Dynamische Einbindung von Aspekten zur Laufzeit
- Eine “intuitive” Syntax
- Die Möglichkeit Java mit Hilfe von JRuby anzusprechen
- Die Möglichkeiten von AOP innerhalb Ruby zu demonstrieren

Vor allem der letzte Punkt ist wichtig für dieses Projekt. Ruby unterstützt durch Metaprogrammierung das Erweitern von Methoden. Somit ist ein Sprachkonstrukt von AOP bereits durch integrierte Ruby-Sprachmittel abgedeckt. Jedoch fehlt dieser Sprache die Möglichkeit Advices zu definieren, , womit das Definieren der Eintrittspunkte in die Programmlogik dem Entwickler überlassen bleibt [Wam08].

Aquarium unterstützt sowohl direkte Einbinden von Aspekten in eine Klasse, als auch die Definition von Aspekten in einer eigenständigen Datei. Das Präsentieren der vielen Möglichkeiten der Aspekteingliederung ist nicht Bestandteil dieser Ausarbeitung. Folglich konzentriert diese sich auf eine spezielle Form der Einbindung von Aspekten, die im Folgenden vorgestellt wird.

Listing 4.5: Ein Beispielaspekt Aquarium [Git]

```
1 require 'aquarium'
2 Aspect.new :around, :calls_to => :all_methods, :on_types => [Foo, Bar] do |
      join_point, object, *args|
3     p "Entering: #{join_point.target_type.name}##{join_point.method_name} for
          object #{object}"
4     result = join_point.proceed
5     p "Leaving: #{join_point.target_type.name}##{join_point.method_name} for
          object #{object}"
6     result  # block needs to return the result of the "proceed"!
7 end
```

In Zeile 1 wird das benötige Aquarium Paket eingebunden. Dies ist essentiell. Im Folgenden wird ein neuer Aspekt per Konstruktor erstellt (Zeile 2-7). Hierbei wird ein Around-Advice für alle Methoden von Objekten erstellt, die entweder vom Typ “Foo” oder “Bar” sind. Beim Aufruf der Methode wird auf die Standardausgabe eine Nachricht ausgegeben. Nach dem Ausführen der Methode wird erneut eine Nachricht, die das Verlassen der Methode signalisiert, gesendet.

4.6 Vergleich

Zuvor wurden vier unterschiedliche AOP-Lösungen für drei verschiedene Programmiersprachen vorgestellt. Die Mächtigkeit dieser Lösungen wird in diesem Kapitel untersucht. Hierbei geht es speziell um die Funktionalität, sprich, mit welchen Sprachkonstrukten sie den Entwickler bei seiner Arbeit unterstützen. Die gezeigte Tabelle vergleicht die vier vorgestellten Lösungen.

	AspectJ	SpringAOP	YAPAF	Aquarium
Definition von Pointcuts	✓	✓	✓	✓
Definition von Advices	✓	✓		✓
Zugriff auf private Attribute	✓			
Zugriff auf Joinpoint-Signatur	✓	✓	✓	✓
Deklaration von Vererbung	✓			
Deklaration von Annotationen	✓			
Nutzung primitiver Pointcuts	✓	✓	✓	✓
Dynamische Einbindung von Aspekten				✓

Tabelle 4.1: Vergleich der vorgestellten AOP-Lösungen (eigene Darstellung)

Es zeigt sich, dass drei von vier Frameworks weitreichende Sprachkonstrukte zur Erstellung eigener AOP-Implementationen anbieten. Speziell Aquarium tritt mit der Möglichkeit hervor, Aspekte dynamisch zur Laufzeit einzubinden und zu entfernen. Dieses Kapitel nimmt jedoch keine weitere Wertung vor. Die Auswahl der AOP-Frameworks für die Implementationen für 101companies findet speziell in Kapitel 5.2 statt.

Kapitel 5

Vorbereitungen für die Implementationen

Im vorherigen Kapitel wurden verschiedene AOP-Technologien, Sprachen sowie Frameworks vorgestellt und diskutiert. Ebenfalls wurden diese auf ihre Sprachmittel hin miteinander verglichen. Dieses Kapitel befasst sich mit einer Selektion der vorgestellten Frameworks. Hierbei werden in Kapitel 5.2 Frameworks für die Implementationen in Kapitel 7 ausgesucht. Anhand dieser werden Codebeispiele bezüglich der Adaptierung von AOP in eine gegebene Plattform demonstriert. Zuvor werden hierfür in Kapitel 5.1 die Features aus dem Featuremodell abgeleitet, welche für AOP relevant sind, da sie nämlich einen Cross-Cutting-Concern darstellen (vgl. Kapitel 2.2).

5.1 Ableitung relevanter Features

In Kapitel 3.2 wurden die Features von 101companies vorgestellt. AOP ist eine relevante Spracherweiterung für CCCs und somit ist nicht jedes Feature sinnvoll durch AOP umsetzbar beziehungsweise erweiterbar. In diesem Kapitel werden die für AOP relevanten Features vorgestellt. Die Sinnhaftigkeit der Umsetzung des jeweiligen Features wird mit Hilfe von AOP wird zusammenfassend aufgezeigt.

Total

Mittels Inter-type Deklaration können verschiedene Entitäten, beispielsweise Firma oder Abteilung, mit zusätzlicher Funktionalität ausgestattet werden. Diese ist dann zentralisiert definiert und entspricht somit dem Modularitätsprinzip.

Depth

siehe **Total**

Cut

siehe **Total,Depth**

Median

siehe **Total,Depth,Cut**

Ranking

Das Ranking soll eine Firma, beziehungsweise deren Abteilungen nach einem bestimmten Schema validieren. Validierung ist ein CCC im klassischen Sinne. Im klassischen Ansatz muss der Entwickler den Code mit Validierungsaufrufen durchziehen. Mittels AOP kann die Validierungslogik zentral implementiert werden.

History

Der Mechanismus der Historisierung durchstreift verschiedene Objekte. Der Entwickler muss Objekte definieren, welche bei Veränderung einen Eintrag erstellen und somit Unterschiede sichtbar machen. Mittels AOP kann diese Funktionalität zentralisiert und einheitlich implementiert werden.

Persistence

Bei Persistence spielen verschiedene Faktoren eine Rolle, die für AOP relevant sind. Beispielsweise lassen sich Ressourcen wie Verbindungen und Transaktionen zentral und vereinheitlicht verwalten. Dadurch lässt sich dem Entwickler die Möglichkeit bieten, das Schreiben von fehleranfälligem Code zu vermeiden. Außerdem muss dieser sich nicht um die Verwaltung von Ressourcen kümmern.

Logging

Das Protokollieren von Ereignissen innerhalb eines Systems stellt ebenfalls einen CCC dar. Anstatt das System mit Logik für das Logging zu durchziehen, kann der Entwickler einheitlich Geschäftslogik für dieses Feature implementieren. Systemereignisse können so in einheitlicher Form automatisiert protokolliert werden. Das mindert unter anderem den Wartungsaufwand.

Weitere Features kommen für eine Implementierung mittels AOP-Technologien nicht infrage. Entweder stellen gewisse Features einerseits keinen CCC dar, wie beispielsweise Anforderungen an das Datenmodell oder Interface, oder andererseits bietet AOP für gewisse Features keinen Mehrwert. Ein Beispiel hierfür ist das **Parallelism** Feature (vgl. Kapitel 3.2.3). Zwar ist der Entwickler in der Lage mittels AOP hier ebenfalls eine Ressourcenverwaltung zu implementieren, jedoch werden diese bereits durch spracheigene Mittel, beispielsweise in Java, hinreichend Verwaltet.

5.2 Ableitung relevanter Technologien

Zuvor wurde das Featuremodell hinsichtlich seiner Tauglichkeit für eine AOP Unterstützung diskutiert. Nun werden die in Kapitel 4 gezeigten AOP-Technologien hinsichtlich 101companies untersucht. Es wird diskutiert, inwiefern die gelistete Technologie sich hinreichend für eine Implementation einsetzen lässt.

AspectJ

AspectJ ist die erste existierende Lösung für Aspekt-orientierte Unterstützung. Es wurde von Kiczales et al. entworfen und wird seitdem aktiv weiterentwickelt. Java ist eine der am meisten genutzten Objekt-orientierten Sprachen (vgl. Kapitel 4.2). Darüber hinaus bietet AspectJ ein breites Spektrum an Sprachmitteln, die sich anhand der 101companies Plattform demonstrieren lassen (vgl. Kapitel 4.6). Als Referenzimplementierung wird AspectJ den Kriterien an eine AOP-Technologie gerecht. Deshalb wird diese Technologie für eine Implementation in Kapitel 7.1 herangezogen. Wie in Kapitel 6.1 gezeigt wird, existiert innerhalb der 101companies Plattform bereits eine Implementierung mit AspectJ auf die in dem entsprechenden Kapitel noch detailliert eingegangen wird.

Spring AOP

Spring AOP bietet eine Lösung, die an das Spring Framework angepasst ist. Dadurch ist es durchaus in seinen Möglichkeiten eingeschränkt. Jedoch erfreut sich die Plattform des Spring-Frameworks großer Beliebtheit (vgl. Kapitel 4.3). Mittels dieser Erweiterung wird AOP in das Spring-Framework adaptiert und bietet eine gute Möglichkeit zur Nutzung von AOP innerhalb von Spring-Komponenten. Als weitere Lösung innerhalb der Java-Welt bietet sich Spring AOP hiermit ideal an, um einen Vergleich zu AspectJ aufzuzeigen. In Kapitel 7.2 wird eine Implementation von 101companies mit Hilfe von Spring AOP vorgestellt.

YAPAF

Das YAPAF-Framework für Aspekt-Orientierung in PHP ist nicht geeignet, um eine eigene Implementation für die 101companies Plattform zu stellen. Zwar ist das Framework ansich erweiterbar, jedoch bietet es nur die Möglichkeit, Methodenaufrufe mittels des Call-Pointcuts zu realisieren (vgl. Kapitel 4.4). Die Nutzung dieses Sprachmittels wird in der bereits vorhandenen Implementation zu YAPAF, welche in Kapitel 6.2 analysiert wird, hinreichend dargestellt. Eine weitere Verdeutlichung dieses Sprachmittels ist nicht notwendig. Da es nicht das Ziel dieser Arbeit ist, AOP-Lösungen mit weiteren Sprachmitteln zu erweitern, wird von einer Implementation mit YAPAF abgesehen.

Aquarium

Die AOP-Erweiterung Äquariumfür die Programmiersprache Ruby bietet dank ihrer Pointcut-Sprache eine sinnvolle Erweiterung zur klassischen Metaprogrammierung in Ruby. Zudem wird ungeachtet des Mangels einer eigenen Implementation

für YAPAF eine AOP-Lösung für eine dynamisch typisierte Programmiersprache präsentiert. Ebenfalls ist Ruby eine der am häufigsten genutzten Objekt-orientierten Programmiersprachen (vgl. Kapitel 4.5). Zwar wird Aquarium zur Zeit nicht weiterentwickelt, jedoch wäre aufgrund der Tatsache, dass das Projekt Open-Source ist, eine Weiterentwicklung denkbar. Aufgrund der vorhandenen weitgreifenden Sprachmittel und den bereits genannten Gründen, eignet sich Aquarium für eine eigene Implementation in Ruby mit Hilfe von AOP. Diese wird in Kapitel 7.3 vorgestellt.

Von vier vorgestellten AOP-Lösungen werden also drei zur Umsetzung mittels Implementation für die 101companies Plattform herangezogen. Mit AspectJ und Spring AOP werden zwei Lösungen für Java vorgestellt. Während AspectJ die Referenzimplementierung für AOP in Java darstellt, erweitert Spring AOP das hauseigene Spring Framework mit AOP-Sprachmitteln. Mit Aquarium wird eine Lösung für AOP in einer dynamisch typisierten Sprache – Ruby – vorgestellt, wohingegen Java eine statisch typisierte Sprachplattform darstellt (vgl. Kapitel 4.2).

Kapitel 6

Existierende Aspekt-orientierte Implementationen

In Kapitel 5.1 wurden relevante Features sowie die relevanten Aspekt-orientierten Technologien für den Kontext dieser Arbeit abgeleitet. Im Repository von 101companies exitstieren bereits Implementationen für die Aspekt-orientierte Programmierung. Namentlich handelt es sich um die Implementationen AspectJ [Impa], welche, wie der Name es verrät die Technologie AspectJ behandelt, sowie YAPAF [Impb], welche eine AOP Implementiertung in der Sprache PHP umsetzt. Beide Technologien wurden in Kapitel 4.2 sowie 4.4 thematisiert. Im Folgenden werden beide 101companies Implementationen analysiert. Dabei wird speziell gezeigt, welche Bestandteile der Technologie anhand der Implementation demonstriert werden, und welche noch zu veranschaulichen sind.

6.1 AspectJ

Wie bereits geschildert, existiert auf der 101companies Plattform eine Implementation für aspectJ. Dieses Kapitel zeigt die wesentliche Umsetzung der Features mithilfe von AspectJ.

aspectJ setzt bereits vier Features von 101companies mit Hilfe von AOP um. Dazu werden drei Entitäten innerhalb des Projektes bereigestellt. Im Package *org.softlang.company* werden die Klassen Company, Department und Employee bereitgestellt. Diese sind gewöhnliche POJOs mit Datenfeldern gemäßdes Featuremodells von 101companies. Zusätzlich enthalten die Klassen die jeweiligen Getter und Setter für die Datenfelder. Weitere Funktionalität wird konkret ausgeschlossen.

Im Folgenden werden die vier implementierten Features vorgestellt.

6.1.1 Logging

Das Logging-Feature soll mögliche Ereignisse innerhalb der Plattform protokollieren. Dieses wird mit Hilfe von zwei verschiedenen Aspekten umgesetzt. Im Package *org.softlang.features* befinden sich die Aspekte Logging und Polymorphism.

Der Aspekt Polymorphism dient zur Vereinheitlichung der verschiedenen Entitäten. Mittels einer einheitlichen Schnittstelle werden gleiche Methoden innerhalb der verschiedenen Entitäten polymorph zusammengefasst.

Listing 6.1: Polymorphism.aj - Nachträgliche Veerbung einer Schnittstelle

```
1   declare parents: Company implements Operations;
2   declare parents: Department implements Operations;
3   declare parents: Employee implements Operations;
```

Hier wird den Entitäten Company, Department und Employee die Anweisung gegeben, die Schnittstelle Operations zu erben. Dadurch werden Methoden unter einer gemeinsamen Schnittstelle definiert und können bei der Umsetzung des Logging-Features einheitlich verwendet werden.

Der Logging-Aspekt ist eine Kapselung der Logging-Funktionalität und besteht im Wesentlichen aus drei Teilen.

Listing 6.2: Logging.aj - Pointcut für Aufruf der cut-Methode

```
1   pointcut cut(Operations o):
2     target(o) && call(void Operations.cut());
```

Hier wird ein Pointcut namens “cut” definiert. Dieser tritt ein sobald, auf einer beliebigen Operations-Instanz die cut-Methode aufgerufen wird. Diese Methoden werden im Cut-Aspekt definiert (vgl. Kapitel 6.1.3) und hier nicht näher beschrieben. Zusätzlich wird im Pointcut festgelegt, dass das “target” des Aufrufes das Operations-Objekt selbst ist, welches in der Pointcut-Signatur als Parameter übergeben wird. Damit lassen sich parametrisierte Advices erstellen, wie im Folgenden noch demonstriert wird. Zunächst eine Veranschaulichung des Advices, der die Logging-Funktionalität erweitert.

Listing 6.3: Logging.aj - Around-Advice zur Umsetzung des Loggings

```
1   void around(Operations o): cut(o) {
2     message("BEGIN",o);
3     proceed(o);
4     message("END",o);
5   }
```

Zu Beginn wird eine Message-Methode mit dem Operations-Objekt als Parameter aufgerufen. Im Anschluss wird der Kontext des Joinpoints fortgeführt, sprich der Me-

thodenaufruf wird forgesetzt. Im Anschluss daran wird ein weiterer Aufruf der Message-Funktion ausgeführt.

Listing 6.4: Logging.aj - Message-Methode

```
1   private void message(String prefix, Operations o) {
2     System.out.println(
3     "> "
4     + prefix
5     + " Cut "
6     + o.getClass().getSimpleName()
7     + " \""
8     + o.getName()
9     + "\". Total: "
10    + o.total());
11  }
```

Wie aufgrund dieser Tatsache ersichtlich wird, wird beim Aufruf des Around-Advices zunächst der Status quo des Objektes an die Standardausgabe weitergereicht. Dabei werden der tatsächliche Klassenname der Operations-Instanz, dessen Name sowie das aufsummierte Einkommen ausgegeben. Nach dem erfolgreichen Abschluss der Methode wird diese Ausgabe erneut getätigt. Hierbei jedoch in nun verändertem Zustand, sprich die Gehälter der jeweiligen Instanz wurden gekürzt.

Es wurde hiermit also gezeigt, dass das Logging-Feature konkret vor und nach dem Methodenaufruf den Zustand der Operations-Instanz auf der Standardausgabe protokolliert. Dabei wurden als Sprachkonstrukte ein Pointcut sowie ein Around-Advice genutzt. Zusätzlich wurde mittels Inter-type Declaration ein Erbe der Operations-Schnittstelle für die Entitäten Company, Department sowie Employee definiert.

6.1.2 Total

Das Total-Feature hat den Zweck der Aufsummierung von Gehältern der jeweiligen Entität, sei es Company, Department oder Employee. Im klassischen Stil der Komposition würde für jede Entität eine eigene Methode zum Aufsummieren der Gehälter implementiert, und in dieser auch hinterlegt werden. Im Stil von AOP können alle Methoden, die im Zusammenhang stehen, als eigenes Modul innerhalb eines Aspektes gruppiert werden.

Listing 6.5: Total.aj - Inter-type Deklarationen von Total-Methoden

```
1   public double Company.total() {
2     double total = 0;
3     for (Department dept : getDepts())
4       total += dept.total();
5     return total;
6   }
7
8   public double Department.total() {
9     double total = 0;
10    total += getManager().getSalary();
```

```
11      for (Department s : getSubdepts())
12        total += s.total();
13      for (Employee e : getEmployees())
14        total += e.getSalary();
15      return total;
16    }
17
18    public double Employee.total() {
19      return getSalary();
20    }
```

Im hier gezeigten Total-Aspekt werden mittels Inter-type Deklarationen die entsprechenden Methoden für die jeweilige Klasse angelegt. Hierdurch wird die Klasse um diese Methode erweitert. Diese können dann entsprechend auf den konkreten Objektinstanzen aufgerufen werden.

Der Vorteil dieser Vorgehensweise liegt in der Grupperung gleichbedeutender Funktionalität. Dadurch kann diese als eigenständiges Modul gruppiert werden, wodurch sich der Wartungsaufwand und die Übersichtlichkeit aus der Perspektive des Entwicklers deutlich verbessern.

6.1.3 Cut

Das zuvor erwähnte Cut-Feature wird in dieser Implementation ebenfalls umgesetzt. Auch hier gelten gleiche Absichten, wie bei der zuvor analysierten Umsetzung des Total-Features.

Listing 6.6: Cut.aj - Inter-type Deklarationen von Cut-Methoden

```
1   public void Company.cut() {
2     for (Department dept : getDepts())
3       dept.cut();
4   }
5
6   public void Department.cut() {
7     getManager().cut();
8     for (Department s : getSubdepts())
9       s.cut();
10    for (Employee e : getEmployees())
11      e.cut();
12  }
13
14  public void Employee.cut() {
15    setSalary(getSalary() / 2);
16  }
```

Wie schon zuvor, werden hier mittels Inter-type Deklarationen die jeweiligen Entitäten um die Funktionalität der Cut-Operation erweitert. Innerhalb des Aspektes werden die verschiedenen Methoden somit als eigenständiges Modul gegliedert. Hierbei werden die Methoden für alle drei Klassen, also Company, Department sowie Employee umgesetzt.

6.1.4 Depth

Das Depth-Feature hat die Anforderung, die "Tiefe" einer Entität zu berechnen. Analog zu den Implementationen von Cut und Total wurde auch hier ein eigener Aspekt für das Depth-Feature implementiert.

Listing 6.7: Depth.aj - Inter-type Deklarationen von Depth-Methoden

```
1   public int Company.depth() {
2     int depth = 0;
3     for (Department d : getDepts())
4       depth = Math.max(depth, d.depth());
5     return depth;
6   }
7
8   public int Department.depth() {
9     int depth = 0;
10    for (Department d : getSubdepts())
11      depth = Math.max(depth, d.depth());
12    return ++depth;
13  }
14
15  public int Employee.depth() {
16    return 0;
17  }
```

Ebenfalls werden auch hier Inter-type Deklarationen umgesetzt, welche die drei bekannten Klassen mittels der Depth-Methoden erweitern.

6.1.5 Fazit

Die hier demonstrierte AspectJ-Implementation für 101companies setzt vier Features aus dem Featuremodell mit Hilfe von AOP um. Dabei zeigt lediglich ein Feature, nämlich das Logging, die Umsetzung von AOP mit Pointcuts und Advices und Inter-type Deklarationen. Die übrigen Features zeigen ausschließlich die Erweiterung von Klassen mittels Inter-type Deklarationen.

Die umgesetzten Features demonstrieren die Möglichkeiten von AspectJ nicht ausreichend. Somit ist die Umsetzung einer eigenen Implementation, wie sie in Kapitel 7.1 gezeigt wird, sehr sinnvoll, um weitere Sprachkonstrukte zu präsentieren.

6.2 YAPAF

Die Implementation für YAPAF zeigt im Gegensatz zur AspectJ-Implementation nur die Demonstration des Logging-Features. Da YAPAF lediglich ein konzeptionelles Werkzeug darstellt, sind entsprechend vielfältige Sprachkonstrukte vergleichbar mit denen aus AspectJ nicht vorhanden. Deswegen genügt die Umsetzung eines einzelnen Features zur vollständigen Demonstration von AOP mit YAPAF.

6.2.1 Logging

Der Zweck des Logging-Features wurde bereits ausreichend beschrieben. Die Umsetzung dieses Features mittels YAPAF geschieht in einer eigenen PHP-Klasse. Ähnlich wird dies auch in AspectJ realisiert.

Listing 6.8: DbLogAspect.php - Logging-Feature Implementation mittels YAPAF

```
1  <?php
2
3    class DbLogAspect {
4
5      /**
6      * @Call(class="Db", method="getCompany")
7      * @Call(class="Db", method="getDepartment")
8      * @Call(class="Db", method="getEmployees")
9      * @Call(class="Db", method="update")
10     * @Call(class="Db", method="insert")
11     * @Call(class="Db", method="delete")
12     * @Call(class="Db", method="cutSalary")
13     * @Call(class="Db", method="cutCompanySalary")
14     * @Call(class="Db", method="companyTotal")
15     * @Call(class="Db", method="refreshDepartmentManager")
16     */
17     public function log() {
18       $file = fopen('/tmp/101companiesDbLog.txt', 'a');
19       fwrite($file, __METHOD__ . ': ' . print_r(func_get_args(), true));
20       fclose($file);
21     }
22 }
```

Die PHP-Klasse *DbLogAspect* besteht aus lediglich einer einzigen Methode, die die Funktionalität für den jeweiligen Pointcut stellt. Konkret werden auf der Methode "log" mehrere Pointcuts definiert. Diese werden mittels *@Call-Annotationen* umgesetzt. Innerhalb der Annotation wird die Klasse markiert, die beim Aufruf des Programmes gewoben werden soll. Diese wird durch die zu erweiternde Methode ergänzt.

Die hier vorgestellte Implementation protokolliert verschiedene Methoden der Db-Klasse, beispielsweise der Aufruf zum Aufsummieren der Company. Jeder entsprechend deklarierte Methodenaufruf wird dabei in einer Textdatei protokolliert. Dabei werden die Funktionsparameter mit in den Logeintrag integriert.

6.2.2 Fazit

Die vorgestellte Implementation setzt das Logging-Feature mittels AOP in YAPAF um. Aufgrund der geringen Umsetzung von Sprachkonstrukten für YAPAF ist eine Demonstration des Werkzeugs beziehungsweise der Sprache hinreichend abgeschlossen. Somit bedarf es keiner weiteren Implementation zur Demonstration.

Kapitel 7

Implementationen

Bisher wurden nötige Grundlagen erläutert, relevante AOP Technologien vorgestellt sowie Features zur Implementation ausgewählt und diskutiert. Ebenfalls wurden bestehende Implementationen von AOP auf 101companies analysiert. Dieses Kapitel beschäftigt sich mit den Implementationen, die als Teil dieser Arbeit angefertigt wurden. Dabei wird diskutiert, inwiefern sich eine Verbesserung eines Projektes durch AOP ergibt.

7.1 AspectJ

Dieses Kapitel beschäftigt sich mit der angefertigten Implementation für AspectJ. Für AspectJ existiert bereits eine Implementation auf der 101companies Plattform, wie Kapitel 6.1 bereits gezeigt hat. Dort wurde jedoch auch aufgezeigt, dass die existierende Implementation unvollständig ist und die AOP-Konzepte nicht hinreichend abdeckt. Die im Folgenden gezeigte, eigens angefertigte, Implementation erweitert die bestehende um weitere Features. Hierzu wird die existierende Implementation als Grundlage für die folgenden Erweiterungen genutzt, sprich der existierende Code wird, soweit möglich, in die neue Implementationen einfließen.

Die grundlegenden Entitätsklassen, sprich Company, Department und Employee, sind denen aus der existierenden Implementation gleichzusetzen. Zusätzlich wurden diese durch Annotationen erweitert. Diese sind für die Persistence-Implementation notwendig, da diese auf der Java Persistence API, kurz JPA, [API] aufbaut. Folgendes Beispiel verdeutlicht die Anpassungen, welche an den Klassen vorgenommen wurden:

Listing 7.1: Company.java - Modifizierte Klasse zur Verwendung mit JPA

```
1  @Entity
2  public class Company implements Serializable {
3
4    private static final long serialVersionUID = -200889592677165250L;
5
6    @Id
7    @GeneratedValue(strategy = GenerationType.AUTO)
```

```
8     private Long id;
9
10    private String name;
11
12    @OneToMany(cascade = CascadeType.ALL)
13    private List<Department> depts = new LinkedList<Department>();
14
15    public String getName() {
16      return name;
17    }
18
19    public void setName(String name) {
20      this.name = name;
21    }
22
23    public List<Department> getDepts() {
24      return depts;
25    }
26
27    public Long getId() {
28      return id;
29    }
30
31    public void setId(Long id) {
32      this.id = id;
33    }
34  }
```

Die grundlegende Klasse wurde um JPA-spezifische Annotationen erweitert, um die Möglichkeit zur Persistenz zu gewinnen. Dadurch können entsprechende Objektinstanzen in der Datenbank gespeichert und aus dieser wieder geladen werden, ohne SQL Kenntnisse erforderlich zu machen. Die Darstellung der Funktionsweise der JPA ist nicht Bestandteil dieser Arbeit und wird somit nicht weiter diskutiert.

Analog zu der vorgestellten Company-Klasse wurden Department und Employee ebenfalls modifiziert und für die JPA erweitert.

Da die existierende AspectJ-Implementation bereits einige Features umsetzt, jedoch bei weitem nicht alle Möglichkeiten von AspectJ aufzeigt, werden nun mit Hilfe der folgenden Implementationen weitere Möglichkeiten aufgezeigt, mit AspectJ ein Softwareprojekt sinnvoll zu erweitern.

7.1.1 Logging

Das Logging-Feature wurde bereits in der existierenden AspectJ-Implementation thematisiert. Allerdings basiert diese Implementation auf einer veralteten Spezifikation der Anforderungen für Logging. Deshalb wird in diesem Kapitel eine erneuerte Version des Logging-Features gezeigt und deren Sinnhaftigkeit diskutiert.

Die angepasste Spezifikation des Logging erfordert das prokollieren von Gehaltsänderungen.

Dies geschieht ähnlich wie in der alten Spezifikation, jedoch wird hierbei der Hintergrund der Analyse von Protokolldaten thematisiert. Im Endeffekt sollen Gehaltsänderungen von Mitarbeitern der Firma analysiert werden. Beispielsweise soll der Median der Differenzen von altem zu neuem Gehalt sichtbar gemacht werden.

Eine solche Spezifikation erfordert, dass Gehaltsänderungen möglichst gekapselt abgelegt, und für eine spätere Verwendung verfügbar gemacht werden. Hierfür wurde die Klasse *LogEntry* angelegt.

Listing 7.2: LogEntry.java - Klasse zur Speicherung von Gehaltsänderungen

```
1  package org.softlang.utils;
2
3
4  public class LogEntry {
5
6    private String name;
7
8    private Double oldSalary;
9
10   private Double newSalary;
11
12   public LogEntry(String name, Double oldSalary, Double newSalary) {
13     this.name = name;
14     this.oldSalary = oldSalary;
15     this.newSalary = newSalary;
16   }
17
18   public String getName() {
19     return name;
20   }
21
22   public Double getOldSalary() {
23     return oldSalary;
24   }
25
26   public Double getNewSalary() {
27     return newSalary;
28   }
29
30   @Override
31   public String toString() {
32     return "LogEntry {name=\"" + name + "\", oldSalary=\"" + oldSalary
33         + "\", newSalary=\"" + newSalary + "\"}";
34   }
35 }
```

Die Klasse besteht aus simplen Datenfeldern, welche den Namen des Mitarbeiters, sein altes sowie das neue Gehalt speichern. Diese können im Nachhinein mittels Getter ausgelesen werden.

Die eigentliche Funktionalität des Loggings wird mit der Klasse *CompanyLogger* bereitgestellt. Diese stellt notwendige Operationen zur Protokollierung und weiteren Aufberei-

tung gespeicherter Daten bereit.

Listing 7.3: CompanyLogger.java - Bietet notwendige Logging-Funktionen

```
1  package org.softlang.utils;
2
3  import java.util.LinkedList;
4  import java.util.List;
5
6  public class CompanyLogger {
7
8    private List<LogEntry> loggedEntries;
9
10   private CompanyLogger() {
11     loggedEntries = new LinkedList<>();
12   }
13
14   public static CompanyLogger getInstance() {
15     return new CompanyLogger();
16
17   }
18
19   public void logCut(String employeeName, Double employeeOldSalary,
20       Double employeeNewSalary) {
21     LogEntry log = new LogEntry(employeeName, employeeOldSalary,
22         employeeNewSalary);
23     loggedEntries.add(log);
24
25     System.out.println(log);
26   }
27
28   public List<Double> getDeltas() {
29     List<Double> deltas = new LinkedList<>();
30     for (LogEntry entry : loggedEntries) {
31       deltas.add(entry.getNewSalary() - entry.getOldSalary());
32     }
33     return deltas;
34   }
35
36   public List<LogEntry> getLogs() {
37     return loggedEntries;
38   }
39
40 }
```

Eine Instanz der CompanyLogger Klasse hält eine Liste mit LogEntry Instanzen. Bei jedem Aufruf der Methode “logCut” wird für den Mitarbeiter ein eigenständiger LogEntry erzeugt und in der Liste abgelegt. Die Methode “getLogs” stellt alle gespeicherten LogEntry Instanzen zur Verfügung, wohingegen die ”getDeltasMethode eine Liste mit den Deltas der Gehälter in den gespeicherten Einträgen bereitstellt.

Die beiden vorgestellten Klassen stellen notwendige Hilfsklassen dar. LogEntry kapselt Protokolldaten und CompanyLogger stellt die Funktionen zur Protokollierung und Aufbereitung der Daten bereit. Die eigentliche, essentielle Logik enthält jedoch der Logging-

Aspekt, der im Folgenden diskutiert wird.

Listing 7.4: Logging.aj - Eine Instanz des CompanyLoggers als privates Feld

```
1   private CompanyLogger logger = CompanyLogger.getInstance();
```

Der Aspekt “Logging” beinhaltet zunächst eine Instanz des CompanyLoggers (Zeile 18). Die Instanz wird hier für demonstrative Zwecke gesetzt. Alternativ besteht die Möglichkeit eine konkrete Instanz des CompanyLoggers beispielsweise in einer Service-Klasse unterzubringen, von welcher aus direkt auf die Logging-Daten zugegriffen werden kann.

Listing 7.5: Logging.aj - Erweiterung der Company-Klasse um zwei weitere Datenfelder

```
1   /*
2    * Company's mean gets updated after each company cut
3    */
4   public double Company.deltaMean;
5
6   /*
7    * See Company.mean
8    */
9   public double Company.deltaMedian;
```

Im Folgenden wird mit Hilfe des Aspekts die Company Klasse um zwei weitere Datenfelder erweitert. Diese tragen im Anschluss an das Logging die entsprechenden Werte, die aus den Protokolleinträgen berechnet werden können. Hier ist ebenfalls eine alternative Möglichkeit gegeben. Zu Demonstrationszwecken werden die hier errechneten Werte jedoch direkt an die jeweilige Instanz der Company gespeichert. Einserseits sind diese Daten dann gespeichert und müssen nicht bei jedem Abruf neu berechnet werden, was bei sehr großen Unternehmen mitunter lange Berechnungszeiten erfordert, andererseits werden diese so direkt nach dem Aufruf der “cut”-Methode auf der Company Instanz automatisch berechnet, wie es im späteren Verlauf demonstriert wird.

Listing 7.6: Logging.aj - Pointcuts für Aufrufe von der cut-Methode

```
1   /*
```

Hier werden zwei Pointcuts definiert. Der Pointcut “cutEmployee” definiert den Joinpoint des Aufrufes der cut-Methode auf einer Employee-Instanz, während der Aufruf der gleichnamigen cut-Methode auf der Company-Instanz durch den Pointcut “cutCompany” definiert wird.

Listing 7.7: Logging.aj - Around-Advice für den cutEmployee Pointcut

```
1   void around(Employee e) : cutEmployee(e) {
2     double oldSalary = e.getSalary();
3     proceed(e);
4     logger.logCut(e.getName(), oldSalary, e.getSalary());
```

```
5     }
```

Der gezeigte Advice wird während des Aufrufs des Pointcuts “cutEmployee” aufgerufen. Hierbei wird zunächst das aktuelle Gehalt des Mitarbeiters temporär zwischengespeichert. Anschließend wird mittels “proceed” Anweisung der Methodenaufruf abgeschlossen. Im Anschluss daran wird mittels des bereits vorgestellten CompanyLoggers ein neuer Protokolleintrag erzeugt.

Listing 7.8: Logging.aj - After-Advice für den cutCompany Pointcut

```
1     after(Company c) : cutCompany(c) {
2       List<Double> deltas = logger.getDeltas();
3
4       c.deltaMedian = CompanyUtils.calculateMedian(deltas);
5       c.deltaMean = CompanyUtils.calculateMean(deltas);
6     }
```

Dieser After-Advice dient zur Berechnung der Statistikdaten im Anschluss an den Aufruf der cut-Methode einer Company. Der CompanyLogger stellt hierbei eine Liste mit Deltas der Protokolleinträge bereit. Mit Hilfe definierter statischer Hilfsmethoden werden hieraus sowohl der Median als auch der Modus aus den Werten errechnet und auf die zuvor definierten Datenfelder als Werte gesetzt.

Der hier aufgeführte Aspekt stellt eine Kapselung der Logging-Funktionalität dar. Sowohl die eigentliche Instanz des CompanyLoggers als auch die Advices zur Protokollierung der Ereignisse werden in einem Aspekt gegliedert. Diese Implementation demonstriert die Umsetzung eines Pointcuts mittels verschiedener primitiver, durch AspectJ vordefinierter Pointcuts. Dabei wird mittels des “target” Pointcuts der Abruf einer konkreten Employee-Instanz verfügbar gemacht.

7.1.2 History

Die Anforderung an das History-Feature ist, ähnlich dem Logging, das Protokollieren von Änderungen an Entitäten. Die Umsetzung dieses Features demonstriert speziell das Verändern von Daten der Employee Klasse. Die dabei angewendete Architektur lässt sich flexibel auf weitere Entitäten wie Department oder Company mit nur wenig Aufwand erweitern.

Die entwickelte Architektur basiert auf einer abstrakten Basisklasse *Historizable*, welche im folgenden Listing gezeigt wird.

Listing 7.9: Historizable.java- Generische, abstrakte Basisklasse für die Historisierung von Änderungen

```
1  package org.softlang.utils;
2
3  import java.io.Serializable;
```

```
4  import java.util.LinkedList;
5  import java.util.List;
6
7  public abstract class Historizable<T> implements Serializable {
8
9    private static final long serialVersionUID = 1L;
10
11   private List<T> history;
12
13   protected Historizable() {
14   }
15
16   public abstract T getCopy();
17
18   public List<T> getHistory() {
19     if (history == null)
20       history = new LinkedList<>();
21     return history;
22   }
23
24 }
```

Die Klasse wurde generisch parametrisiert, wodurch sich die Klasse beliebig vererben lässt. Darüber hinaus lassen sich die Instanzen vom entsprechend richtigen Typ in der Liste abspeichern. Weiterhin speichert die Klasse die History-Einträge in der entsprechenden Liste “history”, welche per Getter abgerufen werden kann. Der Konstruktor wurde entsprechend “protected” markiert. Ein Aufruf dessen aufgrund der Abstraktheit der Klasse nicht möglich. Die sogenannten History-Einträge entsprechen hierbei den eigentlichen Klasseninstanzen selbst. Der Vorteil daraus resultierende Vorteil ergibt sich dadurch, dass die Analyse der Klassen “herkömmlich” stattfinden kann, die Einträge also wie die eigentlich aktuelle Instanz behandelt werden.

Ein weiterer Bestandteil dieser Implementation ist der Aspekt selbst. Er dient der einheitlichen Formulierung, zu welchem Zeitpunkt Änderungen an den Entitäten entsprechend protokolliert werden. Die gezeigte Implementation demonstriert das Vorgehen konkret nur für die Employee-Klasse, lässt sich aber, wie bereits erwähnt, entsprechend für weitere Entitätsklassen erweitern.

Listing 7.10: History.aj- Inter-type Deklaration zur Vererbung der Historizable Basisklasse

```
1    declare parents: Employee extends Historizable<Employee>;
```

Die hier gezeigte Inter-type Deklaration definiert, dass die Employee-Klasse von der zuvor beschriebenen Historizable-Basisklasse erbt. Somit wird festgelegt, dass jede Instanz der Employee-Klasse bei entsprechenden Änderungen an den Datenfeldern historisiert wird.

Listing 7.11: History.aj - Überschreiben der abstrakten getCopy-Methode

```
1   public Employee Employee.getCopy() {
2     Employee copy = new Employee();
3     copy.setName(getName());
4     copy.setSalary(getSalary());
5     copy.setAddress(getAddress());
6
7     return copy;
8   }
```

Der Aspekt übernimmt die Implementierung der getCopy-Methode, welche in der Historizable-Klasse abstrakt definiert wurde. Für jede Klasse, welche anschließend Historizable erweitern soll, steht der Entwickler in der Pflicht, eine Implementation der getCopy-Methode bereitzustellen. Durch mögliche Unterschiede in der konkreten Implementierung bleibt dieses Vorhaben somit sehr flexibel.

Weiterhin werden drei verschiedene Pointcuts definiert. Diese beschreiben mit Hilfe von Wildcards verschiedene Programmeintrittspunkte.

Listing 7.12: History.aj - Definition von Pointcuts

```
1   /*
2    * Pointcut where getCopy method gets called
3    */
4   pointcut copyEntity() : call (* *.getCopy());
5
6   /*
7    * Pointcut where initialization of entity's constructor is pending
8    */
9   pointcut initializeEntity() : initialization(*.new(..));
10
11  /*
12   * Pointcut where any field is set. Consider to avoid calling this pointcut
         when in the control flow of above pointcuts
13   */
14  pointcut modification() : set(* *.*) && !cflow(copyEntity()) && !cflow(
        initializeEntity());
```

Zunächst wird ein Pointcut definiert, welcher den Aufruf einer getCopy-Methode einer beliebgen Instanz beschreibt. Weiterhin definiert der nächste Pointcut, namentlich "initializeEntity", den Joinpoint, während ein beliebiger Konstruktor aufgerufen wird. Der dritte Pointcut kombiniert die ersten beiden Pointcuts mit einer Erweiterung zu einem neuen. Im Zuge dessen wird festgelegt, dass jeglicher Schreibvorgang eines Attributes einer beliebigen Klasse, sofern dieser nicht während einem der beiden zuvor definierten Pointcuts geschieht, mit einem Joinpoint versehen wird. Damit wird im Folgenden der Mechanismus des automatischen Protokollierens der Änderungen möglich gemacht. Das Historisieren der Instanzen geschieht allerdings in dem folgenden Advice.

Listing 7.13: History.aj - Around-Advice

```
1   void around(Employee e) : target(e) && modification() {
2     Employee copy = e.getCopy();
```

```
3       proceed(e);
4       // create only history entries where changes were actually made
5       if (copy != null && !copy.equals(e)) {
6         e.getHistory().add(copy);
7       }
8     }
```

Der hier gezeigte Advice wird beim Auftritt des zuvor definierten modification-Pointcuts in den Programmcode eingewoben. Dieser erweitert herkömmliche Aufrufe und somit Modifikationen von Datenfeldern um folgende Funktionalität:

Zunächst wird die Employee-Instanz mittels der getCopy-Methode dupliziert. Anschließend wird das Programm mittels proceed-Anweisung fortgeführt. Im Anschluss daran wird das zuvor erstellte Duplikat der eigentlichen Instanz als History-Eintrag hinzugefügt. Bedingung dafür ist jedoch, dass sich das Duplikat wesentlich vom Original unterscheidet, sprich der Aufruf der "equals"-Methode *false* zurück gibt.

Die Implementation des History-Features demonstriert das Historisieren von Entitätsklassen in einer weitgehend gekapselten und flexibeln Lösung. Das System ist dank der gewählten Architektur mit wenigen Handgriffen flexibel erweiterbar. Der Entwickler, welcher das System durch weitere Entitätsklassen erweitert, muss lediglich eine Inter-type Deklaration, eine getCopy-Methode, sowie einen eigenen Advice implementieren. Da es durchaus unterschiedliche Lösungsansätze und Anforderungen an die entsprechenden Entitätsklassen gibt, ist es hier durchaus sinnvoll, die drei erwähnten Anweisungen manuell implementieren zu lassen. Auf einen Automatismus durch Wildcards oder Basisklassen wird hier bewusst verzichtet.

7.1.3 Persistence

Das Persistence-Feature stellt ansich keinen klassischen CCC dar. Allerdings lässt sich mit Hilfe von AOP eine weitaus flexiblere Architektur entwickeln. Dieser Ansatz wird in diesem Kapitel demonstriert.

Das übliche Vorgehen im Zuge der Nutzung einer Datenbank API ist die Kapselung der Funktionalität in einer eigenständigen Klasse, welche als Schnittstelle fungiert. Die in dieser Implementation umgesetzte Architektur sieht die prinzipielle Nutzung einer Service-Klasse vor, welche die entsprechende Entität aus der Datebank lädt sowie Änderungen dieser entsprechenden Instanzen in der Datenbank speichert. Das folgende Listing zeigt eine Solche.

Listing 7.14: CompanyDataAccess.java - Eine Service-Klasse für Datebankoperationen

```
1  package org.softlang.utils;
2
3  import javax.persistence.EntityManager;
```

```
4
5  import org.softlang.company.Company;
6
7  public class CompanyDataAccess {
8
9    private EntityManager em;
10
11   public Company findCompany() {
12     Company result = em.createQuery("FROM Company c WHERE c.name = :name",
           Company.class)
13         .setParameter("name", "meganalysis").getResultList().get(0);
14
15     return result;
16   }
17
18   public void insertCompany(Company c) throws Exception {
19     if(em.contains(c)) {
20       em.merge(c);
21     } else {
22       em.persist(c);
23     }
24     em.flush();
25   }
26
27 }
```

Diese Service-Klasse hat eine Instanz eines EntityManagers der JPA als Datenfeld, welcher als Schnittstelle zu den Datenbankoperationen genutzt wird. Weiterhin enthält diese Klasse simple Methoden zum Laden und Speichern einer Company. Diese sind nur schematisch umgesetzt, da zumindest die "find"-Methode nur eine hart kodierte Company aus der Datenbank laden kann. Eine solche Umsetzung reicht für die Demonstration von AOP jedoch vollkommen aus. Die Methode "insertCompany" soll durch AOP mit einer automatischen Steuerung der Datenbank-Transaktion unterstützt werden. Die dafür benötigte Programmlogik befindet sich im Persistence-Aspekt.

Listing 7.15: Persistence.aj - Erweiterungen durch Annotationen

```
1    /*
2     * Declare DAOs entity manager as auto created, hence we demonstrate
3     * dependency injection, or inversion of control
4     */
5    declare @field : * CompanyDataAccess.em : @AutoCreate;
6
7    /*
8     * Declare auto transaction on the insertCompany method, hence transaction
9     * begins, commits and rolls back automatically, see below
10    */
11   declare @method : * CompanyDataAccess.insertCompany(..): @AutoTransaction;
```

Hier werden Felder und Methoden mit eigenen Annotationen versehen. Zunächst wird die Instanz des EntityManagers einer jeden Instanz der CompanyDataAccess-Klasse mit der *@AutoCreate*-Annotation versehen. Dies stellt die automatische Injektion dessen sicher. Weiterhin wird die "insertCompany"-Methode mit der *@AutoTransaction*-Annotation versehen. Dadurch wird eine automatische Transaktionssteuerung während

des Methodenaufrufs gesichert.

Darüber hinaus wird eine eigene Instanz des EntityManagers als privates Feld innerhalb des Aspektes gespeichert. Diese Instanz wird dann an alle CompanyDataAccess-Instanzen injiziert.

Listing 7.16: Persistence.aj - EntityManager Instanz

```
1   /*
2    * Hold a single entity manager instance to inject
3    */
4   private EntityManager localEm = javax.persistence.Persistence
5       .createEntityManagerFactory("101PU").createEntityManager();
```

Das automatische injizieren des EntityManagers in die jeweilige Instanz wird mittels folgendem Pointcut und Advice sichergestellt.

Listing 7.17: Persistence.aj - Pointcut und Advice für die Injizierung des EntityManagers

```
1   /*
2    * Pointcut for accessing entity manager's instance
3    */
4   pointcut accessEntityManager(CompanyDataAccess dao) : target(dao) && (get(
        @AutoCreate * *.*) || within(Persistence) && get(* *.em));
5
6   /*
7    * Before-Advice to create entity manager automatically, hence inject it
8    */
9   before(CompanyDataAccess dao) : accessEntityManager(dao) {
10    dao.em = localEm;
11  }
```

Der Pointcut "accessEntityManager" beschreibt den Joinpoint, der beim Zugriff auf die konkrete EntityManager Instanz auftritt. Dabei wird sichergestellt, dass es nur Instanzen betrifft, welche die *@AutoCreate*-Annotation besitzen. Außerdem soll die Instanz bei Zugriff innerhalb des Aspektes sichergestellt werden, was beim Zugriff durch die Transkationssteuerung geschieht. Der zugehörige Before-Advice stellt eine Zuweisung, sprich Injektion der EntityManager-Instanz auf das entsprechende Datenfeld in der Instanz der CompanyDataAccess-Klasse sicher.

Die automatische Transaktionssteuerung wird durch den folgenden Pointcut sowie Advice definiert.

Listing 7.18: Persistence.aj - Pointcut und Advice für die Injizierung die automatische Transaktionssteuerung

```
1   /*
2    * Pointcut for calling methods with AutoTransaction annotation
```

```
3     */
4    pointcut callAutoTransactionMethod(CompanyDataAccess dao) : target(dao)
5                                   && call(@AutoTransaction * CompanyDataAccess
                                        .*(..));
6
7    /*
8     * Begin, commit and rollback transaction automatically, rethrow exceptions
9     * as runtimeexceptions to omit catching elsewhere
10    */
11   Object around(CompanyDataAccess dao) : callAutoTransactionMethod(dao) {
12     EntityManager em = dao.em;
13     EntityTransaction tx = em.getTransaction();
14     try {
15       if (!tx.isActive()) {
16         tx.begin();
17       }
18       Object result = proceed(dao);
19       if (tx.isActive()) {
20         tx.commit();
21       }
22       return result;
23     } catch (Exception e) {
24       if (tx.isActive()) {
25         tx.rollback();
26       }
27       // form exception message
28       final String exceptionMessage = "Exception thrown at " + thisJoinPoint.
             getSignature()+ ". Transaction rolling back.";
29       throw new RuntimeException(exceptionMessage, e);
30     }
31   }
```

Der zugehörige Pointcut definiert einen Joinpoint beim Aufruf jeglicher Methoden vom CompanyDataAccess, die die *@AutoTransaction*-Methode besitzen. Der entsprechende Around-Advice greift auf den EntityManager der Service-Klasse zu und beginnt eine Transakation. Nach erfolgreichem Methodenaufruf wird die Transaktion abgeschlossen. Beim Auftritt einer Exception wird die Transkation abgebrochen und ein Abschluss bleibt aus.

Die Implementation des Persistence Features zeigt eine gute Möglichkeit, AOP einzusetzen, um eine flexible Architektur zu erhalten. Das Erstellen und Verwalten der Framework-Ressourcen, hier der EntityManager und die entsprechend zugehörige Transaktion, können vom Container verwaltet werden. Dieses Vorgehen erhöht die Flexibilität, reduziert den Wartungsaufwand und vermeidet Fehler, die durch den Mangel einer automatischen Steuerung der Ressourcen entstehen können. Beispielsweise wird der Start oder der Abschluss einer Transaktion übergangen. Eine mögliche Verbesserung an dieser Stelle ist eine weitere Abstraktion der Service-Klasse. Durch den Einsatz einer Schnittstelle kann der Mechanismus weiter abstrahiert werden. Für die Demonstration von AOP anhand dieses Features ist dies jedoch nicht von Nöten.

7.1.4 Ranking

Beim Ranking-Feature soll beim Erstellen und Modifizieren von Instanzen der Employee-Klasse, speziell dem salary-Attribut, eine Überprüfung des sogenannten "Rankings" stattfinden. Dabei sollen bestimmte Bedingungen auf ihre bestehende Gültigkeit hin überprüft werden. Dieses Feature stellt, wie bereits beschrieben, einen CCC dar und kann durch eine AOP-Implementation flexibel umgesetzt werden.

Die Umsetzung einer effizienten Überprüfung benötigt eine eigene Klasse, welche die Kriterien für eine entsprechende Validierung implementiert. Das folgende Listing stellt eine solche Klasse, in dieser Implementation verwendete, Klasse dar.

Listing 7.19: SimpleRankingConstraint.java - Ein Beispiel einer Validierungsklasse

```
1  package org.softlang.utils;
2
3  import org.softlang.company.Department;
4  import org.softlang.company.Employee;
5
6  /*
7   * This Constraint is used for testing purposes - validate if given employee's
        salary isn't higher than any employee above him,
8   * hence if his manager and any employee in a higher department earns more money
       .
9   */
10 public class SimpleRankingConstraint implements RankingConstraint {
11
12   private static final String QUALIFIER = "TESTING_CONSTRAINT";
13
14   @Override
15   public boolean align(Employee e) {
16     Department topDep = e.getDepartment();
17     while (topDep != null) {
18       if (!align(topDep, e, topDep.getParent() != null)) {
19         return false;
20       }
21       topDep = topDep.getParent();
22     }
23     return true;
24   }
25
26   private boolean align(Department d, Employee e,
27       boolean ignoreEmployeeSalaries) {
28     if (d == null) {
29       return true;
30     }
31     if (d.getManager() != null
32         && d.getManager().getSalary() < e.getSalary()) {
33       return false;
34     }
35     if (!ignoreEmployeeSalaries) {
36       for (Employee empl : d.getEmployees()) {
37         if (!align(empl, e))
38           return false;
```

```
39          }
40        }
41        return true;
42      }
43
44      private boolean align(Employee e1, Employee e2) {
45        if (e1.getSalary() < e2.getSalary())
46          return false;
47        return true;
48      }
49
50      @Override
51      public String getConstraintQualifier() {
52        return QUALIFIER;
53      }
54
55  }
```

Die Validator-Klasse implementiert hierbei eine Schnittstelle, welche zuvor ebenfalls angelegt wurde und zur Abstrahierung der Architektur dient. Durch das Implementieren der Schnittstelle können so beliebige Validatoren geschrieben und das System somit erweitert werden. Anhand dieser Klasse wird eine spätere Validierung innerhalb des Aspektes vorgenommen.

Zunächst werden im Ranking-Aspekt die Department- und Employee-Klasse um jeweils ein Feld erweitert.

Listing 7.20: Ranking.aj - Erweiterung von Department und Employee

```
1     /*
2      * Add bijection to gain ability to validate through department employee is
3      * belonging to
4      */
5     private Department Employee.department;
6
7     /*
8      * Use bijection here, too
9      */
10    private Department Department.parent;
```

Die Department-Klasse erhält dabei eine Referenz auf ein eventuell höher gestelltes Department, sprich eine übergeordnete Abteilung. Die Klasse Employee erhält eine entsprechende Referenz auf die Abteilung, der sie zugehörig ist. Diese Datenfelder sind essentiell. Mit ihnen kann im weiteren Verlauf eine Validierung des Mitarbeiters durch die Validator-Instanz geschehen. Die beiden Datenfelder werden zusätzlich mit Getter- und Setter-Methoden erweitert.

Die folgenden Pointcuts definieren Eintrittspunkte, an denen eine nachträgliche Validierung des Mitarbeiters geschehen soll.

Listing 7.21: Ranking.aj - Pointcuts, bei denen eine Validierung stattfinden soll

```
1   /*
2    * Pointcut for Employee instantiation
3    */
4   pointcut createNewEmployee(Employee e) : execution(Employee.new(*)) && target(
        e);
5
6   /*
7    * Pointcut where employee's salary gets cut
8    */
9   pointcut cutEmployee(Employee e) : call(* Employee.cut()) && target(e);
10
11  /*
12   * Modification of employee's salary through setter call
13   */
14  pointcut setEmployeeSalary(Employee e) : call(* Employee.setSalary(*)) &&
        target(e);
```

Wie dieses Listing zeigt, werden die Instanziierung einer neuen Employee-Instanz, der Aufruf der cut-Methode sowie eine Veränderung des Mitarbeitergehaltes durch den entsprechenden Setter, als Methoden mit Validierungsoption festgelegt. Der nachfolgende Advice führt eine Validierung des Mitarbeiters beim Eintritt dieser Pointcuts durch.

Listing 7.22: Ranking.aj - After-Advice zur Validierung der Employee-Instanz

```
1   /*
2    * Advice above pointcuts through employee validation
3    */
4   after(Employee e) : createNewEmployee(e) || cutEmployee(e) ||
        setEmployeeSalary(e) {
5     // possibility to use factory method
6     RankingConstraint constraint = new SimpleRankingConstraint();
7     if (!constraint.align(e))
8       throw new RankingConstraintException(constraint);
9   }
```

Hierbei wird der Mitarbeiter nach seiner Modifikation, beziehungsweise Instanziierung, auf sein Ranking hin validiert. Das Beispiel-Ranking aus dieser Implementation stellt eine simple Prüfung dar. Das Gehalt des Mitarbeits darf nicht höher sein, als das seines Vorgesetzten, im Fachterminus "Manager" genannt, oder jeglicher Mitarbeiter höher gestellter Abteilungen liegen. Innerhalb des gezeigten Advices wird dementsprechend die Validierung ausgeführt. Bei Verletzung der Ranking-Bedingungen wird eine RankingConstraintException geworfen. Diese ist eine eigens angelegte Exception, welche die RuntimeException von Java erweitert.

Die hier gezeigte Implementation baut auf eine flexible Softwarearchitektur. Verschiedene Methoden, bei denen eine Validierung stattfinden soll, werden innerhalb eines Aspektes zusammengefasst. Die Erweiterung der Methodendefinitionen für die Validierungsoption ist somit flexibel erweiterbar. Der in der Anforderung beschriebene CCC lässt

sich hierbei mit Hilfe von AOP sauber kapseln und vermindert somit den Wartungsaufwand deutlich. Auch werden bestimmte Fehlerquellen vermieden. Beispielsweise kann der Entwickler das Einbauen der Validierungsoption vergessen. Durch die Tatsache, dass alle Methoden, welche validiert werden müssen, an einer Stelle definiert werden, verringert sich die Fehlerbehaftung der Programmlogik.

7.1.5 Median

Die Umsetzung des Median-Features ist im Vergleich zu den zuvor vorgestellten Feature-Implementationen sehr klein. Die Umsetzung besteht aus lediglich einem Aspekt, welcher drei Inter-type Deklarationen für die Klassen Company, Department sowie Employee festlegt.

Listing 7.23: Median.aj - Inter-type Deklarationen für Median-Methoden

```
1   public List<Double> Company.getAllSalaries() {
2     List<Double> companySalaries = new ArrayList<>(getDepts().size());
3     for (Department d : getDepts()) {
4       companySalaries.addAll(d.getAllSalaries());
5     }
6     Collections.sort(companySalaries);
7     return companySalaries;
8   }
9
10  public List<Double> Department.getAllSalaries() {
11    List<Double> departmentSalaries = new ArrayList<>();
12    departmentSalaries.add(getManager().getSalary());
13    for (Employee e : getEmployees()) {
14      departmentSalaries.add(e.getSalary());
15    }
16    for (Department subDep : getSubdepts()) {
17      departmentSalaries.addAll(subDep.getAllSalaries());
18    }
19    return departmentSalaries;
20  }
21
22  public Double Company.getMedian() {
23    List<Double> allCompanySalaries = getAllSalaries();
24    return allCompanySalaries.isEmpty() ? null : allCompanySalaries
25        .get(allCompanySalaries.size() / 2);
26  }
```

Für die Company- sowie die Department-Klasse werden jeweils eine Methode “getAllSalaries” angelegt. Diese erweitert die Klassen um eine Funktion, welche alle Gehälter der jeweiligen Klasse zusammenfasst und in Form einer Liste zurückgibt. Anschließend wird die Company-Klasse durch eine Methode “getMedian” erweitert. Diese dient dazu, den Median des Unternehmens zu berechnen. Die zuvor eingeführten Methoden dienen zur Bereitstellung und Verarbeitung der vorbereiteten Daten.

Die Umsetzung dieses Features mit AOP zeigt erneut die Möglichkeit, durch Inter-type Deklarationen die bestehende Funktionalität einer Klasse nutzbringend zu erweitern. Dabei werden alle benötigten Methoden in einer einheitlichen Form – hier der Aspekt – gekapselt.

7.2 Spring AOP

Dieses Kapitel diskutiert die Implementation für Spring AOP. Im Gegensatz zu AspectJ ist Spring AOP eine maßgeschneideterte AOP-Lösung für ein spezielles Framework, nämlich das Spring Framework. Dementsprechend ist es konzeptuell auf eben dieses Framework abgestimmt, was zur Folge hat, dass die Möglichkeiten, die Spring AOP bietet, im Vergleich zu AspectJ stark begrenzt sind.

Spring AOP bietet ausschließlich Spring-Komponenten Unterstützung mittels AOP. Die Möglichkeit, die Klassen Company, Department und Employee als solche Komponenten zu erstellen ist hierbei aufgrund softwaretechnischer Prinizipien unzulässig. Eine sauber getrennte Softwarearchitektur garantiert getrennte Schichten. Die Datendomäne hierbei als Service-Komponente zu markieren widerspricht der sauberen Trennung der Schichten und wird somit vermieden. Allerdings resultieren hieraus diverse Nachteile. Einerseits lassen sich hierdurch nur Methoden einer speziell angelegten Service-Klasse abfangen. Dadurch wird ein Entwickler, welcher an dem Softwareprojekt mitarbeitet, an die Nutzung der speziellen Methoden aus dieser Service-Klasse gebunden. Hierbei erhöht sich Dokumentations- und Koordinationsaufwand. Durch den Einsatz von AOP reduziert sich jedoch der Wartungsaufwand der Software.

In dieser Implementation werden Features, welche mittels Inter-type Deklarationen realisiert werden sollen, aufgrund von mangelnder Unterstützung nicht umgesetzt. Ebenfalls wird das Persistence-Feature nicht realisiert, da ein flexibler Ansatz nicht möglich ist. Dies liegt unter anderem an der Tatsache, dass es keine Möglichkeit gibt, einen Zugriff auf Datenfelder abzufangen. Außerdem ist das injizieren von Annotationen, oder sonstigen Meta-Informationen nicht realisierbar.

Die präsentierte Implementation von Spring AOP wird als Maven-Projekt [Mav] ausgeliefert. Die in Kapitel 7.1 demonstrierten POJO-Klassen werden in diesem Projekt ebenfalls eingesetzt. Diese enthalten jedoch keine JPA-spezifischen Meta-Informationen, sondern sind reine Java-Klassen.

Die folgende Klasse stellt den Employee-Service dar, eine Spring-Komponente, die mittels verschiedener Aspekt-Implementationen erweitert wird.

Listing 7.24: EmployeeService.java - Spring-Komponente zur Bearbeitung von Employee-Instanzen

```
1  package org.softlang.services;
2
3  import org.softlang.company.Department;
4  import org.softlang.company.Employee;
5  import org.softlang.utils.annotations.Historizing;
6  import org.springframework.stereotype.Component;
7
8  @Component
9  @Historizing
10 public class EmployeeService {
11
12   public Employee cut(Employee e) {
13     e.setSalary(e.getSalary() / 2);
14
15     return e;
16   }
17
18   public Employee setEmployeeSalary(Employee e, double salary) {
19     e.setSalary(salary);
20     return e;
21   }
22
23   public Employee setEmployeeName(Employee e, String name) {
24     e.setName(name);
25     return e;
26   }
27
28   public Employee setEmployeeAddress(Employee e, String address) {
29     e.setAddress(address);
30     return e;
31   }
32
33   public Employee createEmployee(String name, String address, double salary,
         Department dept) {
34     Employee e = new Employee();
35     e.setName(name);
36     e.setAddress(address);
37     e.setSalary(salary);
38     e.setDepartment(dept);
39     dept.getEmployees().add(e);
40     return e;
41   }
42
43 }
```

Diese Komponente bietet Operationen auf einer bestimmten Employee-Instanz. Dabei ist es, wie bereits erwähnt, wichtig, dass der Entwickler in einem entsprechenden Szenario ausschließlich diese Methoden innerhalb seiner Erweiterungen verwendet, damit das Softwareprojekt konfliktfrei bleibt und die Funktionalität gewährleistet wird.

7.2.1 Ranking

Die Umsetzung des Ranking-Features basiert teilweise auf Klassen der AspectJ-Implementation dieses Features. So werden die Klassen *SimpleRankingConstraint*, *RankingConstraintException* sowie das Interface *RankingConstraint* aus der AspectJ-Implementation zur Verwendung in diesem Projekt herangezogen. Der Aspekt "Ranking" ist eine Java-Klasse, welche die nötigen Meta-Informationen zur Erweiterung des Employee-Services für die Umsetzung des Ranking-Features bereitstellt.

Listing 7.25: EmployeeService.java - Spring-Komponente zur Bearbeitung von Employee-Instanzen

```
1  package org.softlang.aspects;
2
3  import org.aspectj.lang.annotation.AfterReturning;
4  import org.aspectj.lang.annotation.Aspect;
5  import org.aspectj.lang.annotation.Pointcut;
6  import org.softlang.company.Employee;
7  import org.softlang.utils.RankingConstraintException;
8  import org.softlang.utils.SimpleRankingConstraint;
9  import org.springframework.stereotype.Component;
10
11 @Aspect
12 @Component
13 public class Ranking {
14
15   @Pointcut("execution(public * org.softlang.services.EmployeeService.
         createEmployee(..))")
16   public void createNewEmployee() {
17   }
18
19   @Pointcut("execution(public * org.softlang.services.EmployeeService.
         setEmployeeSalary(..))")
20   public void setEmployeeSalary() {
21   }
22
23   @Pointcut("execution(public * org.softlang.services.EmployeeService.cut(..))")
24   public void cutEmployee() {
25   }
26
27   @AfterReturning(pointcut = "createNewEmployee() || setEmployeeSalary()",
         returning = "e")
28   public void checkForRanking(Employee e) {
29     SimpleRankingConstraint constraint = new SimpleRankingConstraint();
30     if (!constraint.align(e))
31       throw new RankingConstraintException(constraint);
32   }
33
34 }
```

Zunächst werden drei Pointcuts definiert. Sie beschreiben die Eintrittspunkte im Programm, die erweitert werden sollen – hier entsprechend ein Aufruf der Methode "createNewEmployee", das Setzen des Mitarbeitergehalts sowie das Beschneiden des Gehalts mittels der entsprechenden Service-Methoden. Der nachfolgende After-Advice (Zeile 27-

32) erweitert die entsprechenden Methodenaufrufe um eine Prüfung des Rankings. Bei Verletzung der Ranking-Bedingungen wird eine Exception geworfen. Der Advice bezieht seine Employee-Instanz aus dem Rückgabewert, welcher von der entsprechenden Service-Methode zurückgegeben wird.

Die Umsetzung dieses Features ist entsprechend auf die Nutzung der Service-Methoden eingeschränkt. Eine Erweiterung der POJO-Klassen ist nicht möglich. Damit ist diese Implementation bei weitem nicht so flexibel einsetzbar, wie die gezeigte Implementation in AspectJ. Dennoch bietet diese Umsetzung durchaus Vorteile, die zuvor schon diskutiert wurden. Die Implementation dieses Features zeigt die Definition von Pointcuts und Advices innerhalb von Spring AOP.

7.2.2 History

Die Umsetzung des History-Features unterscheidet sich in der internen Verarbeitung der History-Einträge für Mitarbeiter. Da Spring AOP die Erweiterungen mittels Proxy-Mechanismus realisiert, ist eine Vererbung von Basisklassen nicht realisierbar. Dementsprechend wird die History-Struktur auf eine Hilfsklasse ausgelagert, welche die Einträge für jeden Mitarbeiter verwaltet.

Listing 7.26: HistoryEntry.java - Datenkapsel für History-Einträge

```
1  package org.softlang.utils;
2
3  import org.softlang.company.Employee;
4
5  public class HistoryEntry {
6
7    private Employee employee;
8
9    private String name;
10
11   private String address;
12
13   private double salary;
14
15   public HistoryEntry(Employee toHistorize) {
16     this.employee = toHistorize;
17     this.name = new String(toHistorize.getName());
18     this.address = new String(toHistorize.getAddress());
19     this.salary = toHistorize.getSalary();
20   }
21
22   public Employee getEmployee() {
23     return employee;
24   }
25
26   public String getName() {
```

```
27      return name;
28    }
29
30    public String getAddress() {
31      return address;
32    }
33
34    public double getSalary() {
35      return salary;
36    }
37
38  }
```

Diese Klasse stellt eine Datenkapsel für History-Einträge dar. Sie speichert zum Zeitpunkt der Erstellung die aktuellen Daten eines Mitarbeiters, welcher als Parameter übergeben wird. Darüber hinaus wird eine Referenz zur aktuellen Instanz des Mitarbeiters hergestellt. Somit kann von jedem History-Eintrag direkt auf die aktuelle Revision der Employee-Instanz zugegriffen werden.

Die Klasse “HistoryMaintainer” ist eine Hilfsklasse, welche die verschiedenen History-Einträge verwaltet.

Listing 7.27: HistoryMaintainer.java - Hilfsklasse zur Historisierung von Mitarbeitern

```
1   package org.softlang.utils;
2
3   import java.util.Collections;
4   import java.util.LinkedList;
5   import java.util.List;
6   import java.util.Map;
7   import java.util.concurrent.ConcurrentHashMap;
8
9   import org.softlang.company.Employee;
10
11  public enum HistoryMaintainer {
12
13    INSTANCE;
14
15    private Map<Employee, List<HistoryEntry>> history = new ConcurrentHashMap<
          Employee, List<HistoryEntry>>();
16
17    public static HistoryMaintainer getInstance() {
18      return INSTANCE;
19    }
20
21    public synchronized List<HistoryEntry> getHistoryForEmployee(Employee e) {
22      if(!history.containsKey(e)) {
23        List<HistoryEntry> result = Collections.synchronizedList(new LinkedList<
              HistoryEntry>());
24        history.put(e, result);
25        return result;
26      }
27      return history.get(e);
```

```
28    }
29
30    public void historize(Employee e) {
31      HistoryEntry entry = new HistoryEntry(e);
32      getHistoryForEmployee(e).add(entry);
33    }
34  }
```

Diese Klasse ist als Enum angelegt. Dies stellt ein Hilfsmittel dar, um ein echtes Singleton-Objekt zu erzeugen. Die Thread-Sicherheit eines Enums stellt sicher, dass jeder Aufruf der getInstance-Methode auch die einzige Instanz der Klasse als Rückgabewert erhält. Die Klasse verwaltet eine Map, welche Employee-Instanzen einer Liste von History-Einträgen eindeutig zuordnet. Damit wird eine effiziente Suche nach History-Einträgen für eine Employee-Instanz sichergestellt.

Weiterhin wird für dieses Feature eine Annotation *@Historizing* bereitgestellt. Diese wird zur Laufzeit ausgewertet und ist ausschließlich zur Markierung von Typen – sprich Klassen – einsetzen. Diese Annotation soll eine Spring-Komponente entsprechend markieren. Damit wird sichergestellt, dass innerhalb dieser Klasse Methoden zur Historisierung von Mitarbeitern aufgerufen wurden können.

Der Aspekt "History" definiert, wie auch in AspectJ, die Pointcuts und Advices für dieses Feature. Zunächst wird eine Referenz auf die HistoryMaintainer-Instanz gesetzt.

Listing 7.28: History.java - Referenz der HistoryMaintainer-Instanz

```
1   private HistoryMaintainer history = HistoryMaintainer.getInstance();
```

Da ein Aspekt in Spring ebenfalls einer Spring-Komponente entspricht, lässt sich auf diese im Regelfall zugreifen. Dadurch kann man mit Hilfe eines Zugriffs auf die History-Klasse History-Einträge erhalten.

Listing 7.29: History.java - Methode zum Einholen von History-Einträgen

```
1   public List<HistoryEntry> getHistory(Employee e) {
2     return history.getHistoryForEmployee(e);
3   }
```

Diese Methode gibt eine Liste mit History-Einträgen für einen Mitarbeiter zurück. Dieser wird als Parameter an die Methode übergeben. Im weiteren Verlauf werden verschiedene Pointcuts definiert. Diese beschreiben die Joinpoints, an denen ein History-Eintrag für den entsprechenden Mitarbeiter erstellt werden soll.

Listing 7.30: History.java - Pointcuts für Historisierung

```
1   @Pointcut("execution(public * org.softlang.services.EmployeeService.
        setEmployeeSalary(..))")
```

```
2    public void setEmployeeSalary(){}
3
4    @Pointcut("execution(public * org.softlang.services.EmployeeService.
         setEmployeeName(..))")
5    public void setEmployeeName(){}
6
7    @Pointcut("execution(public * org.softlang.services.EmployeeService.
         setEmployeeAddress(..))")
8    public void setEmployeeAddress(){}
9
10   @Pointcut("execution(public * org.softlang.services.EmployeeService.cut(..))")
11   public void cutEmployeeSalary(){}
12
13   @Pointcut("setEmployeeSalary() || setEmployeeName() || setEmployeeAddress() ||
          cutEmployeeSalary()")
14   public void modifyEmployee(){}
```

Die Pointcuts in Zeile 19 bis 32 definieren Joinpoints von Methodenaufrufen, die Datenfelder vom Mitarbeiter manipulieren. Der Pointcut “modifyEmployee” fasst diese verschiedenen Pointcuts zu einem zusammen.

Listing 7.31: History.java - Advice-Definition

```
1    @Before("@target(org.softlang.utils.annotations.Historizing) && modifyEmployee
         () "
2        + "&& !execution(public * org.softlang.services.EmployeeService.
             createEmployee(..)) && args(emp,..)")
3    public void historizeEmployee(Employee emp) {
4      history.historize(emp);
5    }
```

Der definierte Advice erweitert die in den Pointcuts definierten Methodenaufrufe. Dabei wird sichergestellt, dass der Advice nur ausgeführt wird, wenn auf der Target-Klasse – hier der Instanz von EmployeeService – die zuvor beschriebene *@Historizing*-Annotation gesetzt wurde. Somit lässt sich die Historisierung auf bestimmte Spring-Komponenten beschränken. Bei der Ausführung des Advices wird für die Employee-Instanz ein History-Eintrag erstellt und mit Hilfe der HistoryMaintainer-Instanz verwaltet und archiviert.

Diese Implementation demonstriert den Einsatz von verschiedenen Pointcuts und deren Grupperiung. Weiterhin wird ein Before-Advice eingesetzt. Dieser wird mit einer Bedingung eingeschränkt. Die Spring-Komponente, deren Methodenaufrufe abgefangen werden, muss die *@Historizing*-Annotation tragen.

7.2.3 Logging

Das Logging-Feature wurde analog der Implementation in AspectJ umgesetzt. Entsprechend sind beide Hilfsklassen “LogEntry” sowie “CompanyLogger” in dieser Implementation vertreten. Mit Hilfe dieser beiden Klassen werden Log-Einträge erzeugt und innerhalb der CompanyLogger-Instanz verwaltet. Entsprechende Log-Einträge lassen sich,

analog zum History-Feature, aus dem entsprechenden Aspekt beziehen, welcher eine Spring-Komponente darstellt.

Listing 7.32: Logging.java - Implementation des Logging-Aspekts

```
1  package org.softlang.aspects;
2
3  import org.aspectj.lang.ProceedingJoinPoint;
4  import org.aspectj.lang.annotation.Around;
5  import org.aspectj.lang.annotation.Aspect;
6  import org.softlang.company.Employee;
7  import org.softlang.utils.CompanyLogger;
8  import org.springframework.stereotype.Component;
9
10 @Aspect
11 @Component
12 public class Logging {
13
14   private CompanyLogger logger = CompanyLogger.getInstance();
15
16   @Around("(execution(public * org.softlang.services.EmployeeService.cut(org.
         softlang.company.Employee)) "
17       + "|| execution(public * org.softlang.services.EmployeeService.
             setEmployeeSalary(..))) "
18       + "&& args(emp,..)")
19   public void beforeCallAnyMethod(ProceedingJoinPoint pjp, Employee emp) throws
         Throwable {
20     double oldSalary = emp.getSalary();
21     pjp.proceed();
22     logger.logCut(emp.getName(), oldSalary, emp.getSalary());
23
24   }
25
26   public CompanyLogger getLogger() {
27     return logger;
28   }
29
30 }
```

Der implementierte Aspekt besteht aus einem Around-Advice. Dieser erstellt während eines Aufrufs der cut- beziehungsweise der setEmployeeSalary-Methode einen entsprechenden Log-Eintrag im CompanyLogger. Dabei werden der Name, das alte sowie das neue Gehalt des Mitarbeiters gespeichert. Im weiteren Verlauf lassen sich die Differenzen der Gehaltsänderungen entsprechend analysieren.

Die demonstrierte Implementation des Logging-Features demonstriert einen Around-Advice, der verschiedene Pointcuts kombiniert. Dabei werden bei entsprechender Ausführung definierter Methoden Log-Einträge erstellt und archiviert, um im weiteren Verlauf erhoben und analysiert zu werden.

7.3 Aquarium

Dieses Kapitel demonstriert und diskutiert die angefertigte Implementation für das Aquarium-Framework. Dabei werden drei verschiedene Features, nämlich Ranking, History sowie Logging implementiert. Anhand dieser Features werden die grundlegenden Konzepte des Aquarium-Frameworks veranschaulicht.

Analog der beiden zuvor diskutierten Implementationen verwendet diese Implementation drei Basisklassen “Employee”, “Department” sowie “Company”. Diese werden jedoch, anders als in den AspectJ und Spring AOP Implementationen, bereits mit entsprechender Funktionalität bereichert – sprich alle drei Klassen implementieren die cut-Methode.

Eine Demonstration von Inter-type Deklarationen entfällt, da Ruby bereits das Konzept offener Klassen bereitstellt. Klassen können in Ruby beliebig ohne weitere Hilfsmittel erweitert werden. Ebenfalls wird auch hier das Persistence-Feature nicht umgesetzt. Ruby bietet keine Mechanismen für Annotationen. Ein sinnvoller Zugriff auf Attribute ist nur über syntaktischen Zucker realisiert [Git].

Ein besonderes Konzept, was in den zuvor demonstrierten Implementationen nicht vorhanden ist, stellt das Einbinden von Aspekten an konkrete Objektinstanzen dar. Während in AspectJ oder Spring AOP das Erweitern von Methoden nur innerhalb von Klassen möglich ist, stellt Aquarium einen Mechanismus zur Erweiterung konkreter Objekt-Instanzen bereit. Dieser Mechanismus wird in der Implementation des Ranking-Features demonstriert.

7.3.1 Ranking

Die Implementation des Ranking-Features mit Hilfe des Aquarium-Frameworks verläuft grundsätzlich analog zu den bereits gezeigten Implementationen. Zusätzlich wird hierbei das spezielle Sprachkonstrukt von Aquarium, nämlich das Einfügen von Aspekten an konkrete Objekt-Instanzen demonstriert. Dafür muss ein sinnvolles Ranking gestaltet werden, welches die Überprüfung eines kompletten Unternehmens vermeidet. Hierzu weicht die Implementation von den Vorgaben ab.

Listing 7.33: ranking_constraint.rb - Constraint zur Validierung eines Mitarbeiters

```
1 require 'ranking_error'
2
3 class RankingConstraint
4   def initialize(company)
5     @company = company
6   end
7
8   def align(employee)
```

```
9       check_employee(employee) if self.all_employees.include? employee
10    end
11
12    def check_employee(employee)
13      medium_salary = all_employees.keep_if{|emp| emp != employee}.compact.map{|
            emp| emp.salary}.reduce(:+) / all_employees.size
14
15      p "medium salary #{medium_salary}"
16      p "medium salary * 1.25 #{medium_salary*1.25}"
17
18      filtered_employees = all_employees.keep_if { |emp| emp != employee and 1.25
            * medium_salary >= employee.salary }.compact
19
20      raise RankingError.new,"Employee #{employee.name} earns too much: #{employee
            .salary}." if filtered_employees.empty?
21    end
22
23    def all_employees
24      all_depts = Array.new
25      all_depts << @company.departments
26      @company.departments.each do |dep|
27        all_depts << collect_departments(dep)
28      end
29      collect_employees(all_depts.flatten)
30    end
31
32    def collect_departments(department)
33      depts = Array.new
34      depts << department.sub_departments
35      department.sub_departments.each do |sub_dep|
36        depts << collect_departments(sub_dep)
37      end
38      depts
39    end
40
41    def collect_employees(departments)
42      emps = Array.new
43      departments.each do |dep|
44        emps << dep.manager
45        emps << dep.employees
46      end
47      emps.flatten
48    end
49
50  end
```

Die demonstrierte RankingConstraint-Klasse übernimmt konkret eine Aufgabe. Ein Mitarbeiter, welcher zum Ranking herangezogen wird, darf ein Gehalt maximal 25 Prozent über dem Unternehmensdurchschnitt beziehen. Dementsprechend werden in dieser Klasse alle Mitarbeiter eines Unternehmens zusammengefasst, und der Durchschnitt der Gehälter gebildet. Das Gehalt des Mitarbeiters, wird im Falle einer Mitarbeit bei einer Abteilung auf die Gehaltsspanne hin validiert. Bei entsprechendem Überschreiten des Limits wird ein RankingError mit einer Nachricht geworfen.

Darüber hinaus wird das Ranking-Modul definiert. Dieses Modul beherbergt entsprechende Aspekte, die der Implementation dienen.

Listing 7.34: ranking.rb - Ranking-Modul mit entsprechenden Aspekten

```
1  require 'aquarium'
2  require 'employee'
3  require 'ranking_error'
4
5  module Ranking
6    include Aquarium::Aspects
7    def self.enforce_ranking(to_rank, company)
8      Aspect.new :after, :object => to_rank, :calls_to => [:cut,:salary=] do |jp,
           emp, *args|
9        ranking_constraint = RankingConstraint.new(company)
10       ranking_constraint.align to_rank
11     end
12   end
13 end
```

Dieses Listing beinhaltet eine Methode "enforce_ranking" welche eine Employee- und eine Company-Instanz als Parameter annimmt. Diese Methode soll aufgerufen werden, wenn das Gehalt des entsprechenden Mitarbeiters anhand des Rankings eines Unternehmens bemessen werden soll. Ohne einen Aufruf dieser Methode findet auch keine Ranking–Überprüfung statt. Der demonstrierte Aspekt zeigt einen After-Advice, welcher auf der konkreten Objekt-Instanz aufgerufen wird. Dabei wird eine Instanz einer RankingConstraint erstellt und das Gehalt des Mitarbeitrs validiert.

Diese Implementation zeigt, wie sich mit Hilfe von Aquarium konkrete Objekt-Instanzen mit Hilfe von Aspekten erweitern lassen. Es lassen sich konkrete Methodenaufrufe definieren. Dabei muss sichergestellt werden, dass diese Methoden innerhalb der entsprechenden Klasse der Objekt-Instanz tatsächlich enthalten sind, da keine Validierung der Aspekte stattfindet. Dies ist Aufgabe des Entwicklers, jedoch bietet dieser Mechanismus einen hilfreichen und flexiblen Ansatz, um konkrete Instanzen unabhängig ihres tatsächlichen Objekttyps zu erweitern.

7.3.2 History

Die Implementation des History-Features ist ebenfalls ähnlich zu den beiden vorhergehenden Implementationen aufgebaut. Eine Klasse namens "HistoryFactory" definiert eine Hilfsklasse, welche History-Einträge erstellt und für jeden Mitarbeiter verwaltet. Weiterhin wird ein Modul "History" bereigestellt. Dieses enthält den Aspekt, welcher den History-Mechanismus ermöglicht.

Listing 7.35: history.rb - History-Aspekt Implementation

```
1  require 'aquarium'
2  require 'employee'
3  require 'history_factory'
```

```
4
5  module History
6
7    include Aquarium::Aspects
8
9    @history = HistoryFactory.new
10
11   def self.history
12     @history.history
13   end
14
15   # define a list with historizable classes
16   historizable = [Employee]
17
18   # before advice, any attribute access via reader, and cut method on employee
         and subtypes gets advised
19   Aspect.new :before, :attribute => /.*/, :attribute_options => [:writers], :
         calls_to => [:cut], :on_types_and_descendents => historizable do |
         join_point, emp, *args|
20     @history.historize emp
21   end
22
23 end
24
25 # reopen employee class to add new method
26 class Employee
27   def copy
28     copy = Employee.new(@name,@address,@salary)
29     copy
30   end
31 end
```

Zunächst wird eine Instanz der HistoryFactory-Klasse instantiiert und als Datenfeld gesetzt (Zeile 9). Mit Hilfe dieser Hilfsklasse können im weiteren Verlauf die Einträge entsprechend ausgelesen und verarbeitet werden. Die history-Methode (Zeile 11-13) gibt die aktuelle Instanz der HistoryFactory zurück. Im weiteren Verlauf wird eine Liste bereitgestellt, die Typen enthält, deren Instanzen durch den nachfolgenden Aspekt erweitert werden sollen (Zeile 16). Der genannte Aspekt aus Zeile 19 bis 21 definiert einen Before-Advice. Dieser erweitert die genannten Objekttypen um einen Historisierungsmechanismus, welcher mit Hilfe der zuvor erwähnten HistoryFactory einen Eintrag für den Mitarbeiter erstellt. Der Aspekt definiert mit Hilfe eines Regulären Ausdrucks, dass alle Zugriffe auf Attribute eines Objekttyps als Joinpoint definiert werden, wobei dieser jedoch auf schreibende Zugriffe der Attribute beschränkt wird. Darüber hinaus wird der Pointcut um einen Joinpoint, bei dem die Methode “cut” auf einem Objekt aufgerufen wird, erweitert. Weiterhin wird mit definiert, dass der Pointcut für alle Objekttypen inklusive dessen Erben gilt. Dabei wird als Parameter die zuvor definierte Liste mit Objekttypen übergeben.

Speziell der letzte Ausdruck ist besonders, da ohne die direkte Anweisung, alle Erben der Employee-Klasse in den Aspekt mit einzuschließen, nur die Klasse selbst erweitert

wird. Die Implementation zeigt dies mit einem Beispiel, wo die Klasse “Employee” durch die Klasse “Freelancer” erweitert wird. Diese soll einen Freiberufler darstellen. Wird die Anweisung des Einbezugs der direkten Erben entfernen, werden ausschließlich Objektinstanzen des Typs “Employee” historisiert.

Die Implementation des History-Features mit Ruby zeigt weitere Konzepte des Aquarium-Frameworks. Speziell wird hier der Ausdruck *on_types_and_descendents* und dessen Unterschiede zu *on_types* demonstriert.

7.3.3 Logging

Das Umsetzung des Logging-Features zeigt ein weiteres Konzept von Aquarium, nämlich das Definieren von Pointcuts, welche dann innerhalb eines Aspektes als Liste an einen Advice übergeben werden.

Die Hilfsklassen für diese Implementation sind den beiden bereits demonstrierten Implementationen von AspectJ und Spring AOP nachempfunden. Es existiert eine Klasse “LogEntry”, welche die Datenkapsel für die Log-Einträge darstellt. Des weiteren enthält das Projekt Modul “CompanyLogger”. Dieses Modul stellt die Funktionalität für das Erstellen und Verwalten von Log-Einträgen bereit.

Listing 7.36: logging.rb - Logging-Aspekt Implementation

```
1  require 'aquarium'
2  require 'company_logger'
3
4  module Logging
5    include Aquarium::Aspects
6
7    @logger = CompanyLogger.new
8
9    def self.logger
10     @logger
11   end
12
13   call_cut = JoinPoint.new :type => Employee, :method => :cut, :on_types => [
         Employee]
14   write_salary = JoinPoint.new :type => /^Employee/, :method => :salary=, :
         on_types => [Employee]
15
16   Aspect.new :around, :pointcuts => [call_cut, write_salary] do |join_point, emp
         , *args|
17     old_salary = emp.salary
18     join_point.proceed
19     p @logger.log(emp.name,old_salary,emp.salary)
20   end
21 end
```

Zunächst wird, ähnlich der Implementation des History-Features, eine Instanz des CompanyLoggers referenziert und mittels Methode verfügbar gemacht. Daraufhin werden zwei Pointcuts explizit definiert. Einerseits wird der Joinpoint der cut-Methode, andererseits der Aufruf der Methode, mit welcher das Datenfeld "salary" beschrieben wird, gesetzt (Zeile 13+14). Dabei wird die Klasse, auf der die Methoden aufgerufen werden, auf zwei verschiedene Arten definiert. Einerseits wird der Objekttyp "Employee" direkt angegeben. Andererseits wird im zweiten Pointcut hingegen ein regulärer Ausdruck als Parameter überreicht. Dies demonstriert die verschiedenen Möglichkeiten, mit Hilfe von Aquarium Objekttypen für Pointcuts festzulegen. Darüber hinaus wird in Zeile 16 bis 20 ein Aspekt mit einem Around-Advice erstellt. Dieser Advice nimmt die zuvor erstellten Pointcuts als Parameter und erweitert deren Joinpoints um die Funktionalität des Logging-Features.

Die Implementation des Logging-Features zeigt weitere Konzepte von Aquarium. Einerseits wird ein Around-Advice demonstriert. Des weiteren wird die Möglichkeit der Definition einzelner Pointcuts sowie der Suchmechanismus mittels Regulärer Ausdrücke innerhalb der Pointcuts aufgezeigt.

Kapitel 8

Zusammenfassung

In dieser Arbeit wurden drei verschiedene Implementationen mit Hilfe Aspekt-orientierter Technologien für die Softwareplattform 101companies ausgearbeitet und vorgestellt. Dabei wurden grundlegende Konzepte des jeweiligen Werkzeugs im Rahmen der Ausarbeitung veranschaulicht. Dennoch existieren diverse Sprachkonstrukte, die mit dieser Arbeit nicht abgedeckt wurden. Diese Sprachkonstrukte lassen sich mit Hilfe der 101companies Plattform nicht sinnvoll veranschaulichen. Beispielsweise ist das nachträgliche Hinzufügen von Compiler-Fehlern mit AspectJ nicht einsetzbar, womit eine Verbesserung durch AOP zustande kommt.

Die Arbeit demonstriert drei verschiedene AOP-Werkzeuge, welche allesamt aufgrund diverser Kriterien zu einer Implementation herangezogen wurden.

Die Implementation mit AspectJ hat die bereits existierende Ausarbeitung um einige Features erweitert. Dabei wurden weitere Konzepte und Sprachkonstrukte, die AspectJ unterstützt, demonstriert. Speziell wurden bereits gezeigte Konzepte, wie das Hinzufügen von Methoden zu einer existierenden Klasse, erneut aufgezeigt. Ein Beispiel hierfür ist die Umsetzung des Features “Median”.

Die Implementation mit Hilfe von Spring AOP zeigt alltagsähnliche Implementationen. Dabei wurde eine Spring-Komponente durch verschiedene Aspekte sinnvoll und flexibel erweitert. Durch den speziellen Proxy-Mechanismus ist Spring AOP in Blick auf die Sprachkonstrukte eingeschränkt. Viele der in AspectJ unterstützten Sprachkonstrukte werden von Spring AOP nicht unterstützt. Beispielsweise lassen sich nachträgliche Vererbungsstrukturen nur mit Interfaces aufbauen, wohingegen AspectJ auch konkrete Klassen vererben kann. Dennoch konnte die 101companies Plattform durch speziell an das Spring Framework angepasste Implementationen sinnvoll bereichert werden.

Die Implementation des Aquarium-Frameworks zeigt eine sinnvolle Erweitertung der sehr mächtigen Sprache Ruby durch AOP. Trotz der existierenden und bereits weitreichenden Metaprogrammierung bringt das Aquarium-Framework sinnvolle Sprachkonstrukte ein,

um ein Softwareprojekt in Ruby mit Hilfe von AOP zu verbessern. Speziell ist hier das Konstrukt zu nennen, mit dem Aspekte dynamisch an Objektinstanzen gebunden werden können. Die drei gezeigten Feature-Implementationen zeigen, wie ein Softwareprojekt, wie 101companies, durch AOP bereichert werden kann.

Alle drei demonstrierten AOP-Werkzeuge sind innerhalb eines Softwareprojektes sinnvoll einsetzbar. Allerdings unterscheiden diese sich speziell im Hinblick auf die Breite der Auswahl der unterstützten Sprachkonstrukte und Konzepte. Deswegen fällt die Menge der implementierten Features bei allen Implementationen unterschiedlich aus.

Abschließend lässt sich sagen, dass AOP eine große Bereicherung für ein Softwareprojekt darstellt. Mit AOP lassen sich CCCs sinnvoll kapseln, was eine Verminderung des projektspezifischen Wartungsaufwands sowie eine saubere Trennung innerhalb der Software nach sich zieht. Allerdings ist die Auswahl eines geeigneten Frameworks zu diskutieren. Wird das Spring Framework für ein Softwareprojekt genutzt, steht dem Entwickler die Auswahl zwischen Spring AOP und AspectJ bereit. Während Spring AOP für diverse Anforderungen sinnvoll umsetzbar ist, bleibt AspectJ die primäre Wahl, um weitereichende Umsetzungen mit Hilfe Aspekt-orientierter Programmierung vorzunehmen.

Literatur

[101a] 101companies. URL: http://101companies.org.

[101b] 101repo. URL: http://101companies.org/wiki/@repo.

[101c] 101system. URL: http://101companies.org/wiki/@system.

[101d] 101worker. URL: http://101companies.org/wiki/@worker.

[API] Oracle - Java Persistence API. URL: http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html.

[Asp] AspectJ. URL: http://www.eclipse.org/aspectj/.

[Boo86] Grady Booch. "Object-Oriented Development". In: *IEEE Transactions on Software Engineering*. IEEE, 1986.

[Dev] AOSD Tools for Developers. URL: http://www.aosd.net/wiki/index.php?title=Tools_for_Developers.

[Fea] 101companies Features. URL: http://101companies.org/wiki/Namespace:Feature.

[Git] Aquarium on GitHub. URL: https://github.com/deanwampler/Aquarium.

[Impa] 101companies AspectJ Implementation. URL: http://101companies.org/resources/contributions/aspectJ?format=html.

[Impb] 101companies YAPAF Implementation. URL: http://101companies.org/resources/contributions/YAPAF?format=html.

[Int] The Java EE 6 Tutorial – Overview of Interceptors. URL: http://docs.oracle.com/javaee/6/tutorial/doc/gkigq.html.

[Jul] TIOBE Index for July 2014. URL: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html.

[Kic+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm und William G. Griswold. “An Overview of AspectJ”. In: *ECOOP 2001 — Object-Oriented Programming*. Springer Verlag, 2001.

[Kic+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier und John Irwin. “Aspect-Oriented Programming”. In: *ECOOP’97 — Object-Oriented Programming*. Springer Verlag, 1997.

[Lan] The Ruby Language. URL: https://www.ruby-lang.org/en/.

[Mav] Apache Maven. URL: http://maven.apache.org/.

[Pat] Wikipedia: Proxy Pattern. URL: http://en.wikipedia.org/wiki/Proxy_pattern.

[PGA02] Andrei Popovici, Thomas Gross und Gustavo Alonso. “Dynamic Weaving for Aspect-Oriented Programming”. In: *AOSD ’02 Proceedings of the 1st international conference on Aspect-oriented software development* (2002).

[PZ03] Eduardo Kessler Piveta und Luiz Carlos Zancanella. “Aspect Weaving Strategies”. In: *Journal of Universal Computer Science* vol. 9 no. 8 (2003). URL: http://www.jucs.org/jucs_9_8/aspect_weaving_strategies/Piveta_E_K.pdf [Letzter Zugriff: 29.06.2013].

[Ref] Java Reflection. URL: http://docs.oracle.com/javase/7/docs/api/java/lang/reflect/package-summary.html.

[Req] 101companies Data Requirement. URL: http://101companies.org/wiki/Data_requirement.

[Sch] Spring AOP XML Schema. URL: http://docs.spring.io/spring/docs/3.2.3.RELEASE/spring-framework-reference/html/aop.html#aop-schema.

[Sch12] Markus Schulte. “Aspekt-Orientierung in PHP”. Diplomarbeit. Universität Koblenz-Landau, 2012.

[Spr] Aspect Oriented Programming with Spring. URL: http://docs.spring.io/spring/docs/3.2.3.RELEASE/spring-framework-reference/html/aop.html.

[Sup] Spring AOP @AspectJ Support. URL: http://docs.spring.io/spring/docs/3.2.3.RELEASE/spring-framework-reference/html/aop.html#aop-ataspectj.

[The] The Curious Coders Java Web Frameworks Comparison: Spring MVC, Grails, Vaadin, GWT, Wicket, Play, Struts and JSF. URL: http://zeroturnaround.com/rebellabs/the-curious-coders-java-web-frameworks-comparison-spring-mvc-grails-vaadin-gwt-wicket-play-struts-and-jsf/.

[Wam07] Dean Wampler. "Aspect-Oriented Design in AspectJ and Ruby". In: *International Conference on Software Engineering (ICSE)* (2007).

[Wam08] Dean Wampler. "Aquarium AOP for Ruby". In: *AOSD 2008* (2008). URL: http://aspectprogramming.com/papers/Aquarium_AOP_for_Ruby_presentation.pdf [Letzter Zugriff: 03.08.2013].