Hochschule Kempten

Hochschule für angewandte Wissenschaften

Diplomarbeit

Studiengang Informatik

(Wirtschaftsinformatik)

Kacper Bak

Scala: Eine objektfunktionale Sprache für die JVM – Java Virtual Machine

Aufgabensteller	Prof. Dr. rer. nat. Ulrich Göhner
Arbeit vorgelegt am	30.04.2012
durchgeführt bei	PENTASYS AG, München
Betreuer	Günther Reisner
Anschrift des Verfassers	Höfatsstraße 46 86163 Augsburg

Zusammenfassung

Die Erstellung der Arbeit erfolgte mit Unterstützung der PENTASYS AG, deren primäres Interesse es war, Scala in der Domäne der Java-Webtechnologien zu testen. Dazu wird im ersten Teil ein kurzer Überblick aufkommender JVM-Sprachen abgebildet und begründet, warum sich diese Arbeit mit Scala befasst.

Das zweite Kapitel umfasst die Theorie um Scala. Darin werden allgemeine Merkmale veranschaulicht und ein kurzer Ausblick auf die Mächtigkeit der Sprache geboten. Abschließend werden die Details in objektorientierte und funktionale aufgeteilt und anhand von Beispielen erklärt. Dabei erfolgt eine konstante Retrospektive zum bisher vermittelten Inhalt. Auch wenn nicht alle Sprachdetails der objektfunktionalen Programmiersprache Scala darin enthalten sind, bildet es die Grundlage für weiterführende Betrachtungen im Praxisteil.

Der Praxisteil beschreibt die Problemstellung unterschiedlich ausgeprägter Entwicklungsumgebungen und Werkzeuge, die für ein mehrschichtiges, Projekt wie es sich die PENTASYS AG vorstellt, notwendig sind. Für diesen Zweck wird der Analytische Hierarchieprozess, beschrieben, auf die Problemstellung angewendet und als Beispielanwendung implementiert. Die dabei entstandenen Vergleichsdaten werden abschließend in die kommerzielle Software Expert Choice eingegeben und mit dem Ergebnis der Beispielanwendung verglichen. Weiter folgt die Beschreibung der Technologien welche dafür in Verbindung mit Scala gebracht wurden. Das Resultat ist eine klassische 3-Schicht-Architektur mit dem Wicket-Framework (Java) als Benutzeroberfläche, Scala als Funktions- und Steuerungsschicht und der Anwendung von JDBC unter Java zur Persistierung der Daten. Diese Konstellation überprüft gleichzeitig, wie kompatibel Java und Scala miteinander auf der Java Virtual Machine zusammen harmonieren und an welchen Stellen Probleme auftreten. Neben den genannten Entwicklungsumgebungen werden zwei Werkzeuge zum Testen und Erstellen komplexer Java/Scala-Projekte evaluiert und eine verwendbare Kombination vorgestellt. Weitere Ergebnisse sind ein Performanzvergleich zwischen der Java/Scala Parallelisierung. Dieser ist fachlich nicht unbedingt notwendig, zeigt aber Unterschiede in Leistung und Implementierung auf. Daneben werden zwei weitere

spezifische Anwendungsfälle erörtert, bei denen Scala, Java überlegen ist. Außerdem wurde die Funktions- und Steuerungs-Schicht zusätzlich in Java implementiert, um einen direkten Vergleich zwischen beiden Sprachen vollziehen zu können. Untersucht werden dafür die Komplexität zwischen den Objekten und der tatsächliche Implementierungsaufwand für den Entwickler. Mit diesen unternommenen Anstrengungen kommt der Autor zu folgendem Ergebnis:

Die Sprache selbst kann als sehr umfangreich verstanden werden. Bis der objektorientierte Java Entwickler seine gewohnte Denkweise in Scala anwenden kann, muss er neben der Zeit für die neue Syntax auch ein, zwei neue Konzepte lernen. Eine Affinität gegenüber der funktionalen Programmierung benötigt er dafür nicht, ablehnend sollte er dem neuen Paradigma aber nicht gegenüberstehen. Insofern ist ein ehrliches Maß an Motivation zwingend für die effektive Anwendung von Scala. Die zweite Hürde sind die noch neuen und unausgereiften Werkzeuge des Scala-Ökosystems, weshalb zu den Java-Pendants gegriffen wurde. Ein flüssiger Entwicklungslebenszyklus, mit Editieren, Erstellen bis hin zum Testen ist mit der hier vorgestellten Kombination möglich. Dass Scala in naher Zukunft Java ablösen wird, ist aus Sicht des Autors unwahrscheinlich. Dafür erweitert Scala aber die Java Virtual Machine sehr gut.

Vorwort

Als Informatik-Student der Hochschule Kempten kam der Autor dieser Arbeit mit zahlreichen imperativen Programmiersprachen in Kontakt. Dazu zählten C/C++, Java aber auch deklarative wie SQL und zum Teil auch JavaFX. Im Praxissemester und in seiner Zeit als Werksstudent, kam er dann mit dem .NET Framework in Berührung dessen Sprache C#, primär objektorientiert ausgerichtet ist, aber zudem funktionale Ansätze aufweist. Als die PENTASYS AG das Thema „Scala: Eine objektfunktionale Sprache für die JVM – Java Virtual Machine“ ausschrieb, war das Interesse des Autor geweckt, die funktionale Programmierung kennen zu lernen und Synergieeffekte zu bekannten Konzepten herzustellen.

Inhaltsverzeichnis

Vorwort 1

Inhaltsverzeichnis 2

Abkürzungsverzeichnis 9

1 Einleitung 10

1.1 Problemstellung 10

1.2 Ziele dieser Arbeit 11

1.3 Methodik 12

1.4 Das Unternehmen: PENTASYS AG 12

1.5 Anmerkungen 13

2 Scala eine objektfunktionale Programmiersprache 14

2.1 Scala Hauptmerkmale 14

2.1.1 Herkunft und Intention 14

2.1.2 Plattformen: JVM & .NET 15

2.1.3 Trennung zwischen veränderbaren & unveränderbaren Daten 16

2.1.4 Statische Typisierung 16

2.1.5 Objektfunktional 17

2.1.6 Skalierbarkeit 19

2.1.6.1 Erweiterbarkeit anhand von Datentypen 19

2.1.6.2 Erweiterbarkeit durch Kontrollstrukturen 21

2.2 Scala Sprachfeatures 23

2.2.1 Objektorientierte Programmierung 24

2.2.1.1 Klassen und Felder 24

2.2.1.2 Standardwerte 27

2.2.1.3 Sichtbarkeit 28

2.2.1.4 Methoden 30

2.2.1.5 Zusätzliche Konstruktoren und Standardwerte 33

2.2.1.6 Singleton-Objekt 34

2.2.1.7 Begleit-Objekt 36

2.2.1.8 Vererbung 37

2.2.1.9 Member überschreiben 38

2.2.1.10 Abstrakte Klassen 39

2.2.1.11 Konstruktorverkettung 40

2.2.1.12 Polymorphie 41

2.2.1.13 Scala Typhierarchie 43

2.2.1.14 Traits 45

2.2.1.15 Mehrfachvererbung mit Traits 49

2.2.2 Funktionale Programmierung 53

2.2.2.1 Definition 53

2.2.2.2 Funktionen 56

2.2.2.2.1 Lokale Funktionen 58

2.2.2.2.2 Closures 60

2.2.2.2.3 Einsatz von Funktionen und Closures 61

2.2.2.2.4 Currying und Kontrollstrukturen 65

2.2.2.3 Implicits 67

2.2.2.4 Mustererkennung 70

3 Scala in der Praxis 74

3.1 Beispielanwendung 74

3.1.1 Problemstellung 74

3.1.2 Vorgaben 75

3.1.3 Ziele 75

3.1.3.1 Umsetzung einer modularen Webanwendung 75

3.1.3.2 Scala spezifische Anwendungsfälle 76

3.1.3.3 Java – Scala Vergleich 76

3.1.3.3.1 Quelltextzeilen 77

3.1.3.3.2 Kopplung zwischen Objekten 77

3.1.3.3.3 Performanz 77

3.1.4 Lösungsansatz: Analytic Hierarchy Process - AHP 78

3.1.4.1 Beschreibung des AHP 78

3.1.4.2 Anwendung des AHP auf die Problemstellung 78

3.1.4.2.1 Überblick des AHP Ablaufs 79

3.1.4.2.2 Aufstellung der Elementhierarchie 81

3.1.4.2.3 Die AHP-Skala 82

3.1.4.2.4 Bewertung der Elementhierarchie 83

3.1.4.2.4.1 Prioritätenschätzung 83

3.1.4.2.4.2 Ablauf der Prioritätenschätzung 85

3.1.4.2.5 Berechnung der Gewichte 86

3.1.4.2.6 Beschreibung und Gewichtung der Attribute / Kriterien 88

3.1.4.2.6.1 Hauptkriterien 88

3.1.4.2.6.2 K1 IDE technische Merkmale 90

3.1.4.2.6.3 K2 Maven Unterstützung 91

3.1.4.2.6.4 K3 Quelltextbearbeitung 92

3.1.4.2.6.5 K4 Navigation 93

3.1.4.2.6.6 K5 Refactoring 94

3.1.4.2.6.7 K6 Fehlerdiagnose und Analyse 96

3.1.4.2.7 Beschreibung und Gewichtung der Alternativen 97

3.1.4.2.7.1 K1 IDE technische Merkmale 98

3.1.4.2.7.2 K2 Maven Unterstützung 100

3.1.4.2.7.3 K3 Quelltextbearbeitung 101

3.1.4.2.7.4 K4 Navigation 103

3.1.4.2.7.5 K5 Refactoring 105

3.1.4.2.7.6 K6 Fehlerdiagnose und Analyse 107

3.1.4.2.8 Konsistenzprüfung 108

3.1.4.2.9 Berechnung der globalen Attributgewichte 111

3.1.4.2.10 Erstellung der Alternativrangfolge.....112

3.1.5 Umsetzung des AHP.....114

3.1.5.1 Anwendungsfälle 114

3.1.5.2 Anwendungsschichten 117

3.1.5.2.1.1 Wicket in der Benutzeroberfläche 117

3.1.5.2.1.2 Funktions- und Steuerungs-Schicht 121

3.1.5.2.1.3 Datenbank-Schicht 124

3.1.5.3 Projektkonfiguration mit Maven 127

3.1.5.3.1 Die Projektkonfigurationsdatei: pom.xml.....127

3.1.5.3.2 Standardisierte Verzeichnisstruktur 129

3.1.5.3.3 Das Multimodulprojekt.....130

3.1.5.3.4 Fazit: Maven.....133

3.1.6 Ergebnisse.....133

3.1.6.1 Umsetzung einer modularen Java/Scala-Webanwendung 133

3.1.6.1.1 Probleme mit der Serialisierbarkeit.....133

3.1.6.1.2 Modulkapselung.....136

3.1.6.1.3 Fazit der Integration.....139

3.1.6.2 Scala spezifische Anwendungsfälle 140

3.1.6.2.1.1 Parallelisierung mit dem Fork/Join-Framework 140

3.1.6.2.1.2 Erweiterung bestehender Bibliotheken.....146

3.1.6.2.1.3 Erstellung eines Parsers 147
3.1.6.3 Java – Scala Vergleich 151
3.1.6.3.1 Quelltextzeilen 151
3.1.6.3.2 Kopplung zwischen Objekten 153
3.1.6.3.3 Performanz 155
3.1.6.3.3.1 Parallelisierung 155
3.1.6.3.3.2 Übersetzung 157
3.1.6.4 Beste IDE für modulare Java/Scala Projekte 158
3.2 Toolsupport 160
3.2.1 Einsatz von Maven unter Scala 160
3.2.2 Scala - Entwicklung in der Kommandozeile 163
3.2.3 Scala - Entwicklung mit IDE - Unterstützung 165
3.2.3.1 Installation der IntelliJ IDE 165
3.2.3.2 Installation der Eclipse IDE 165
3.2.3.2.1 Installation der Eclipse IDE mit Maven Unterstützung 167
3.2.3.2.2 Installation der Eclipse IDE mit Maven aus der Kommandozeile 168
3.2.3.3 Installation der Netbeans IDE 170
3.2.4 Testen mit JUnit 171
4 Fazit 174
5 Anhang 178
5.1 Klassendiagramm der Scala-Typhierarchie 178

5.2 Objektmodel der Beispielanwendung 179

5.3 Datenbankschema der Beispielanwendung 180

5.4 Inhalte des beigefügten Datenträgers 180

6 Abbildungsverzeichnis 181

7 Formelverzeichnis 184

8 Quellenverzeichnis 184

9 Quelltextverzeichnis 192

10 Tabellenverzeichnis 200

Abkürzungsverzeichnis

Akronym:	**Bedeutung:**
AHP	engl.: Analytic Hierarchy Process
BSD	Berkeley Software Distribution
CBO	Coupling Between Objects
CSS	Cascading Style Sheets
DRY	engl.: Don't repeat yourself
EC	Expert Choice
EPFL	École Polytechnique Fédérale de Lausanne
FP	Funktionale Programmierung
IDE	engl.: Integrated Development Environment
JVM	Java Virtual Machine
KISS	Keep it small and simple
LOC	Lines Of Code
OOP	Objektorientierte Programmierung
RAM	Random-Access Memory
REPL	Read-Evolve-Print-Loop
SBT	Simple-Built-Tool

1 Einleitung

1.1 Problemstellung

Seit dem Release 1995 ist die Java Technologie eine Erfolgsgeschichte. 2010 gab Sun Microsystems bekannt, dass die Verbreitung neun Millionen Entwickler umfasst [Redaktion JAXenter2011a]. Optimistische Hochrechnungen erheben Java sogar zur Programmiersprache Nr.1 [TIOBE2011]. Realistischere Zahlen stammen aus dem Unternehmensbereich. Im Jahre 2010 stand Java auf gleicher Augenhöhe mit der Konkurrenzplattform .NET aus dem Hause Microsoft [Schwichtenberg2010]. Fakt ist, dass die Innovationsgeschwindigkeit der .NET Plattform größer ist, als dessen Pendants. Während Version 1.0 des .NET-Frameworks 2002 veröffentlicht wurde, fanden ab der Version 3.5 funktionale Aspekte, wie Lambda-Ausdrücke, darin Einzug [Hilyard2007]. Gegenwärtig kann Oracle, der neue Eigentümer der Java-Plattform, diesem Trend nur schwer folgen. Das vor kurzem veröffentlichte Java 7 wird von der Fachpresse mehr als „Sammelsurium kleinerer Erweiterungen“ gewertet [Eisele2011a]. Die „Revolution“ soll Mitte 2013 mit Java 8 folgen [Neumann2011].

Die Innovationsgeschwindigkeit der Sprache Java ging lange Zeit gegen Null. Bestätigt wird diese Aussage von den vielen Sprachen, welche die Java Virtual Machine (JVM) als Basis verwenden und bis zum heutigen Tag entwickelt wurden. Neben weniger verbreiteten Vertretern, wie Gosu, Kotlin und Ceylon mit fehlender Community, Dokumentation und Umsetzung, gibt es auch vielversprechende Kandidaten unter den neuen Sprachen [Redaktion JAXenter2011b]. Dazu zählen Groovy, Clojure und Scala.

Diese Sprachen setzen alle auf die JVM und versuchen mit funktionalen Ansätzen spezifische Probleme, wie Nebenläufigkeiten, einfacher zu lösen und nebenbei die Sprache selbst prägnanter und erweiterbarer als Java zu machen.

Abbildung 1 veranschaulicht das gegenwärtige Interesse im Web für die genannten Sprachen. Die Programmiersprache Ceylon wurde nicht berücksichtigt, da zur Erstellung dieser Arbeit weder Compiler noch Projekt-Homepage vorlagen.

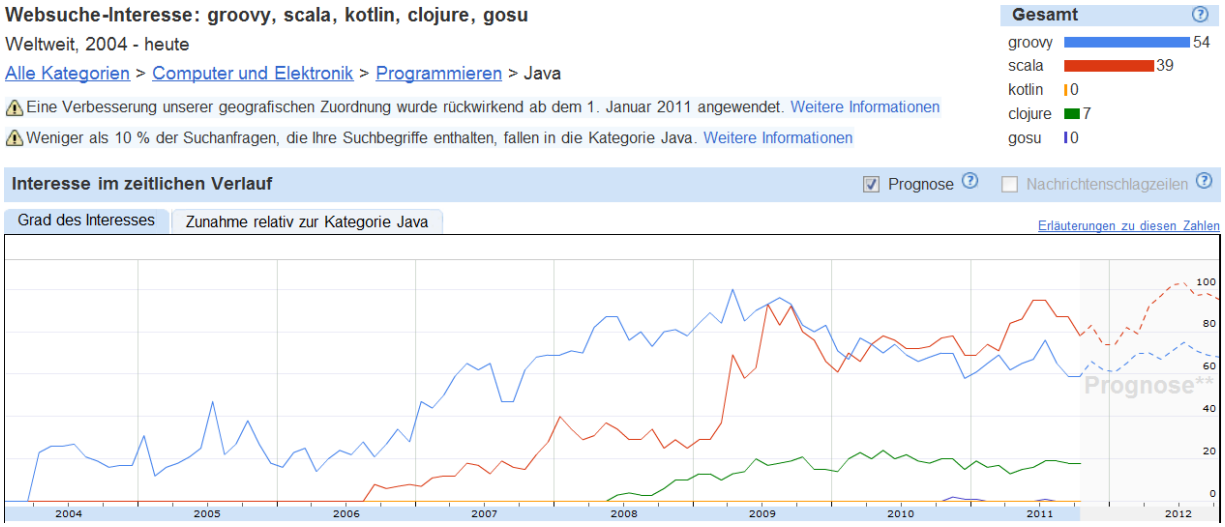


Abbildung 1 Programmiersprachen Trends [Google Inc.2011]

Die Grafik veranschaulicht einen starken Trend hin zu Scala. Dieser ist jedoch nicht das ausschlaggebende Kriterium für die Entscheidung, Scala als Basis dieser Arbeit zu verwenden.

Je nachdem, mit welcher Programmiersprache sich der heutige Softwareentwickler befasst, bekommt er „Die“ Sprache und „Das“ Paradigma als Lösung all seiner Probleme präsentiert. So verfolgt Groovy primär einen objektorientierten Ansatz und ergänzt diesen mit Lambda-Ausdrücken in Form von Closures [Codehaus 2011]. Die Sprache Clojure hingegen ist ein weiterer Lisp-Dialekt in dem das funktionale Paradigma dominiert. Das Verändern von Zuständen gilt hier wiederum als Ausnahme und wird über das Konzept, Software-Transactional-Memory realisiert [Hickey2011].

Scala hingegen vereint objektorientierte und funktionale Programmierung und bietet dem Entwickler/Architekten die Freiheit zum aktuellen Problem das beste Konzept anzuwenden.

1.2 Ziele dieser Arbeit

Primäres Ziel dieser Arbeit ist die praktische Evaluation der Programmiersprache Scala und die Vor- und Nachteile gegenüber Java darzustellen. Scala verspricht komplexe Problemstellungen mit weniger Quellcode zu lösen. Die dafür notwendigen theoretischen Grundlagen werden dafür erörtert und im Praxisteil umgesetzt. Dabei richtet sich diese

Arbeit primär an den routinierten Java-Entwickler, der mit den Konzepten der objektorientierten Programmierung (kurz OOP) vertraut ist und dem Software-Entwurfsmuster nicht fremd sind. Weder die PENTASYS AG noch der Autor dieser Arbeit sind der Ansicht, dass Scala die Java-Programmiersprache im Enterprise-Sektor von heute auf morgen ersetzen wird. Dafür ist Java zu stark verbreitet. Jedoch soll untersucht werden, wie gut sich Scala in bestehende Java Projekte integrieren lässt, welche Auswirkungen der Einsatz von zwei unterschiedlichen Sprachen auf ein Projekt haben und in welchen Anwendungsfällen sich die Umstellung auf Scala lohnt.

1.3 Methodik

Im ersten Teil werden die theoretischen Grundlagen für die Erörterung der zu erstellenden Beispielanwendung und Scala spezifischer Anwendungsfälle geschaffen. Da es sich bei Scala um einen Sprachhybrid aus objektorientierter und funktionaler Programmierung handelt, werden beide Paradigmen, so weit wie möglich, getrennt voneinander betrachtet. Dabei findet an passenden Stellen ein Vergleich zur objektorientierten Sprache Java statt, indem Vorteile anhand von Codebeispielen veranschaulicht werden. Die Synthese der Arbeit erfolgt im Kapitel 3.1.6.

Die Beispielanwendung ist eine modulare Java/Scala Webanwendung. Ziel der PENTASYS AG war es, Scala im Kontext der Internettechnologien zu testen. Zu diesem Zwecke konnte sich der Autor für eine beliebige Persistenzlösung und ein beliebiges Java-Webframework entscheiden. Die Entwicklung der Anwendung sollte mit Hilfe einer gängigen Java-Entwicklungsumgebung stattfinden. Zur Verwaltung der Abhängigkeiten und Modulbildung sollte ein Erstellungswerkzeug verwendet werden. Der Einsatz eines Testframeworks deckt den Entwicklungslebenszyklus grob ab.

1.4 Das Unternehmen: PENTASYS AG

Gegründet 1995 versteht sich die PENTASYS AG als Systemhaus für Projektmanagement, agile Entwicklung und maßgeschneiderte Softwarelösungen unter anderem in den Bereichen Telekommunikation, Finanzen und Logistik. Durch eine auf Nachhaltigkeit ausgelegte

Personalentwicklung motiviert die PENTASYS AG ihre mehr als 200 Mitarbeiter, sich kontinuierlich weiterzubilden und damit Fachexpertise aufzubauen. Diese Methode überzeugte auch die Jury des Best-Quality-Institute, welche die PENTASYS AG unter die Top 3 Unternehmen des „European IT-Workplace of the Year 2012“ wählte. Mit einer Steigerung des Jahresumsatz von 22.0 Mio. € (2010) auf 32.0 Mio. € (2011) wächst die PENTASYS AG überdurchschnittlich und wurde für diese Dynamik mit der „Bayerns Best 50“ Urkunde vom bayrischen Wirtschaftsminister ausgezeichnet [PENTASYS AG2012].

1.5 Anmerkungen

Zum Zweck der besseren Lesbarkeit dieser Arbeit, findet keine explizite Unterscheidung zwischen der männlichen/weiblichen Form statt (Bsp.: Benutzer/ -innen). Alle verwendeten Bezeichnungen gelten an entsprechender Stelle für weibliche Personen.

Diagramme die sich auf den Programmquelltext beziehen, sind in englischer Sprache verfasst, da der Quelltext selbst in Englisch erstellt wurde.
Alle Eigenamen und Bezeichnungen aus dem Kontext nicht offensichtlicher Entitäten, wie beispielsweise Schlüsselwörter aus Programmiersprachen, Klassenamen, Bezeichner usw. aus Quelltextbeispielen oder formal Verweise, sind in Anführungszeichen „“ gefasst.

Weiterhin werden Quelltextbeispiele auch Listings bzw. Code-Listings genannt. Diese beginnen immer mit der Zeilennummer, welche nicht Gegenstand des nachkommenden Inhaltes ist. Konsolenausgaben werden dabei in *kursiver Schrift* abgebildet.

2 Scala eine objektfunktionale Programmiersprache

2.1 Scala Hauptmerkmale

2.1.1 Herkunft und Intention

Martin Odersky, Schöpfer der Programmiersprache Scala, weist eine lange Historie mit der JVM auf. Diese entwickelte er zusammen mit Philip Wadler und war Hauptverantwortlicher für die Entwicklung des Java-Compilers. Die Forschung an der JVM ging weiter, so wurden Konzepte der generischen Programmierung ein Teil von Java SE 1.5 [Pollak2009]. Seit 2001 entwickelte Martin Odersky und seine Teammitglieder nun die Sprache Scala am École Polytechnique Fédérale de Lausanne (kurz EPFL) in der Schweiz. Ziele der Forschung sind es, eine für Entwickler optimierte Programmiersprache zu entwerfen und dabei ein Werkzeug für kommerzielle Softwareentwicklung herzustellen. Scala stellt deshalb Mechaniken zur höheren Abstraktion von Typen, als auch Werten bereit und fördert dabei die Komposition/ Dekomposition von Softwarebausteinen. Die Umsetzung dieser Skalierbarkeit erfolgt durch die Verallgemeinerung und Vereinigung des objektorientierten und funktionalen Programmierparadigma in der Sprache Scala. Das Resultat dieser Forschung ist Open Source Software, die unter der BSD-artigen Lizenz verfügbar ist und die Erstellung proprietärer Programme ermöglicht [EPFL2012b].

Scala selbst versteht sich, identisch zu Java, als Sprache für alle Anwendungsfälle (engl.: General Purpose Language), welche in Java-Bytecode übersetzt wird und auf der Java Virtual Machine (kurz JVM) lauffähig ist. Sie wird in den Punkten Stabilität, Testbarkeit und Performanz mit Java gleichgestellt und eignet sich sowohl zum Erstellen von Anwendungen, die auf einem Webservern ausgeführt werden, als auch zur Abbildung und Bearbeitung von Fachlogik [Evans2011].

LAYER	DESCRIPTION	EXAMPLES
DOMAIN-SPECIFIC	DOMAIN-SPECIFIC LANGUAGE (DSL); TIGHTLY COUPLED TO A SPECIFIC PART OF THE APPLICATION DOMAIN	APACHE CAMEL DSL, DROOLS, WEB TEMPLATING
DYNAMIC	RAPID, PRODUCTIVE, FLEXIBLE DEVELOPMENT OF FUNCTIONALITY	GROOVY, JYTHON, CLOJURE
STABLE	CORE FUNCTIONALITY, STABLE, WELL-TESTED, PERFORMANT	JAVA, SCALA

Abbildung 2 Einsatzzwecke der unterschiedlichen JVM-Sprachen

LAYER	EXAMPLE PROBLEM DOMAINS
DOMAIN-SPECIFIC	BUILD, CONTINUOUS INTEGRATION, AND CONTINUOUS DEPLOYMENT DEV-OPS ENTERPRISE INTEGRATION PATTERN (EIP) MODELING BUSINESS RULES MODELING
DYNAMIC	RAPID WEB DEVELOPMENT PROTOTYPING INTERACTIVE ADMINISTRATIVE/USER CONSOLES SCRIPTING TEST-DRIVEN DEVELOPMENT (TDD) AND BEHAVIOR-DRIVEN DEVELOPMENT (BDD)
STABLE	CONCURRENT CODE APPLICATION CONTAINERS CORE BUSINESS FUNCTIONALITY

Abbildung 3 Mögliche Aufgaben entsprechend der Einsatz-Schicht

2.1.2 Plattformen: JVM & .NET

Scala bietet die Möglichkeit sowohl in JVM-Bytecode zu übersetzen, als auch die direkte Interpretation innerhalb einer Read-Evolve-Print-Loop (kurz REPL), wie es aus Skriptsprachen bekannt ist. Auf die Entwicklung in der Scala REPL wird im Kapitel 3.2.2 detaillierter eingegangen. Der Scala Quelltext wird mittels des Scala-Übersetzers (engl.: Compiler) in Java-Bytecode transformiert. Das Ergebnis sind analog zu Java *.class Dateien, welche JAR-Artefakte bilden, die wiederum unter der JVM-Laufzeitumgebung lauffähig sind [Pollak2009]. Scala verspricht 100% Kompatibilität zur Java Plattform und die Wiederverwendbarkeit jeglicher in Java geschriebenen Bibliotheken und Frameworks. Somit wären alle bisherigen Java-Investitionen eines Unternehmens nicht verloren und könnten als gesichert gelten. Zu einem bestimmten Teil kann auch von Java auf Scala zugegriffen werden. Die Ermittlung der Grenzen dieser Interoperabilität ist ein Gegenstand dieser Arbeit, dessen Ergebnis unter

Kapitel 3.1.6.1 eingesehen werden kann. Eine Übersetzung in die Common Language Runtime (kurz CLR), welches das Bytecode-Pendant im Microsoft .NET-Framework darstellt, ist auch möglich. Zum Zeitpunkt der Erstellung dieser Arbeit, lag aber keine 100% Kompatibilität vor [EPFL2012a].

2.1.3 Trennung zwischen veränderbaren & unveränderbaren Daten

Grundsätzlich unterscheidet Scala bereits bei einfachen Daten, wie Variablen zwischen veränderbaren und unveränderbaren. Empfehlenswert ist ein Programmierstil, bei dem auf veränderbare Daten verzichtet wird. Damit entfällt nämlich ein Stück Komplexität, die man zur Laufzeit nicht beachten muss. Unveränderbare Variablen werden in Scala mit dem Schlüsselwort „val“ eingeleitet [Seeberger2011].

```
1 val myval = "Hello Scala World"
2 myval: java.lang.String = Hello Scala World
```

Listing 1 Erstellung einer unveränderbaren Variable

An dieser Stelle zeigt sich gleichzeitig, dass der Datentyp String aus Java verwendet wird. Versuchen wir nun Änderungen an dieser Variablen vorzunehmen, unterbindet dies der Compiler.

```
1 scala> myval = "foo"
2 <console>:8: error: reassignment to val
3        myval = "foo"
4                ^
```

Listing 2 Zuweisung eines neuen Wertes an eine unveränderbare Variable

Analog dazu verhält sich die Erstellung von veränderbaren Variablen mit dem Unterschied, dass dafür das Schlüsselwort „var“ verwendet wird.

2.1.4 Statische Typisierung

Statische Typisierung stellt sicher, dass Daten sich gemäß ihrer Definition - Typs - der entsprechenden Programmiersprache verhalten. So unterscheidet sich der Informationsgehalt eines binären Datentyps, wie „Boolean“, erheblich von dem eines Gleitkommazahl Datentyps, wie „Double“. Damit auf Daten operiert werden kann, müssen deren Eigenschaften zueinander passen. Dies stellt Scala mit der Festlegung auf statische Datentypen sicher. Obwohl der Scala-Übersetzer Typinferenz beherrscht, ist eine explizite

Angabe des verwendeten Datentyps möglich und manchmal auch erforderlich. Dies geschieht durch die Angabe des Datentyps nach dem Bezeichner, getrennt von einem Doppelpunkt wie aus Listing 3 ersichtlich ist [Seeberger2011].

```
1 var value:Boolean = true
2 value: Boolean = true
```

Listing 3 Erstellung einer booleschen Variable

Wird nun versucht der booleschen Variable „value“ ein Double-Wert zuzuweisen, quittiert dies der Scala-Übersetzer mit einem Typfehler.

```
1 value = 1.0
2 <console>:8: error: type mismatch;
3  found   : Double(1.0)
4  required: Boolean
5        value = 1.0
6                ^
```

Listing 4 Zuweisung einer Fließkommazahl an eine boolesche Variable

Das Gegenteil wäre eine dynamische Typisierung, welche für den Entwickler in erster Linie mehr Freiheit bei der Quelltexterstellung bedeuten würde. Dabei findet aber eine Verschiebung des Problems statt, da spätestens zur Laufzeit die Datentypen zueinander passen müssen. Das Resultat sind Testfälle, die wesentlich aufwendiger sind, als bei Sprachen mit statischer Typisierung, da neben der Fachlogik zusätzlich die Datentypen getestet werden müssen. Mit zunehmender Komplexität der Anwendung steigt damit der Aufwand der Testfälle an [Pollak2009].

2.1.5 Objektfunktional

Scala unterstützt das objektorientierte Paradigma, indem der Java-Entwickler gewohnt imperativ arbeiten kann und ist zugleich strikter. Ein Beispiel dafür zeigte bereits Listing 3, indem die Variable „value“ einem „scala.Boolean“ Wert zugeordnet wurde. Im Gegensatz zu Java handelt es sich um ein echtes Objekt und nicht um ein Primitive (int , float,...), wie in Java [Seeberger2011]. Weiter sind in Scala alle Funktionen auch Werte und diese wiederum Objekte. Diese können als Parameter übergeben und das Ergebnis eines Rückgabewertes sein. Damit besteht die Möglichkeit Funktionen höherer Ordnung abzubilden und funktional zu programmieren [Odersky2010].

Dies ermöglicht dem Entwickler entsprechend der Art der Problemstellung, das für ihn pragmatischere Paradigma auszuwählen, um zu einer Lösung zu gelangen. Ein sanfter Einstieg in Scala ist für den routinierten objektorientierten Entwickler möglich. Als Beispiel erstellen wir die Klasse „Person“ und eine zugehörige Instanz „p“ mit dem Personennamen „Kacper“.

```
1 class Person (val name: String,val age: Int){
2         override def toString = name + ":" + age
3 }
4 defined class Person
5
6 val p = new Person("Kacper",29)
7 p: Person = Kacper:29
```

Listing 5 Erstellung der Klasse „Person“ und einer Instanz „p“

Wir erstellen eine zweite Instanz mit dem Personennamen „Bak“ und gruppieren beide in einer Liste „persons“.

```
1 val persons = List( p , new Person("Bak",29) )
2 persons: List[Person] = List(Kacper:29, Bak:29)
```

Listing 6 Erstellung einer Liste mit zwei Personen

Bisher konnte der Java Entwickler, abgesehen von einer anderen Syntax, seine bisherigen Denkmuster anwenden. Als nächstes lassen wir die Liste nach Namen sortieren, indem wir der Methode „sortWith“ eine Vorschrift übergeben, anhand die Liste „persons“sortiert werden soll.

```
1 persons sortWith { (p1,p2) => p1.name < p2.name }
2 res1: List[Person] = List(Bak:29, Kacper:29)
```

Listing 7 Sortieren der Liste

Erzeugt wird in Listing 7 eine neue Liste „res1“, bei der die lexikalische Reihenfolge der Objekte bezüglich des Namensfeldes eingehalten wird.
Mittels dieses Beispiels ist bereits deutlich geworden, dass mit relativ wenig Quelltext viel erreicht wird. Viel wichtiger ist aber, dass der Java Entwickler mit diesem Quelltext bereits etwas anfangen kann, da Scala ihm bekannte Konzepte benutzt. Dies ist einer der Hauptvorteile der Sprache, die es dem Entwickler ermöglichen Schritt für Schritt auf Basis

der OOP Ergebnisse zu erzielen und mit wachsenden Kenntnissen der funktionalen Programmierung produktiver zu werden [Seeberger2011].

2.1.6 Skalierbarkeit

2.1.6.1 Erweiterbarkeit anhand von Datentypen

Viele Sprachen haben eine abgeschlossene Mächtigkeit, da deren Grundtypen in die Sprache integriert sind. Ein Beispiel dafür ist die Integration der komplexen und rationalen Zahlen in Haskel, Lisp und Python. Der Vorteil dieses Ansatzes besteht in dem nahtlosen Übergang bei der Anwendung. Der Hauptnachteil ist die schlechte Erweiterbarkeit, die eine Anpassung der Sprache voraussetzt. In Java gibt es aus diesem Grund Primitive, also Grunddatentypen, wie „int“ oder „double“, welche ein direkter Bestandteil der Sprache sind. Spezifischere Datentypen, wie „BigInteger“ oder „BigDecimal“, sind als Klassen innerhalb der Java Bibliothek definiert und können über einen Import in den Kontext geladen werden.

```
1 //definition
2 public static BigInteger factorial(BigInteger x) {
3         if(x.equals(BigInteger.ZERO)){
4                 return BigInteger.ONE;
5         }else{
6                 return x.multiply( factorial( x.subtract(BigInteger.ONE)));
7         }
8 }
9 ...
10 //call
11 System.out.println("5! " +  factorial(new BigInteger("5")));
12 5! 120
```

Listing 8 Implementierung der Fakultät unter Java

Listing 8 implementiert die Fakultätsfunktion rekursiv:

$$n! = \begin{cases} 1, & n = 0 \\ n \cdot (n-1)!, & n > 0 \end{cases}$$

Formel 1 Fakultät

Aus der Java-Implementierung sind zwei Nachteile dieses Ansatzes ersichtlich. Die Handhabung der Datentypen ist nicht intuitiv, wie es zwischen Primitives der Fall ist. Um die

Multiplikation und Subtraktion auf dem Datentyp auszuführen, müssen die Methodennamen angegeben werden. Dies führt bei der Ausführung vieler Operationen hintereinander zu regelrechten Methodenketten. Besonders bei aussagekräftigen, also langen Methodennamen, werden diese Ketten immer länger. Der zweite Nachteil ist eine Unterscheidung zwischen Objekten und Primitiven. Während der Vergleich zwischen zwei Primitiven (1 == 1) zum wahren Ergebnis führt, wäre der Einsatz des „==“ Operators in Zeile 3 zwar intuitiv, jedoch falsch. Im Objekt-Kontext muss der Wert des Objekts mit der „equals“ Methode verglichen werden, da der „==“ Operator sich auf die Referenzen der zu vergleichenden Objekte bezieht, welche in diesem Fall unterschiedlich sind [Odersky2010].

```
1 //definition
2 implicit def IntToBigInteger(i: Int) = new java.math.BigInteger(i.toString)
3 implicit def BigIntToBigInteger(bi: BigInt) = new java.math.BigInteger(bi.toString)
4
5 trait FacultyExtension{
6         private def factorial(x: BigInt):BigInt = {
7                 if(x == 0) 1 else x * factorial(x - 1)
8         }
9         def !():BigInt with FacultyExtension = {
10                new BigInt(factorial(this.asInstanceOf[BigInt])) with FacultyExtension
11        }
12 }
```

Listing 9 Implementierung der Fakultät in Scala

In Scala wird die Implementierung der Fakultät mit den Operatoren der linearen Algebra implementiert, wie aus Listing 9 Zeile 7 ersichtlich ist. Dies hängt einerseits mit der Tatsache zusammen, dass jede Operation in Scala ein Methoden-Aufruf ist und, dass alles ein Objekt ist, was einen Wert darstellt. Durch diesen einheitlichen Einsatz wird auch der „==“ Operator auf zwei Objekten angewendet [Odersky2010].

Weiter ist es möglich auf bestehende Klassen, z.B. aus importierten Bibliotheken, Einfluss zu nehmen, ohne von Ihnen direkt zu erben. Im Listing 9 Zeile 9 wurde die „!“ Methode definiert, welche die Fakultätsmethode aufruft. Dies geschieht innerhalb eines Traits. Ohne auf dessen Definition genau einzugehen, kann er in diesem Kontext als Analogie zu einer abstrakten Klasse in Java gesehen werden. Mittels des „FacultyExtension“ Traits kann die „scala.BigInt“ Klasse aus der Scala-API um die „!“ Methode erweitert werden, indem bei der Erstellung einer „scala.BigInt“ Variable der Trait durch „with“ hineingemixt wird. Im

Anschluss kann die hinzugefügte Methode „!“auch ohne den obligatorischen Punkt in Java, zwischen Bezeichner und Methode, ausgeführt werden, was der direkten mathematischen Notation gleichkommt. Neben dieser „Infix“ Schreibweise ist dies durch symbolische Namen für Methoden und Funktionen möglich, die nicht auf den ASCII-, sondern Unicode-Zeichensatz beschränkt sind [Odersky2010]. Dies ermöglicht beispielsweise ein griechisches Summenzeichen als Methodenname. Das Ergebnis dieser Merkmale verhält sich ähnlich der Operator-Überladung aus Sprachen wie C++.

```
1 //call
2 scala> val x = new BigInt(5) with FacultyExtension
3 x: scala.math.BigInt with FacultyExtension = 5
4
5 scala> x!
6 res0: BigInt with FacultyExtension = 120
```

Listing 10 Aufruf der erweiterten „!“ Methode aus Scala

Die Anpassung der Datentypen, welche sowohl im Trait (Kapitel 2.2.1.14), als auch bei der Erstellung des neuen Typs nötig wird, geschieht mittels einer impliziten Umwandlung: Implicits (Kapitel 2.2.2.3), die ein sehr mächtiges, aber auch gefährliches Merkmal von Scala darstellt. Damit simuliert Scala Eigenschaften von dynamischen Sprachen, wie Ruby [Wähner2012].

2.1.6.2 Erweiterbarkeit durch Kontrollstrukturen

Durch die bereits beschriebenen Eigenschaften und der Möglichkeit funktional zu programmieren, ist es in Scala möglich Kontrollstrukturen zu definieren bzw. „fühlt“ sich das Ergebnis für den Entwickler so an. Eine Kontrollstruktur ist nur dann sinnvoll, wenn sie einen bestimmten Anwendungsfall abdeckt. Betrachtet wird als Beispiel das „loan“ Entwurfsmuster, welches durch das „using“ Schlüsselwortes in C# .NET umgesetzt wird. Der Anwendungsfall ist der Einsatz von Ressourcen, welcher eine Initialisierung voraussetzt. Nach der Bereitstellung erfolgt der eigentliche Einsatz der Ressource mit abschließender Freigabe. Diese Art von Zugriffs-Muster tritt sehr häufig in der IT ein, beispielsweise bei Datenbank/Webserver- Verbindungen oder bei I/O-Zugriffen auf das Dateisystem und lohnt deshalb einer Vereinfachung.

```
1 using (TextReader textReader = new StreamReader(filename))
```

```
2 {
3       return textReader.ReadLine();
4 }
```

Listing 11 Anwendung des „using“ Schlüsselwortes unter C# .NET

Dazu wird die entsprechende Quelle, in Listing 11 der „textReader“, als Parameter an das Schlüsselwort übergeben und die Ressource wird allokiert. Innerhalb der geschweiften Klammern erfolgt der Zugriff auf die Ressource, indem der Entwickler mit dieser arbeitet. Damit nach dem Durchlaufen des Anweisungsblockes die Ressource wieder freigegeben werden kann, muss diese die „close“ Methode implementieren.

Wir bauen diese Kontrollstruktur nun mit den in Scala verfügbaren Mitteln nach. Die Theorie dazu wird im Kapitel 2.2.2.2.2 vertieft.

```
1 def using[A <: {def close(): Unit}, B](param: A)(f: A => B): B =
2 try {
3       f(param)
4 } finally {
5       param.close()
6 }
```

Listing 12 Definition der Methode „using“, welche als Kontrollstruktur verwendet werden kann

Im Prinzip werden die redundanten Abschnitte „try“ und „finally“ unter dem Schlüsselwort „using“ verfügbar gemacht. Der übergebene Parameter wird über die Funktion „f“ ausgeführt und muss die Methode „close“ für die Freigabe der Ressource enthalten. Die konkrete Anwendung ist dem Listing Listing 13 zu entnehmen [Pollak2009].

```
1 //definition
2 class Person(val name: String,val age: Int)
3
4 object Person {
5   def using...(Siehe: vorangegangenes Listing)
6
7   def findPeople(conn: Connection): List[Person] =
8     using(conn.createStatement) {
9       statement =>
10        using(statement.executeQuery("SELECT * FROM person")) {
11          resultSet =>
12            val ret = new ListBuffer[Person]
13            while (resultSet.next()) {
14              ret += new Person(resultSet.getString("name"), resultSet.getInt("age"))
15            }
16            ret.toList
17        }
18    }
```

```
19 }
20 //call
21 val con = DriverManager.getConnection("jdbc:db2:scaladb");
22 Person.findPeople(con).foreach( p => println(p))
```

Listing 13 Anwendung der „using“ Methode als Kontrollstruktur

Da Kontrollstrukturen in Scala beliebig hinzugefügt werden können, sind bestimmte Schlüsselwörter im Vergleich zu Java nicht enthalten, beispielsweise „break“. Auf diese Kontrollstruktur kann im Grunde verzichtet werden, da deren Funktion auch durch den Einsatz eines binären Flags ersetzt werden kann. Sollte jedoch der Entwickler nicht auf sein vertrautes Werkzeug verzichten können, existiert die Möglichkeit diese in den aktuellen Kontext einzubinden. Das folgende Listing veranschaulicht die Anwendung [Odersky2010].

```
1 def foo = {
2         var flag = true
3         var i = 0;
4         while(flag){
5                 i = i + 1
6                 println("i:" + i)
7                 if(i == 3) flag = false
8         }
9 }
10 ...
11 foo
12 i:1
13 i:2
14 i:3
```

```
1 import scala.util.control.Breaks._
2
3 def bar = {
4         var i = 0
5         while(true){
6                 i = i + 1
7                 println("i: " + i)
8                 if(i == 3) break;
9         }
10 }
11 ...
12 bar
13 i: 1
14 i: 2
15 i: 3
16 scala.util.control.BreakControl
```

Listing 14 Scala-Iteration mit und ohne „break“ Kontrollstruktur

Im Falle von „break“ handelt es sich wieder um eine Implementierung in einer Klasse, die aus der Scala Bibliothek entnommen werden kann.

2.2 Scala Sprachfeatures

Die hier behandelten Spracheigenschaften decken den Sprachumfang soweit ab, wie es für die Erstellung der Beispielanwendung und die spezifischen Scala Anwendungsfälle notwendig war. Um die hier dargestellten Konzepte zu testen, empfiehlt sich die Installation der Scala-Laufzeitumgebung und die Auswertung in der REPL. Genauere Anweisungen finden sich im Kapitel 3.2.2.

2.2.1 Objektorientierte Programmierung

Da die Zielgruppe dieser Arbeit ist der OOP kundige Java-Entwickler ist, wird auf eine ausführliche Erläuterung von Grundprinzipien der Objektorientierung, wie Abstraktion, Polymorphie, Vererbung und Kapselung, verzichtet. Das Kapitel beschränkt sich auf die Demonstration dieser Basis, wie sie in Scala Anzuwenden ist und an welchen Stellen, Scala im Vergleich zu Java prägnanter, objektorientierter und flexibler angewendet werden kann. Nach diesem Kapitel sollte der Java Entwickler im Stande sein, seine gewohnten Denkmuster in Scala umsetzen zu können, was eine Grundvoraussetzung für den produktiven Einsatz von Scala darstellt.

2.2.1.1 Klassen und Felder

Klassen fassen Daten in Feldern und Operationen in Methoden zusammen. Beide werden weiterhin als Member der Klasse bezeichnet. Klassen sind Schablonen für Objekte. Das Schlüsselwort für die Erstellung einer Klasse in Scala ist „class“. Neue Instanzen einer Klasse werden mit „new“ erzeugt. Felder sind Klassen-Member und enthalten Informationen oder Zustände eines Objektes. Ihre Definition beginnt mit den Schlüsselwörtern „val“ für unveränderbar und „var“ für veränderbare Felder [Odersky2010].

```
1 //definition
2 class Dog{
3         var name = "Beethoven"
4 }
5
6 //call
7 val dog1 = new Dog
8 val dog2 = new Dog
9 dog2.name = "Jacky"
```

Listing 15 Erstellung und Verwendung einer Klasse

Erstellt wird ein Klasse mit dem Namen „Dog“, welche ein veränderbares Feld „name“ enthält, auch Instanzvariable genannt. Der Primär-Konstruktor, ähnlich zum „Java-Default“ Konstruktor, ergibt sich implizit durch die Klassendefinition. Im Gegensatz zu Java gibt es kein „public“ Schlüsselwort, welches den Zugriff beschränkt, da der offene Zugang zu Klassen-Membern der Standardfall in Scala ist. Dies erkennt man daran, dass in der letzten Zeile, von Listing 15,der äußere Zugriff auf die interne Instanzvariable möglich ist. Abbildung

4 stellt die Repräsentation der Objekte im Speicher, nach deren Erstellung (Zeile 7 und 8) dar.

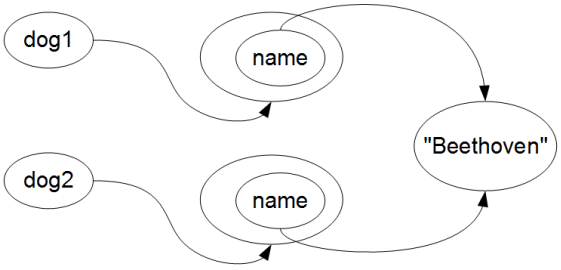


Abbildung 4 Objekte im RAM

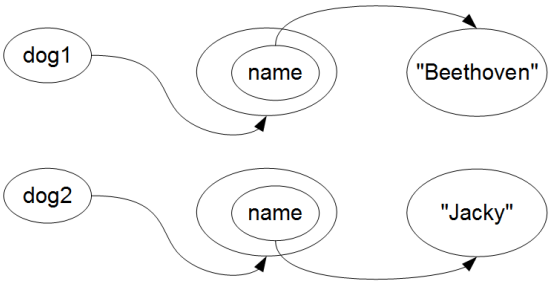


Abbildung 5 Objekte im RAM

In Zeile 9 wird der Name des zweiten Hundes verändert. Dies ist möglich, obwohl die Variablen „dog1“ und „dog2“ Konstanten sind welche mit „val“ definiert wurden. Das Feld „name“ eines Objektes, ist jedoch eine veränderbare Instanzvariable „var“. Auf das Feld kann also nachdem das Objekt erstellt wurde Einfluss genommen werden, siehe Abbildung 5. Eine erneute Zuweisung der Konstanten „dog1“ oder „dog2“ schlägt jedoch fehl [Odersky2010].

Als nächstes wird eine Klasse definiert, deren Feld bei der Objekterstellung initialisiert werden kann.

```
1 class Dog(n:String){
2         val name = n
3 }
```

```
1 class Dog(val name: String)
```

Listing 16 Initialisierende Klassen mit konstantem, öffentlichen Feld

Die Parameterliste des Konstruktors gehört zur Klassendefinition. Es handelt sich weiter um den Primär-Konstruktor. Der Konstruktorrumpf wird auf gleicher Ebene mit den Klassenattributen definiert. Die Initialisierung der Attribute, gehört also zur Konstruktordefinition dazu [Piepmeyer2010]. Die Syntax der Parameterliste lautet: Parametername, Doppelpunkt, Parametertyp. Listing 16 zeigt zwei Möglichkeiten zur Definition des gleichen Konstruktors. Dessen Feld „name“ ist öffentlich, unveränderbar und wird bei der Objekterstellung initialisiert [Seeberger2011].

```
1 class Dog(var name: String){
2         require(name.length > 0, "Name is to short!")
```

```
3 }
```
Listing 17 Initialisierende Klasse mit variablem, öffentlichen Feld

Das Feld des Konstruktors aus Listing 17 ist hingegen veränderbar. Zusätzlich wird mittels der „require“ Methode geprüft, ob das Feld nicht mit einem leeren String initialisiert wird. Die Methode erwartet zwei Argumente. Das erste ist ein Boolean-Ausdruck, welcher als Ergebnis „true“ erwartet. Das zweite Argument ist ein String, welcher als Nachricht für eine „IllegalArgumentException“ benötigt wird, sollte der Boolean-Ausdruck „false“ ergeben [Seeberger2011]. Mittels dieser Methode kann der Entwickler die Plausibilität der Konstruktorparameter sehr effizient prüfen. Sie ist innerhalb des „scala.Predef“ Singleton-Objektes definiert, dessen Member automatisch importiert werden. Somit stehen diese dem Entwickler jederzeit zur Verfügung. Das Singleton-Objekt wird in Kapitel 2.2.1.6 beschrieben.

```
1 public class Dog {
2       public String name;
3
4       public Dog(String name) {
5               if(!(name.lenght > 0)){
6                       throw new IllegalArgumentException("Name is to short!");
7               }
8               this.name = name;
9       }
10 }
```
Listing 18 Initialisierende Klasse mit variablem, öffentlichen Feld in Java

Listing 18 zeigt die identische Klassendefinition, wie Listing 17 in der Sprache Java. Bereits ab diesem Punkt wird ersichtlich, dass Scala identischen Informationsgehalt in weniger Quelltext vermittelt.
Als letztes Beispiel folgt die Definition einer Klasse, deren Feld initialisiert wird jedoch, weder öffentlich noch veränderbar ist. Diese Eigenschaft erfolgt durch das Auslassen von „var“ und „val“ innerhalb der Parameterliste.

```
1 class Dog(name: String)
```
Listing 19 Initialisierende Klasse mit konstantem, nicht öffentlichen Feld

Das Beispiel lässt die Frage offen, wie veränderbare, private Felder erstellt werden können. Die Antwort darauf wird im Kapitel 2.2.1.3 erläutert, jedoch werden davor noch benötigte Grundlagen dargelegt.

2.2.1.2 Standardwerte

Die folgende Definition einer Klasse würde scheitern, da das Feld „age“ und „name“ bei der Objekterstellung nicht definiert sind. Dem Entwickler bleibt hier die Möglichkeit einen Standardwert mit dem „_“ Wildcard-Operator zu vergeben oder die Klasse als „abstract“ zu deklarieren, siehe dazu Kapitel 2.2.1.10.

```
1 class Dog{
2         var name: String
3         var age: Int
4         override def toString = name + " " + age
5 }
```

Listing 20 Klassenfelder mit fehlenden Standardwerten

Die Entscheidung fällt im nächsten Listing auf den Wildcard-Operator. Zusätzlich wird die Methode „toString“ überschrieben, von der jedes Scala-Objekt ableitet und bei der Objekterzeugung innerhalb der REPL aufgerufen wird.

```
1 //definition
2 class Dog{
3         var name: String =_
4         var age: Int =_
5         override def toString = name + " " + age
6 }
7
8 //call
9 val d = new Dog
10 d: Dog = null 0
```

Listing 21 Klassenfelder mit zugewiesenen Standardwerten

An diesem Beispiel erkennt man sehr gut, dass der Datentyp „scala.Int“, welcher ein Wertetyp ist, von „scala.AnyVal“ erbt und als Standardwert die „0“ zugewiesen bekommt (siehe letzte Zeile in Listing 21). Der Referenztyp „java.lang.String“ ist Rückgabewert der „toString“ Methode und erbt von „scala.AnyRef“, dessen Standardwert „null“ ist. Auf die Zusammenhänge zwischen den Datentypen wird in Kapitel 2.2.1.13 eingegangen. Das System der Standardwerte ist damit weitestgehend identisch zu Java. Eine Ausnahme bildet der Typ „Unit“, der in Java nicht vorhanden ist und dessen Standardwert „()“ lautet [Piepmeyer2010].

2.2.1.3 Sichtbarkeit

Das aus Java bekannte Schlüsselwort „public“ existiert in Scala nicht. Als Standard ist in Scala alles öffentlich zugänglich und muss über die übrigen Zugriffs-Schlüsselwörter (engl.: Access Modifier) deklariert werden. Dabei verhält sich „private“ analog zu Java und sperrt den äußeren Zugriff auf die Klassenmember. Die Sichtbarkeit von „protected“ hingegen wird strenger gehandhabt, da sie den Zugriff ausschließlich auf Subklassen beschränkt. In Java ist der Zugriff aus dem identischen Paket (engl.: Package) möglich [Seeberger2011].

```
1 //def
2 class Dog{
3         private var name: String =_
4         private var age: Int =_
5         override def toString = name + " " + age
6 }
7
8 //call, test der sichtbarkeit
9 val d1 = new Dog
10 d1: Dog = null 0
11
12 d1.
13 asInstanceOf   isInstanceOf   toString (x)
```

Listing 22 Klassendefinition mit privaten, aber mit initialisierten Standardwerten

Innerhalb der REPL können die sichtbaren Klassen-Member eines Objektes, mit der Betätigung der Tabulator-Taste, dargestellt werden. Wie aus der letzten Zeile zu erkennen ist, sind die definierten Felder „name“ und „age“ nicht in der Menge der sichtbaren Klassen-Member enthalten. Diese Definitionen sind möglich, aber für die Initialisierung der Werte impraktikabel. Identisch dazu verhält sich folgende Klassendefinition, deren Felder zumindest bei Objekterstellung beeinflusst werden können.

```
1 class Dog(private var name: String, private var age:Int){
2         override def toString = name + " " + age
3 }
```

Listing 23 Klassendefinition mit privaten, aber initialisierten Feldern

In Java wird das Problem der Kapselung von Feldern mit korrespondierenden Methoden gelöst, die auch als Getter und Setter bekannt sind. Die Getter-Methode stellt den lesenden Zugriff auf den Zustand des Feldes (Inspektor) dar. Während die Setter-Methode (Mutator) den Zustand verändert und somit schreibend auf das Feld zugreift. In Scala erzeugt der

Compiler, Mutatoren und Inspektoren implizit für jedes Feld. Betrachtet wird dazu folgender Quelltext:

```
1 class Dog(val name: String, private var a: Int){
2         def age = a                                    //Getter
3         def age_=(ae:Int){ a = ae }                    //Setter
4         override def toString() = name + " " + age
5 }
```

Listing 24 Klassendefinition mit Properties

Der lesende Zugriff auf das im Konstruktor definierte Attribut „a“, erfolgt in diesem Fall über die Inspektor-Methode „age“ und der schreibende Zugriff über die Mutator-Methode „age_=“. Auf Methoden wird im nächsten Kapitel näher eingegangen. Bei der Anwendung dieser Klasse wird ersichtlich, dass sich die beiden Methoden beim Zugriff auf ein erstelltes Objekt, wie ein variables Feld verhalten [Piepmeyer2010]. Dieses Konzept ist aus der Programmiersprache C# als Properties bekannt [Hilyard2007].

```
1 val beethoven = new Dog("Beethoven", 10)
2 beethoven: Dog = Beethoven 10
3
4 beethoven.age                          //read
5 res0: Int = 10
6
7 beethoven.age = 5                      //write
8 beethoven.age: Int = 5
```

Listing 25 Anwendung von Scala Properties

Bei der Erstellung des Objektes erfolgt die Initialisierung der Felder „name“ und „a“, deren Werte abschließend von der „toString“ Methode ausgegeben werden (siehe Listing 25, Zeile 2). In Zeile 4 wird auf das Feld lesend zugegriffen, indem die Methode „age“ aufgerufen wird. Beim schreibenden Zugriff ersetzt der Compiler den Ausdruck „beethoven.age = 5“ mit „beethoven.age_=(5)“, wie aus Zeile 7 zu entnehmen ist [Piepmeyer2010]. Abschließend erfolgt der Vergleich der fachlich identischen Klasse „Dog“ in Java und dem Zugriff mittels Getter und Setter in Listing 26.

```
1 //definition
2 public class Dog {
3         public final String name;
4         private int a;
5
6         public Dog(String name, int a) {
```

```
7                 this.name = name;
8                 this.a = a;
9         }
10
11        public int getAge() { return a; }
12
13        public void setAge(int size) { this.a = size; }
14
15        @Override
16        public String toString(){ return name + " " + getAge(); }
17 }
18
19 //call
20 Dog d = new Dog("Beethoven", 10);
21 int age = d.getAge();                          //read
22 d.setAge(5);                                   //write
```

Listing 26 Anwendung von Java Getter- und Setter-Methoden

2.2.1.4 Methoden

Methoden beschreiben die Funktionalität innerhalb von Klassen und gelten als deren Member. Sie werden mit dem Schlüsselwort „def“ eingeleitet und enthalten ausführbaren Quellcode [Seeberger2011]. Methoden können auch innerhalb von Skripten oder eigenständig in der REPL definiert werden. Streng genommen bezieht sich der Methoden-Begriff auf ein Objekt [Piepmeyer2010].

```
1 def increment(i:Int):Int = { i + 1 }
```

Listing 27 Definition einer Methode

Die Deklaration der Methodensignatur beginnt mit „def“, gefolgt vom Namen, Parameterliste und meist optionalem Rückgabewert mit vorstehendem Doppelpunkt (siehe Listing 27). „=“ leitet die Implementierung einer Methode ein, wobei diese ein einzelner Ausdruck oder ein Code-Block in geschweiften Klammern sein kann [Seeberger2011]. Die Parameter einer Methode sind „val“s und somit Konstant. Jeder Versuch das übergebene Argument innerhalb der Methode neu zuzuweisen scheitert [Piepmeyer2010].

Der letzte Ausdruck innerhalb der geschweiften Klammern bildet den Rückgabewert der Methode. Im Gegensatz zu Java muss kein explizites „return“ Statement angegeben werden. An dieser Stelle wird versucht den Entwickler davon abzubringen, seinen Quelltext mit vielen dieser Ausdrücke komplex zu gestalten und ihn damit in Richtung kurzen Methodenimplementierungen zu bewegen. Für Prozeduren, also Methoden, welche wegen

ihrer Seiteneffekte oder Benutzerinteraktion verwendet werden, gibt es den Datentyp „Unit“. Dieser ist analog zu „void“ in Java. Folgende Methode hat beispielsweise „Unit“ als Rückgabewert [Seeberger2011].

```
1 def foo = println("bar")
2 foo: Unit
```

```
1 def foo():Unit = { println("bar") }
2 foo: ()Unit
```

Listing 28 Definition einer Prozedur in Scala

Während jede Funktion (siehe Kapitel 2.2.2.2) einen Rückgabewert hat, kann eine Methode auch eine Konstante zurückgeben.

```
1 def one = 1
2 one: Int
```

Listing 29 Konstante als Rückgabewert

Diese ist zwar kein eigenständiges Objekt, hat aber ihren eigenen Typ, siehe Listing 29, Zeile 2. Der Rückgabewert ist an vielen Stellen optional und nur selten zwingend, wie bei rekursiven Methoden. Es ist aber guter Stil diesen immer mit anzugeben, da er die Methodendefinition verständlicher macht. Die Angabe der Typparameter, sprich die Datentypen der übergebenen Argumente an die Methode, ist hingegen zwingend, da Scala eben Polymorphie und Überladung unterstützt [Piepmeyer2010].

```
1 //definition
2 class Dog(val name: String, var age: Int){
3
4         def minus(that: Dog): Int = this.age - that.age
5
6         def -(min: Int): Int = this.age - min
7
8         def -:(min: Int): Int = this.age - min
9 }
10
11 //call
12 val d1 = new Dog("bello", 10)
13 val d2 = new Dog("jacky", 3)
14 d1.minus(d2)
```

Listing 30 Definition und Methodenanwendung

Die Methode „minus“ wird innerhalb der Klasse „Dog“ definiert, um zwei Hunde-Instanzen voneinander abziehen zu können. Wie in Zeile 4 aus Listing 30 zu erkennen ist, wird das Feld „age“ der zugehörigen Instanz vom Feld des übergebenen Objektes subtrahiert. Da es sich

um einen Ausdruck handelt, der in eine Zeile passt, kann auf die Verwendung von geschweiften Klammern verzichtet werden [Seeberger2011].

An dieser Stelle muss kurz über Operatoren ausgeholt werden, um die zwei weiteren Methoden erklären zu können. Wie bereits erwähnt, erscheint für den Programmier auch ein „scala.Int“ als Objekt. Demnach ist folgendes Listing möglich.

```
1 val x = 1
2 val y = 2
3 x.+(y)          /* == */          x + y
4 res0: Int = 3
```

Listing 31 Operator-Notation

Der „+“ Operator ist demnach ein Methodenaufruf auf einer Integer-Instanz (siehe Listing 31, Zeile 3 linke Seite). Für die Operator-Notation spricht die einfache Lesbarkeit in diesem Zusammenhang (Zeile 3, rechte Seite). Sie sollte aber nur benutzt werden, wenn die Methode genau einen Parameter aufnimmt, ein Ergebnis zurückliefert und frei von Seiteneffekten ist [Seeberger2011].

Zurück zur Beschreibung der zwei übrigen Methoden in das Klasse „Dog“ aus Listing 30. Die zweite Methode ist mit dem Minuszeichen „-“ benannt. Sie erwartet ein „scala.Int“ als Parameter, welcher innerhalb der Methode vom Alter des Hundes abgezogen und das Ergebnis zurückgegeben wird. Nach den bisherigen Erkenntnissen funktioniert die „-“ Methode, solange diese auf einem Dog-Objekt aufgerufen wird, jedoch nicht umgekehrt.

```
1 d1 -  1
2 2  -  d1
3 2  -: d1
```

Listing 32 Demonstration der links- und rechtsassoziativen Methoden

Dies hängt mit der Assoziativität der Methoden zusammen. Operatoren die mit einem „:“ enden, sind rechtsassoziativ, alle anderen linksassoziativ [Braun2011]. Damit schlägt Zeile 2 in Listing 32 fehl, während in Zeile 3 die Methode „-:“ auf dem rechten Objekt „d1“ ausgeführt wird.

2.2.1.5 Zusätzliche Konstruktoren und Standardwerte

Sollten weitere Konstruktoren neben dem Primär-Konstruktor zur Erzeugung notwendig werden, erfolgt deren Definition als Methode „def“ und die Kennzeichnung des Konstruktors mit dem Schlüsselwort „this“ mit anschließender Parameterliste. Diese Argumente sind in der gesamten Klasse sichtbar.

```
1 //definition
2 class Dog(var name: String){
3         def this(name: String, age: Int) = this(name + " is " + age + " years old")
4         def this() = {this("defaultName"); println("auxiliary")}
5         println("primary")
6 }
7
8 //call
9 val d1 = new Dog("Bello", 5)
10 primary
11
12 d1.name
13 res0: String = Bello is 5 years old
14
15 val d2 = new Dog
16 primary
17 auxiliary
18
19 d2.name
20 res1: String = defaultName
```

Listing 33 Anwendung der „Auxiliary Constructors“

Zusätzliche Konstruktoren werden in Scala „Auxiliary Constructors“ genannt. Die erste Aktivität innerhalb dessen muss der Aufruf des Primär-Konstruktors sein, siehe Listing 33, Zeile 3. Nach der Konstruktordefinition, erfolgt der Aufruf des Primär-Konstruktors, auch wenn dieser hier fachlich keinen Sinn ergibt. Der zweite zusätzliche Konstruktor erstellt einen Standardwert, welcher dem Feld „name“ immer zugewiesen wird, wenn der parameterlose Konstruktor eingesetzt wird, Zeile 15. Auffällig dabei ist die Reihenfolge, welche in den Zeilen 16 und 17 ersichtlich wird. So wird der Primär-Konstruktor immer zuerst durchlaufen, erst danach folgt die Abarbeitung weiterer Anweisungen [Seeberger2011].

Analog zu Java kann auch die Sichtbarkeit des Primär-Konstruktors begrenzt und die Erstellung des Objektes durch einen Zusätzlichen übernommen werden.

```
1 //definition
2 class Dog private (name: String, age: Int){
3         def this() = this("Bello",42)
4         override def toString = name + " " + age
5 }
6
7 //call
8 val d = new Dog
9 d: Dog = Bello 42
```

Listing 34 Sperrung des Primär-Konstruktors

Nach dem Klassennamen folgt der Access Modifier „private“, welcher den Primär-Konstruktor unzugänglich macht. Die Klasse kann danach nur noch mittels des parameterlosen Konstruktors instanziiert werden.

Um Standardwerte für Felder festzulegen, gibt es seit Scala 2.8 die sogenannten „named and default arguments“. Dabei werden die Standardwerte direkt nach dem Datentyp innerhalb des Primär-Konstruktors durch den „=“ Operator zugewiesen.

```
1 //definition
2 class Dog(val name: String = "Bello", var age: Int = 0){
3         override def toString = name + " " + age
4 }
5
6 //call
7 val d1 = new Dog
8 d1: Dog = Bello 0
9
10 val d2 = new Dog("Beethoven")
11 d2: Dog = Beethoven 0
12
13 val d3 = new Dog(age = 7)
14 d3: Dog = Bello 7
```

Listing 35 Initialisierung mit „named and default arguments“

Wie zu erwarten, werden die Standardwerte beim Einsatz des parameterlosen Konstruktors, siehe Zeile 7 und 8 in Listing 35, übernommen. Die Definition zusätzlicher Konstruktoren für diesen Zweck ist demnach obsolet. In Zeile 13 und 14 wird dem Feld „age“ der Wert direkt zugewiesen, während das Feld „name“ den Standardwert übernimmt [Seeberger2011].

2.2.1.6 Singleton-Objekt

Im Vergleich zu Java haben Klassen in Scala keine „static“ Member. Dabei handelt es sich um Klasseneigenschaften, welche für alle Instanzen der Klasse identisch sind. Als Beispiel wird

das Feld „PI“ aus „java.lang.Math“ betrachtet. Es besteht keine Notwendigkeit für „PI“ in unterschiedlichen Objekten von „Math“ seinen Wert zu ändern, da dieser für alle gleich sein soll. In dem Fall ist es nicht einmal notwendig, dass von „Math“ überhaupt eine Instanz erstellt wird, weshalb dessen Konstruktor mit „private“ unzugänglich ist.

```
1 double pi = Math.PI;
```

Listing 36 Aufruf eines „static“ Members in Java

Wie man erkennen kann, ist der Zugriff auf das Feld ohne die Instanziierung der Klasse möglich. Identisch verhält es sich mit Methoden, wie „sin(double a)“. Für statische Methoden gilt allerdings die Einschränkung, dass diese nicht auf Instanzvariablen zugreifen dürfen [Ullenboom2011]. Um diese und weitere Einschränkungen zu umgehen, besteht das Singleton-Entwurfsmuster. Dabei handelt es sich handelt es sich um ein Objekt, von dem zur Laufzeit ausschließlich eine Instanz erstellt werden kann. Während das Prinzip schnell erklärt ist, muss besonders bei der Implementierung in Multi-Thread-Anwendungen besonders darauf geachtet werden, dass zur Laufzeit doch nicht mehrere Objekte instanziiert werden können [Eilebrecht2010].

All diese Probleme tauchen in Scala nicht auf. Das Schlüsselwort „static“ existiert in Scala nicht [Odersky2010]. Stattdessen stellt Scala das Schlüsselwort „object“ bereit, welches anstelle von „class“ eingesetzt wird. Damit wird ein Singleton-Objekt erstellt, welches, analog zum genannten Entwurfsmuster, einmalig ist. Es können somit keine Instanzen mittels „new“ erzeugt werden. Aus diesem Grund nimmt der Konstruktor eines Singleton-Objektes keine Parameter entgegen [Odersky2010]. Statische Member aus Java werden in Scala innerhalb eines Singleton-Objektes definiert.

```
1 //definition
2 object Dog{
3         val name = "Beethoven"
4         println("init Dog")
5         override def toString = "name: " + name
6 }
7
8 //call
9 val d1 = Dog
10 init Dog
11 d1: Dog.type = name: Beethoven
12
```

```
13 val d2 = Dog
14 d2: Dog.type = name: Beethoven
15
16 (d1 eq d2)
17 res0: Boolean = true
```

Listing 37 Definition und Anwendung eines Singleton-Objektes

Die Bereitstellung erfolgt nachdem das Objekt zum ersten Mal verwendet wird. Bei der Initialisierung wird der Konstruktor einmalig durchlaufen, wobei der String „init Dog“ ausgegeben wird, siehe Zeile 10 aus Listing 37. Trotz weiterer Zugriffe erscheint keine weitere Ausgabe des genannten Strings. Abschließend wird in Zeile 16 überprüft, ob es sich tatsächlich um dasselbe Objekt handelt, indem mit der „eq“ Methode die Referenzen der Konstanten „d1“ und „d2“ verglichen werden. Das Resultat, bestätigt dies mit dem Wert „true“, da beide auf dasselbe Objekt referenzieren [Piepmeyer2010].

Einzelne Singleton-Objekte definieren keinen Typ von dem eine Instanz erstellt werden kann. Jedoch können diese von einer Klasse oder einem Trait erben [Odersky2010]. Dieses Verhalten entspricht dem OOP-Paradigma, hingegen können in Java statisch definierte Member nicht polymorph überschrieben werden [Piepmeyer2010].

Wird ein Singleton-Objekt einzeln, ohne begleitende Klasse verwendet, spricht man von einem „standalone object“ [Odersky2010]. Dieses wird beispielsweise für den Einstiegspunkt einer Anwendung benutzt, nachdem durch das fehlende „static“ Schlüsselwort, nicht die typische Java-Signatur für die „main“-Methode definiert werden kann. Das nachfolgende Listing zeigt die entsprechende Scala-Version.

```
1 object TestDrive{
2         def main(args: Array[String]){
3                 ...
4         }
5 }
```

Listing 38 Scala Singleton-Object mit „main“ Methode als Einstiegspunkt

2.2.1.7 Begleit-Objekt

Teilt sich das Singleton-Objekt den Namen mit einer Klasse, bezeichnet man dieses als Begleitobjekt (engl.: Companion-Object). Die Definition der Klasse und des Singleton-Objekts in einer Datei ist in Scala möglich. Deren Namensraum ist nämlich nicht zwingend an den

Namen der Quelltext-Datei gebunden, wie es in Java der Fall ist. Werden beide innerhalb der gleichen Quellcode-Datei definiert, hat die Begleitklasse Zugriff auf die privaten Member des Begleitobjektes [Odersky2010].

Mit diesem Begleitobjekt ist es nun wie in Java möglich, statische und nicht statische Member weiterhin gemischt zu verwenden [Piepmeyer2010].

```
1 //definition
2 class Dog(val name: String){
3         def bark: Unit = println(name + " says woof, count: " + Dog.counter)
4 }
5
6 object Dog{
7         private var counter = 0
8         def createDog(n:String):Dog = {
9                 Dog.counter = Dog.counter + 1
10                new Dog(n)
11        }
12 }
13
14 //call
15 val d1 = Dog.createDog("Beethoven")
16 val d2 = Dog.createDog("Jacky")
17 d2.bark
18 Jacky says woof, count: 2
```

Listing 39 Anwendung des Singleton-Object als Begleitobjekt einer Klasse

Die Erstellung des Objektes erfolgt über eine Factory-Methode „createDog“, siehe Listing 39, Zeile 8. Diese zählt nebenbei die Anzahl der erstellten Objekte. Innerhalb einer Objekt-Instanz wird durch die Methode „bark“ auf den Member „counter“ des Begleitobjektes zugegriffen. Als Einschränkung gilt, dass der Zugriff vom Begleitobjekt auf die Member der Begleitklasse nicht möglich ist. Die Ausnahme bildet der Konstruktor der Begleitklasse, wie anhand der „createDog“ Methode in Zeile 15 und 16 zu erkennen ist.

2.2.1.8 Vererbung

Die Vererbung von Membern einer Basisklasse an eine Subklasse ist ein wesentliches Prinzip der OOP. In den meisten Fällen ist die Anwendung dieser analog zum Java-Pendant. Das Schlüsselwort für diese Beziehung lautet „extends“. Wird dieses bei der Definition einer Klasse ausgelassen, erbt jede Instanz von dem Objekt „scala.AnyRef“ (siehe Kapitel 2.2.1.13). Dieses ist mit dem „java.lang.Object“ Datentyp gleichbedeutend. Die Intention dahinter ist

wie in Java, Grundfunktionalität, wie die „toString“ Methode, für alle Objekte bereitzustellen [Piepmeyer2010].

```
1 //definition
2 class Animal{
3         def makeNoise = println("what noise ?")
4 }
5 class Dog extends Animal
6
7 //call
8 val d1 = new Dog
9 d1.makeNoise
```

Listing 40 Vererbung mit „extends“

Listing 40 zeigt die Definition der Basisklasse „Animal“ und der Subklasse „Dog“, welche die Methode „makeNoise“ übernimmt. Es werden alle Member der Basisklasse geerbt, welche standardmäßig öffentlich sichtbar oder durch den Access Modifier „protected“ beschränkt sind. Identisch zu Java können Scala-Klassen nicht mehr als von einer Basisklasse erweitert werden. Weiter kann mit dem Schlüsselwort „final“ die Ableitung von einer Klasse verhindert werden [Seeberger2011].

```
1 final class Dog
```

Listing 41 Beschränkung der Vererbung mit „final“

Andere Konzepte verhalten sich jedoch abweichend zu Java, diese werden weiterhin veranschaulicht.

2.2.1.9 Member überschreiben

Liegt in der Basisklasse ein Member vor, dessen Eigenschaften in der Subklasse spezifisch für seinen Typ sein sollen, muss dieser Member mittels des „override“ Schlüsselwortes überschrieben werden. Betrachtet wird als erstes die Java-Implementierung der Klasse „Dog“, in welcher die „toString“ Methode aus der Klasse „java.lang.Object“ überschrieben wird [Piepmeyer2010].

```
1 public class Dog{
2
3         public String name;
4
5         public Dog(String name){
6                 this.name = name;
```

```
1 class Dog(var name: String){
2         override def toString: String =
"dog.name: " + name
3 }
```

```
7       }
8       @Override public String toString(){
9               return "dog.name: " + name;
10      }
11 }
```

Listing 42 Überschreiben der „toString“ Methode in Java und Scala

Die Annotation „@Override“ stellt sicher, dass die Subklasse „Dog“ die „toString“-Methode der Oberklasse „Object“ überschreibt. Diese Annotation ist allerdings optional und führt bei versehentlichem Abweichen von der Methodensignatur zu einer Überladung, anstatt einer Überschreibung der Methode. Mit der Annotation überprüft der Compiler, ob tatsächlich eine Methode der Oberklasse überschrieben wird [Ullenboom2011].

In Scala ist der Einsatz von „override“ beim Überschreiben eines Members obligatorisch. Wird es im Fall von Scala aus Listing 42 ausgelassen, tritt folgende Fehlermeldung auf [Seeberger2011].

```
1 <console>:8: error: overriding method toString in class Object of type ()java.lang.String;
2 method toString needs `override' modifier
3                       def toString: String = "dog.name: " + name
4                           ^
```

Listing 43 Fehlermeldung bei fehlendem „override“ Schlüsselwort

Bei „java.lang.String“ handelt es sich um einen Subtyp von „scala.AnyRef“, welcher, wie bereits erwähnt, dem Typ „java.lang.Object“ auf der JVM entspricht [Seeberger2011]. Details zur Scala-Typhierarchie folgen in Kapitel 2.2.1.13.

2.2.1.10 Abstrakte Klassen

In Scala findet das Schlüsselwort „abstract“ nur im Zusammenhang mit der gesamten Klasse Verwendung, nicht mit einzelnen Membern. Sobald ein Feld nicht mit einem konkreten Wert initialisiert wird oder eine Methodenimplementierung ausbleibt, muss die Klasse mit dem Schlüsselwort „abstract“ versehen werden. Bleibt die vollständige Definition eines Members aus, ist die Klasse demnach nicht instanziierbar [Braun2011]. Für Felder spielt es dabei keine Rolle, ob diese un- oder veränderbar sind [Piepmeyer2010].

```
1 abstract class Animal{
2       val name: String
3       def makeNoise
```

```
4 }
```
Listing 44 Abstrakte Klassendefinition

In diesem Kontext ergibt die Implementierung der Methode „makeNoise“ keinen Sinn, da ein unbestimmtes Tier, kein typisches Tiergeräusch von sich gibt. Durch den Verzicht des Methodenkörpers, als auch durch die fehlende Zuweisung eines Wertes an das Feld „name“, ist die „abstract“ Deklaration der Klasse „Animal“ zwingend. Der Datentyp eines Feldes, welches keine Initialisierung durch einen Wert oder Parameter erfährt, muss angegeben werden [Piepmeyer2010]. In diesem Fall ist es der Datentyp „java.lang. String“, siehe Listing 44, Zeile 2.

```
1 //var 1
2 class Dog(override val name: String) extends Animal{
3         override def makeNoise = println("wau wau, says " + name)
4 }
5
6 //var 2
7 class Dog(val name: String) extends Animal{
8         def makeNoise = println("wau wau, says " + name)
9 }
10
11 //var 3
12 class Dog extends Animal{
13         override def makeNoise = println("wau wau, says " + name)
14 }
```
Listing 45 Beispiele für das Überschreiben der Member der Basisklasse

In Listing 45 findet nun der Einsatz des „override“ Schlüsselwortes statt, indem die Klasse „Dog“ durch „extends“ von „Animal“ erbt. Die ersten beiden Varianten überschreiben die Member „name“ und „makeNoise“. Der Verzicht auf das „override“ Schlüsselwort in der zweiten Variante ist möglich, weil der Compiler erkennt, dass beide Member in der Subklasse zwingend überschrieben werden müssen. In der letzten Version findet keine Überschreibung von „name“ statt, weshalb diese Klassendefinition fehlschlägt [Piepmeyer2010].

2.2.1.11 Konstruktorverkettung

Enthält der Basisklassenkonstruktor Parameter müssen diese bei der Vererbung der Subklasse an ihn übergeben werden. Das folgende Listing demonstriert diesen Zusammenhang [Seeberger2011].

```
1 class Dog(val name: String) extends Animal{
2         println("Dog")
3         def makeNoise = println("wau wau, says " + name)
4 }
5
6 class SaintBernard(name: String,val barrelSize: Int) extends Dog(name){
7         println("SaintBernard")
8         override def makeNoise = println("wuff wuff, says " + name)
9 }
```

Listing 46 Übergabe des „name“ Parameters von „SaintBernard“ an „Dog“

Der erste Parameter des Bernhardiners (engl.: „SaintBernard“) ist dessen „name“ welcher an die Basisklasse „Dog“ weitergegeben wird. Darauf folgt die Definition eines Feldes „barrelSize“, welches nur innerhalb der Subklasse vorhanden ist [Piepmeyer2010].

```
1 val sb1 = new SaintBernard("beethoven",5)
2 Dog
3 SaintBernard
```

Listing 47 Verarbeitungsreihenfolge der Konstruktoren

Wie man aus Listing 47 erkennen kann, wird zuerst der Konstruktor der Klasse „Dog“ initialisiert und durchlaufen, abschließend der von „SaintBernard“. Um den aus Java bekannten „super“ Konstruktor aufzurufen, in diesem Fall den von „Dog“, müssen all dessen Argumente an ihn übergeben werden. In diesem Fall geschieht das über den Parameter „name“ der Klasse „SaintBernard“. Der darin enthaltene Wert wird direkt an die Basisklasse „Dog“ weitergegeben [Odersky2010]. Natürlich muss der Wert für den Basisklassenkonstruktor „dog“ nicht von einem Parameter stammen, sondern kann auch innerhalb der Klasse definiert werden.

2.2.1.12 Polymorphie

Analog zu Java, kann ein Bezeichner abhängig von seinem zugewiesenen Datentyp, zu unterschiedlichen Fähigkeiten imstande sein. In Java beschränkt sich diese Eigenschaft jedoch auf Methoden. Folgender Quelltext stellt daher keine Abweichung zu Java dar.

```
1 val beethoven:Dog = new SaintBernard("Beethoven",0)
2 beethoven.makeNoise
3 wuff wuff, says Beethoven
```

Listing 48 Java-Polymorphie

Auf dem Objekt „beethoven“, dessen Typ „Dog“ ist und sich am Listing 46 orientiert, wird die Implementierung des Subtyps „SaintBernard“ ausgeführt. Ersichtlich ist dieser Zusammenhang durch die Ausgabe nach dem Methodenaufruf „beethoven.makeNoise“ aus Listing 48 in Zeile 3 und dem Vergleich der Implementierungen beider Hunde-Typen.
Mit identischer Herangehensweise wird nun das Verhalten eines Attributes, einer Java- und einer Scala-Definition verglichen.

```
1 //Java definition
2 public class Hi {
3         protected int v = 42;
4         public void out(){
5                 System.out.println(v);
6         }
7 }
8
9 public class Lo extends Hi{
10        protected int v = 4711;
11 }
12
13 //call
14 public class testDrive {
15        public static void main (String[] args){
16
17                Hi h = new Hi();
18                Hi l = new Lo();
19
20                h.out();        //output: 42
21                l.out();        //output: 42
22        }
23 }
```

```
1 //Scala definition
2 class Hi{
3         protected val v = 42
4         def out = println(v)
5 }
6
7 class Lo extends Hi{
8         override protected val v = 4711
9 }
10
11 //call
12 val h = new Hi
13 val l = new Lo
14 h.out
15 42
16
17 l.out
18 4711
```

Listing 49 Polymorphie-Vergleich von Attributen in Java und Scala

In beiden Listings enthält die Basisklasse „Hi“ und die Subklasse „Lo“ das Attribut „v“ mit unterschiedlichen Wertzuweisungen (42 und 4711). Abhängig vom verwendeten Datentyp sollte der initialisierte Wert in der Ausgabe erfolgen. Da dies in Java nicht der Fall ist, besteht keine Polymorphie für Java-Attribute. In Scala hingegen findet die Umsetzung des „Prinzips des einheitlichen Zugriffes“ (engl.: Uniform Access Principle) statt. Dieses verlangt, dass jedes Objekt seine Dienste mit einer einheitlichen Syntax anbietet. Konkret bedeutet dies syntaktisch identischen Zugriff auf Attribute und Methoden eines Objektes. Aus diesem Grund ist der Effekt des „override“ Schlüsselwortes auf alle Scala-Felder wie Scala-Methoden ohne Parameter identisch [Piepmeyer2010]. Auf der rechten Seite des Listing 49 ist anhand

der Ausgabe des Wertes „4711“, das polymorphe Verhalten für Attribute in Scala nachgewiesen.

2.2.1.13 Scala Typhierarchie

Der nachfolgende Text bezieht sich auf das Klassendiagramm der Scala-Typhierarchie, welches dem Anhang 5.1 zu entnehmen ist.

Die Scala-Typhierarchie teilt sich in zwei markante Stränge auf. Die Klasse „scala.AnyVal“ bildet die Basisklasse für alle aus Java bekannten „Primitives“ bzw. Wertetypen, wie „int“, „float“, usw. Der zweite Strang leitet sich von der Klasse „scala.AnyRef“ ab. Es handelt sich hierbei um Referenztypen und entspricht genau dem „java.lang.Object“ der Java-Plattform, von dem alle Objekte in Java eine Subklasse sind [Seeberger2011].

Ist in Scala nicht alles ein Objekt? Für den Entwickler kann diese Frage, zu 99% mit „true“ beantwortet werden. Intern jedoch wandelt der Compiler an jeder möglichen Stelle, beispielsweise ein „scala.Int“ in einen 32-bit Word („int“ Primitive) um [Odersky2010]. Die Repräsentation als Objekt benutzt der Compiler ausschließlich, wenn es sich nicht vermeiden lässt. Die Ursache dieses Vorgehens verdeutlicht folgender Java-Code.

```
1 int hi = Integer.MAX_VALUE
2 for(int i = 0; i !=hi; i++)
3 for(Integer i = 0; i.compareTo(hi); i = i + 1)
```

Listing 50 Iteration mit Primitives und Objekten

Die for-Schleife zählt in beiden Versionen einfach nur hoch, bis der Wert von „hi“ erreicht ist. In der zweiten Version wird allerdings ein sogenannter Wrappertyp verwendet. Dieser bildet den Primitive-Wert „int“ als Objekt ab. Das ist beispielsweise bei der Definition von generischen Typen zwingend. Der Hauptunterschied besteht allerdings in der Laufzeit der Schleifen, wobei die „int“-Schleife bis um den Faktor drei performanter ist. Da in jeder Iteration ein neues Integer-Wrapperobjekt erstellt und der variablen „i“ eine neue Referenz zugewiesen wird, entsteht eine enorme Belastung seitens der Garbage Collection der JVM, welche für die Speicherverwaltung zuständig ist. Eine Differenzierung ist also unumgänglich, jedoch führt in Scala diese der Compiler selbstständig durch und setzt nur dort Objekte ein, wo es nötig ist.

Alle aus Java bekannten Wertetypen leiten von „scala.AnyVal“ ab. Für Prozeduren, also Funktionen welche, aufgrund von Seiteneffekten erstellt wurden und keinen sinnvollen Rückgabewert haben, existiert der „Unit“ Datentyp. In einer rein funktionalen Programmiersprache gäbe es dieses Konzept nicht (Siehe Kapitel 2.2.2) [Piepmeyer2010].

Alle Referenztypen in Scala (Objekte) erben von der Basisklasse „scala.AnyRef“. Wie bereits in Kapitel 2.1 erwähnt wurde, können Objekte durch die „==“ Methode auf Gleichheit ausgewertet werden. Diese ist in der Klasse „scala.Any“ definiert und delegiert den Aufruf an „equals“ aus „scala.AnyRef“ weiter. Da „==“ als „final“ deklariert ist, erfolgt die Anpassung des Gleichheits-Verhaltens durch Überschreiben von „equals“. Mittels der „eq“ Methode aus „scala.AnyRef“ kann die Referenz zweier Objekt verglichen werden [Seeberger2011].

Alle Referenztypen in Java haben den Wert „null“, welcher als Schlüsselwort implementiert ist. In Scala ist der Wert „null“ ein Objekt vom Typ „scala.Null“, welcher von „scala.AnyRef“ und allen Subklassen von „scala.AnyRef“ erbt [Piepmeyer2010]. Der Typ „scala.Null“ kann nur für Referenztypen, nicht aber für Wertetype verwendet werden, da er in keiner Beziehung zum Strang „scala.AnyVal“ steht. In Scala besteht allerdings die Möglichkeit, beide Stränge der Typhierarchie zu vermischen. Dies wird beispielsweise benötigt, wenn eine Liste sowohl aus Werte- als auch Referenztypen besteht.

```
1 val list = List(1,2,"drei")
2 list: List[Any] = List(1, 2, drei)
3
4 list.foreach( x => println(x) )
5 1
6 2
7 drei
```

Listing 51 Polymorpher Zugriff auf eine Liste mit Werte- und Referenztypen

In Listing 51 werden zwei „scala.Int“ Werte „1“ und „2“ mit dem Basistyp „scala.AnyVal“ und ein „java.lang.String“ und dem Wert „drei“, dessen Basistyp „scala.AnyRef“ ist, in eine Liste gesteckt. Bei der Angabe des Types „scala.Any“ in eckigen Klammern, in Zeile 2, handelt es sich um einen Typparameter, welcher als Java-Generic betrachtet werden kann. Er beschreibt den Datentyp der in der Liste enthaltenen Elemente. Abschließend erfolgt die Ausgabe jedes Elementes der Liste, wobei auf jedem Element, welches kein

„java.lang.String“ ist, die „toString“ Methode aus „scala.Any“ aufgerufen wird. Dies trifft für die beiden „scala.Int“ Typen zu.

Soll hingegen eine leere Liste erstellt werden, kann keine Vorhersage über die zukünftig enthaltenen Typen getroffen werden. Für diesen Fall besteht der Datentyp „scala.Nothing“. Er ist Untertyp aller Typen des Scala-Typsystems und gehört zu den zwei „Bottomtypes“ der Scala-Typhierarchie. „scala.Null“ und „scala.Nothing“, beides „Bottomtypes“, sind als „final“ definiert und können nicht weiter subtypisiert werden [Piepmeyer2010].

```
1 val list = List()
2 list: List[Nothing] = List()
```

Listing 52 Erstellung einer leeren Liste

2.2.1.14 Traits

Das Konzept der Traits ähnelt in vielerlei Hinsicht dem von Java-Schnittstellen (engl.: Interface) [Piepmeyer2010]. Hauptmerkmal dieser ist die Beschreibung verfügbarer Funktionalität, welche sie jedoch nicht selbst implementieren. Ein Klasse „A“ implementiert das Interface „I“ und bietet diese Funktionalität über „I“ an weitere Komponenten an [Ullenboom2011]. Diesen Standardfall bilden wir als nächstes mittels Traits in Scala ab. Im ersten Schritt definieren wir den Trait „Noise“, welcher ausschließlich abstrakte Member enthält. Danach folgt die Klasse „Dog“, welche die vom Trait bereitgestellte Funktionalität implementiert, indem die Member überschrieben werden. Abschließend wird der Nutzer „ParkArea“ definiert, welche über den Trait „Noise“ auf die Implementierung der Methode „makeNoise“ in der Klasse „Dog“ zugreift. Die Methode verwendet die Attribute, indem sie das Geräusch „noise“ ausgibt und zwar so oft, wie es in „count“ definiert ist.

```
1 //definition
2 trait Noise{
3         val count:Int
4         val noise:String
5         def makeNoise: Unit
6 }
7
8 class Dog(val name: String) extends Noise{
9         override val count = 2
10        override val noise = " wau "
11        def makeNoise: Unit = for(i <- (1 to count)) print( noise )
12 }
```

```
13
14 class ParkArea(val list:List[Noise]){
15       def listenForNoise = list.foreach( animal => animal.makeNoise )
16 }
17
18 //call
19 val dogList = List(new Dog("Beethoven"), new Dog("Jacky"))
20 val park = new ParkArea(dogList)
21 park.listenForNoise
22  wau  wau  wau  wau
```

Listing 53 Anwendung eines Traits zur Kapselung von Funktionalität

Wie wir in Listing 53 sehen, wird das Schlüsselwort „trait“ anstelle von „class“ verwendet, um den Trait zu erstellen. Weiter müssen alle abstrakten Member einen Rückgabewert angeben. In Zeile 19 wird eine Liste mit zwei Hunden erstellt und diese an die Nutzer-Klasse „ParkArea“ übergeben. Wird nun die Methode „listenForNoise“ der Nutzer-Klasse aufgerufen, erfolgt für jedes Element der Liste die Ausführung der Implementierung in „Dog“. Daraus lässt sich schließen, dass Kapselung mit Traits in Scala möglich ist. Die Anwendung dieser ist Analog zu Java, wie folgendes UML-Diagramm darstellt [Piepmeyer2010].

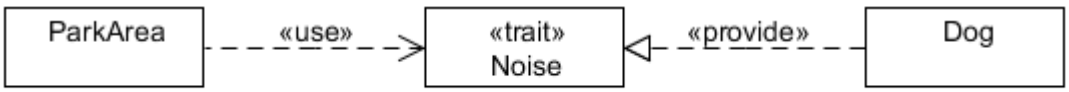


Abbildung 6 Kapselung mittels Traits anstelle eines Java-Interface

Doch Traits bieten weitaus mehr an, es ist nämlich möglich bereits bekannte Werte oder Methoden innerhalb eines Traits nicht nur zu deklarieren, sondern zu definieren. Einem Attribut wird dabei ein Wert zugewiesen, während eine Methode implementiert werden kann. Damit erweitert sich die Mächtigkeit eines Traits, auf die einer abstrakten Klasse in Java. Identisch zu dieser kann auch von einem Trait keine direkte Instanz erstellt werden [Piepmeyer2010].

```
1 //definition
2 trait Noise{
3       val count = 1
4       val noise:String
```

```
5         def makeNoise: Unit = for(i <- (1 to count)) print(noise)
6 }
7
8 class Dog(val name: String) extends Noise{
9         override val noise = " wau "
10 }
11
12 class Cat(val name: String) extends Noise{
13        override val noise = " miau "
14 }
15 //call
16 val animalList = List(new Dog("Beethoven"), new Cat("Garfield"))
17 val park = new ParkArea(animalList)
18 park.listenForNoise
19  wau  miau
```

Listing 54 Definierte Member innerhalb eines Traits

Die Member „count“ und „makeNoise“ wurden nun konkret im Trait implementiert, Zeile 3 und 5. Dies hat den Vorteil, dass hinzukommende Datentypen, wie „Cat“, welche die identische Implementierung der Methode „makeNoise“ benötigen, einfach mit dem Trait „Noise“ gemixt werden können. Traits sind also Werkzeuge, um Duplizierung von Quellcode zu vermeiden [Seeberger2011]. Natürlich wäre dies auch in Java mittels einer abstrakten Klasse möglich. Doch was passiert, wenn weitere Eigenschaften hinzukommen, wie beispielsweise Schwimmen?

```
1 //definition
2 trait Swimmer{
3         def swim = print(" plansch ")
4 }
5
6 class Dog(val name: String) extends Noise with Swimmer{
7         override val noise = " wau "
8 }
9
10 //call
11 val dog = new Dog("Bello")
12 dog.swim
13  plansch
14 dog.makeNoise
15  wau
```

Listing 55 Erweiterung einer Klasse um zwei Traits

Wie man in Zeile 6 erkennen kann, werden in die Klasse „Dog“ zwei Traits hineingemixt, „Noise“ und „Swimmer“. Dadurch sind die Member beider Traits nun Bestandteile von Dog, wie in den Zeilen 12 bis 15 sichtbar ist. Der zweite Trait wird durch das Schlüsselwort „with“

in die Klasse gemixt. Dabei gilt die Syntaxregel, dass die erste Klasse/ der erste Trait mit „extends“ gemixt, während die nachfolgende Klassen/Traits mit „with“ den Datentyp erweitern. Zur Anwendung des „Swimmer“ Traits in der Klasse „ParkArea“ fehlen noch ein paar Mechaniken aus der funktionalen Programmierung. Diese wird im Kapitel 2.2.2.4 angepasst.

Traits lassen sich auch anonym implementieren, indem ein neuer Typ erstellt wird, ohne dass dafür eine explizite Definition erfolgt. Dieses Konzept ist auch in Java bekannt. Betrachtet wird dafür die Klasse „Animal“ aus dem Kapitel 2.2.1.10, um den bisherigen Kontext zu wahren.

```
1 //definition
2 abstract class Animal(val name:String)
3
4 //call
5 val bello = new Animal("Bello") with Noise{
6         override val noise = "tap"
7 }
8 bello: Animal with Noise = $anon$1@1613fe7
```

Listing 56 Erstellung einer Anonyme-Klasse mittels eines Traits

Hinter der Konstante „bello“ steht nun eine Anonyme-Klasse, wie aus Listing 56, Zeile 5 zu entnehmen ist. Die „extends“ danach „with“ Syntaxregel gilt hier nicht. Mit dieser Methode ist es nun möglich Datentypen dynamisch zur Laufzeit um Eigenschaften zu erweitern [Piepmeyer2010]. In Java müsste dieses Verhalten zuerst mit dem Dekorator-Entwurfsmuster nachimplementiert werden [Eilebrecht2010].

Durch das Mixen von mehreren Traits in eine Klasse, entsteht ein Problem, welches in Java nicht auftaucht. Was geschieht, wenn beide Traits den gleichen Member haben, welcher wird ausgeführt? In Java kann eine Klasse zwar von mehreren Interfaces erben, aber die Implementierung erfolgt eben in der Klasse, welche die Schnittstellen implementiert. Dieses Problem der Mehrfachvererbung ist auch als Diamond-Problem bekannt und wird im nächsten Kapitel gelöst [Piepmeyer2010].

2.2.1.15 Mehrfachvererbung mit Traits

In den bisherigen Kapiteln sind immer wieder Situationen entstanden, welche auf Mehrfachvererbung in Scala hindeuten. Ob das nun die „Bottomtypes“ aus Kapitel 2.2.1.13 waren oder wie im vorhergehenden Kapitel die Member eines Datentypes von mehreren Traits erweitert werden konnten. Bevor das Prinzip erklärt wird, warum sich Scala trotz dieser Eigenschaften zu jedem Zeitpunkt deterministisch verhält, werden Traits und ihre Beziehung zur Basisklasse erläutert.

Alle bisher definierten Traits erbten implizit von der Basisklasse „scala.AnyRef“. Diese Beziehung ist der Standardfall solange ein Trait nicht explizit mittels „extends“ von einer anderen Basisklasse erbt [Seeberger2011].

```
1 class A
2 trait T extends A
3 class B extends T
```

Listing 57 Vererbung mit Traits

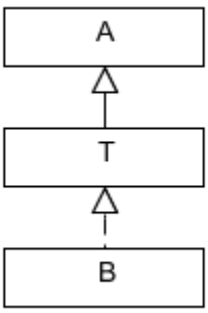


Abbildung 7 Vererbung mit Traits

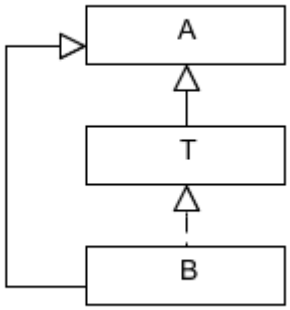
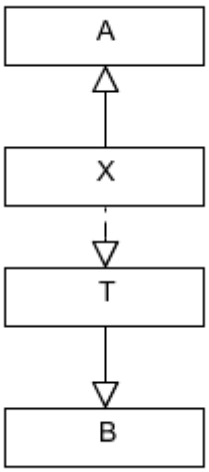


Abbildung 8 Folgen der Vererbung mit Traits

In Zeile 2 aus Listing 57 erbt Trait „T“ von der Basisklasse „A“. Darauf wird die Klasse „B“ von Trait „T“ erweitert. Nach bisherigem Stand bildet Abbildung 7 die Vererbungsbeziehung ab. An diesem Punkt übernimmt aber Klasse „B“, den Basistyp von „T“, als eigene Basisklasse, siehe Abbildung 8. Dieses Vorgehen ist nicht unproblematisch, da es passieren kann, dass Klasse und Trait zwei explizite Basistypen haben, die in keiner Vererbungsbeziehung zueinander stehen. Dies macht das folgende Listing und Diagramm ersichtlich.

```
1 class A
2 class B
3 trait T extends B
4 class X extends A with T
5 <console>:10: error: illegal inheritance; superclass A
6  is not a subclass of the superclass B
7  of the mixin trait T
8       class X extends A with T
9                        ^
```

Listing 58 Fehlende gemeinsame Basistypen



Listing 59 Unterschiedliche Basistypen

Im Regelfall, haben allerdings Klasse und Trait beide „scala.AnyRef“ als gemeinsamen Basistyp [Seeberger2011].

Als letzten Punkt der OOP wird nun das Hauptproblem der Mehrfachvererbung betrachtet. Dieses ist in Scala durch Traits umgesetzt, da von diesen mehrere in eine Klasse hinein gemixt werden können. Es stellt sich die Frage, wie Scala mit einer Klasse umgeht die zwei Traits hinein mixt, welche beide denselben konkreten Member haben. Gelöst wird das Problem durch die Erstellung einer linearen Reihenfolge (Linearisierung), entlang der Vererbungshierarchie. Dazu wird ein konkretes Beispiel betrachtet.

```
1 //definition
2 abstract class Animal{
3       val name: String
4       def makeNoise: Unit
5 }
6
7 trait Noise extends Animal{
8       val count = 5
9       val noise:String
10      def makeNoise: Unit = for(i <- (1 to count)) print(noise)
11 }
12
13 trait Swimmer{
14      def swim = print(" plansch ")
15 }
16
17 class Dog(override val name:String, override val noise:String) extends Animal with Swimmer with
```

```
Noise
18
19 //call
20 val beethoven = new Dog("Beethoven", " wuff ")
21 beethoven.makeNoise
22  wuff  wuff  wuff  wuff  wuff
```

Listing 60 Linearisierungsbeispiel

Wie der Ausgabe aus Zeile 22, Listing 60, entnommen werden kann wurde die Implementierung aus „Noise“ ausgeführt. Was passiert nun wenn mehrere Elemente den Member „makeNoise“ implementieren? Dazu werden nun folgende Änderungen vorgenommen.

```
1 //definition
2 abstract class Animal{
3         val name: String
4         def makeNoise: Unit = println("a very abstract noise")
5 }
6
7 class Dog(override val name:String, override val noise:String) extends Animal with Swimmer with
Noise{
8         override def makeNoise = println("Grrr... makes the dog")
9 }
10
11 //call
12 val cujo = new Dog("Cujo", " ROAR ")
13 cujo.makeNoise
14 Grrr... makes the dog
```

Listing 61 Erstellung eines Memberkonfliktes

In der Klasse „Dog“, als auch in ihrer Basisklasse „Animal“, wird die Methode „makeNoise“ nun konkret implementiert, siehe Listing 61, Zeile 4 und 8. Die letzte Zeile der Ausgabe verdeutlicht aber, dass die Klassen-Implementierung aus Zeile 8 den Zuschlag zur Ausführung bekommt. Dieser kann aus der Linearisierung der Vererbungshierarchie bestimmt werden. Für das bisherige Beispiel erstellt der Compiler folgende lineare Reihenfolge, anhand der Klasse „Dog“.

Als erstes wird die Basisklasse „Animal“ betrachtet, diese leitet von „scala.AnyRef“ ab, wobei ganz oben in der Hierarchie „scala.Any“ steht.

Animal > AnyRef > Animal

Weiter wurde der Trait „Swimmer“ hineingemixt, wobei existierende Typen unberücksichtigt werden.

Swimmer > Animal > AnyRef > Animal

Als nächstes folgt der Trait „Noise“, wobei dieser von „Animal“ erweitert wird, jedoch unberücksichtigt bleibt, da er bereits vorhanden ist.

Noise > Swimmer > Animal > AnyRef > Animal

Zum Schluss kommt die Klasse „Dog“ selbst. Die endgültige Reihenfolge sieht für die Klasse so aus:

Dog > Noise > Swimmer > Animal > AnyRef > Animal

Abbildung 9 zeigt die bestehenden Vererbungsbeziehungen. Die expliziten Vererbungsbeziehungen sind gekennzeichnet, alle anderen Beziehungen finden implizit statt.

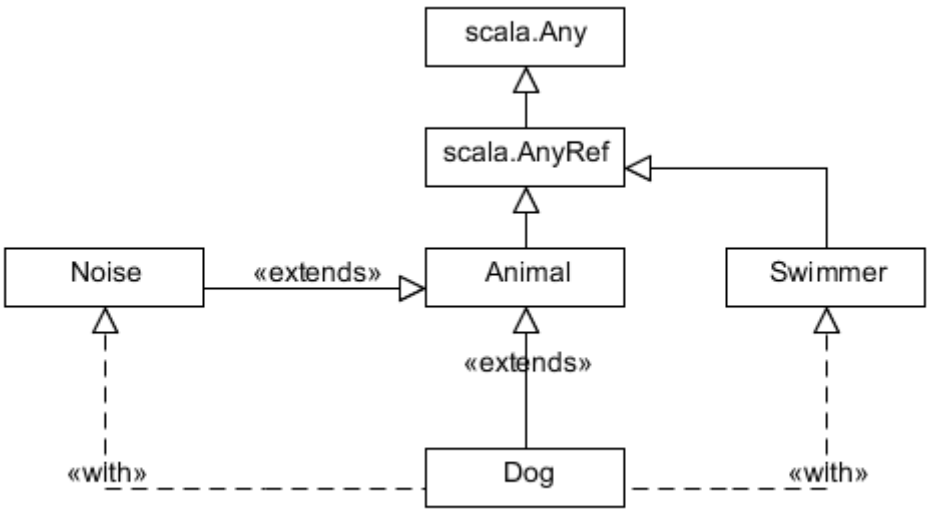


Abbildung 9 Vererbungshierarchie der Klasse „Dog“ aus Listing 60

Für den Einsatz der Mehrfachvererbung mittels Traits gilt demnach folgende Regel: "Je weiter rechts ein Trait beim Hinein-Mixen steht, desto vorrangiger ist er. Den höchsten Vorrang hat jedoch die Klasse selbst, in die hineingemixt wird." [Seeberger2011].

2.2.2 Funktionale Programmierung

Nachdem nun die wichtigsten Grundlagen für den Umgang mit Scala erörtert wurden, folgt eine kurze Definition der funktionalen Programmierung (kurz FP) und ihren Eigenschaften. Danach wendet sich der Autor dem Hauptaspekt dieses Paradigmas in Scala zu, der Funktion. Der Schluss dieses Kapitels konzentriert sich auf Konzepte, welche im Rahmen der Beispielanwendung und der Scala spezifischen Anwendungsfälle angewandt wurden.

2.2.2.1 Definition

Die Grundlagen einer jeden funktionalen Programmiersprache lauten:

1. Jedes Programm ist eine Funktion.
2. Jede Funktion kann weitere Funktionen aufrufen.
3. Funktionen werden wie Daten behandelt.

Dies führt zur Definition einer Funktion, welche folgende Merkmale aufweist. Eine Funktion f nimmt eine Liste von Werten für die Parametern p1,...,pn auf: f(p1,...,pn). Solche Werte werden weiterhin als Argumente bezeichnet. Für die Parameter besteht eine Menge an zulässigen Werten, welche der Typ des Parameters ist. Verschiedene Parameter können den gleichen Typ haben. Das Ergebnis einer Funktion, welcher aus den Parametern ermittelt wird, hat auch einen Typ, welcher Rückgabetyp genannt wird [Piepmeyer2010]. Dazu werden drei Java-Beispiele betrachtet.

```
1 //1
2 int a() = {
3         return 42
4 }
5 //2
6 double twice(int v){
7         return 2.0 * v;
8 }
9 //3
```

```
10 void displayValue(int v){
11        System.out.println("2*v: " + (2*v));
12 }
```

Listing 62 Funktionen und Prozedur

Die ersten zwei Beispiele entsprechen den aufgestellten Kriterien an eine Funktion, während am dritten Beispiel der Fehler bereits in der Signatur steckt. Diese enthält den Rückgabewert „void“, der bekanntlich für keinen Wert in Java steht.
Somit handelt es sich um eine Prozedur, welche wegen ihrer Wirkung aufgerufen wird, den Konsolenzustand durch eine Ausgabe zu verändern. Streng genommen handelt es sich damit um keine echte Funktion [Piepmeyer2010]. Dieses Verhalten wird auch Seiten- oder Nebeneffekt genannt, da die Prozedur wegen einer Zustandsänderung, also eines Effektes, angewendet wird. Ein weiteres Beispiel wäre die Manipulation einer globalen Variable innerhalb einer Prozedur [Klaeren2007]. Reine Funktionen hingegen verändern keine Zustände.

Betrachtet wird nun die zweite Grundlage. Funktionen können miteinander verknüpft werden, indem der Rückgabewert einer Funktion als Parameter einer anderen Funktion dient. Voraussetzung dafür ist, dass der Rückgabetyp zum Typ des Parameters der darauf folgenden Funktion passt.

```
1 //definition inside Math
2 public static long abs(long a) ...
3 public static double sqrt(double a) ...
4
5 //call
6 Math.sqrt((Math.abs(-25)))          /* != */          Math.abs((Math.sqrt(-25)))
```

Listing 63 Verknüpfung von Funktionen

Das Beispiel zeigt die Signaturen beider Funktionen, deren Rückgabewerte und Parametertyp zwar nicht identisch, aber miteinander vereinbar sind. Die Reihenfolge der Verknüpfung, deren Auswertung von Innen nach Außen stattfindet, ist vom Kontext abhängig und nicht willkürlich. So schlägt auf der rechten Seite die Ausführung fehl, da aus einer negativen Zahl keine Wurzel gezogen werden kann. Daraus folgt, dass eine Funktion aus Funktionsverknüpfungen und Aufrufen besteht [Piepmeyer2010]. Voraussetzung dafür

sind nicht-weiter-zerlegbare bzw. atomare Funktionen. Ein Beispiel dafür sind die Operatoren, wie „*“ und „+“, welche durch „add“ und „mult“ abgebildet werden können.

```
1 3*3+4*4                  /* == */          add(mult(3,3),mult(4,4)
```
Listing 64 Anwendung atomarer Funktionen

In imperativen Programmiersprachen bestehen die Anweisungen aus Folgen von Anweisungen. Hingegen in der funktionalen Programmierung werden diese Anweisungsfolgen durch geschachtelte Funktionen ersetzt. Werte verändern sich nicht, es gibt deshalb auch keine Variablen nur Konstanten. Diese Definition scheint im ersten Augenblick radikal. Konstrukte wie Schleifen, welche von Iteration zu Iteration auf einem unterschiedlichen Zustand arbeiten, sind nicht mehr möglich. Allerdings bestehen dazu Alternativen, wie Rekursion [Piepmeyer2010].

```
1 //imperative
2 int sum(int v){
3         int result = 0;
4         for(int i=v; i>0; i--){
5                 result += 1;
6         }
7         resturn result;
8 }
9
10 //functional
11 int sum(int v){
12        return v <= 0 ? 0 : v +sum( v-1 );
13 }
```
Listing 65 Imperative und rekursive Berechnung einer Summe

Die Rekursion ist mit der Kurzschreibweise für if-else-Konstrukte in Java implementiert. Wenn der Ausdruck „v <= 0“ den Wert „true“ ergibt, ist das Ergebnis „0“, ansonsten wird der else-Zweig mit der Anweisung „v +sum(v-1)“ ausgeführt. Es stellt sich nun die Frage, ob alle bisherigen Probleme, welche unter Java, C++ oder einer anderen imperativen Programmiersprache die Turing-vollständig ist, mittels der funktionalen Programmierung gelöst werden können? Die Antwort lautet ja, denn die Basis der FP liegt im Lambda-Kalkül, welches äquivalent zur Turing-Maschine ist [Piepmeyer2010]. Für die parallele Verarbeitung von Daten lösen sich damit sehr viele Probleme gleichzeitig, da es keine veränderbaren Werte gibt, entstehen keine Konflikte, um den Schreib- und Lesezugriff einer Variablen. Die

Ausführung der Problemlösung kann auf beliebig viele Kerne skaliert werden. Eine praktische Anwendung dieser Art findet in dem Kapitel 3.1.6.2.1.1 statt.

Aus den bisherigen Definitionen entsteht nun die Eigenschaft der referentiellen Transparenz. Diese besagt, dass für gleiche Argumente die Funktion immer das gleiche Ergebnis zurückgeben muss, auch bei wiederholter Anwendung. Der Zeitpunkt, also wann der Aufruf/ die Auswertung erfolgt, darf dabei keine Rolle spielen! Somit sind jegliche Art von Nebeneffekten ausgeschlossen. Die Folge für Objekte ist, dass diese keine veränderbaren Attribute und somit keinen Zustand abbilden [Piepmeyer2010]. Was folgt daraus für die Praxis? In Scala können Variablen als auch veränderbare Attribute in Objekten erstellt und neu zugewiesen werden. Demnach ist Scala auch keine „reine“ funktionale Programmiersprache. Das ist auch gut so, da für relative einfache Aufgaben, wie Ein- und Ausgabe, Konzepte wie Monaden angewendet werden müssten [Piepmeyer2010], welche den Umfang dieser Arbeit sprengen würden. Trotzdem bietet Scala die Möglichkeit, an sinnvollen Stellen den imperativen Stil zu verlassen und das Problem mit Funktionen höherer Ordnung anzugehen, was in Java nur begrenzt möglich ist, da Funktionen nicht direkt wie Daten behandelt werden können.

2.2.2.2 Funktionen

In Scala definieren Funktionen einen eigenen Typ, aus dem ein konkretes Objekt erstellt werden kann. Dafür wird als erstes das Funktionsliteral betrachtet. Das Literal bildet einen „scala.Int“ auf einem „java.lang.String“ ab, in der Kurzschreibweise: Int => String.

```
1 ( i: Int) => { "i + 1: " + ( i+1 ).toString }
```

Listing 66 Ein Funktionsliteral

Das Literal nimmt einen Parameter „i“ vom Typ „scala.Int“ entgegen und hat als Rückgabewert eine „java.lang.String“. Das „=>“ Zeichen leitet den Funktionskörper ein, der identisch zu den Methoden, ein Statement ohne Klammern, oder mehrere mit Klammern enthalten kann. Der letze Ausdruck ist der Rückgabewert.

Die Syntax eines Literals lautet demnach: Parameterliste, „=>“, Funktionskörper

Aus dem Funktionsliteral wird zur Laufzeit ein Funktionswert, dies ist vergleichbar mit dem Klasse/Objekt-Konzept der OOP [Odersky2010]. Das folgende Listing zeigt diesen Vorgang in drei unterschiedlichen Schreibweisen.

```
1 val f = ( i: Int) => { "i + 1: " + ( i+1 ).toString }
2 f: Int => java.lang.String = <function1>
3
4 /* == */
5
6 val f: Function1[Int,String] = ( i: Int) => { "i + 1: " + ( i+1 ).toString }
7 f: Int => String = <function1>
8
9 /* == */
10
11 val f: Int => String = i =>  "i + 1: " + ( i+1 ).toString
12 f: Int => String = <function1>
```

Listing 67 Erstelllung eines Funktionswertes (Objektes) aus einem Funktionsliteral

Listing 67 erstellt einen Funktionswert, welcher über den Konstanten Bezeichner „f“ angesprochen werden kann. Dabei handelt es sich in Scala um ein Objekt. Das Literal wird nämlich in eine Klasse kompiliert, in die, abhängig von der Parameteranzahl, Traits hineingemixt werden. Für diesen Zweck stellt die Scala-Bibliothek spezielle Traits bereits. Für eine Funktion, welche keinen Parameter entgegen nimmt, existiert der Typ „Function0“, dies geht weiter bis „Function22“, was der maximalen Parameteranzahl entspricht. Die zugehörige Abbildung für die Funktion mit maximaler Parameteranzahl ist: (P1,...,P22) => R, wobei P für Parameter- und R für Rückgabetyp steht [Piepmeyer2010].

```
1 //definition
2 val f: Function1[Int,String] = ( i: Int) => {
3         "i + 1: " + ( i+1 ).toString
4 }
5 f: Int => String = <function1>
6
7 val g: Function1[Int,Unit] = ( i: Int) => {
8         println("i + 1: " + ( i+1 ).toString)
9 }
10 g: Int => Unit = <function1>
11
12 //call
13 g(1,2)
14 res0: String = i + 1: 2 j + 2: 4
```

Listing 68 Definition von Funktionswerten mit expliziter Typangabe

Bei der Ausgabe von „f“ aus Listing 68, Zeile 5 wird ersichtlich, dass es sich um den identischen Funktionswert, wie in Listing 67, Zeile 2,7 und 12 handelt. „g“ zeigt die Definition einer Funktion mit einem Nebeneffekt auf der Konsole. Das Erstellen solcher ist demnach möglich. Die Scala-Bibliothek nimmt an sehr vielen Stellen Funktionen als Parameter entgegen. Beispielhaft sind dafür die Scala-Collections, welche alle eine „foreach“ Methode bereitstellen. Diese iteriert durch alle Elemente der Kollektion (engl.: Collection) und führt die übergebene Funktion auf jedem Element aus [Odersky2010].

```
1 val list = List(10,20,30)
2
3 list.foreach(g)
4 i + 1: 11
5 i + 1: 21
6 i + 1: 31
7
8 list map f
9 res2: List[String] = List(i + 1: 11, i + 1: 21, i + 1: 31)
```

Listing 69 Anwendung der Funktionen auf eine Collection

Neben der „foreach“ Methode wird auch die „map“ Funktion in Listing 69, Zeile 8 ausgeführt mit dem Unterschied, dass als Rückgabewert eine neue Collection mit den Ergebnissen der Funktion „f“, erstellt wurde.

Funktionsliterale können weiterhin mit dem Platzhalter „_“ versehen werden, um noch prägnanter zu sein. Man kann sich diesen als freistehende Lücke vorstellen, welche besetzt wird, wenn die Funktion tatsächlich aufgerufen wird [Odersky2010].

```
1 list.filter( _ > 10)                    /* == */        list.filter( x => x > 10)
2 res3: List[Int] = List(20, 30)
```

Listing 70 Anwendung des „_“ Platzhalters

Für den Platzhalter werden demnach Werte aus der Liste eingesetzt, um die Funktion auszuführen. Natürlich stehen hier noch zahlreiche weitere Methoden zur Verfügung. Aus Gründen des Umfanges wird aber das Thema Scala-Collections nicht näher vertieft [Odersky2010].

2.2.2.2.1 Lokale Funktionen

An dieser Stelle soll ein Design-Prinzip der FP vorgestellt werden. Probleme sollten in kleine Teile zerlegt werden, um die Lösung dann aus den Teilergebnissen zusammenzusetzen. In Java ist dieses Prinzip ebenso anwendbar. Mittels des „private“ Access Modifiers werden Teil-Methoden, welche nach außen nicht sichtbar sein sollen, verborgen. In Scala lässt sich dieses Prinzip wunderbar auf Methoden bzw. Funktionen anwenden [Odersky2010].

```
1 //definition
2 object Calc{
3
4       def cylinderVolume(radius: Double, height: Double): Double = {
5
6               val PI = scala.Math.Pi
7
8               def circleArea(r: Double) = square(r) * PI
9
10              circleArea(radius) * height
11      }
12
13      private def square(x:Double) = x * x
14 }
15
16 //call
17 Calc.
18 asInstanceOf    cylinderVolume   isInstanceOf    toString
19
20 Calc.cylinderVolume(2.0,3.0)
21 res1: Double = 37.69911184307752
```

Listing 71 Definition von lokalen Funktionen

Das Listing zeigt die Zylindervolumen-Berechnung, welche sich in Teilprobleme zerlegen lässt, für die einzelne Methoden innerhalb des Singleton-Objektes „Calc“, erstellt wurden. Die Formel dafür lautet:

$$V_{Zylinder} = r^2 \cdot PI \cdot h$$

Formel 2 Volumen eines Zylinders

Dabei wird der Radius in der Methode „square“ quadriert, die Grundfläche in „circleArea“ und das Volumen in Zeile 10 berechnet und zurück gegeben [Klaeren2007]. Wie aus Zeile 18 entnommen werden kann, beschränkt sich die Sichtbarkeit allein auf die Methode „cylinderVolume“. Funktionen, welche innerhalb von Funktionen definiert werden, verhalten sich unter diesem Aspekt wie lokale Variablen [Odersky2010].

Aus den bisherigen Erkenntnissen ist bekannt, dass Methoden ein Bestandteil von Objekten sind. Da Funktionen Objekte sind, können diese auch Methoden als ihre Member enthalten. Dazu wird die Methode „cylinderVolume“ mittels des „_“ Platzhalters, welcher im vorhergehenden Kapitel eingeführt wurde, in einen Funktionswert überführt, was ein Objekt ist [Piepmeyer2010]. An dieser Stelle ersetzt der Platzhalter die gesamte Parameterliste [Odersky2010].

```
1 val cv = Calc.cylinderVolume _
2 cv: (Double, Double) => Double = <function2>
3
4 scala> cv(2.0,3.0)
5 res2: Double = 37.69911184307752
```

Listing 72 Erstellung einer Funktion aus einer Methode

Damit enthält die Konstante „cv“ ein Objekt, das weitere Methoden zur Ermittlung seines Wertes verwendet. Dies ist die einfachste Möglichkeit, um aus einer Methode ein Objekt zu erstellen.

2.2.2.2.2 Closures

Bisherige Beispiele bezogen sich immer auf Parameter, welche direkt übergeben wurden. Das ist nicht zwingend, denn mit Closures können Variablen innerhalb des sichtbaren Bereiches referenziert werden.

```
1 //definition
2 var more = 1
3 val inc = (x: Int) => x + 1
4 val addMore = (x:Int) => x + more
5
6 //call
7 addMore(10)
8 res0: Int = 11
9
10 more = 1000
11 addMore(10)
12 res1: Int = 1010
```

Listing 73 Abgrenzung einer Funktion von einer Closure

Das Code-Listing zeigt die Definition zweier Funktionsliterale. Das Erste in Zeile 3 ist ein geschlossenes Funktionsliteral, da es bereits mit seiner Niederschrift in Quelltext konkret ist. Aus diesem Grund wird es auch „geschlossenes“ Funktionsliteral genannt. Anders hingegen

das zweite Funktionsliteral aus Zeile 4. Dessen Variable „more“ ist eine „freie“ Variable deren Kontext außerhalb des Funktionsliterals beeinflusst werden kann. Es handelt sich hier um einen „offenen“ Term, dessen Ergebnis von der Referenz „more“ abhängig ist. Zeile 10 und 12 demonstriert diesen Zusammenhang. Das Objekt, welches nun für das Funktionsliteral zur Laufzeit erstellt wird, nennt sich „Closure“ und leitet von dem Begriff „close“ ab, da es die freien Variablen mit einschließt. Scala erfasst dabei die Referenz einer freien Variable, nicht den Wert dahinter. Deshalb ist auch der inverse Zugriff, indem Fall schreibend, auf die freie Variable aus der Closure möglich, wie dem nachfolgenden Listing zu entnehmen ist [Odersky2010].

```
1 val list = List(1,1,1)
2 list.foreach( x =>  more = more + x)
3 more
4 res6: Int = 1003
```

Listing 74 Zuweisung einer freien Variable innerhalb einer Closure

Erstellt wird eine Liste mit drei Ganzzahlen. Auf der Liste wird die „foreach“ Methode ausgeführt, welche für jedes Element der Liste, das übergebene Closure ausführt. Dieses liest die freie Variable „more“ aus addiert, das aktuelle Listenelement hinzu und schreibt das Ergebnis wieder in „more“. Die Ausgabe der letzten Zeile bestätigt den schreibenden Zugriff auf die freie Variable.

2.2.2.2.3 Einsatz von Funktionen und Closures

Nachdem nun die Begriffe Funktion und Closure beschrieben wurden, stellt sich die Frage, was sich damit anstellen lässt. Zum Einstieg werden zwei Funktionen „f“ und „value42“ definiert und miteinander verknüpft.

```
1 val f = (x: Int) => { "Call funktion f(" + x + ")"}
2 f: Int => java.lang.String = <function1>
3
4 val value42 = ( p: Int => String ) => { p(42) }
5 value42: Int => String => String = <function1>
6
7 value42(f)              /* == */        value42( x => f(x) )
8 res19: String = Call funktion f(42)
```

Listing 75 Verknüpfung von Funktionen

Die Funktion „f“ aus Listing 75, bildet einen „scala.Int“ auf einem „java.lang.String“ ab, wie aus Zeile 2 zu entnehmen ist. „value42“ nimmt eine Funktion genau diesen Types entgegen, wie man anhand des Parametertyps von „p“ aus Zeile 4 erkennen kann, und ruft diesen Parameter mit der Konstante 42 auf. Die Verknüpfung beider Funktionen findet in Zeile 7 statt [Pollak2009].

Es stellt sich nun die Frage, welchen praktischen Nutzen dieses Vorgehen hat. Dazu wird ein Problem untersucht, welches auch in imperativen Sprachen vorkommt. Identische Algorithmen sollten nicht wiederholt definiert, sondern abstrahiert und wiederverwendet werden. Dies verkörpert das Programmierprinzip: „Don't repeat yourself“ (kurz DRY) [Piepmeyer2010]. Betrachtet wird dafür das Objekt „NumberMatcher“ aus folgendem Listing.

```
1 object NumberMatcher{
2
3       private val numbers = List(-5,-4,-3,-2,-1,0,1,2,3,4,5)
4
5       def positiveNumbers = {
6               for(number <- numbers
7                       if (number > 0)
8               )print(number + " ")
9       }
10
11      def evenNumbersInterval(start: Int, end: Int){
12              for(number <- numbers
13                      if (number >= start)
14                      if (number <= end)
15                      if (number % 2 == 0)
16              )print(number + " ")
17      }
18 }
```

Listing 76 Beispielobjekt mit abstrahierbarem Algorithmus

Dieses enthält eine Liste mit den ganzen Zahlen von -5 bis 5 und zwei Methoden, welche auf dieser Liste arbeiten. Beide enthalten eine for-Schleife, welche über die Elemente der Liste iteriert und die aktuelle Zahl ausgibt, sofern die Filter-Bedingungen in den if-Klauseln einen wahren Wert ergeben.

```
1 NumberMatcher.positiveNumbers
2 1 2 3 4 5
3 NumberMatcher.evenNumbersInterval(-3,3)
4 -2 0 2
```

Listing 77 Anwendung des Beispielobjektes

Dem Namen entsprechend verhalten sich die beiden Methoden, wie man der Ausgabe aus Listing 77 entnehmen kann. Die Unterschiede an den beiden Methoden sind ihre Namen und ihre Filter-Bedingungen, während der Rest identisch ist. An diesem Punkt setzt der erfahrene Programmierer an, da er sich bewusst ist, dass bei hinzukommenden Methoden dieser Art, eine Abstraktion wiederverwendet werden kann. Der Hauptvorteil dieses Ansatzes tritt erst bei Veränderungen der Programmlogik in Erscheinung. Wird nämlich eine Funktion/Methode wiederverwendet, erfolgt die Modifikation an zentraler Stelle.

```
1 object NumberMatcher{
2
3       private val numbers = List(-5,-4,-3,-2,-1,0,1,2,3,4,5)
4
5       private def numberMatching(matcher: Int => Boolean) = {
6               for(number <- numbers
7                       if matcher(number)      //generic
8               )print(number + " ")
9       }
10
11      def positiveNumbers{
12              val f = (n: Int) => { (n > 0) }
13              numberMatching(f)
14      }
15
16      def evenNumbersInterval(start: Int, end: Int){
17              val cl = (n: Int) => { (n % 2 == 0) && (n >= start) && (n <= end)}
18              numberMatching(cl)
19      }
20 }
```

Listing 78 Beispielobjekt mit abstrahiertem Algorithmus der for-Schleife

Wie Listing 78 zu entnehmen ist, wurde die Iteration und Ausgabe in eine für den Anwender des Objektes unzugängliche Methode „numberMatching“ abstrahiert. Die beiden bisherigen Methoden „positiveNumbers“ und „evenNumbersInterval“ rufen nun diese Abstraktion auf und übergeben dabei die Funktion „f“ und das Closure „cl“, welche beim Aufruf erstellt werden. Die Funktion „f“ ist kein Closure, da diese keine „freie“ Variable bei deren Erstellung einbindet. Im Falle von „cl“ sind die übergebenen Parameter „start“ und „end“ die freien Variablen [Odersky2010].

Natürlich lässt sich diese Art von generischem Algorithmus auch in einer objektorientierten Programmiersprache wie Java lösen. Dazu wird im Folgenden das Schablonenmethode-Entwurfsmuster (engl.: Template Method Pattern) auf das identische Problem angewendet [Eilebrecht2010]. Der generische Schritt aus Listing 78 in Zeile 7, wird in einer eigenen abstrakten Klasse ausgelagert, dessen Methode in einer Unterklasse überschrieben wird.

```
1 public abstract class NumberMatcher {
2
3    protected int start = 0;
4    protected int end = 0;
5    protected int[] numbers = {-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5};
6
7    protected abstract void matcher(int i);
8
9    private void numberMatching() {
10       for (int i = 0; i < numbers.length; i++) {
11          matcher(i);  //generic
12       }
13    }
14
15    public void positiveNumbers(){
16       numberMatching();
17    }
18
19    public void evenNumbersInterval(int start, int end){
20       this.start = start;
21       this.end = end;
22       numberMatching();
23    }
24 }
```

Listing 79 Abstrakte Klasse mit generischer Methode „matcher“ in Java

Listing 79 zeigt eine solche abstrakte Klasse, welche den dynamischen Teil ihres „numberMatching“ Algorithmus, in die abstrakte Methode „matcher“ auslagert.

```
1 public class EvenNumbersIntervalMatcher extends NumberMatcher{
2
3    @Override
4    protected void matcher(int i){
5       if ( (numbers[i] % 2 == 0) && (numbers[i] >= start) && (numbers[i] <= end)) {
6          System.out.print(numbers[i] + " ");
7       }
8    }
9 }
```

Listing 80 Implementierung der generischen Methode „matcher“ im Untertyp: „EvenNumbersIntervalMatcher“

Listing 80 zeigt einen Untertyp von „NumberMatcher“, welcher die „matcher“ Methode implementiert. Wird nun eine der „public“ Methoden aus dem Basistyp aufgerufen, erfolgt die Delegation des Kontrollflusses vom Basistyp an den Untertyp. In Zeile 3 aus Listing 81, wurde das identische Muster für den Untertyp „EvenNumbersIntervalMatcher“ angewendet.

```
1 //call
2 NumberMatcher pnm = new PositiveNumberMatcher();
3 NumberMatcher enm = new EvenNumbersIntervalMatcher();
4 NumberMatcher primes = new NumberMatcher() {
5         @Override
6         protected void matcher(int i) {
7                 if ((numbers[i] == 2)||(numbers[i]==3)||(numbers[i]==5)) {
8                         System.out.print(numbers[i] + " ");
9                 }
10        }
11 };
12
13 pnm.positiveNumbers();                  //1 2 3 4 5
14 enm.evenNumbersInterval(-3,3);         //-2 0 2
15 primes.positiveNumbers();              //2 3 5
16 primes.evenNumbersInterval(-3,3);      //2 3 5
```

Listing 81 Anwendung des „template method“ Entwurfsmuster

Aus Zeile 4 ist zu entnehmen, dass eine Implementierung der „matcher“ Methode auch innerhalb des aktuellen Kontextes möglich ist. In diesem Fall wird eine anonyme Klasse mit dem Basistyp „NumberMatcher“ erstellt und die „matcher“ Methode überschrieben. Dieses verkürzte Vorgehen ist allerdings nur zu empfehlen, wenn auf einen konkreten Datentyp verzichtet werden kann.

Der Vergleich des funktionalen und objektorientierten Ansatzes hat gezeigt, dass dieses Problem in Scala durch das funktionale Paradigma wesentlich eleganter gelöst werden kann. Für Scala spricht nicht nur die intuitivere Implementierung, sondern auch fachlich saubere Anwendung, wie den Zeilen 15 und 16 aus Listing 81 entnommen werden kann. Auf dem Objekt „primes“ werden zwei unterschiedliche Methoden aufgerufen, jedoch gilt für beide die gleiche Implementierung.

2.2.2.2.4 Currying und Kontrollstrukturen

Mit Methoden, welche Funktionen als Argumente entgegen nehmen können, lassen sich nun neue Kontrollstrukturen erstellen. Als Beispiel wird hier das „loan“ Entwurfsmuster

verdeutlicht, welches eingesetzt wird damit Ressourcen einheitlich gehandhabt und abschließend immer freigegeben werden. Dafür wird im Folgenden die Kontrollstruktur „using“ erstellt.

```
1 //definition
2 def using(file :File, op: PrintWriter => Unit){
3         val writer = new PrintWriter(file)
4         try{
5                 op(writer)
6         }finally{
7                 writer.close()
8         }
9 }
10
11 //call
12 using( new File("date.txt"), writer => writer.println(new java.util.Date) )
```

Listing 82 Erstellung einer Kontrollstruktur

Die definierte Methode nimmt eine Datei „file“ und eine Funktion, welche einen „PrintWriter“ auf „Unit“ abbildet, als Parameter entgegen. Die Anwendung dieser erfolgt durch den Aufruf der definierten Methode und der Übergabe passender Argumente [Odersky2010].

Nun wirkt diese Art von Aufruf, aus Listing 82 in Zeile 12, nicht besonders wie ein Bestandteil der Sprache. Dafür unterstützt Scala das Curry-Prinzip, bekannt durch Haskell Brooks Curry. Damit lässt sich eine Methode oder Funktion mit n Parametern in eine Methode oder Funktion mit n elementigen Parameterlisten transformieren [Piepmeyer2010]. Angewendet auf das vorhergehende Listing, folgt die gleiche Methode in Curry-Form.

```
1 def using(file :File)(op: PrintWriter => Unit){
2         val writer = new PrintWriter(file)
3         try{
4                 op(writer)
5         }finally{
6                 writer.close()
7         }
8 }
```

Listing 83 Erstellung einer Kontrollstruktur in Curry-Form

Einziger Unterschied ist, dass die Methode „using“ in der Curry-Form nun zwei Parameterlisten mit einem Parameter, anstatt einer Parameterliste mit zwei Parametern

enthält. Bei der Anwendung kann jetzt die letzte Parameterliste in geschweiften Klammern geschrieben werden, ohne dass am Kontext etwas verändert wurde [Odersky2010].

```
1 var file = new File("date.txt")
2
3 using(file){
4         writer => writer.println(new java.util.Date)
5 }
```

Listing 84 Anwendung von „using“ in Curry-Form

Ein Praxisbeispiel und dessen Anwendung wurde bereits in Kapitel 2.1.6.2 demonstriert.

```
1 def using[A <: {def close(): Unit}, B](param: A)(f: A => B): B =
2 try {
3         f(param)
4 } finally {
5         param.close()
6 }
```

Listing 85 Typsichere Definition des Schlüsselwortes „using“

Nachdem nun das Prinzip des „loan“ Entwurfsmuster erläutert wurde, soll noch der Rest der Methoden-Signatur beschrieben werden. Darin enthalten sind beispielsweise die Typparameter „A“ und „B“. Der zuletzt genannte, kann jeden Typen darstellen, wobei „A“ einer Einschränkung unterliegt. „A“ kann demnach jeder Typ sein, welcher die Methode „close“ beinhaltet. Dies nennt sich in Scala strukturelle Typisierung [Pollak2009]. Beide Konzepte, das der Typparameter, als auch der strukturellen Typisierung sind nicht mehr Gegenstand dieser Arbeit.

2.2.2.3 Implicits

In Kapitel 2.2.1.4 wurde die Klasse „Dog“ in Listing 30 vorgestellt welche die Methode „-:“ implementiert. Diese diente dem Zweck der Rechtsassoziativität des Operators, womit ein Ganzzahlentyp von einem Hund subtrahiert werden kann.

```
1 val bello = new Dog("Bello",7)
2 5 -: bello
3 res3: Int = 2
```

Listing 86 Demonstration der Rechtsassoziativität

Allerdings ist dies keine besonders elegante Lösung. Aus diesem Grund wird nach einer Alternative gesucht. Betrachtet werden dafür die folgenden Aufrufe.

```
1 1 -> 'a'
2 res4: (Int, Char) = (1,a)
3
4 "kacper" reverse
5 res5: String = repcak
```
Listing 87 Betrachtung rechtsassoziativer Aufrufe

Die erste Zeile erstellt ein Tupel mit den Datentypen „scala.Int“ und „scala.Char“. Der dafür verwendete Operator ist allerdings nicht, wie zu erwarten, in „scala.Int“ definiert. Identisch verhält es sich bei der „reverse“ Methode aus „java.lang.String“ in Zeile 4. Wie kann es sich hierbei um validen Code handeln? Wie bereits in Kapitel 2.2.1.1 erwähnt, steht das „scala.Predef“ Singleton-Objekt jederzeit bereit. Dieses erbt von der Basisklasse „LowPriorityImplicits“, welche folgende Methodendefinition bereitstellt:

```
1 implicit def wrapString(s: String): WrappedString = if (s ne null) new WrappedString(s) else null
```
Listing 88 Implizite Typumwandlung

Es handelt sich hierbei um eine Konvertierung von einem Datentyp „String“ in einen mächtigeren „WrappedString“. Dieser stellt die Methode „reverse“ aus Listing 87, Zeile 4 bereit [Piepmeyer2010]. Die Anwendung von Listing 88 findet nur statt, weil der Quelltext ohne sie ungültig wäre! Daraufhin durchsucht der Compiler den Sichtbarkeitsbereich (engl.: Scope) nach bestehenden Regeln zur Umwandlung, den impliziten Methoden-Definitionen. Findet er eine passende Umwandlung, erfolgt die Ausführung ohne Fehlermeldung, da von bewusster Anwendung ausgegangen wird. Voraussetzung dafür ist die Kennzeichnung der Konvertierungsmethode mit dem Schlüsselwort „implicit“. Die Konvertierungsmethode muss mit den passenden Datentypen definiert sein, in dem Fall ist „String“ der Ausgangsdatentyp und „WrappedString“ der Zieldatentyp. Abschließend muss der Sichtbarkeitsbereich der Konvertierungsmethode definiert werden. Dies ist durch die Anwendung von Listing 88 innerhalb es aktuellen „Scopes“ bzw. Codeblockes möglich. Die zweite Möglichkeit wäre die Sichtbarkeit durch Member eines importierten Objekts verfügbar zu machen, nämlich als Begleit-Objekt des Ausgangs- oder Zieldatentypen [Seeberger2011]. Mit diesen Kenntnissen wird nun das Problem der Rechtsassoziativität auf den Typ „Dog“ gelöst.

```
1 class Dog(val name: String, var age: Int){
2
3         def minus(that: Dog): Int = this.age - that.age
```

```
4
5         def -(min: Int): Int = this.age - min
6
7         def -(that: Dog): Int = this.age - that.age
8 }
9
10 object Dog{
11        implicit def intToDog(i: Int) = new Dog("defaultName", i)
12 }
13
14 //call
15 val beethoven = new Dog("Beethoven",10)
16
17 12 - beethoven
18 res8: Int = 2
19
20 beethoven - 2
21 res9: Int = 8
```

Listing 89 Definition des „Implicits“ im Begleit-Objekt der Klasse „Dog“

Die „-:“ Methode wird in der Klasse „Dog“ gegen eine Überladung der Mehthode „-“ ausgetauscht. Damit kann ein Hund von einem anderen und anhand des Alters subtrahiert werden. Die Konvertierungsmethode „intToDog“ innerhalb des Begleit-Objektes, wandelt bei Bedarf einen „scala.Int“-Wert in ein Hunde-Objekt um, auf dem die hinzugekommene Methode „-“ angewendet wird. Abschließend ist in der Zeile 17 und 20 aus Listing 89 zu erkennen, dass die Methode sowohl links- als auch rechtsassoziativ funktioniert.

Weiterhin anzumerken ist, dass „Implicits“ nicht transitiv angewendet werden. Eine Konvertierung von A nach B, B nach C, was zu A nach C führen würde, findet aus Gründen der Einfachheit nicht statt. Eine Mehrfach-Definition von „Implicits“ bzw. von Methoden im Scope deren Signatur identisch zu vorhandenen „Implicits“ ist, führt zu Fehlern beim Übersetzen [Odersky2010].

„Implicits“ sind ein extrem mächtiges Werkzeug in Scala, die eine funktionale Alternative zum expliziten Casten mit „asInstanceOf[T]“ bereitstellen. Die Sprache selbst lässt sich mit diesem Konzept nach den eigenen Vorstellungen umgestalten. Aus Sicht des Autors sollte dieses Konzept nur dort eingesetzt werden, wo es auch wirklich Sinn macht. Effektiv kommt für den Entwickler eine weitere Dimension hinzu, welche er bei der Erstellung, vor allem aber bei der Einvernehmung von Fremdquelltext berücksichtigen muss. Somit steigt der Dokumentationsaufwand und Einarbeitungsaufwand.

2.2.2.4 Mustererkennung

Die Verarbeitung von Mustern (engl.: Pattern Matching) nutzt das Konzept der bedingten Verzweigung [Piepmeyer2010]. Dies kann zu sehr ausufernden „if/else“ Konstrukten führen, welche in Java mittels der „switch“ und „case“ Anweisung reduziert werden sollen. Diese war in Java 6 auf primitive Datentypen, ihre Wrapper-Pendants und Enums beschränkt. Seit Java 7 können auch Zeichenketten („java.lang.String“) innerhalb des „switch“ Konstruktes unterschieden werden [Eisele2011a]. Scala stellt dafür das Schlüsselwort „match“ zur Verfügung, welches wesentlich vielseitiger eingesetzt werden kann.

Eine häufige Aufgabe ist es einen Text, der in unserem Beispiel „42+21“ eine einfache Rechenaufgabe enthält, in seine Grundbestandteile zu zerlegen und entsprechend dem Element ('42', '+', '21'), eine Aktion auszuführen. Wie die Zerlegung/Parsen von Zeichenketten in Scala aussehen kann, wird in Kapitel 3.1.6.2.1.3 erklärt. Für das Beispiel reicht die Zuordnung des Elementes zur Operation [Piepmeyer2010].

```
1 def compute(operator: Char, op1: Int,  op2: Int){
2         operator match{
3                 case '+' => op1 + op2
4                 case '-' => op1 - op2
5                 case '*' => op1 * op2
6                 case '/' | ':' => op1 / op2
7                 case _   =>  println("not defined")
8         }
9 }
10
11 compute('%',42,21)
12 not defined
```

Listing 90 Mustererkennung anhand von Konstanten

Verglichen wird der „operator“ Parameter mit den Zeichen-Konstanten der Grundrechenarten. Stimmt der Vergleich überein, wird die entsprechende Rechenoperation ausgeführt. Zeile 11 aus Listing 90 wendet die Methode falsch an, indem ein Zeichen als Operator übergeben wird, welches nicht explizit definiert ist. In diesem Fall tritt die Aktion hinter dem Standardplatzhalter „_“ ein, welcher hier dem Schlüsselwort „default“ in Java entspricht. Wichtig bei dessen Verwendung ist, dass er am Ende als letzter Fall definiert wird, da er sonst auf alle Argumente des „operator“ Parameters zutrifft. In Zeile 6 ist eine ODER-Operator definiert, welche hinter den Zeichen „/“ und „:“ die Teilungsoperation ausführt

[Piepmeyer2010]. Diese Art von Konstanten-Matching wurde auch innerhalb der Beispielanwendung, zur Zuweisung von AHP-Werten zu Textinterpretation (siehe Kapitel 3.1.4.2.3) und umgekehrt, in der Klasse „scalaProject.serviceFacade.compareValue.**AhpCompareValueMatcher**“ angewendet.

Dies entsprach nun weitestgehend der Java-Mächtigkeit von „switch“. Das nächste Beispiel zeigt die Anwendung auf Datentypen. Es bezieht sich auf die Fachlichkeit aus dem Trait-Kapitel 2.2.1.14. Definiert wurden dort mehrere Datentypen („Noise“, „Swimmer“, „Cat“, „Dog“) mit unterschiedlichen Eigenschaften. Beide Tierarten konnten Geräusche von sich geben, da sie die „makeNoise“ Methode aus dem Trait „Noise“ erbten. Naturgemäß sind Hunde affin gegenüber Wasser eingestellt und erben aus dem Trait „Swimmer“ die „swim“ Methode. Katzen hingegen nicht. Zur Anwendung kommen beide Datentypen in der „ParkArea“, deren Implementierung mittels der Mustererkennung erfolgt.

```
1 class ParkArea(val list:List[AnyRef]){
2
3       private def matchNoiseMaker(animal: AnyRef) = animal match{
4               case x: Noise    => x.makeNoise
5               case _ => print(" noNoise ")
6       }
7
8       private def matchSwimmer(animal: AnyRef) = animal match{
9               case x: Swimmer => x.swim
10              case _ => print(" noSwimmer ")
11      }
12
13      def listenForNoise = list.foreach( animal => {
14              matchNoiseMaker(animal)
15              matchSwimmer(animal)
16      })
17 }
```

Listing 91 Mustererkennung anhand des Objekttypes

An die Klasse „ParkArea“ kann eine Liste mit beliebigen Objekten übergeben werden (Grund dafür ist „scala.AnyRef“ als Typparameter für „list“). Innerhalb der Klasse ist die einzige öffentliche Methode „listenForNoise“, welche zwei weitere Methoden für jedes Objekt in der Liste aufruft. Betrachtet wird eine dieser Methoden: „matchSwimmer“.

Diese vergleicht den Typ des übergebenen Objekts mit dem Datentyp der lokalen „case“-Variable „x“. Stimmen die Typen überein, wird dessen Member „swim“ ausgeführt. Dazu

muss das Objekt um den Trait „Swimmer“ erweitert worden sein. Ist dies nicht der Fall erfolgt die Ausgabe „noSwimmer“. Analog dazu verhält sich die zweite Methode mit Bezug auf den Trait „Noise“. Es folgt die Anlage einer Liste mit einem Hund und einer Katze, und abschließender Ausführung der „listenForNoise“ Methode.

```
1 val animalList: List[AnyRef] = List(new Dog("Beethoven"), new Cat("Garfield"))
2 val park = new ParkArea(animalList)
3 park.listenForNoise
4  wau  plansch  miau  noSwimmer
```

Listing 92 Anwendung der Mustererkennung mit unterschiedlichen Objekttypen

Wie an der Ausgabe zu erkennen ist, verhält sich die Implementierung wie beschrieben. Eine feinere Granularität kann durch „Guards“ erreicht werden. Diese folgen auf das Muster nach „case“, beginnend mit „if“ und erwarten einen booleschen Ausdruck. Ergibt die Auswertung dessen „true“, wird die zugeordnete Anweisung ausgeführt. Ein Beispiel wäre dafür die Überprüfung eines Hunde- oder Katzennamens [Odersky2010].

```
1 //definition
2 def listenForNoise = list.foreach( animal => {
3         matchNoiseMaker(animal)
4         matchSwimmer(animal)
5
6         animal match{
7                 case x: Dog if ( x.name == "Beethoven" ) => println("Beethoven is in the park!")
8                 case _ => println("Unknown is in the park!")
9         }
10 })
11
12 //call
13 park.listenForNoise
14  wau  plansch Beethoven is in the park!
15  miau  noSwimmer Unknown is in the park!
```

Listing 93 Anwendung der Mustererkennung mit Guards

Dafür wurde die Implementierung der „listenForNoise“ um einen Guard auf das Attribut „name“ ausgebaut, wie aus Listing 93, Zeile 7 zu erkennen ist. Trifft nun die Zeichenkette „Beethoven“ auf einen Hundenamen zu, wird dessen Anwesenheit im Park kundgetan.

Als Letztes wird das Zusammenspiel von Pattern Matching und Funktionen betrachtet. Die bisherigen vorgestellten Funktionen waren total. Sie waren für jedes übergebene Argument gültig, auch wenn bei der Verarbeitung Ausnahmen (engl.: Exceptions) möglich waren. Bei

Funktionen mit nur einem Parameter kann auch mit partiellen Funktionen gearbeitet werden. Diese schließen bestimmte Werte aus dem verwendeten Datentyp aus.

```
1 //definition
2 def reciprocal: PartialFunction[Double,Double] = {
3         case x  if(x!= 0) => 1/x
4 }
5
6 //call
7 reciprocal(2)
8 res7: Double = 0.5
9
10 reciprocal(0)
11 scala.MatchError: 0.0 (of class java.lang.Double)
```

Listing 94 Mustererkennung mit Funktionen

Die Kehrwertfunktion „reciprocal“ nimmt einen „scala.Double“ Wert auf und bildet diesen auf einem „scala.Double“ ab. Mit dem Guard „if(x!= 0)“ wird geprüft, ob das übergebene Argument zulässig ist. Da kein „_“ Platzhalter definiert wurde, tritt bei ungültigen Werten eine „MatchError“ Exception auf, siehe Listing 94, Zeile 11. Aus diesem Grund, ist die Funktion vom Typ „PartialFunction“. Dieser Trait definiert die Methode „isDefinedAt“, mit welcher zur Laufzeit getestet werden kann, ob die Funktion einen Wert innerhalb ihrer Domäne erlaubt. Den Einsatz verdeutlicht das folgende Listing.

```
1 reciprocal.isDefinedAt(0)
2 res9: Boolean = false
```

Listing 95 Anwendung einer Funktion mit Mustererkennung

Weitere Typen von Mustern existieren für Tupel, Listen, Variablen und Case-Klasse auf die aber nicht eingegangen wird.

3 Scala in der Praxis

3.1 Beispielanwendung

3.1.1 Problemstellung

Gleich zu Beginn dieser Arbeit stand die Auswahl einer geeigneten Entwicklungsumgebung, IDE (engl.: Integrated Development Environment), mit der die Beispielanwendung umgesetzt werden sollte. Das Hauptaugenmerk richtete sich dabei auf die drei populärsten Java-IDE's: Eclipse, IntelliJ IDEA und Netbeans (siehe Abbildung 10).

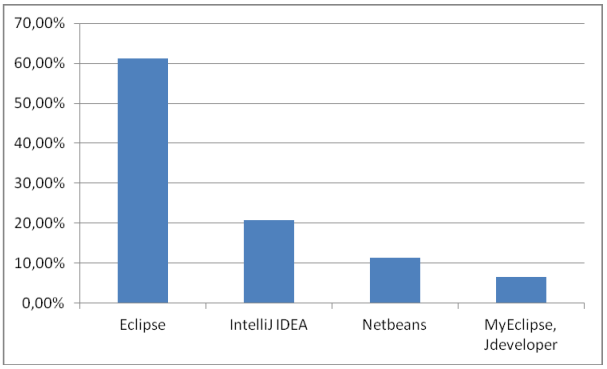


Abbildung 10 Verbreitung der Java-IDE's [Redaktion JAXenter2011a] (bearbeitet)

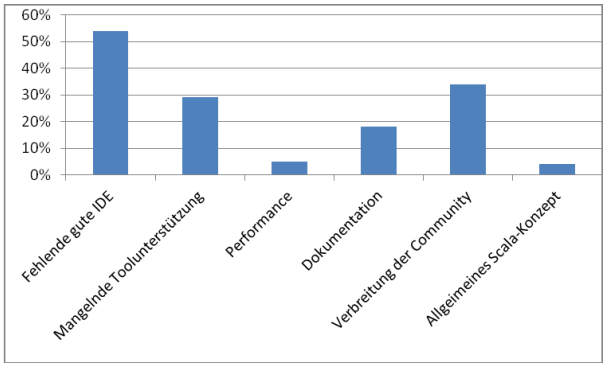


Abbildung 11 Schwächen von Scala [CamelCaseCon2011]

Alle aufgeführten Entwicklungsumgebungen haben ihren Ursprung in der Java-Programmiersprache. Daher wird die Kompatibilität für Scala durch das Hinzufügen einer Erweiterung (engl.: Plugin) hergestellt. Die Qualität des Plugins entscheidet darüber, wie gut Scala unter einer bestimmten Entwicklungsumgebung unterstützt wird. Da das Ziel dieser Arbeit nicht die alleinige Scala-Betrachtung ist, sondern auch eine praktische Evaluierung, in wie weit sich Scala mit etablierten Technologien vereinbaren lässt, besteht der Anspruch an die IDE's diese Technologien zu unterstützen. Das erste befriedigende Resultat, beinhaltete einen langen Try-And-Error Prozess. Bis ein Entwicklungstool gefunden wurde, das grundlegende Ansprüche erfüllte und sich dabei stabil verhielt. Diese Erkenntnis wird auch von der Umfrage gestützt, welche nach der Entwicklerveranstaltung „CamelCaseCon 2011" veröffentlicht wurde. An der Umfrage nahmen 153 Entwickler teil, von denen über 85%

Erfahrungen mit Scala gesammelt haben. Die Umfrage bestätigt die Problematik im Bereich der Entwicklungsumgebung und Toolunterstützung (siehe Abbildung 11).

Nachdem die Entscheidung für ein Werkzeug gefallen war, stand auch fest, dass zukünftige Releases des Scala-Plugins auf jeden Fall besser werden müssen. Die Entscheidung für eine IDE hängt jedoch von sehr vielen Kriterien ab, die unterschiedlich gewichtet werden. Um eine rationale Entscheidung für eine von mehreren Alternativen zu fällen, bedarf es einer Methodik, welche gewichtete Kriterien berücksichtigt und als Resultat eine Empfehlung für eine Alternative ausgibt. Diese Merkmale erfüllt der „Analytische Hierarchieprozess“ (AHP - engl.: Analytic Hierarchy Process).

3.1.2 Vorgaben

Ein Großteil der PENTASYS AG – Projekte basieren auf Internettechnologien, bzw. Client-Server-Lösungen. In diesem Sinne ist eine praktische Umsetzung mittels dieser Technologien von Interesse. Gefordert wird der Einsatz eines beliebigen Java-Webframeworks, die Persistierung der Ein- und. Ausgabedaten anhand einer Datenbank sowie das Einbinden der Scala-Technologie in eine 3-Schichtarchitektur. Diese Konstellation ist sehr stark verbreitet und deckt den Großteil heutiger EDV-Lösungen, auch außerhalb der PENTASYS AG, ab. Für die Konfiguration dieses Projektes, wird das Erstellungswerkzeug Maven verwendet, dass sich um die Verwaltung der Abhängigkeiten kümmert und alle Schritte im Entwicklungslebenszyklus unterstützt. Als Testtool fällt die Wahl auf JUnit, da es sich um das am meisten verbreitete Java-Framework zur Testrealisierung handelt.

3.1.3 Ziele

3.1.3.1 Umsetzung einer modularen Webanwendung

Aus den Vorgaben heraus ergibt sich folgende Aufteilung der Anwendung:

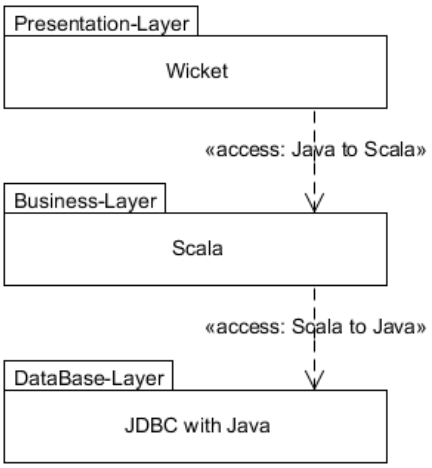


Abbildung 12 Schichtenarchitektur der Beispielanwendung

Jede Schicht greift auf die Dienste der darunterliegenden zu, bzw. reicht die Funktionalität nach oben weiter. Durch den Einsatz von Schnittstellen besteht eine konsistente Kapselung zu jeder Schicht. Die Umsetzung dieser Konstellation impliziert die Kompatibilität zwischen den eingesetzten Technologien, da diese voneinander abhängig sind. Der Einsatz von Scala in der mittleren Schicht wird bewusst angewendet, da deren Aufgaben sowohl Kommunikation zwischen der Benutzeroberfläche und Datenbank, als auch das Ausführen von Berechnungen und Plausibilitätsprüfungen ist. Gelingt die Umsetzung dieser Struktur kann eine klare Aussage über die Kompatibilität von Scala in Verbindung mit den eingesetzten Technologien getroffen werden.

3.1.3.2 Scala spezifische Anwendungsfälle

Scala verspricht nicht nur bekannte Konstrukte aus der OOP-Welt eleganter auszudrücken, sondern durch die Vereinigung von FP und OOP Synergieeffekte zu bilden. Es sollen Anwendungsfälle gefunden werden, in denen Scala der Programmiersprache Java überlegen und somit vorzuziehen ist.

3.1.3.3 Java – Scala Vergleich

Der Vergleich beider Programmiersprachen findet in der Funktions- und Steuerungs-Schicht (engl.: Business-Layer) der Beispielanwendung statt. Eine Implementierung der genannten

Schicht erfolgt sowohl in Java als auch in der Programmiersprache Scala. Mit dieser Methodik wird ein direkter Vergleich möglich. Es folgen Kriterien anhand derer beide Sprachen verglichen werden.

3.1.3.3.1 Quelltextzeilen

Die Anzahl der effektiven Quelltextzeilen (LOC - engl.: Lines of Code) gehört zu den Umfangsmetriken der Softwareanalyse. Diese wird häufig zur Quantifizierung der Modulgröße und Produktpreisbildung herangezogen. So steht die Menge der effektiven Codezeilen im direkten Verhältnis zur Produktivität. Sie ist stark vom Programmierstil und der verwendeten Programmiersprache abhängig und wird daher als angemessen für den Vergleich empfunden [Grechenig2010].

3.1.3.3.2 Kopplung zwischen Objekten

Die Kopplung zwischen Objekten (CBO - engl.: Coupling Between Objects), beziffert die Anzahl der Klassen, die mit der gemessenen Klasse verbunden sind. Je niedriger der Wert desto geringer sind die Abhängigkeiten zum gemessenen Objekt. Dies spielt eine wesentliche Rolle im gesamten Entwicklungszyklus. Das Maß der Kopplung reflektiert die Komplexität eines Objektes wider und hat direkten Einfluss auf die Testbarkeit und Wiederverwendbarkeit [Grechenig2010]. Im Sinne des KISS-Prinzipes („Keep it small and simple“) ist die Senkung von Abhängigkeiten oberstes Design-Ziel, um erweiterbare und wartbare Anwendungen herstellen zu können. Dies erreicht man in der Praxis durch eine Abstraktion von Problemstellungen und der Anwendung bekannter Problemlösungen mittels Entwurfsmustern.

Sowohl die CBO, als auch die LOC Metrik, werden mit dem Tool stan4j [Odysseus Software2011b] analysiert, das auf strukturelle Analysen von Java-basierten Projekten ausgelegt ist.

3.1.3.3.3 Performanz

Der AHP beinhaltet eine Matrixmultiplikation, mit der eine Problemgröße gut skaliert werden kann. Anhand einer steigenden Problemgröße wird die Zeit gemessen, wie lange

eine Java- und eine Scala-Implementierung für die Berechnung eines identischen Problems benötigt. Dieses Feature wird auch innerhalb der Beispielanwendung eingesetzt, um mögliche Leistungssteigerungen zu demonstrieren.

3.1.4 Lösungsansatz: Analytic Hierarchy Process - AHP

3.1.4.1 Beschreibung des AHP

Der „Analytische Hierarchieprozess“ ist eine Methode zur mehrkriteriellen Entscheidungsfindung. Er dient als Entscheidungshilfe bei Problemstellungen mit mehreren Zielen (Alternativen) mit beliebig tiefer Komplexität (Kriterien/Attribute der Alternativen). Somit ist er für Problemstellungen geeignet, bei denen die Intuition versagt, da durch zunehmende Details der Überblick für das Relevante verloren geht. Damit dieser Fall nicht eintritt, werden die Kriterien in einer hierarchischen Struktur abgebildet und untereinander über eine vordefinierte Bewertungsskala durch Paarvergleiche gewichtet. Nach der Bewertung der Alternativen zu jedem Kriterium kann eine quantitative Aussage über eine Alternative getroffen und eine Empfehlungsrangfolge erstellt werden [Weber1993]. Der AHP ist eine etablierte Methode bei der Entscheidungsfindung. Aufgrund des relativ hohen Aufwandes wird dieser nur bei komplexen Entscheidungen eingesetzt. Das Unternehmen Expert Choice offeriert eine DV-technische Unterstützung des AHP, welche die Anwendung des AHP um ein Vielfaches beschleunigt. Eine akademische Desktop-Demo-Version kann über die Internetseite des Unternehmens bezogen werden [EC2012]. Da die PENTASYS AG die Evaluierung von Scala in Bezug auf Internettechnologien fordert, ist die Überführung des AHP auf eine Client-Server-Architektur naheliegend. Damit steht in Zukunft ein Werkzeug bereit, das simultanes Bearbeiten eines AHP ermöglicht und das nicht unerhebliche Aufwand/Nutzen-Verhältnis verbessert [Lütters2008].

3.1.4.2 Anwendung des AHP auf die Problemstellung

Wie bereits in der Problemstellung erwähnt wurde, standen für die Implementierungen der Beispielanwendung mehrere Entwicklungsumgebungen zur Auswahl. Diese bilden die zur Verfügung stehenden Alternativen ab. Darunter fallen die genannten IDE’s IntelliJ IDEA (kurz

IntelliJ), Eclipse und Netbeans. Da sich große Unterschiede während der Umsetzung der Beispielanwendung zwischen den Alternativen ergeben haben, wurden Kriterien festgelegt, die mehr und minder wichtige Attribute der IDE's abbilden. Die gefundenen Attribute werden entsprechend ihrer Semantik in eine hierarchische Struktur gebracht. Die Gewichtung, ob ein Attribut einer Alternative mehr oder weniger relevant ist, erfolgt durch Paarvergleiche über die Elemente der Hierarchie. Ebenso müssen die Alternativen bezüglich jedes Kriteriums untereinander verglichen und priorisiert werden. Dabei wird die Gewichtung über eine quantitative Skala ermittelt. Die aus dem Paarvergleich entstandenen Teilgewichte werden hinterher auf Konsistenz geprüft um sicherzustellen, dass der getroffene Vergleich zumindest besser ist, als ein zufälliger Vergleich der Attribute. Sind alle Kriterien untereinander als auch alle Alternativen bezüglich der Kriterien erfolgreich verglichen worden, erfolgt die Aggregation der Teilgewichte zu globalen Gewichten. Diese bilden eine Rangfolge welche die Lösung für die hier beschriebene Problemstellung darstellt [Meixner2002]. An dieser Stelle endet die Implementierung des AHP, da dieser nicht Hauptgegenstand der Arbeit ist und das Ergebnis mit Hilfe der Expert Choice Software validiert wird.

3.1.4.2.1 Überblick des AHP Ablaufs

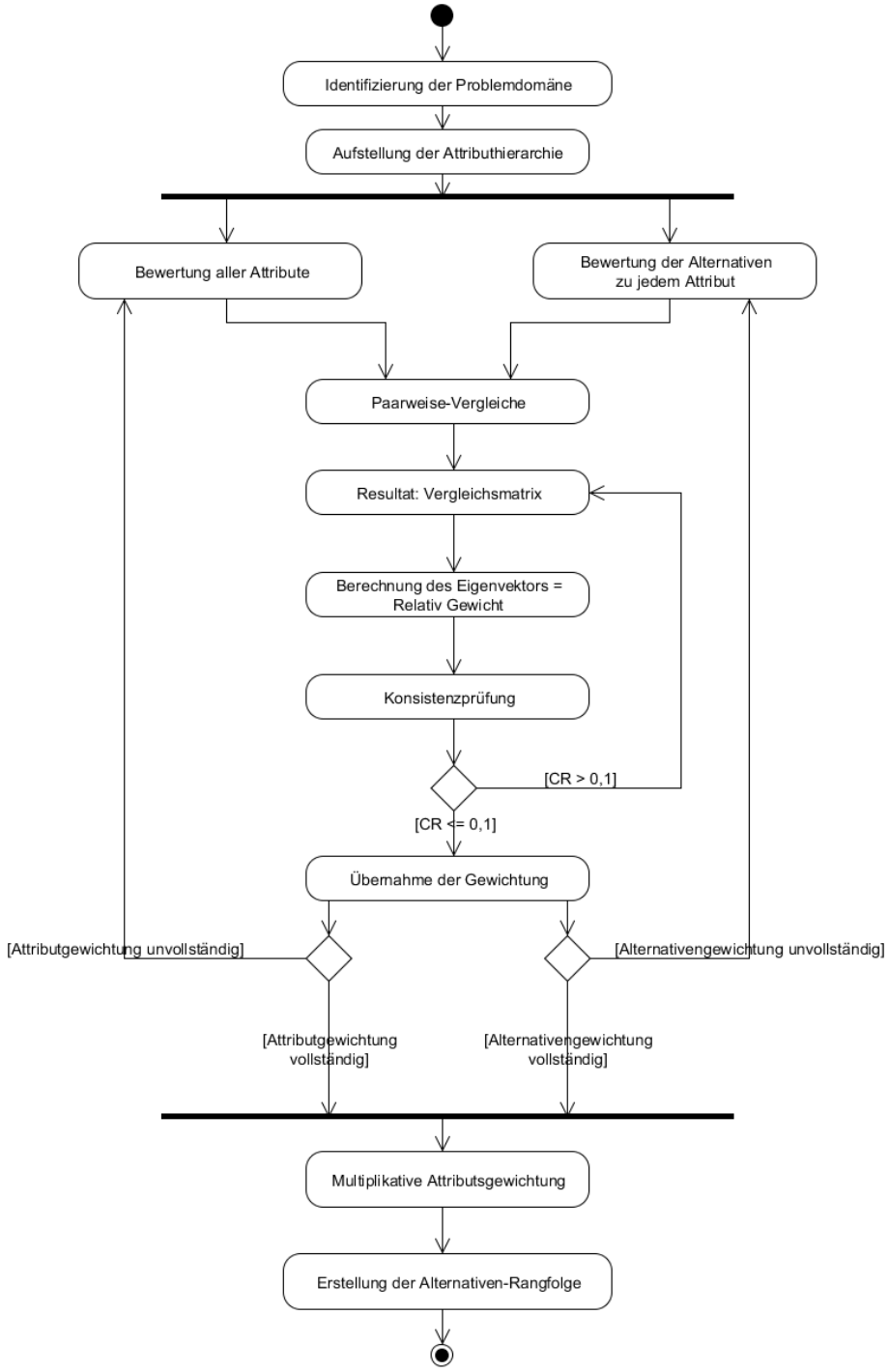


Abbildung 13 Modifizierter Ablauf des AHPs für eine Webanwendung

Abbildung 13 ist an das Ablaufschema des AHPs mittels Expert Choice angelehnt, deckt dieses im Umfang aber nicht vollständig ab [Meixner2002]. Der Ablauf wurde dahingehend modifiziert, dass die Bewertung der Kriterien untereinander und die Bewertung der

Alternativen bezüglich ihrer Attribute, unabhängig einer Reihenfolge stattfinden können. Da die Webanwendung von mehreren Clients gleichzeitig bedient werden kann, die gemeinsam einen AHP bearbeiten, ist diese Abweichung sinnvoll. Auf das sogenannte Synthesegewicht, also die finale Rangfolge der Alternativen, wirkt sich die Modifikation nicht aus. Dies wird gewährleistet indem davor geprüft wird, ob alle Elemente der AHP-Struktur gewichtet wurden.

3.1.4.2.2 Aufstellung der Elementhierarchie

Der erste Schritt ist die Problemdefinition. Es muss ein Oberziel gesetzt werden, in diesem Fall die „Beste Java/Scala IDE“ zu evaluieren. Dieses Ziel ergibt sich aus einem Angebot an Java-Entwicklungsumgebungen, die durch ein Plugin die Scala Unterstützung erweitern. Diese IDE’s bilden die Alternativen ab. Ausgehend vom Oberziel werden Kriterien/Attribute definiert, welche die Alternativen gemeinsam haben und von Relevanz für das Ergebnis sind. Die Attribute werden solange in Subattribute unterteilt, bis eine klare Aussage (besser, schlechter, gleich) über ein Merkmal zwischen zwei Alternativen getroffen werden kann [Meixner2002]. Ebene 1 bildet das oberste Ziel ab, während die darauf folgenden Ebenen die Attribute darstellen. Auf unterster Ebene befinden sich die zur Auswahl stehenden Alternativen.

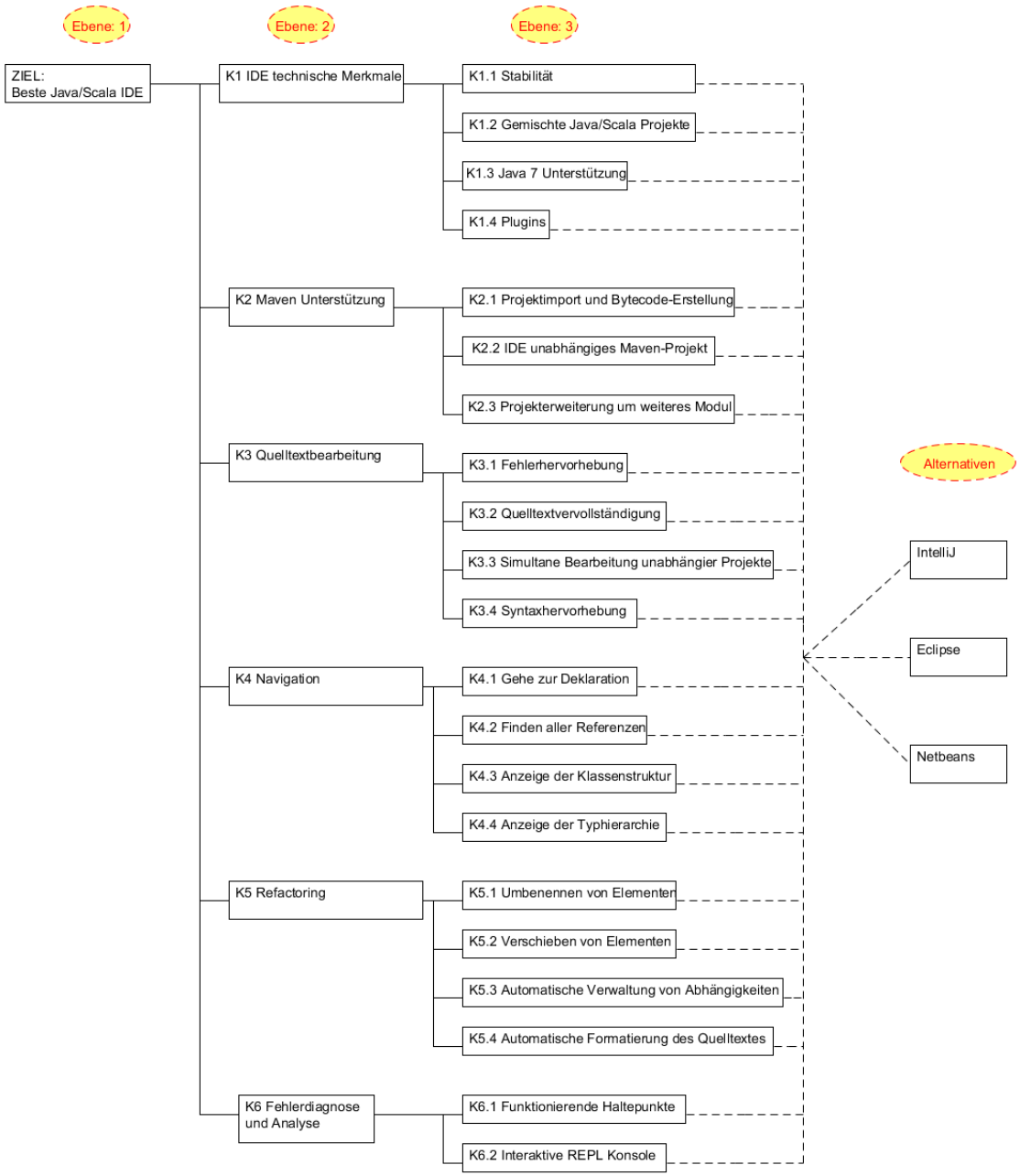


Abbildung 14 Elementhierarchie des AHPs: Beste Java/Scala IDE

3.1.4.2.3 Die AHP-Skala

Damit eine Aussage über die Qualität getroffen werden kann, muss eine geeignete Skala die Abstufungen definieren. Tabelle 1 überführt dabei eine qualitative Aussage in eine

quantitative, indem ein AHP-Wert dafür eingesetzt wird. Dadurch wird die qualitative Aussage in eine quantitative überführt.

Das Zustandekommen der Aussage erfolgt durch Gegenüberstellung von Elementen (A,B) die bezüglich ihrer Eigenschaften bewertet werden. Hierbei spricht man von einem Paarvergleich. A ist besser oder relevanter als B, wenn der AHP-Wert > 1 ist. Umgekehrt ist A schlechter oder weniger relevant als B, wenn der Kehrwert verwendet wird [Meixner2002].

AHP-Werte	AHP-Kehrwerte	Interpretation
1	1	A ist gleichbedeutend zu B
3	1/3	A ist etwas größer/relevanter zu B
5	1/5	A ist erheblich größer/relevanter zu B
7	1/7	A ist sehr viel größer/relevanter zu B
9	1/9	A ist größtmöglich dominierend über B
2,4,6,8	1/2, 1/4, 1/6, 1/8	Zwischenstufen

Tabelle 1 AHP Skala nach Thomas L. Saaty (bearbeitet) [Saaty2006]

3.1.4.2.4 Bewertung der Elementhierarchie

3.1.4.2.4.1 Prioritätenschätzung

Für den Vergleich mehrerer Elemente wird eine Matrix erstellt, die Paarvergleiche in Beziehung zu ihren Elementen abbildet. Ziel ist eine Ableitung von Gewichten, welche „die Wichtigkeit eines Elements im Verhältnis zu den Vergleichselementen“ [Meixner2002] widerspiegeln.

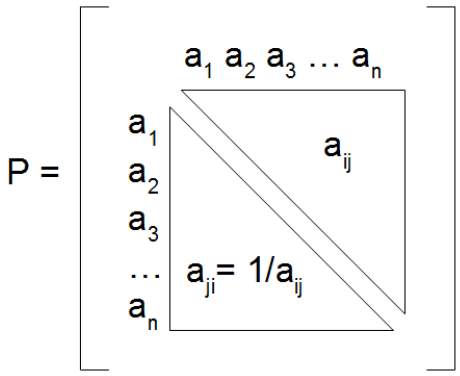


Abbildung 15 Allgemeiner Aufbau der Vergleichsmatrix

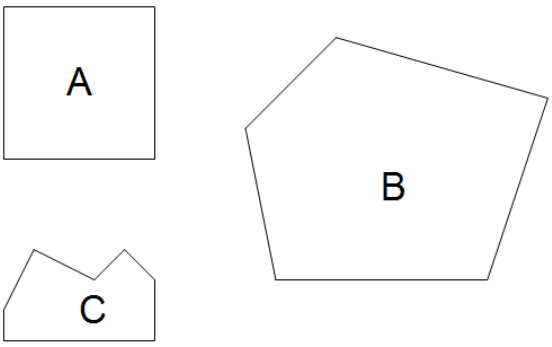


Abbildung 16 Beispielobjekte

a_n = Spalten / Zeilenelement (aus der Hierarchie)

a_{ij} = Paarvergleich

Abbildung 15 stellt den Aufbau einer solchen Matrix dar. Es genügt die Paarvergleiche zwischen den Elementen der a_{ij} Hälfte der Matrix vorzunehmen. Die gegenüberliegende Hälfte a_{ji} ergibt sich über das reziproke Axiom des AHPs [Meixner2002]. Die Diagonale a_{ii} der Matrix entspricht dem Wert 1, da das Element in dem Fall zu sich selbst verglichen wird.
Es folgt nun ein Beispiel für die Interpretation eines Paarvergleiches anhand von Abbildung 16. Dabei ist das Ziel, die größte Fläche durch den Vergleich von A, B und C zu evaluieren. Folgende Matrix ist das Resultat, wobei es sich bei den Paarvergleichen um Schätzungen und nicht um den exakten Flächeninhalt handelt. Die Schätzungen orientieren sich an der Abstufung der AHP-Skala aus Tabelle 1.

	Ziel: Größter Flächeninhalt	A	B	C
1.	A	1	1/5	2
2.	B		1	8
3.	C			1

Tabelle 2 Aufstellung der Evaluationsmatrix: P

Interpretation der 1. Zeile:

A ist im Vergleich zu A gleich.

A ist im Vergleich zu B erheblich kleiner.

A ist im Vergleich zu B etwas größer.

Interpretation der 2. Zeile:

B ist im Vergleich zu B gleich.

B ist im Vergleich zu C dominierend am größten.

Interpretation der 3. Zeile:

C ist im Vergleich zu C gleich.

Es fällt auf, dass aufgrund der AHP Gesetzmäßigkeit $a_{ij} = 1 / a_{ji}$ die übrigen Positionen der Matrix über die Bildung des Kehrwertes bestimmt werden können. Somit sinkt die Anzahl der Paarvergleiche in diesem Fall auf 3.

Allgemein formuliert beträgt die Anzahl der notwendigen Paarvergleiche = n * (n – 1) / 2 [Meixner2002].

Dies ergibt die aufgestellte Evaluationsmatrix mit abgeleiteten Kehrwerten:

Ziel: Größter Flächeninhalt	A	B	C
A	1	1/5	2
B	5	1	8
C	1/2	1/8	1

Tabelle 3 Vollständig verglichene Evaluationsmatrix: P

3.1.4.2.4.2 Ablauf der Prioritätenschätzung

Verglichen werden untereinander die Elemente innerhalb einer Hierarchieebene aus Abbildung 14. Wobei das Ziel für den Vergleich immer in der Ebene darüber liegt. Für Ebene 2 wird eine Vergleichsmatrix aus den Kriterien K1 bis K6 erstellt. Das übergeordnete Ziel

dieser Matrix ist unser Hauptziel: Die „Beste Java/Scala IDE“ auszuwählen (siehe). In Ebene 3 bilden sich die Vergleichsmatrizen aus den Subattributen, z. B.: K1.1 bis K1.4 mit dem Ziel K1 (siehe Tabelle 4). Analog werden die übrigen Matrizen der Ebene 3 gebildet. Die Resultate aus der Berechnung der Gewichte dieser Ebene nennt man: Lokale Attributgewichte. Angekommen auf der untersten Ebene werden nun alle Alternativen, pro Subattribut aus Ebene 3, miteinander paarweise verglichen (siehe Tabelle 11). Das Ziel in diesem Vergleich ist das Subattribut selbst. Die Ergebnisse aus der Berechnung dieser Ebene heißen lokale Alternativgewichte.
Sukzessiv kann mit dieser Methodik eine beliebig tiefe Elementhierarchie bewertet werden.

3.1.4.2.5 Berechnung der Gewichte

Nachdem eine Evaluationsmatrix mit Paarvergleichen gefüllt wurde, kann anhand dieser Daten die Priorität/Relevanz der enthaltenen Elemente berechnet werden. Die Grundlage schafft dafür das Theorem von T. L. Saaty. Dessen sinngemäße Aussage lautet: „Die aus einer positiven, reziproken n•n-Matrix abgeleiteten Prioritäten – die Zeilen der n•n-Matrix repräsentieren Paarvergleichs-Gewichte – entsprechen dem rechten Haupt-Eigenvektor“ [Meixner2002].

Bei der Berechnung der Elementgewichte handelt es sich um einen iterativen Prozess. Die erstellte Matrix wird dabei quadriert/potenziert und anschließend zur Basis = 1 normalisiert, um eine prozentuale Aussage über die Partialgewichte der Matrix zu treffen. Es folgen die notwendigen Rechenschritte.

1. Schritt: Quadrierung/Potenzierung der Matrix

P entspricht hier der Evaluationsmatrix aus Kapitel 3.1.4.2.4.1

$P * P = P^2$

$$\begin{bmatrix} 1 & 0{,}2 & 2 \\ 5 & 1 & 8 \\ 0{,}5 & 0{,}125 & 1 \end{bmatrix}^2 = \begin{bmatrix} 3 & 0{,}65 & 5{,}6 \\ 14 & 3 & 26 \\ 1{,}63 & 0{,}35 & 3 \end{bmatrix}$$

Formel 3 Quadrierung der Beispiel-Evaluationsmatrix P

2. Schritt: Normalisierung

Die Normalisierung erfolgt durch Summenbildung der Zeile und anschließender Division durch die Gesamtzeilensumme (Spaltensumme in Abbildung 17) [Saaty2008].

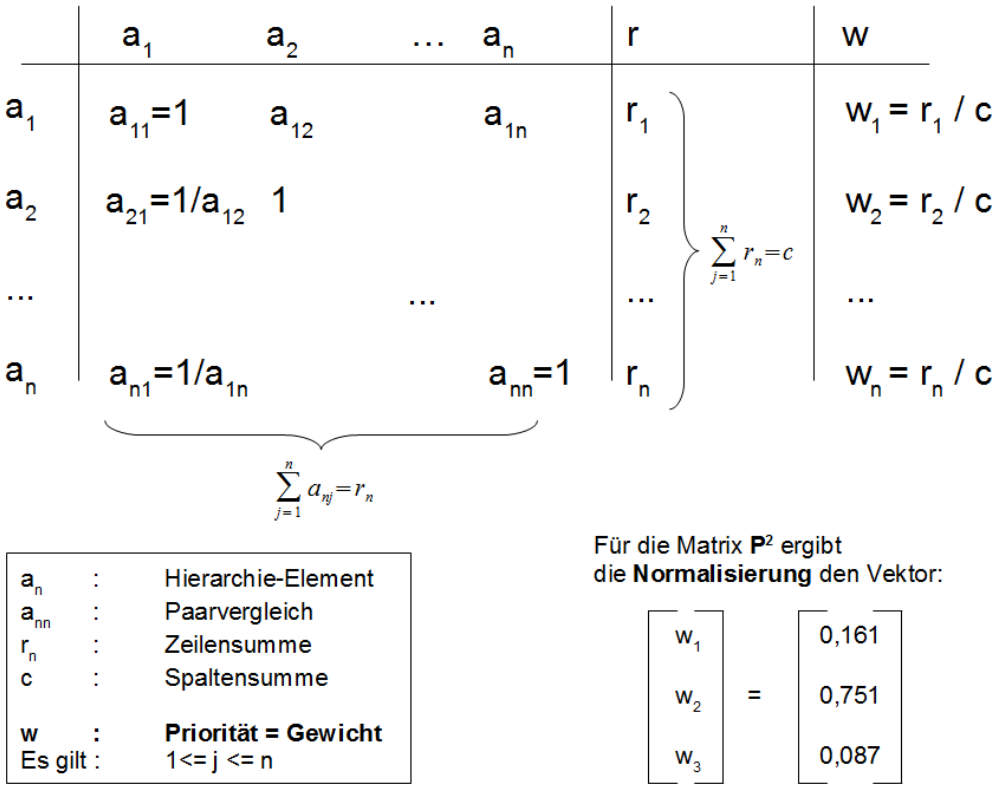


Abbildung 17 Allgemeine Darstellung der Normalisierung von Saaty

3. Schritt: Vergleich der Ergebnisse

Im ersten Schritt wurde die Matrix quadriert, was die Genauigkeit der Gewichte bereits stark erhöhte. In der nächsten Iteration sollte die Matrix mit einem größeren Exponenten als zwei potenziert werden. Das Ergebnis nach der Normalisierung wird eine geringere Differenz,

gegenüber den Gewichten aus der quadrierten Iteration aufweisen. Diese Differenz der Prioritäten nimmt mit zunehmender Potenz ab [Meixner2002].

Eine Nachstellung der Expert Choice Software Ergebnisse kommt zu folgendem Resultat: Für eine dreistellige Kommagenauigkeit der Prioritäten reicht die Potenzierung der Matrix bis zur zehnten Potenz oder weniger [Booz 2003]. Die Berechnung aller Prioritäten der IDE-Evaluation wurde mit der zehnten Potenz durchgeführt. Festgelegt wurde die Genauigkeit der Nachkommastellen auf drei.
In den folgenden zwei Kapiteln werden die Alternativen und deren Attribute beschrieben, miteinander paarweise verglichen und Resultate in Form von lokalen Gewichten erstellt.

3.1.4.2.6 Beschreibung und Gewichtung der Attribute / Kriterien

Da die Qualität des Ergebnisses maßgeblich von dem Wissen der wertenden Person über die Alternativen und deren Eigenschaften abhängt [Meixner2002], folgt eine Beschreibung der Attribute und Alternativen aus der ersichtlich wird, wie der Verfasser auf die Werte der Paarvergleiche kommt.

3.1.4.2.6.1 Hauptkriterien

K1 IDE technische Merkmale

Die technische Unterstützung bildet das Fundament, in wie weit sich eine IDE überhaupt einsetzen lässt und kann ein Schlüsselkriterium für den Einsatz sein. Auf dieses Kriterium fällt das höchste Gewicht.

K2 Maven Unterstützung

Das heterogene Projekt soll über das Projektmanagementwerkzeug Maven verwaltet werden. Getestet wurde, ob ein bestehendes Java/Scala Projekt importiert und direkt übersetzt werden kann. Weiter wurde überprüft, ob aus der IDE ein weiteres Modul dem Beispielprojekt hinzugefügt werden kann. Da der Einsatz von Maven als sehr hilfreich angesehen wird, bildet dieser den zweitwichtigsten Oberbegriff.

K3 Quelltextbearbeitung

Der Grad an Hilfestellung/Automatisierung, welcher dem Entwickler bei der Erstellung des

Programmecodes zur Verfügung steht, entscheidet maßgeblich über dessen Produktivität. Als typische Eigenschaft einer IDE nimmt die Quelltextbearbeitung das dritthöchste Gewicht ein.

K4, K5 Navigation und Refactoring

In umfangreichen Projekten, zu denen eine 3-Schicht-Anwendung zählt, ist die Navigation zwischen semantischen Elementen entscheidend für eine flüssige Entwicklung der Anwendung. Von gleichgewichtiger Notwendigkeit wird die Quelltextaufbereitung (engl.: Refactoring) angesehen. Diese ist ebenso von der Referenzierung der Elemente abhängig und wird meist im gleichen Kontext mit der Navigation, der Quelltextoptimierung, verwendet. Da es sich ebenso um typische Funktionalitäten einer IDE handelt, wird diese gleich mit der Quelltextbearbeitung gewichtet.

K6 Fehleranalyse und Behebung

Die Analyse und Behebung von Fehlern in Software stellt eine selbstverständliche Funktion jeder seriösen IDE dar. Die gegenwärtigen IDE's unterstützen die aus Java bekannten Funktionen auf der OOP Seite von Scala. Was die FP-Seite angeht, besteht noch Verbesserungsbedarf. Die Gewichtung ist mit den Kriterien K3 bis K5 identisch.

Aus dieser ausformulierten Gewichtung heraus, leitet sich bei Anwendung der von Saaty beschriebenen Skala, die Vergleichsmatrix für die Hauptkriterien ab. Die Anwendung des AHP auf die Vergleiche ergibt ein quantitatives Resultat (Eigenvektor). Wie dieses exakt zustande kommt, wurde im vorherigen Abschnitt erläutert.

Ziel: Beste Java/Scala IDE	K1	K2	K3	K4	K5	K6	Attribut-gewicht
K1	1	3	6	6	6	6	0,487
K2	0,333	1	3	3	3	3	0,212
K3	0,167	0,333	1	1	1	1	0,075
K4	0,167	0,333	1	1	1	1	0,075
K5	0,167	0,333	1	1	1	1	0,075
K6	0,167	0,333	1	1	1	1	0,075

Tabelle 4 Paarvergleichsmatrix mit dem Ziel: Beste Java/Scala IDE

3.1.4.2.6.2 K1 IDE technische Merkmale

K1.1 Stabilität

Absolut oberste Priorität hat die Stabilität der Entwicklungsumgebung. Der Entwickler darf niemals das Gefühl bekommen, dass Fehlverhalten auf die IDE und nicht auf die Inputdaten, also den Quelltext, zurückzuführen ist. Solch ein Verhalten ist im Praxiseinsatz nicht hinnehmbar und senkt die Produktivität massiv ab.

K1.2 Gemischte Java/Scala Projekte

Darunter ist die Aufteilung in Maven-Module zu verstehen, wobei innerhalb eines Modules keine Vermischung zwischen den Sprachen stattfindet. Da die Aufteilung in Module ein Kernansatz in größeren Projekten darstellt, kommt dieser das zweitgrößte Gewicht zu.

K1.3 Java 7 Unterstützung

Das JDK 7 ist relativ neu, Erscheinungsjahr: 2011, und findet noch relativ geringe Verbreitung. Die Features sind übersichtlich, selbst Oracle beschreibt es als „evolutionäres Release“ [Neumann2011]. Im Rahmen dieser Arbeit wird das darin enthaltene ForkJoin-Framework verwendet und ein Vergleich anhand dieses Features aufgezeigt. Deshalb und

aus einer zukunftsorientierten Betrachtung wird diesem Kriterium das dritthöchste Gewicht zugesprochen.

K1.4 Plugins

Bei Scala selbst handelt es sich um eine recht wenig verbreitete Sprache, deren Toolsupport erst heranreift. Die Zeitzyklen in denen die Scala-Plugins aktualisiert werden, sind von IDE zu IDE unterschiedlich. Bewertet wird hier die Komplexität des Installationsprozesses, welcher direkten Einfluss auf die Dauer zum Aufsetzen einer Produktivumgebung hat.

Ziel: K1	K1.1	K1.2	K1.3	K1.4	Attribut-gewicht
K1.1	1	2	3	4	0,467
K1.2	0,500	1	2	3	0,277
K1.3	0,333	0,500	1	2	0,160
K1.4	0,250	0,333	0,500	1	0,095

Tabelle 5 Paarvergleichsmatrix mit dem Ziel: K1 IDE technische Merkmale

3.1.4.2.6.3 K2 Maven Unterstützung

K2.1 Projektimport und Bytecode-Erstellung

Grundlegende Maven-Unterstützung belegt die Entwicklungsumgebung, indem sie bestehende Maven-Projekte importieren und verwalten kann. Weiter ist das Erstellen von Artefakten aus Quelltext heraus eine notwendige Bedingung. Diesem Kriterium wird das höchste Gewicht zugesprochen, da es die grundsätzliche Maven-Unterstützung einschließt.

K2.2 IDE unabhängiges Maven Projekt

Ob ein Maven Projekt von der IDE unterstützt wird, sollte nicht von IDE-spezifischen Inhalten innerhalb des Maven Projektes abhängen. Grundsätzlich gilt, ein Projekt, welches mit Maven in der Kommandozeile erstellt werden kann, muss ohne Zusätze oder Änderungen an den Maven Projektdateien, von der Entwicklungsumgebung unterstützt werden. Diese Eigenschaft, die IDE-Unabhängigkeit, ist sehr von Vorteil. Da jeder Entwicklungsrechner eine

unterschiedliche IDE, Plugin-Version oder Laufzeitumgebung aufweisen kann, bedeutet ein unabhängiges Maven Projekt vor allem Gewährleistung von Kompatibilität. Deshalb wird dieses Attribut als zweitgrößtes gewichtet.

K2.3 Projekterweiterung um weiteres Modul

Das Erweitern eines modularen Maven Projektes um ein weiteres Modul ist ein Anwendungsfall, welcher von der IDE unterstützt werden sollte. Diesem Kriterium wird allerdings das geringste Gewicht zugesprochen, da die Erweiterung auch in der Kommandozeile mit einem gewöhnlichen Texteditor stattfinden kann.

Ziel: K2	K2.1	K2.2	K2.3	Attribut-gewicht
K2.1	1	3	5	0,637
K2.2	0,333	1	3	0,258
K2.3	0,200	0,333	1	0,105

Tabelle 6 Paarvergleichsmatrix mit dem Ziel: K2 Maven Unterstützung

3.1.4.2.6.4 K3 Quelltextbearbeitung

K3.1 Fehlerhervorhebung

Als direkter Indikator, ob der geschriebene Quelltext sich übersetzen lässt, ist die korrekt funktionierende Fehlerhervorhebung von größter Bedeutung.

K3.2 Quelltextvervollständigung

Eine sehr nützliche Eigenschaft ist die automatische Vervollständigung bereits definierter Elemente wie Klassen, Methoden, Variablen usw. bei der Erstellung des Quellcodes. Weiter umfasst es die Anzeige der Member eines Elementes, wenn darauf zugegriffen wird. Somit erhöht dieses Attribut nicht nur die Produktivität und senkt die Anzahl von Tippfehlern, sondern kann auch als Ersatz für den schnellen Blick in die API einer Bibliothek genutzt werden. Es erfährt deshalb die zweitgrößte Gewichtung.

K3.3 Simultane Bearbeitung unterschiedlicher Projekte

Die gängige Praxis bei der Erarbeitung neuer Sprachen, Bibliotheken, Module ist das Erstellen eines Prototyps, der gefragte Eigenschaften hervorheben soll. Erfüllt der Prototyp die Erwartungen, wird er als Schablone in eine Produktivumgebung überführt. Da es sich bei Prototyp und Produktivumgebung um zwei voneinander unterschiedliche Projekte im differenzierten Kontext handelt, ist eine IDE, die mehrere Projekte gleichzeitig verwalten kann klar im Vorteil. Zwingend ist diese Unterstützung jedoch nicht, denn zur Not können als Schablone verfasste Quelltexte mit einem externen Editor genutzt werden. Aus diesem Grund wird dieses Kriterium als drittwichtigstes gewichtet.

K3.4 Syntaxhervorhebung

Die Syntax der Programmiersprachen Java und Scala werden farblich hervorgehoben. Diese Unterstützung hilft dem Entwickler bei der semantischen Unterscheidung zwischen Begriffen der verwendeten Sprache und dem fachlichen Kontext. Diese Art von Unterstützung ist zwar sehr nützlich, jedoch von geringster Bedeutung.

Ziel: K3	K3.1	K3.2	K3.3	K3.4	Attribut-gewicht
K3.1	1	3	5	8	0,573
K3.2	0,333	1	3	5	0,259
K3.3	0,200	0,333	1	3	0,116
K3.4	0,125	0,200	0,333	1	0,052

Tabelle 7 Paarvergleichsmatrix mit dem Ziel: K3 Quelltextbearbeitung

3.1.4.2.6.5 K4 Navigation

K4.1 Gehe zur Deklaration

Bei der Analyse von Quelltext ist das automatische Auffinden der Deklaration einer Klasse, Methode, Variablen etc. eines der wichtigsten Werkzeuge, die eine Entwicklungsumgebung

bietet. Die Referenz auf den Ursprung einer Entität wird bei der Navigation mit dem höchsten Gewicht bewertet.

K4.2 Finden aller Referenzen

An zweiter Stelle kommt die Anzeige aller Referenzen einer Entität, also an welcher Stelle eine Entität in welchem Kontext verwendet wird. Dabei sollte es keine Rolle spielen, ob die Referenzen innerhalb oder außerhalb des Modules liegen, von dem aus die Suche gestartet wird. Besonders bei der Analyse und Fehlersuche kommt dieser Funktion eine große Bedeutung zu.

K4.3 Anzeige der Klassenstruktur

Die Struktur einer Klasse setzt sich aus deren Membern zusammen. Die Navigation über diese ist ein entscheidendes Merkmale, wie innerhalb einer Klasse navigiert wird. Innerhalb der Navigation handelt es sich um das drittwichtigste Attribut.

K4.4 Anzeige der Typhierarchie

Eine Übersicht der Typhierarche ist eine gute strukturelle Ergänzung zur Visualisierung der Anwendungsarchitektur. Da es sich um eine optionale Eigenschaft handelt, deren Informationsgehalt auch über die Eigenschaft „Gehe zur Deklaration“ extrahiert werden kann, wird deren Gewicht als geringstes eingestuft.

Ziel: K4	K4.1	K4.2	K4.3	K4.4	Attribut-gewicht
K4.1	1	3	5	8	0,573
K4.2	0,333	1	3	5	0,259
K4.3	0,200	0,333	1	3	0,116
K4.4	0,125	0,200	0,333	1	0,052

Tabelle 8 Paarvergleichsmatrix mit dem Ziel: K4 Navigation

3.1.4.2.6.6 K5 Refactoring

K5.1, K5.2 Umbenennen und Verschieben von Elementen

Da Software heutzutage nur noch selten nach dem Wasserfallmodel entwickelt wird, durchläuft der Quelltext viele Optimierungsschritte. Die zwei wichtigsten Operationen in diesem Prozess sind für den Entwickler das Umbenennen und Verschieben von Elementen wie Klassen, Klassenelemente oder gar ganzer Paketstrukturen. Vor allem in großen Projekten bei denen Änderungen mehrere Module gleichzeitig betreffen, sind diese Funktionen von großer Bedeutung. Beide Operationen sind gleich wichtig und werden als essentiell für den Aufbereitungsprozess bei der Optimierung von Software angesehen.

K5.3 Automatische Verwaltung von Abhängigkeiten

Während der Implementierung von Software bilden Komponenten aus unterschiedlichen Quellen ein gesamtes Ganzes ab. Damit der Entwickler sich um die explizite Angabe der Quellen nicht kümmern muss, bindet er die gebrauchte Bibliothek ein und die Entwicklungsumgebung kümmert sich selbstständig um die eindeutige Pfadangabe. Obwohl diese Eigenschaft als Standardfunktionalität bei Entwicklungsumgebungen angesehen wird, trägt sie zu einer hohen Produktivität bei und wird deshalb als zweitwichtigstes Attribut gewichtet.

K5.4 Automatische Formatierung des Quellcodes

Die selbstständige Formatierung des Quellcodes nach bestimmbaren Regeln ist ein weiteres Standardattribut einer IDE, das für beide Sprachen, Java und Scala von der IDE bereitzustellen ist. Eine Produktivitätssteigerung bietet diese, da der Entwickler die Formatierung einheitlich in einem Schritt veranlassen kann. Da es sich aber um einen kosmetischen Eingriff auf den Quelltext handelt, ist diese Eigenschaft mit dem geringsten Gewicht versehen.

Ziel: K5	K5.1	K5.2	K5.3	K5.4	Attribut-gewicht
K5.1	1	1	2	4	0,364
K5.2	1	1	2	4	0,364
K5.3	0,500	0,500	1	2	0,182
K5.4	0,250	0,250	0,500	1	0,091

Tabelle 9 Paarvergleichsmatrix mit dem Ziel: K5 Refactoring

3.1.4.2.6.7 K6 Fehlerdiagnose und Analyse

K6.1 Funktionierende Haltepunkte

Bei der Fehlerdiagnose von Softwareanwendungen ist das Verhalten zur Laufzeit von größtem Interesse. Zu diesem Zweck unterstützen IDE's das Setzen von Haltepunkten. Die Funktionalität dieser ist in beiden Sprachen, Java und Scala, wünschenswert. Zwar können viele Informationen über das Laufzeitverhalten auch auf anderen Wegen, z.B. Ausgabe von Zuständen über die Konsole oder über den Einsatz von Logging-Werkzeugen, gewonnen werden, jedoch ist der Einsatz von Haltepunkten der gängigste Weg. Deshalb wird diesem Kriterium das höchste Gewicht bei der Fehlerdiagnose und Analyse zugeteilt.

K6.2 Interaktive REPL Konsole

Die Evaluierung von Scala-Quelltext kann auch direkt in der IDE erfolgen, sofern diese Funktionalität unterstützt wird. Im Idealfall erkennt die Entwicklungsumgebung den Projektkontext und unterstützt den Entwickler durch bereits genannte Eigenschaften, wie Quelltextvervollständigung oder Verwaltung der Importe. Da eine Evaluierung auch außerhalb der IDE in der Kommandozeile möglich ist, wird dieses Kriterium mit wenig Gewicht gewertet.

Ziel: K6	K6.1	K6.2	Attributgewicht
K6.1	1	3	0,750
K6.2	0,333	1	0,250

Tabelle 10 Paarvergleichsmatrix mit dem Ziel: K6 Fehlerdiagnose und Analyse

3.1.4.2.7 Beschreibung und Gewichtung der Alternativen

Für die Untersuchung wurde ein Testprojekt erstellt, das die gemeinsame Ausgangsbasis für den Vergleich bot. Die Evaluierung selbst wurde auf einem Windows 7 64 Bit System mit dem JDK 1.6_30 ausgeführt.

IntelliJ IDEA

Die einzige kommerzielle IDE unter den Alternativen ist die IntelliJ IDEA (kurz IntelliJ) Entwicklungsumgebung, welche auch als kostenfreie Community Edition in eingeschränkter Funktionalität bereitgestellt wird [JetBrains s.r.o.2011]. Die Beispielanwendung dieser Arbeit wurde mit der kommerziellen, als auch der kostenlosen Version dieser IDE entwickelt. Signifikantester Unterschied war die fehlende XHTML/CSS Unterstützung in der frei verfügbaren Version. Der folgende IDE-Vergleich bezieht sich auf die Community Edition 11.0.2 mit dem Scala-Plugin in der Version 0.5.369 [Vigdorchik2012].

Eclipse - Scala IDE

Als Java-Primus unter den Entwicklungsumgebungen wird eine Eclipse Java EE IDE for Web Developers in der Version Helios SR2 verwendet. Als einzige IDE wird Eclipse ohne native Maven-Unterstützung angeboten. Dieser Mangel kann aber durch das m2eclipse-Plugin nachgeholt werden [Bentmann2012], Version 1.0.100.20110804-1717. Die Unterstützung für Scala wird durch das Plugin: „Scala-IDE for Eclipse“ bereitgestellt. Verwendung findet die Version 2.0.0-2_09 [Dragos2012]. Damit die Maven Unterstützung auch für Scala-Projekte funktioniert, wird das m2eclipse-scala Plugin benötigt. Dieses befindet sich noch in Entwicklung und wird in der Version 0.2.3-17 verwendet [Bernard2012c].

Netbeans

Nach der Übernahme von Sun Microsystems wird die Netbeans IDE weiter durch Oracle entwickelt. Evaluiert wurde die Version 7.0.1 [Oracle Corp.2012a]. Auch bei dieser IDE muss die Unterstützung für die Sprache Scala erst über ein Plugin hinzugefügt werden. Die verwendete Version lautet 2.9.x-0.9 [Dcaoyuan2012].

Es folgt nun ein Vergleich zwischen den Entwicklungsumgebungen (Alternativen) bezüglich deren Kriterien (Attribute). Dieser sieht eine Beschreibung des Vergleichsergebnisses in Prosa vor und eine quantitative Gewichtung in der Vergleichsmatrix.

3.1.4.2.7.1 K1 IDE technische Merkmale

K1.1 Stabilität

IntelliJ bietet definitiv die beste Stabilität unter den Alternativen. Eclipse verhält sich mit dem m2eclipse-scala Plugin etwas instabil, besonders bei zunehmender Projektgröße und Änderungen an der Projektstruktur. Über Aktualisierungen und Aufräum-Operationen kann dem Fehlverhalten entgegengekommen werden. Richtig stabil läuft Eclipse jedoch nur ohne Maven-Unterstützung. Die Referenzierung der Projektbestandteile, wie Klassen und Pakete, ist bei Netbeans leider sehr instabil. Dieses Problem ist die Hauptursache für weitere Funktionsmakel, welche diese IDE sehr stark abwerten.

Ziel: K1.1	IntelliJ	Eclipse	Netbeans	Alternativen-gewicht
IntelliJ	1	3	8	0,661
Eclipse	0,333	1	5	0,272
Netbeans	0,125	0,200	1	0,067

Tabelle 11 Paarvergleichsmatrix mit dem Ziel: K1.1 Stabilität

K1.2 Gemischte Java/Scala Projekte

Alle drei Entwicklungsumgebungen unterstützen grundsätzlich Projekte, deren Maven-Module in unterschiedlichen Programmiersprachen umgesetzt sein können.

Ziel: K1.2	IntelliJ	Eclipse	Netbeans	Alternativen-gewicht
IntelliJ	1	1	1	0,333
Eclipse	1	1	1	0,333
Netbeans	1	1	1	0,333

Tabelle 12 Paarvergleichsmatrix mit dem Ziel: K1.2 Gemischte Java/Scala Projekte

K1.3 Java 7 Unterstützung

Die Eclipse IDE bietet keine Java 7 Unterstützung an, Netbeans und IntelliJ hingegen schon. Für die Untersuchung der Kompatibilität wurde das JDK 1.7_01 verwendet.

Ziel: K1.3	IntelliJ	Eclipse	Netbeans	Alternativen-gewicht
IntelliJ	1	7	1	0,467
Eclipse	0,143	1	0,143	0,067
Netbeans	1	7	1	0,467

Tabelle 13 Paarvergleichsmatrix mit dem Ziel: K1.3 Java 7 Unterstützung

K1.4 Plugins

IntelliJ IDEA bietet als einzige IDE ein Scala Plugin an, das unabhängig von der verwendeten Scala-Version ist. Es entsteht also kein Administrationsaufwand, sollte von Scala 2.9 auf Scala 2.8 zurückgestellt werden. Bei der Konfigurationszeit der IDE's schneidet Eclipse am schlechtesten ab, da viele Details beachtet und mehr als ein Plugin zu installieren ist. Details zur Installation und Konfiguration der IDE-Plugins finden sich im Kapitel 3.2.3.2 der Arbeit.

Ziel: K1.4	IntelliJ	Eclipse	Netbeans	Alternativen-gewicht
IntelliJ	1	5	3	0,637
Eclipse	0,200	1	0,333	0,105
Netbeans	0,333	3	1	0,258

Tabelle 14 Paarvergleichsmatrix mit dem Ziel: K1.4 Plugins

3.1.4.2.7.2 K2 Maven Unterstützung

K2.1 Projektimport und Bytecode-Erstellung

Die Maven-Integration aller drei IDE's funktioniert für den Import gemischter Java/Scala Projekte gleich gut. Ebenso verhält es sich mit der Erstellung von JVM-Bytecode aus Quelltext.

Ziel: K2.1	IntelliJ	Eclipse	Netbeans	Alternativen-gewicht
IntelliJ	1	1	1	0,333
Eclipse	1	1	1	0,333
Netbeans	1	1	1	0,333

Tabelle 15 Paarvergleichsmatrix mit dem Ziel: K2.1 Projektimport und Bytecode-Erstellung

K2.2 IDE unabhängiges Maven Projekt

Soweit alle IDE-Plugins korrekt installiert sind, ist eine funktionierende Maven-Unterstützung in allen drei Entwicklungsumgebungen verfügbar.

Bei Netbeans und Eclipse sind die Scala-Plugins an eine bestimmte Scala-Version gebunden. Wird die Scala-Version in der Maven-Projektdatei verändert, verfällt somit die Scala-Kompatibilität der IDE. Im ungünstigsten Fall muss die IDE dann komplett neu aufgesetzt werden. Diese Abhängigkeit ist vor allem sehr unvorteilhaft bei teamorientierter Entwicklung, wenn die Änderung eines Programmierers Auswirkungen auf die

Entwicklungsumgebungen der übrigen Entwickler hat.
Einen Sonderfall bildet das m2eclipse-scala Plugin, welches sich in der Entwicklung befindet und negative Wechselwirkungen auf andere Funktionen der IDE ausübt. Aus diesem Grund, wurde auch die Möglichkeit evaluiert Maven aus der Kommandozeile in Verbindung mit Eclipse zu benutzen. Leider erhöht diese Lösung die Abhängigkeit zwischen IDE und Maven-Skript. Grund dafür sind weitere Eclipse-spezifische Ergänzungen in der Maven-Konfigurationsdatei. Genaue Details dieser Problematik finden sich im Toolsupport Kapitel 3.2.

Ziel: K2.2	IntelliJ	Eclipse	Netbeans	Alternativen-gewicht
IntelliJ	1	5	3	0,637
Eclipse	0,200	1	0,333	0,105
Netbeans	0,333	3	1	0,258

Tabelle 16 Paarvergleichsmatrix mit dem Ziel: K2.2 IDE unabhängiges Maven-Projekt

K2.3 Projekterweiterung um weiteres Modul

In allen drei Entwicklungsumgebungen konnte ein bestehendes modulares Projekt um ein weiteres Maven-Modul erfolgreich erweitert werden.

Ziel: K2.3	IntelliJ	Eclipse	Netbeans	Alternativen-gewicht
IntelliJ	1	1	1	0,333
Eclipse	1	1	1	0,333
Netbeans	1	1	1	0,333

Tabelle 17 Paarvergleichsmatrix mit dem Ziel: K2.3 Projekterweiterung um weiteres Modul

3.1.4.2.7.3 K3 Quelltextbearbeitung

K3.1 Fehlerhervorhebung

Als einzige IDE mit einer zuverlässigen Kennzeichnung von Fehlern innerhalb des Quelltextes kann man die IntelliJ IDE nennen. Zweitbeste ist die Eclipse Entwicklungsumgebung, die zwar Inkonsistenzen nach Maven-Operationen aufweist, jedoch mit der Aktualisierung der Maven-Projektkonfiguration wieder in einen stabilen Zustand findet. Als unberechenbar erweist sich Netbeans. Offensichtlich verliert die IDE nach unbestimmter Zeit der Anwendung, Referenzen zu bereits erstellten Elementen, wie Klassen, Schnittstellen und gar ganzen Projektmodulen. Eine Ursache für dieses Verhalten konnte in keinem kausalen Zusammenhang ermittelt werden.

Ziel: K3.1	IntelliJ	Eclipse	Netbeans	Alternativen-gewicht
IntelliJ	1	3	8	0,661
Eclipse	0,333	1	5	0,272
Netbeans	0,125	0,200	1	0,067

Tabelle 18 Paarvergleichsmatrix mit dem Ziel: K3.1 Fehlerhervorhebung

K3.2 Quelltextvervollständigung

Da die Referenzierung der Projektbestandteile die entscheidende Voraussetzung für dieses Kriterium ist, verhält sich die Gewichtung analog zur Fehlerhervorhebung.

Ziel: K3.2	IntelliJ	Eclipse	Netbeans	Alternativen-gewicht
IntelliJ	1	3	8	0,661
Eclipse	0,333	1	5	0,272
Netbeans	0,125	0,200	1	0,067

Tabelle 19 Paarvergleichsmatrix mit dem Ziel: K3.2 Quelltextvervollständigung

K3.3 Simultane Bearbeitung unabhängiger Projekte

Die IntelliJ Entwicklungsumgebung kann als Einzige nicht mehr als ein Projekt gleichzeitig bearbeiten. Natürlich kann das geladene Projekt sich aus Submodulen des gleichen Kontextes zusammensetzen, die wiederum simultan verwaltet werden können. Für Eclipse und Netbeans liegen keine Einschränkungen vor.

Ziel: K3.3	IntelliJ	Eclipse	Netbeans	Alternativen-gewicht
IntelliJ	1	0,125	0,125	0,059
Eclipse	8	1	1	0,471
Netbeans	8	1	1	0,471

Tabelle 20 Paarvergleichsmatrix mit dem Ziel: K3.3 Simultane Bearbeitung unabhängiger Projekte

K3.4 Syntaxhervorhebung

Alle drei IDE's unterstützen die Java und Scala Syntax, sofern die Plugin-Installation erfolgreich durchlaufen wurde.

Ziel: K3.4	IntelliJ	Eclipse	Netbeans	Alternativen-gewicht
IntelliJ	1	1	1	0,333
Eclipse	1	1	1	0,333
Netbeans	1	1	1	0,333

Tabelle 21 Paarvergleichsmatrix mit dem Ziel: K3.4 Syntaxhervorhebung

3.1.4.2.7.4 K4 Navigation

K4.1 Gehe zur Deklaration

Der Sprung zum Ursprung eines Quelltextelementes funktioniert in IntelliJ tadellos. Die Eclipse IDE hat auch hier, wie schon bei der Quelltextbearbeitung beschrieben, lösbare

Probleme nach dem Einsatz von Maven. Netbeans hingegen verliert zeitweise Referenzen und bildet manche sichtbaren Attribute/Methoden einer Klasse gar nicht ab.

Ziel: K4.1	IntelliJ	Eclipse	Netbeans	Alternativen-gewicht
IntelliJ	1	3	5	0,637
Eclipse	0,333	1	3	0,258
Netbeans	0,200	0,333	1	0,105

Tabelle 22 Paarvergleichsmatrix mit dem Ziel: K4.1 Gehe zur Deklaration

K4.2 Finden aller Referenzen

Als einzige IDE findet IntelliJ IDEA eine gesuchte Variable, Methode, Klasse etc. über Modulgrenzen hinweg ohne Einschränkung der verwendeten Sprache. Besonders positiv fällt auf, dass die Art, wie auf eine Referenz zugegriffen wird, lesend oder schreibend, zusätzlich anzeigt wird. Dies ist vor allem bei der Fehlersuche eine große Hilfestellung. Eclipse findet Referenzen zwar über Modulgrenzen hinweg, jedoch nur in jenen Modulen identischer Sprache. Wenn die Referenzierung in Netbeans funktioniert, verhält sich diese analog zur Eclipse IDE. Dies kann aber als kein zumutbarer Zustand gewertet werden.

Ziel: K4.2	IntelliJ	Eclipse	Netbeans	Alternativen-gewicht
IntelliJ	1	5	8	0,742
Eclipse	0,200	1	3	0,183
Netbeans	0,125	0,333	1	0,075

Tabelle 23 Paarvergleichsmatrix mit dem Ziel: K4.2 Finden aller Referenzen

K4.3 Anzeige der Klassenstruktur

Alle IDE‘s zeigen die Klassenstruktur korrekt an. Bei Netbeans jedoch klappt die Navigation zu den Attributen von Scala-Klassen nicht.

Ziel: K4.3	IntelliJ	Eclipse	Netbeans	Alternativen-gewicht
IntelliJ	1	1	2	0,400
Eclipse	1	1	2	0,400
Netbeans	0,500	0,500	1	0,200

Tabelle 24 Paarvergleichsmatrix mit dem Ziel: K4.3 Anzeige der Klassenstruktur

K4.4 Anzeige der Typhierarchie

In Eclipse und IntelliJ werden in Java/Scala sowohl Interfaces/ Traits, als auch Vererbungsbeziehungen korrekt angezeigt. Netbeans unterstützt hingegen nur die Anzeige in Java.

Ziel: K4.4	IntelliJ	Eclipse	Netbeans	Alternativen-gewicht
IntelliJ	1	1	7	0,467
Eclipse	1	1	7	0,467
Netbeans	0,143	0,143	1	0,067

Tabelle 25 Paarvergleichsmatrix mit dem Ziel: K4.4 Anzeige der Typhierarchie

3.1.4.2.7.5 K5 Refactoring

K5.1 Umbenennen von Elementen

IntelliJ unterstützt das Umbenennen von Klassen und deren Member über Sprach- und Modulgrenzen hinweg. Bei Eclipse setzt die Funktion aus, sobald der Wechsel von Java auf Scala und umgekehrt eintritt. Bleibt der Anwender beim Umbenennen in der gleichen

Sprache wirken sich die Änderungen auch auf abhängige Module aus. Solange die Referenzierung der Netbeans-IDE funktioniert, verhält sich diese bei der Umbenennung identisch zu Eclipse.

Ziel: K5.1	IntelliJ	Eclipse	Netbeans	Alternativen-gewicht
IntelliJ	1	5	8	0,742
Eclipse	0,200	1	3	0,183
Netbeans	0,125	0,333	1	0,075

Tabelle 26 Paarvergleichsmatrix mit dem Ziel: K5.1 Umbenennen von Elementen

K5.2 Verschieben von Elementen

IntelliJ erkennt das Verschieben von Elementen unabhängig der Sprache und passt die Referenzen in abhängigen Modulen automatisch an. Das Restrukturieren in Eclipse ist schlecht umgesetzt. Unter Java-Modulen funktioniert die Umsetzung, jedoch versagt die Anpassung aller Abhängigkeiten in Bezug auf Scala-Module. In Netbeans funktioniert das Verschieben unter Java, bricht jedoch die Referenzen zu Scala-Modulen. Unter Scala ist die Operation überhaupt nicht verfügbar.

Ziel: K5.2	IntelliJ	Eclipse	Netbeans	Alternativen-gewicht
IntelliJ	1	5	8	0,742
Eclipse	0,200	1	3	0,183
Netbeans	0,125	0,333	1	0,075

Tabelle 27 Paarvergleichsmatrix mit dem Ziel: K5.2 Verschieben von Elementen

K5.3 Automatische Verwaltung von Abhängigkeiten

IntelliJ und Eclipse unterstützen die automatische Verwaltung von Abhängigkeiten

uneingeschränkt. Im Falle, dass Netbeans die Referenzen erkennt, funktioniert der Auto-Import von Java-Modulen innerhalb Scala-Projekten nicht.

Ziel: K5.3	IntelliJ	Eclipse	Netbeans	Alternativen-gewicht
IntelliJ	1	1	7	0,467
Eclipse	1	1	7	0,467
Netbeans	0,143	0,143	1	0,067

Tabelle 28 Paarvergleichsmatrix mit dem Ziel: K5.3 Automatische Verwaltung von Abhängigkeiten

K5.4 Automatische Formatierung des Quellcodes

Alle drei Entwicklungsumgebungen unterstützen die automatische Formatierung von Java- und Scala-Quelltext.

Ziel: K5.4	IntelliJ	Eclipse	Netbeans	Alternativen-gewicht
IntelliJ	1	1	1	0,333
Eclipse	1	1	1	0,333
Netbeans	1	1	1	0,333

Tabelle 29 Paarvergleichsmatrix mit dem Ziel: K5.4 Automatische Formatierung des Quellcodes

3.1.4.2.7.6 K6 Fehlerdiagnose und Analyse

K6.1 Funktionierende Haltepunkte

Alle drei Entwicklungsumgebungen sind in der Lage, Zustände von Objekten schrittweise zu untersuchen. Dabei spielt es keine Rolle ob, ein Java-Objekt in einem Scala-Abschnitt oder umgekehrt, untersucht wird. Lediglich Eclipse erzeugt die Fehlermeldung „Unable to install breakpoint in ... due to missing line number attributes.“, die jedoch keine Auswirkungen auf den Diagnoseprozess hat.

Ziel: K6.1	IntelliJ	Eclipse	Netbeans	Alternativen-gewicht
IntelliJ	1	1	1	0,333
Eclipse	1	1	1	0,333
Netbeans	1	1	1	0,333

Tabelle 30 Paarvergleichsmatrix mit dem Ziel: K6.1 Funktionierende Haltepunkte

K6.2 Interaktive REPL Konsole

Alle drei IDE haben eine Read-Evolve-Print-Loop, die im Projektkontext startet. IntelliJ IDEA bietet jedoch die beste Interaktivität an, sowohl der automatische Import von Referenzen, als auch die Quelltextvervollständigung unterstützen den Entwickler.

Ziel: K6.2	IntelliJ	Eclipse	Netbeans	Alternativen-gewicht
IntelliJ	1	5	5	0,714
Eclipse	0,200	1	1	0,143
Netbeans	0,200	1	1	0,143

Tabelle 31 Paarvergleichsmatrix mit dem Ziel: K6.2 Interaktive REPL Konsole

3.1.4.2.8 Konsistenzprüfung

Unter der Konsistenz von Paarvergleichen ist deren Widerspruchsfreiheit zu verstehen. Ein Vergleich mir drei Elementen ist konsistent wenn, gilt: A > B , B > C, dann muss gelten A > C. Eine 100 prozentige Konsistenz ist aber nicht immer möglich. Dazu müssten bei jedem Vergleich die Gewichte transitiv verteilt werden: A - (2x) - > B , B - (3x) -> C und A - (6x) -> C. Dies ist schon aufgrund der AHP-Skala nicht immer möglich, da diese mit dem Wert 9 endet.

Es muss also sichergestellt sein, dass es sich bei dem Vergleich um keine Zufallsentscheidung handelt. Dabei wird angenommen, dass ein Vergleich, der durch eine informierte Person vorgenommen wird, konsistenter ist, als eine zufällige Gewichtung [Saaty2006].

Im Falle der vollständigen Konsistenz entspricht der maximale errechnete Eigenwert λ_{max}, der Elementanzahl - n - der Matrix [Meixner2002].

$$\lambda_{max} = n$$

Formel 4 Maximaler Eigenwert

Je größer also die Diskrepanz zwischen λ_{max} und n ist, desto inkonsistenter ist die Evaluierung. Dieses Verhältnis wird durch den Konsistenzindex - CI - ausgedrückt.

$$CI = \frac{(\lambda_{max} - n)}{(n-1)}$$

Formel 5 CI: Konsistenzindex

Verglichen wird CI mit der Zufallszahl R. Diese unterscheidet sich mit wachsender Zahl der Elemente der Matrix, da die zufällige zu erwartende Inkonsistenz mit der Größe der Matrix steigt [Meixner2002]. Das Verhältnis von CI/R = CR wird Konsistenzverhältniszahl genannt und sollte den Richtwert von **0,1** nach Saaty nicht übersteigen.

$$CR = \frac{CI}{R} \leq 0{,}1$$

Formel 6 CR: Konsistenzverhältnis

Der folgenden Tabelle kann entnommen werden, wie sich R mit zunehmender Matrixgröße anpasst.

n:	1	2	3	4	5	6	7	8	9	10
R:	0	0	0,52	0,89	1,11	1,25	1,35	1,40	1,45	1,49

Tabelle 32 Zufälliger Konsistenzverhältnisindex [Saaty2006]

Es folgt ein Beispiel der Konsistenzprüfung anhand der Matrix P, des Flächenvergleiches aus Kapitel 3.1.4.2.4.1, Tabelle 3. Die Prioritäten w_1 ... w_n sind aus Kapitel 3.1.4.2.5, Abbildung 17 entnommen. Im ersten Schritt wird die Evaluationsmatrix mit den Prioritäten multipliziert.

$$P * \begin{bmatrix} w_1 \\ w_2 \\ \dots \\ w_n \end{bmatrix} = \begin{bmatrix} u_1 \\ u_2 \\ \dots \\ u_n \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0{,}2 & 2 \\ 5 & 1 & 8 \\ 0{,}5 & 0{,}125 & 1 \end{bmatrix} * \begin{bmatrix} 0{,}161 \\ 0{,}751 \\ 0{,}087 \end{bmatrix} = \begin{bmatrix} 0{,}485 \\ 2{,}252 \\ 0{,}261 \end{bmatrix}$$

Formel 7 Konsistenzprüfung am Beispiel des Flächenvergleiches - 1. Schritt

Im zweiten Schritt wird der Quotient zwischen den Elementen des Spaltenvektors u und dem Prioritätenvektor w gebildet. Das Skalarprodukt zwischen dem Zeilenvektor (1,1,1) und dem eben erstellten Quotientenvektor ergibt die Summe dessen Elemente. Der maximale Eigenvektor der Evaluationsmatrix entspricht dem Durchschnitt der Summe durch die Anzahl der Matrixelemente [Weber1993].

$$\lambda_{max} = (1,1,...,1) * \begin{bmatrix} u_1/w_1 \\ u_2/w_2 \\ \dots \\ u_n/w_n \end{bmatrix} * (1/n)$$

$$\lambda_{max} = (1,1,1) * \begin{bmatrix} 3{,}012 \\ 2{,}999 \\ 3{,}000 \end{bmatrix} * (1/3) = (9{,}011 / 3) = 3{,}004$$

Formel 8 Konsistenzprüfung am Beispiel des Flächenvergleiches - 2. Schritt

Im letzten Schritt wird die beschriebene Konsistenzverhältniszahl berechnet. In diesem Fall ist CR <= 0,1. Somit ist die Matrix konsistent und gilt als plausibel.

$$CI = \frac{(3{,}004 - 3)}{(3-1)} = 0{,}002$$

Formel 9 Berechnung des Konsistenzindex

$$CR = \frac{0{,}002}{0{,}52} = 0{,}004 \leq 0{,}1$$

Formel 10 Verhältnisbildung und Vergleich mit Grenzwert 0,1

3.1.4.2.9 Berechnung der globalen Attributgewichte

Die Summe aller Gewichte einer Ebene beträgt immer 1. Um das globale Bedeutungsgewicht eines Subattributes der Elementhierarchie zu berechnen, muss sein lokales Gewicht mit den lokalen Gewichten seiner übergeordneten Knoten multipliziert werden. Der Graph über den dabei multipliziert wird, beginnt beim tiefsten Subattribut in der Elementhierarchie und reicht hoch bis zur Wurzel, dem Hauptziel des AHPs. Abbildung 18 zeigt dabei die Berechnung der globalen Gewichte für die Attribute K1.1, K1.2 und K6.2 entlang der Elementhierarchie.

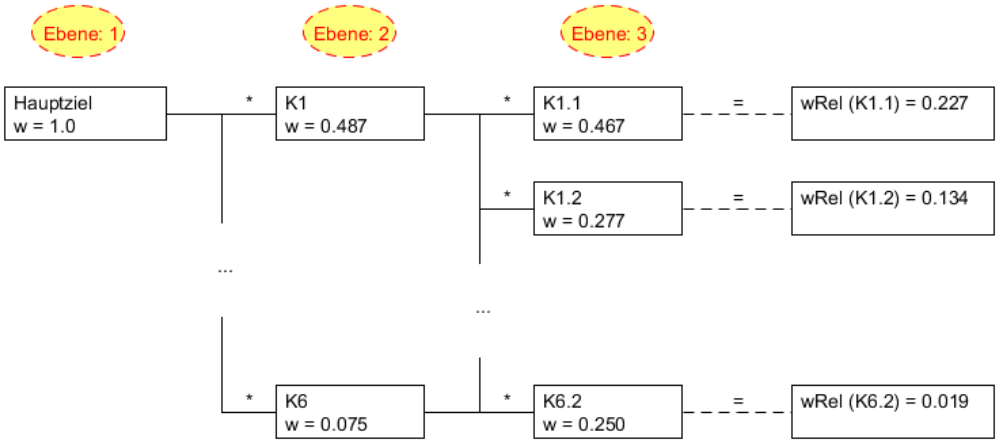


Abbildung 18 Beispielrechnung der globalen Attributgewichte für K1.1, K1.2 und K.6.2

3.1.4.2.10 Erstellung der Alternativrangfolge

Nachdem die globalen Attributgewichte - wRel(Subattribut) - erstellt wurden, werden diese mit den lokalen Alternativgewichten - w(Alternative) - der Subattribute multipliziert.

Subattribut	wRel(Subattribut)	w(IntelliJ)	w(Eclipse)	w(Netbeans)
K1.1	0,227	0,661	0,272	0,067
K1.2	0,134	0,333	0,333	0,333
...	...	...	...	...
K6.2	0,019	0,714	0,143	0,143

Tabelle 33 Ermittlung der globalen Alternativgewichte

Das Ergebnis sind globale Alternativgewichte - wRel(Alternative) -, deren Summe je Alternative die Wichtigkeit jeder Alternative ist und das Ergebnis des AHPs repräsentiert [Meixner2002].

Subattribut	wRel(IntelliJ)	wRel(Eclipse)	wRel(Netbeans)
K1.1	0,150	0,062	0,015
K1.2	0,045	0,045	0,045
...	...	...	...
K6.2	0,014	0,003	0,003
Summe	**0,521**	**0,258**	**0,222**

Tabelle 34 Ermittlung der Alternativrangfolge

3.1.5 Umsetzung des AHP

3.1.5.1 Anwendungsfälle

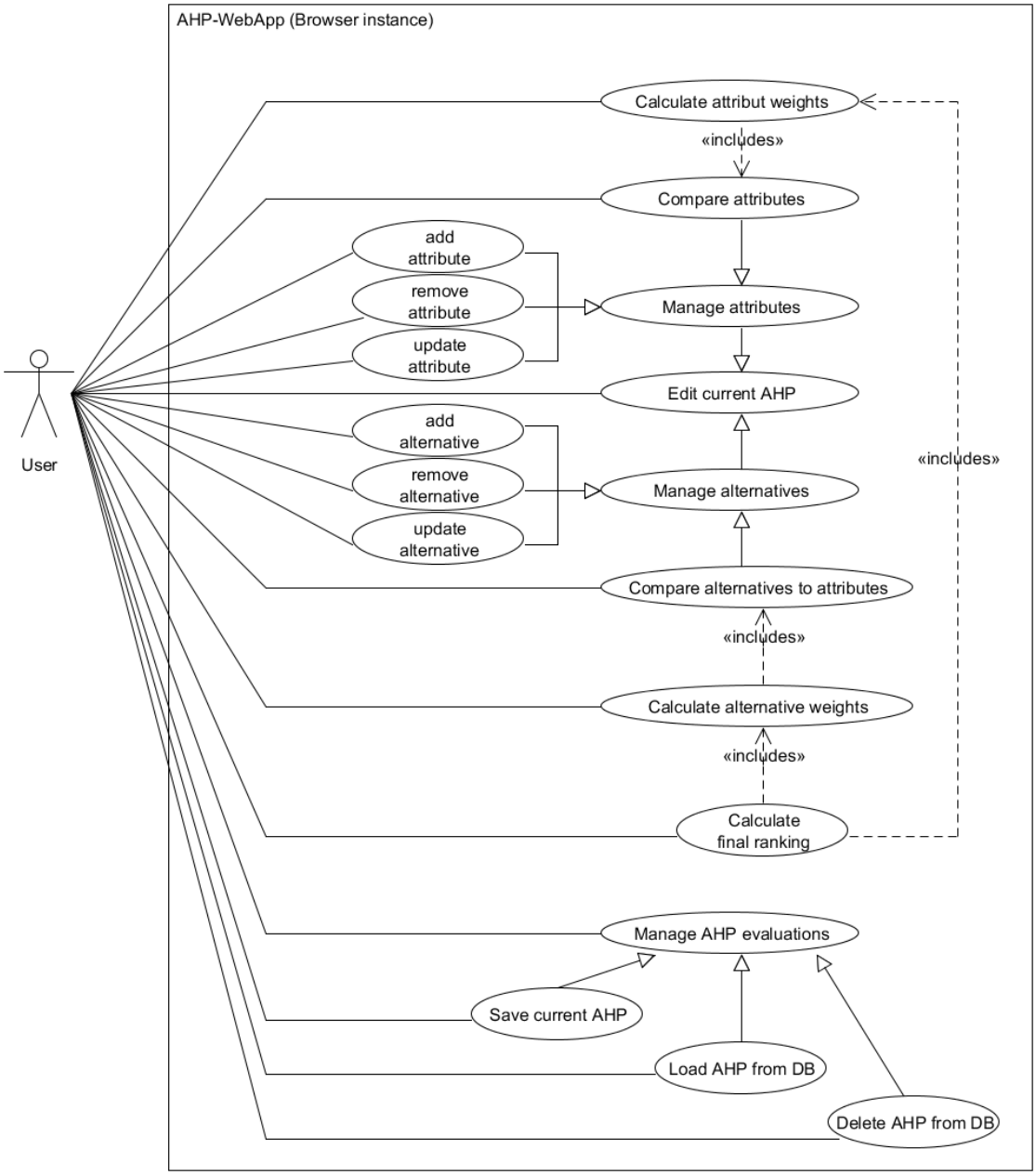


Abbildung 19 Anwendungsfalldiagramm der Beispielanwendung AHP-WebApp

Der Anwender (engl.: User) verbindet sich mittels Webbrowser in der Client-Rolle zum Webserver, auf dem die Beispielanwendung betrieben wird. Theoretisch können beliebig viele Anwender an der Evaluation gleichzeitig mitwirken, getestet wurde allerdings nur der

Single-Client-Betrieb. Nach erfolgreicher Verbindung besteht die Möglichkeit eine bereits persistierte AHP-Evaluation aus der Datenbank in den Kontext zu laden „Load AHP from DB“, diese zu pflegen „Manage AHP evaluations“ oder einen neuen AHP zu editieren „Edit current AHP“. Dafür muss das Ziel des AHPs, die Attribute/Kriterien und die Alternativen innerhalb der Evaluation abgebildet sein. Einfluss kann der Anwender mit den „add“, „remove“ und „update“ Operationen auf den notwendigen Daten ausüben. Abbildung 20 zeigt die Manipulation der Attribut-Hierarchie.

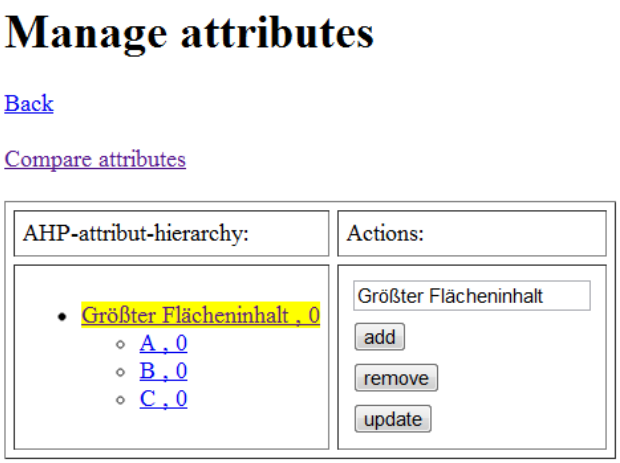


Abbildung 20 Bedienmaske zur Bearbeitung der Attribut-Hierarchie

Nachdem alle notwendigen Daten bezüglich der Evaluationsstruktur vorliegen, finden Paarvergleiche zwischen den Attributen untereinander und den Alternativen zu den Attributen statt. Für die Paarvergleiche wird eine NxN Matrix generiert, welche von dem selektierten Blatt/Knoten innerhalb der Navigation abhängt (siehe Abbildung 21 „Navigation“). Der Benutzer wählt ein Element innerhalb der Matrix aus und wertet dieses, indem er auf einen der Werte in der AHP-Skala klickt. Innerhalb der Matrix wird der ausgewählte Wert und der Kehrwert auf der gegenüberliegenden Position der Diagonalen automatisch eingetragen. Weiter erfolgt eine Benachrichtigung im Ausgabefeld (siehe Abbildung 21 roter Pfeil unten) der semantischen Bedeutung des Vergleiches, entsprechend der AHP-Skala.

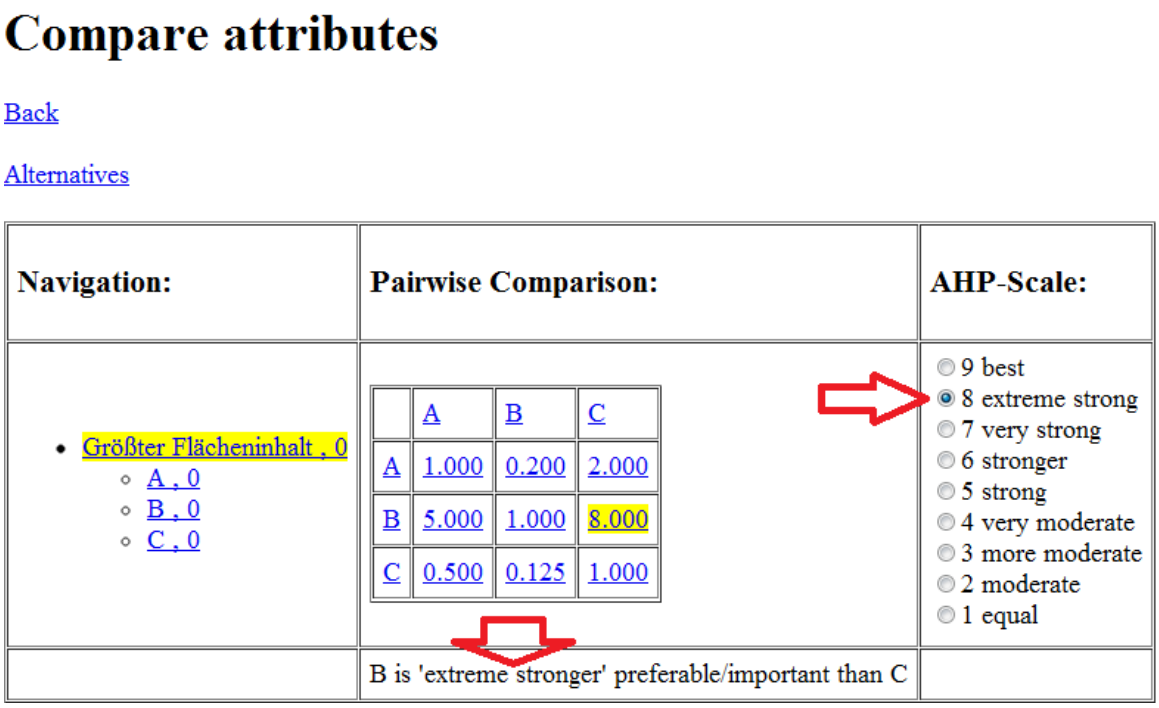


Abbildung 21 Bedienmaske zur Wertung der Paarvergleiche

Nachdem alle Paarvergleiche vorgenommen wurden, kann die Berechnung der Gewichte erfolgen. Dazu müssen lediglich die Schaltflächen „sequential“ und „parallel“ betätigt werden (siehe Abbildung 21). Das darunterstehende Diagramm bildet die Gewichtung des Ergebnisses ab. Abschließend zeigt das Ausgabefeld den Erfolg/Misserfolg der Berechnung, bezüglich der Konsistenz des Ergebnisses an.

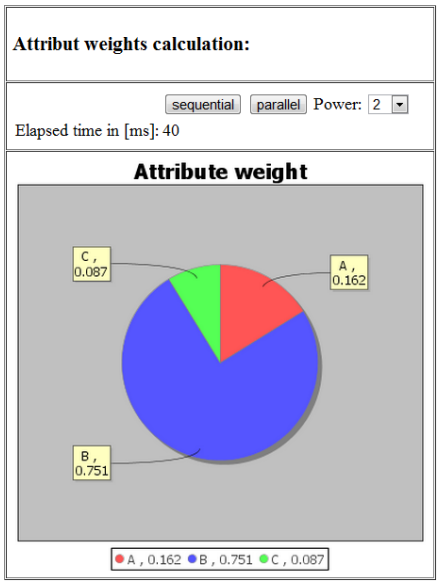


Abbildung 22 Bedienmaske zur Berechnung und Ergebnisdarstellung der Attributgewichte

Analog zu diesen Ausschnitten über die Gewichtung der Attribute, wurden die Anwendungsfälle und Bedienmasken der Alternativen gestaltet. Der finale Anwendungsfall, die Bestimmung der Alternativenrangfolge „Calculate final ranking“, kann erst vollzogen werden, nachdem die Anwendungsfälle „Calculate attribute weights“ und „Calculate alternative weights“ erfolgreich durchlaufen wurden.

3.1.5.2 Anwendungsschichten

Es folgt eine Beschreibung der Aufgaben und Mittel mit denen die Schichtenarchitektur der Beispielanwendung umgesetzt wurde sowie deren Zusammenspiel.

3.1.5.2.1.1 Wicket in der Benutzeroberfläche

Für die Interaktion des Benutzers mit dem AHP wurde eine adäquate Benutzeroberfläche implementiert, deren Zugriff über einen Browser erfolgt.

Umgesetzt wurde die Benutzeroberfläche mit Apache Wicket. Dabei handelt es sich um ein komponentenbasiertes Java Webframework, welches eine klare Trennung pflegt zwischen der Anwendungslogik, die in Java implementiert wird und der Dokumentstruktur in HTML. Die designspezifische Erscheinung des Dokumentes kann mittels Cascading Style Sheets (kurz CSS) beeinflusst werden. Die Kopplung zwischen den Bereichen, welche im Dokument

dynamisch sind und von der Anwendungslogik beeinflusst werden, erfolgt durch die Zuweisung einer „wicket:id“ an ein HTML-Element.

```
1 <html xmlns:wicket="http://wicket.apache.org">
2 ...
3         <a href="#" wicket:id="switchToNewAhp">Edit current AHP</a>
4 ...
5 </html>
```

Listing 96 Definition einer „wicket:id“ innerhalb eines Anker-Tags im HTML-Dokument: Index.html

In Listing 96 wird mit dem Anker-Tag <a> ein HTML-Link, innerhalb des „Index“ HTML-Dokumentes, definiert. Der Tag selbst enthält eine eindeutige „wicket:id“ mit dem Wert „switchToNewAhp“.

```
1 public Index() {
2         add(new Link("switchToNewAhp") {
3                 @Override
4                 public void onClick() {
5                         ...
6                 }
7         });
8 }
```

Listing 97 Verknüpfung der „wicket:id“ mit korrespondierender Java-Klasse: Index.java

Damit zur Laufzeit bestimmt werden kann, welche Aktion ausgeführt werden soll, nachdem der Link betätigt wurde, wird innerhalb des Konstruktors der Java Klasse „Index“ die Link-Komponente hinzugefügt (siehe Listing 97). Die genannte „wicket:id“ wird dazu benutzt, um eine Verknüpfung zwischen Java-Komponente und HTML-Element herzustellen. Die Hierarchie der Komponenten in Java und HTML muss dabei identisch sein [Dashorst2008].

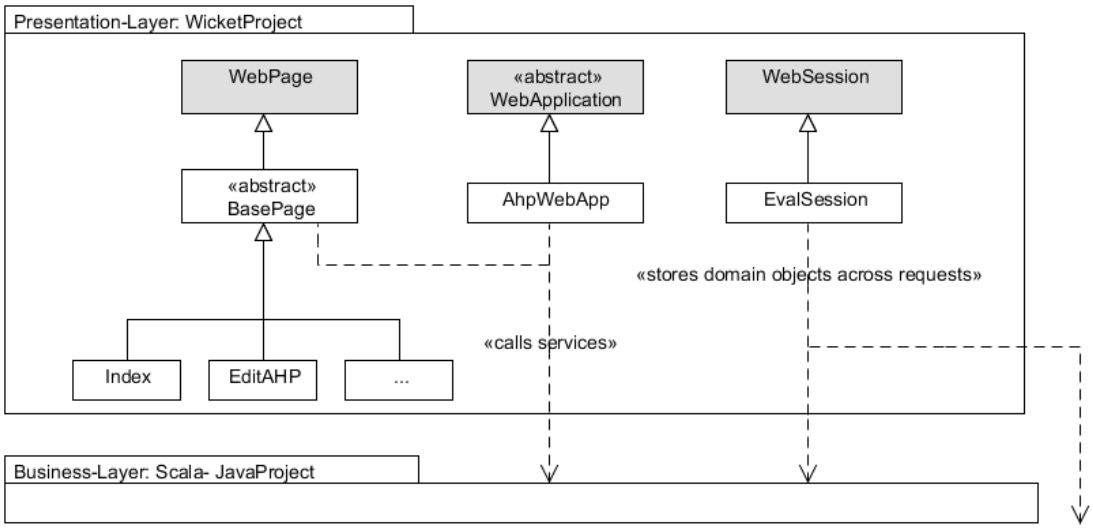


Abbildung 23 Ausschnitt aus der Präsentations-Schicht der AHP-WebApp. Die grau hervorgehobenen Klassen entstammen dem Wicket-Framework

Eine rudimentäre Wicket-Anwendung weist mindestens zwei Klassen auf. Die Erste ist die „WebApplication“. Die Zweite, beispielsweise. „Index“, dient als Startseite, welche von „BasePage“ erweitert wird. Innerhalb der Beispielanwendung (siehe Abbildung 23) wird die „AhpWebApp“-Klasse von der „WebApplication“-Klasse erweitert und dient der Initialisierung der Anwendung sowie der Konfiguration ihrer selbst und weiterer Frameworks/ Komponenten. Zur Beginn der Anwendung stand die Minimalanforderung, dass diese von einem User bedient und später auf mehrere ausgebaut werden soll. Da es sich bei HTTP, dem Kommunikationsprotokoll zwischen Browser und Web-Server, um ein zustandsloses Protokoll handelt, werden die Zustandsänderungen gespeichert. Dies geschieht für jeden Nutzer innerhalb einer Session. Im Falle der Beispielanwendung betrifft das die gesamte Evaluation, welche durch die Klasse „AhpEvaluation“ repräsentiert wird. Die Klasse „EvalSession“ wird dazu von der „WebSession“ Klasse erweitert. Innerhalb der Session werden alle Fachobjekte der Evaluation gespeichert. Dies ist für die Rekonstruktion des Anwendungszustandes, beispielsweise wenn der Benutzer den Zurück-Knopf im Browser betätigt, von Bedeutung. Hierfür muss jedes Domänenobjekt die „java.io.Serializable“ Schnittstelle implementieren. Der damit verbundene Prozessor- und Speicherverbrauch steigt somit linear mit wachsender Benutzerzahl. Diesem Malus kann zwar mit

„LoadableDetachableModel“s kompensiert werden, jedoch ist die Anwendung nicht für Hunderte von gleichzeitigen Nutzern vorgesehen. Außerdem war von Interesse, wie kompatibel sich die Scala-Business-Layer mit der Implementierung der Serializable-Schnittstelle verträgt. Als letztes wird auf die Basisklasse der HTML-Dokumente eingegangen. Eine lauffähige Wicket-Anwendung muss mindestens eine Homepage-Klasse enthalten. Um gemeinsame Funktionalitäten zu bündeln, empfiehlt sich deren Definition innerhalb der „BasePage“-Klasse, von welcher alle folgenden Klassen der Benutzeroberfläche erben [Dashorst2008].

Einer der Hauptvorteile von Wicket ist dessen komponentenbasiertes Design. Die gebräuchlichsten Benutzeroberflächenelemente bietet das Framework bereits an. Für die Umsetzung des AHP wurde aber eine Tabelle benötigt, welche dynamisch in X- und Y-Richtung skaliert und deren Zellen individuell aktualisiert werden konnten. Das Besondere an den Wicket-Komponenten ist die Möglichkeit, diese ineinander zu schachteln. Damit konnte für die AHP-Paarvergleiche benötigte dynamische NxN-Matrix realisiert werden, ohne dass eine entsprechende Komponente dafür existiert. Dafür wurden zwei Listen ineinander gesteckt, welche Veränderungen ihrer Elemente selbstständig aktualisieren. Zusätzlich kam die Anforderung, dass die Elemente der Matrix selektierbar sein sollten. Dazu wurde die Matrix mit „Links“ befüllt, wobei jedes Element seine Position selbst verwalten muss, nachdem es ausgewählt wurde.

```
1 this.dynamicTable = new ListView("rowList", dynamicTableData) {
2
3           @Override
4           protected void populateItem(ListItem item) {
5
6               ArrayList<String> colList = (ArrayList<String>) item.getModelObject();
7               item.add(new ListView("colList", colList) {
8
9                   @Override
10                  protected void populateItem(ListItem item) {
11
12                      ACompareValue compareValue = (ACompareValue) item.getModelObject();
13                      Link link = new Link("selectedCompareValue", item.getModel()) {
14
15                          @Override
16                          public void onClick() {
17                              selectCompareValue(this);
18                          }
```

```
19                  };
20                  link.add(new Label("element", splitCompareValueData(compareValue)));
21                  link.add(new SelectedTableBehavior(currentCompareValue, (ACompareValue)
22                              link.getModelObject()));
23                  item.add(link);
24              }
25          });
26      }
27  };
```

Listing 98 Implementierung der NxN-Matrix in der Java-Klasse: CompareAlternatives.java

Durch „new ListView“ wird in Listing 98, Zeile 1 eine neue sich selbst aktualisierende Liste „rowList“ erstellt. Innerhalb der „populateItem“ Methode, Zeile 4, erfolgt die Instanzierung der zweiten Liste „colList“, welche die gewünschte NxN Eigenschaft ausmacht. Die Elemente der Matrix, bildet der Link „selectedCompareValue“, aus Zeile 13 ab. Wie bereits erwähnt, muss diese Hierarchie der Komponenten identisch im korrespondierenden HTML-Dokument umgesetzt werden.

Um eine flüssige Entwicklung sicherstellen zu können wurde der Entwicklungsprozess auf dem leichtgewichtigen Jetty-Webserver vollzogen. Dieser kann vorkonfiguriert über ein Maven-Quickstart-Projekt von der Apache-Website bezogen werden [Apache2011a].

3.1.5.2.1.2 Funktions- und Steuerungs-Schicht

Die mittlere Schicht der Anwendung kann auch als Business-Layer bezeichnet werden, da ihre primären Aufgaben in der Bereitstellung der fachlichen Funktionalität, wie Berechnungen und Plausibilitätsprüfungen, und der Weitergabe von Diensten besteht [Steger2009]. Sie wurde in Java und in Scala implementiert, um einen Vergleich der beiden Sprachen in der Praxis zu veranschaulichen. Da der Zugriff von Java auf Scala und umgekehrt untersucht werden sollte, bot sich die Business-Layer für diesen Zweck an. Die Ergebnisse des Vergleiches sind dem Kapitel 3.1.6.3 zu entnehmen. Probleme bezüglich der Interoperabilität beider Sprachen finden sich im Kapitel 3.1.6.1.

Umgesetzt wurde sowohl die Business-Layer, als auch die Datenbank-Schicht mit dem „Fassade“-Entwurfsmuster, dessen Aufgabe ist es, die interne Struktur eines komplexen Subsystems nach außen hin zu kapseln und für mehrere Clients bereit zu stellen [Eilebrecht2010]. Die Kapselung erfolgt dabei mittels einer abstrakten Komponente. Je nach

verwendeter Sprache wird dazu ein Interface (Java) oder ein Trait (Scala) verwendet. Das folgende UML-Diagramm stellt die Struktur der Business-Layer in der Scala-Implementierung dar.

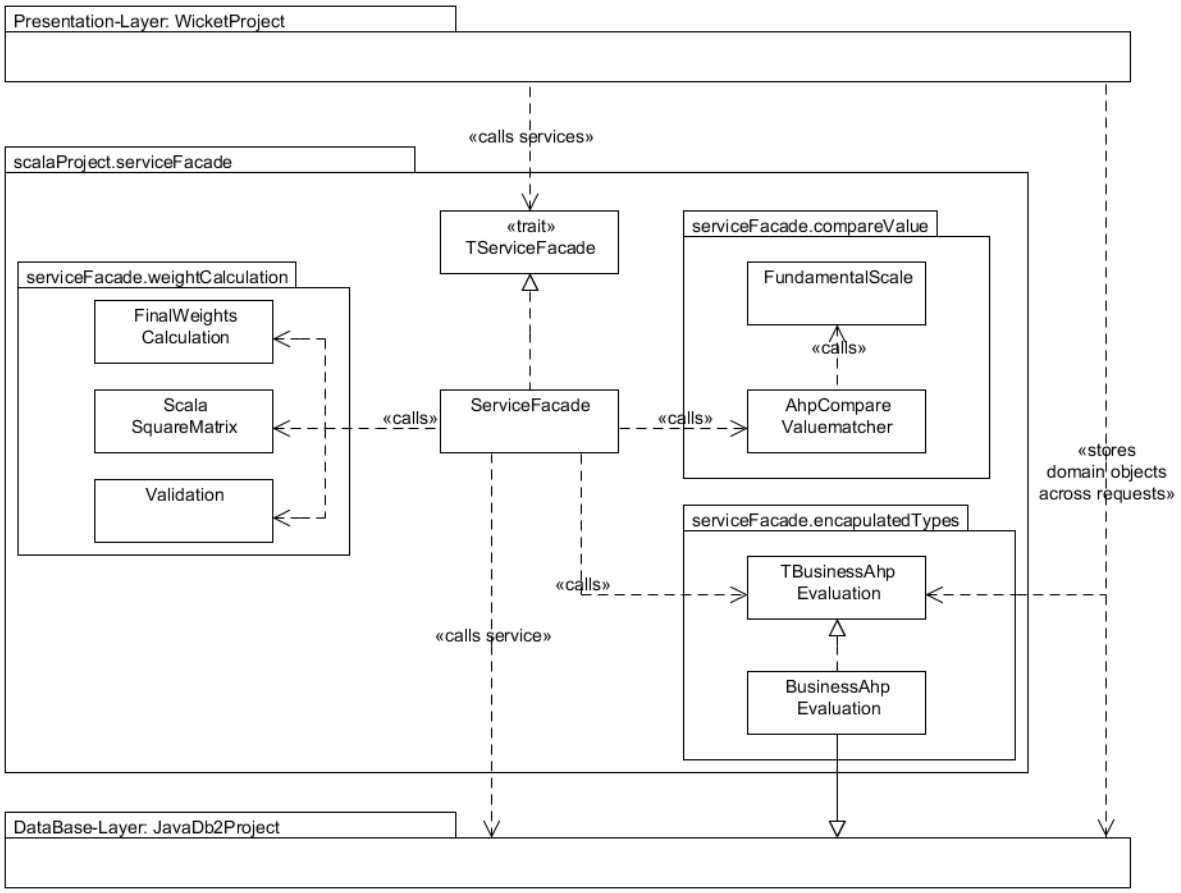
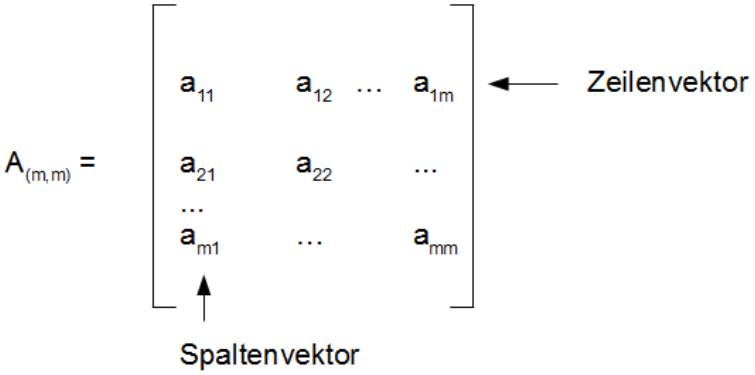


Abbildung 24 Funktions- und Steuerungs-Schicht der Beispielanwendung: AHP-WebApp

Aus Abbildung 24 kann erkannt werden, wie der Trait „TServiceFacade“ die Funktionalität der Business-Layer kapselt sowie die konkrete Implementierung innerhalb der „ServiceFacade“ auf weitere interne Module und die darunterliegende Schicht zugreift.

Die Hauptfunktionalität dieser Schicht ist die Berechnung der AHP-Gewichte (siehe Kapitel 3.1.4.2.5), auf deren Rechenaufwand weiter eingegangen wird.
Als Eingabegröße dient eine quadratische Matrix A, welche die Paarvergleiche abbildet.



Formel 11 Quadratische Matrix

Die Berechnung der Prioritäten erfolgt nun durch Potenzierung, also der Multiplikation der Matrix mit sich selbst und abschließender Normalisierung. Die Skalierung des Rechenaufwandes erfolgt über die Potenzierung der Eingabematrix der Dimension m und über die Höhe der Potenz k. Im Detail werden dafür die Skalarprodukte, c_{ij} mit (1 <= i <= m) und (1 <= j <= m), aus entsprechendem Zeilen- und Spaltenvektor berechnet, die den Elementen eines Matrizenproduktes entsprechen. Der Algorithmus für die Multiplikation der Matrix lautet:

Multipliziere den **1.** Zeilenvektor mit jedem Spaltenvektor.
Multipliziere den **2.** Zeilenvektor mit jedem Spaltenvektor.
...
Multipliziere den **m.** Zeilenvektor mit jedem Spaltenvektor.
[Papula2011]

Die Anzahl der Zeilen- und Spaltenvektor-Komponenten a_{ij} entspricht in der quadratischen Matrix der Dimension m. Mit zunehmender Dimension, steigt deshalb der Rechenaufwand eines Skalarproduktes c_{ij}, wie Formel 12 an einem Beispiel zeigt.

$$c_{11} = a_{11} * a_{11} + a_{21} * a_{12} + \ldots + a_{m1} * a_{1m}$$

Formel 12 Steigender Rechenaufwand, demonstriert am Skalarprodukt

Weiter steigt mit der Größe der Matrix (Dimension m), also mit der Anzahl zu vergleichender Attribute, die Anzahl der zu berechnenden Skalarprodukte c_{ij}, quadratisch an.

Mit der Variation der Potenz k (engl.: Power) kann diese skaliert werden. Die Potenz bestimmt, wie oft die Matrix mit sich selbst multipliziert wird, bevor die Gewichtung berechnet wird. Je nach Matrixgröße und Höhe der Potenz, kann der Benutzer die Berechnung sequentiell oder parallel verarbeiten lassen, um die Laufzeiten beider Methoden zu vergleichen (siehe Abbildung 25).

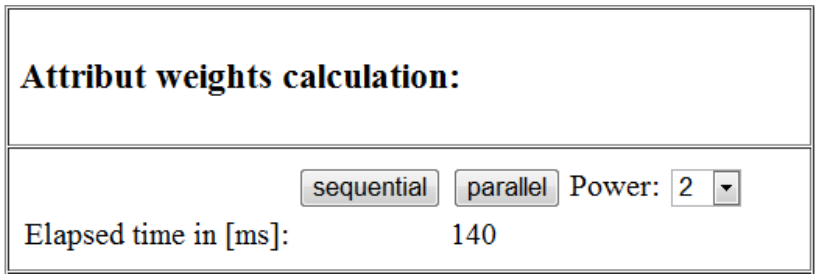


Abbildung 25 Skalierbarkeit der Berechnung innerhalb der AHP-WebApp Beispielanwendung

Der Benutzer kann die Potenz sukzessiv steigern, bis sich das Ergebnis der Gewichte nicht mehr verändert und somit die höchste Genauigkeit erreicht wird. Dies soll am Beispiel des Flächenvergleiches aus Kapitel 3.1.4.2.4.1 demonstriert werden. Abbildung 26 zeigt die Prioritäten nach einer Quadratur der Matrix (k=2) aus der Beispielanwendung AHP-WebApp. Während Abbildung 27 die Gewichtung mit dritter Potenz der Matrix darstellt.

Größter Flächeninhalt , 0
- A , 0.16164264
- B , 0.75141983
- C , 0.08693753

	A	B	C
A	1.000	0.200	2.000
B	5.000	1.000	8.000
C	0.500	0.125	1.000

Abbildung 26 Prioritäten mit k = 2

Größter Flächeninhalt , 0
- A , 0.16180746
- B , 0.75104143
- C , 0.08715111

	A	B	C
A	1.000	0.200	2.000
B	5.000	1.000	8.000
C	0.500	0.125	1.000

Abbildung 27 Prioritäten mit k = 3

Mit zunehmender Potenz wird die Differenz zwischen den Ergebnissen geringer. Umgekehrt steigt die Inkonsistenz der Paarvergleiche mit zunehmender Matrix-Größe [Quelle Meixner]. Mit der Flexibilität der AHP-WebApp lässt sich damit eine sehr hohe Genauigkeit bei beliebiger Matrix-Größe umsetzen.

3.1.5.2.1.3 Datenbank-Schicht

Identisch zur Business-Layer wird auch hier die Funktionalität nach außen hin mit einer Fassade gekapselt. Der Zugriff auf die Datenbankfunktionalität ist demnach ausschließlich über die Schnittstelle „IPersistenceFacade“ möglich (siehe Abbildung 28). Die Aufgaben der Datenbank-Schicht belaufen sich in der Beispielanwendung auf die rudimentäre Persistierung der Daten auf einer IBM-DB2 Datenbank. Die Implementierung eines Session- bzw. Transaktionsmanagements war nicht erforderlich. Außerdem wurde innerhalb dieser Schicht das Objektmodel definiert. Die Gründe für diese Entscheidung, sind dem Kapitel 3.1.6.1 zu entnehmen.

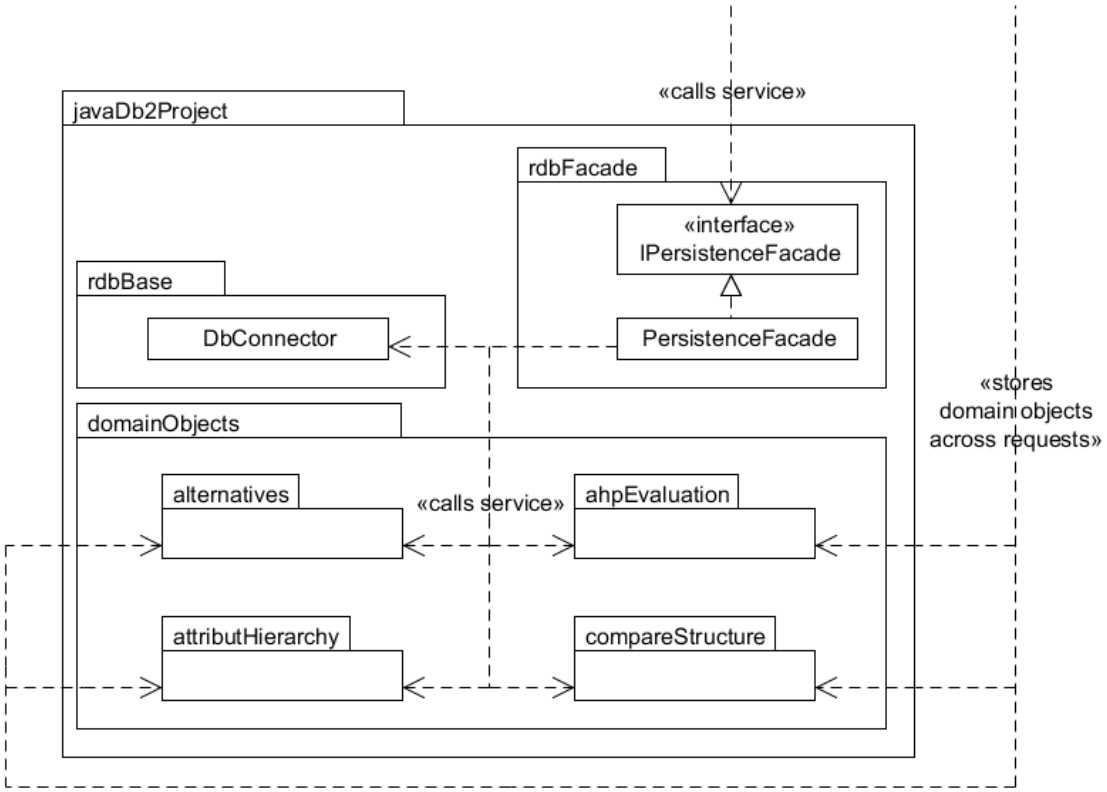


Abbildung 28 Paket-Diagramm der Datenbank-Schicht (DataBase-Layer) der Beispielanwendung AHP-WebApp

Weiter wird auf die Abbildung des AHP durch das Objektmodell und die Unterschiede zum Datenbankschema eingegangen, welche beide dem Anhang zu entnehmen sind. Eine Evaluation besteht aus mindestens einem Attribut, nämlich dem Hauptziel der Auswertung,

welches die Wurzel in der Attributhierarchie darstellt. Die Kriterien/Attribute einer Entscheidung müssen hierarchisch abgebildet werden, wobei ein Attribut, n weitere Attribute enthalten kann (Vergleich Abbildung 29 Nr. 1 mit Objektmodel Nr.1). Dazu wurde das Kompositum Entwurfsmuster innerhalb des Objektmodels verwendet [Eilebrecht2010]. Die Evaluation beinhaltet m Alternativen, die mit jedem Kriterium/Attribut n verglichen werden (Vergleich Abbildung 29 Nr. 2 Objektmodel Nr.2), um eine Rangfolge erstellen zu können.

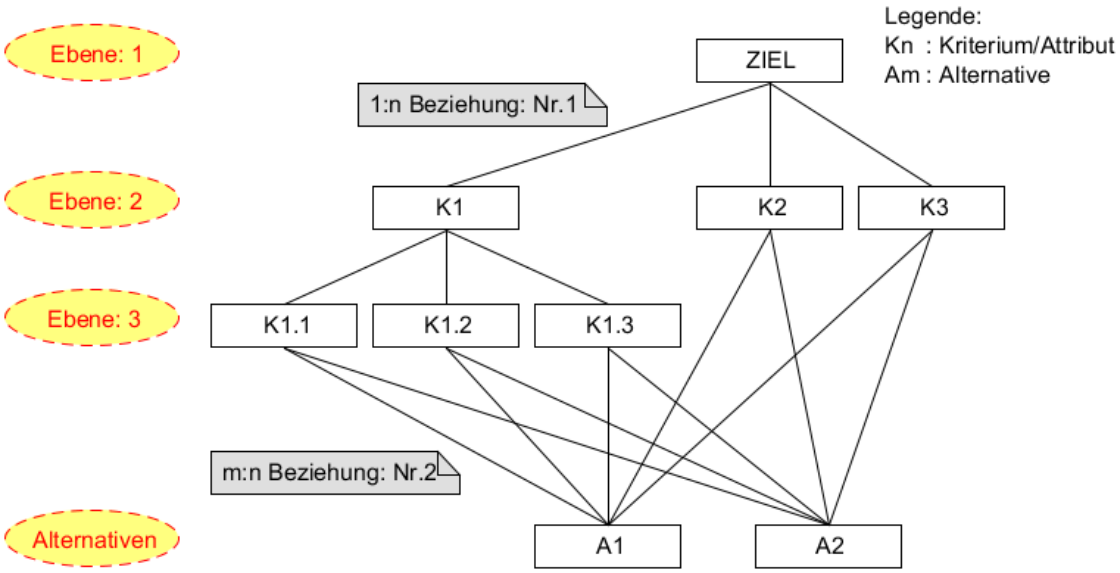


Abbildung 29 Dreistufiges AHP Model [Jayaswal2006]

Aus dem Vergleich der Attribute und Alternativen entsteht eine Vergleichsmatrix, die im Objektmodel mit der Abstraktion „AStructure“ abgebildet und spezialisiert werden kann (siehe „AlternativeStructure“ und „AttributStructure“ im Objektmodel). Diese zweidimensionale Struktur wird als Parameter an die Matrix-Klasse der Business-Layer zur Kalkulation der Gewichte übergeben. Die Matrix selbst besteht aus Elementen, welche ihre Position (x,y) innerhalb der Matrix selbst verwalten.

Im Datenbankschema hingegen wird auf die Persistenz der „AStructur“ verzichtet, da diese allein der Berechnung dienen. Allerdings ist die m:n Beziehung zwischen Alternativen und Sub-Attributen durch zwei 1:n Beziehungen aufgelöst worden. Weiter ist der Impedance

Mismatch, also die Unvereinbarkeit zwischen OOP und Relationaler-Datenbanken-Repräsentation [Steger2009], an der Abbildung der Hierarchie zu erkennen. Statt einer Vererbungsbeziehung enthält ein „Attribut“ das Feld „isLeaf“, welches zwischen Blättern und Knoten unterscheidet.

3.1.5.3 Projektkonfiguration mit Maven

Apache Maven ist ein Werkzeug zur Automatisierung und Standardisierung des Erstellungsvorganges (engl.: Build) von Anwendungssoftware. Dabei sind Abhängigkeiten zwischen erstellten Quelltexten und verwendeten Bibliotheken beim Übersetzen und Verlinken zu beachten. Voraussetzung für den Betrieb von Maven ist eine installierte Java-Laufzeitumgebung, da es sich bei Maven selbst um ein Stück Java-Software handelt. Im Gegensatz zu Apache Ant, welches auf eine detaillierte Beschreibung des Build-Prozesses angewiesen ist, beschränkt sich Maven auf die Angabe der wichtigsten Fakten eines Projektes, welche vom Standardfall abweichen. Der Vorteil dieses Ansatzes ist der wesentlich geringere Aufwand für eine Projektkonfiguration und ein einheitliches Vorgehen. Voraussetzung dafür ist, dass die verwendeten Technologien, Maven in Form eines Plugins unterstützen und die Maven-Konventionen einhalten. Neben der Erstellung kümmert sich Maven um die Verwaltung der internen und externen Abhängigkeiten, wie Projektsubmodulen oder verwendeten Bibliotheken. Sind Letztere nicht auf dem Entwicklungsrechner vorhanden, werden die Ressourcen aus Internetquellen (engl.: Repository) heruntergeladen und für den nächsten Gebrauch lokal gespeichert [Popp2009].

3.1.5.3.1 Die Projektkonfigurationsdatei: pom.xml

Damit eine Verarbeitung der Metainformationen des Projektes möglich ist, müssen diese in standardisierter Form vorliegen. Dies geschieht in der Projektkonfigurationsdatei, welche in XML verfasst ist und weiter pom.xml genannt wird.

```
1 <project ...>
2    <artifactId>DA-App</artifactId>
3    <groupId>de.kacperbak</groupId>
4    <version>0.6.0</version>
5    <packaging>pom</packaging>
6        ...
7    <build>
```

```
8       <plugins>
9         <plugin>
10            <artifactId>maven-compiler-plugin</artifactId>
11            <groupId>org.apache.maven.plugins</groupId>
12            <version>2.3.2</version>
13            <configuration>
14              <verbose>true</verbose>
15              <fork>true</fork>
16              <compilerVersion>1.7</compilerVersion>
17              <executable>C:\Program Files\Java\jdk1.7.0_01\bin\javac</executable>
18            </configuration>
19          </plugin>
20                        ...
21       </plugins>
22    </build>
23      ...
24    <dependencies>
25      <dependency>
26        <groupId>junit</groupId>
27        <artifactId>junit</artifactId>
28        <version>4.8.1</version>
29        <scope>test</scope>
30      </dependency>
31    </dependencies>
32 </project>
```

Listing 99 Auszug aus der pom.xml des Hauptprojektes der Beispielanwendung: AHP-WebApp

Die pom.xml kann auch als Bauplan für eine Funktionseinheit/ ein Modul verstanden werden (siehe Listing 99), dessen Ergebnis nach der Ausführung durch Maven, ein Artefakt darstellt. Es handelt sich dabei um fertige auslieferbare Produkte, die z.B. auf einem Webserver aufgespielt werden können. Der Typ des Artefaktes wird durch das Tag „package“ bestimmt. Ist es beispielsweise das Ziel eine Bibliothek auszuliefern, trägt der Anwender den Wert „jar“ (Java Archive) ein. Die Tags „artefactId“, „goupId“ und „version“ dienen der eindeutigen Identifikation des Artefaktes.

Um auf den Build-Prozess Einfluss nehmen zu können, bedarf es der Integration eines Plugins der verwendeten Technologie. Dieses wird innerhalb des „build“ Tags eingefügt. In diesem Fall (siehe Listing 99, Zeile 16) ist es Ziel, dass alle Java-Quelltextdateien mit dem neusten Java 1.7 zu übersetzen, während Maven selbst mit einer unterschiedlichen Java-Version arbeitet [Apache2012d]. Für diesen Zweck wird das „maven-compiler-plugin“ eingefügt, das der identischen ID-Konvention von „artifactId“, „groupId“ und „version“ folgt, wie die Projektkonfigurationsdatei selbst. Die Angaben über die abweichende Übersetzer-Version erfolgen in den Tags „compilerVersion“ und „executable“.

Als letztes wird der Mechanismus zur Verwaltung der Abhängigkeiten erwähnt, welcher mit dem „dependency“ Tag eingesetzt wird, und die bereits genannte ID-Konvention zur Unterscheidung der Artefakte verwendet. Maven unterscheidet dabei nicht, ob es sich bei der Abhängigkeit um eine externe Bibliothek oder ein Teil- bzw. Sub-Modul des Projektes handelt [Popp2009].

3.1.5.3.2 Standardisierte Verzeichnisstruktur

Die Projektkonfigurationsdatei wiederum ist Bestandteil einer standardisierten Verzeichnisstruktur (Abbildung 30), welche von Maven selbst vorgegeben wird und an die sich das Projekt halten soll.

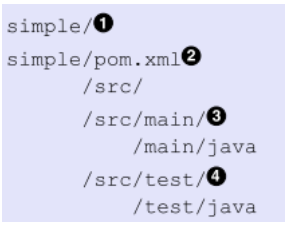


Abbildung 30 Maven standard Verzeichnisstruktur [O’Brien2009]

1. Stellt das Hauptverzeichnis dar, welches dem Wert des „artefactId“ Tags entspricht.
2. Projektmodel-Datei mit Metainformationen über das Projekt.
3. Dieses Verzeichnis enthält Java-Quelltextdateien.
4. Entspricht der gespiegelten Projektstruktur für Testfälle, siehe mehr unter Kapitel 3.2.4.

Angewendet wird Maven über die Kommandozeile oder über die Integration innerhalb einer Entwicklungsumgebung. Im Falle von Abbildung 30 ist die Erstellung des Projektes mit den zwei folgenden Befehlen zu bewerkstelligen.

```
> cd simple
> mvn install
```

Listing 100 Wechsel in das Projektverzeichnis und Maven-Befehl zur Auslieferung der fertigen Artefakte in das lokale Repository [O’Brien2009]

Dabei wechselt der erste Befehl in das aktuelle Projektverzeichnis, während der zweite die Erstellung der Applikation, über den gesamten Standard-Entwicklungslebenszyklus, wie ihn Maven definiert, anstößt. Bei dem Zyklus handelt es sich um ein Durchlaufen von definierten

Phasen, deren Ziele durch die Konfiguration der bereits genannten Maven-Plugins beeinflusst werden können.

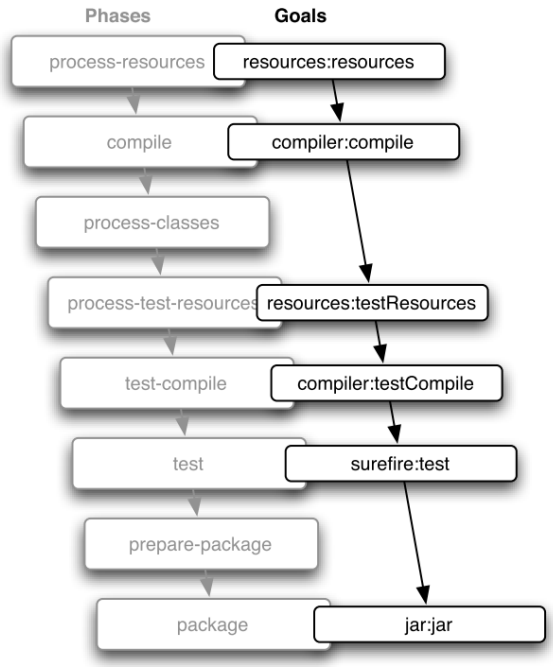


Abbildung 31 Maven Lifecycle [O'Brien2009]

Die Abarbeitung der Phasen erfolgt sequentiell. Den einzelnen Phasen werden beim „Lifecycle-Mapping" den Build-Zielen zugeordnet. Das Resultat ist der konkrete Build-Prozess [Popp2009]. Der Ablauf beginnt mit einer Validierung der Projektintegrität, gefolgt von einer Verpackung der Ressourcen und Übersetzung der Quelltexte. Nachdem die Testfälle alle erstellt und erfolgreich abgeschlossen wurden, erfolgt die Bündelung und Prüfung des Artefaktes. Der letzte Schritt kann eine Auslieferung in ein lokales oder ein remote-Repository im Internet sein [O'Brien2009].

3.1.5.3.3 Das Multimodulprojekt

Die in den vorhergehenden Kapiteln erstellten Schichten sollen nun mittels Maven zu einem Gesamtprojekt zusammengeführt werden, um den Erstellungsprozess der Anwendung zu vereinfachen. Jede Schicht stellt dabei ein einzelnes Maven-Modul mit einer eigenen Projektkonfigurationsdatei dar. Dazu wird ein übergeordnetes Wurzelprojekt erstellt, welches die einzelnen Schichten als Submodule einbindet. Wie in Listing 101 zu erkennen ist, werden die einzelnen Schichten im „modules" Tag des Wurzelverzeichnisses angegeben. Da es sich um ein Projekt handelt, welches der Verwaltung von Submodulen dient, wird das „packaging" Tag auf den Wert „pom" gesetzt. Das Wurzelprojekt bietet auch die Mögichkeit globale Abhängigkeiten, „dependencies", die allen Submodulen zur Verfügung stehen sollen, einzubinden. Geeignet dafür ist die Einbindung des JUnit-Testframeworks für alle

Projektbestandteile. Die Umsetzung dieser Eigenschaften erfolgt in der pom.xml des Wurzelprojektes, wie Listing 101 veranschaulicht [O'Brien2009].

```
1 <project ...>
2       ...
3   <artifactId>DA-App</artifactId>
4   <groupId>de.kacperbak</groupId>
5   <version>0.6.0</version>
6       ...
7   <packaging>pom</packaging>
8   <name>AHP-WebApp</name>
9       ...
10   <modules>
11      <module>wicketProject</module>
12      <module>scalaProject</module>
13      <module>javaDb2Project</module>
14   </modules>
15      ...
16   <dependencies>
17      <dependency>
18         <groupId>junit</groupId>
19         <artifactId>junit</artifactId>
20         <version>4.8.1</version>
21         <scope>test</scope>
22      </dependency>
23   </dependencies>
24      ...
25 </project>
```

Listing 101 pom.xml des Wurzelprojektes: DA-App (alias AHP-WebApp)

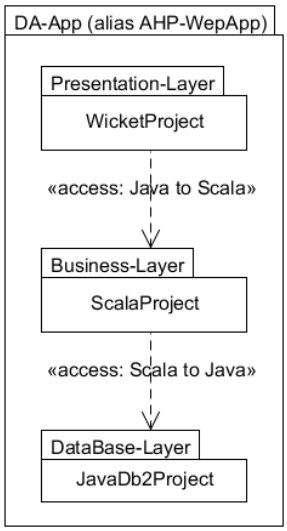
Damit Maven erkennt, zu welchem übergeordneten Projekt die Submodule gehören, erfolgt dessen Angabe mittels der ID-Konvention innerhalb des „parent" Tags (siehe Listing 102). Das oberste Sub-Modul der AHP-WebApp wird von einem Webserver ausgeführt. Dieser nimmt Module in Form des - Web application Archive - Formates (kurz „war") an. Damit dieses Projekt während der Übersetzungsphase in das benötigte Format erstellt wird, ist dessen Angabe „war" im „packaging" Tag nötig. Die Webanwendung, also das Wicket-Projekt, greift auf die darunterliegenden Schichten, wie auf Bibliotheken, zu. Deren Wert im „packaging" Tag ist deshalb „jar". Als letztes muss Maven die Abhängigkeit der internen Sub-Module untereinander bekannt gemacht werden [Popp2009].

```
1 <project ...>
2       <parent>
3             <artifactId>DA-App</artifactId>
4             <groupId>de.kacperbak</groupId>
5             <version>0.6.0</version>
```

```
6       </parent>
7       ...
8       <artifactId>wicketProject</artifactId>
9       <groupId>de.kacperbak</groupId>
10      <version>0.6.0</version>
11      <packaging>war</packaging>
12      <name>wicketProject</name>
13      ...
14      <dependencies>
15              <dependency>
16                      <artifactId>scalaProject</artifactId>
17                      <groupId>de.kacperbak.scalaProject</groupId>
18                      <version>0.6.0</version>
19              </dependency>
20      </dependencies>
21 </project>
```

Listing 102 pom.xml der Webanwendung: WicketProject

Dies erfolgt mit dem identischen Mechanismus, den Maven auch für externe Bibliotheken verwendet. In Listing 102 ist zu erkennen, wie die Abhängigkeit zwischen Webanwendung und der Business-Schicht in Scala, über das „dependency“ Tag angegeben wird. Die nachfolgende Tabelle gibt einen Auszug des Gesamtüberblicks über die Maven-Konfiguration der Beispielanwendung „AHP-WebApp“.



Projektname:	**packaging-Tag:**	**dependency-Tag:**
DA-App	pom	JUnit (Testframework)
WicketProject	war	ScalaProject
ScalaProject	jar	JavaDb2Project
JavaDBProject	jar	db2jcc (Datenbanktreiber)

Listing 103 Auszug Maven-Projektkonfiguration: AHP-WebApp

Die Business-Schicht in Scala stellt eine Abweichung vom Standard im Entwicklungsprozess dar. In diesem Fall muss eine Anpassung mittels entsprechendem Plugin erfolgen, um die

Funktion mit Maven zu gewährleisten. Auf die Konfiguration dieses Plugins wird in Kapitel 3.2.1 eingegangen. Um den Build-Prozess nun anzustoßen, genügt die Ausführung des folgenden Befehls auf dem Wurzelprojektverzeichnis:

```
1 ...\DA-App>mvn clean install
2 ...
3 [INFO] Reactor Build Order:
4 [INFO]
5 [INFO] DA-App
6 [INFO] javaDb2Project
7 [INFO] scalaProject
8 [INFO] wicketProject
9 ...
```

Listing 104 Erstellung der Beispielanwendung AHP-WebApp mittels Maven

Aus Listing 104 ist ersichtlich, wie ein modulares Projekt vom Maven-Reactor untersucht wird. Dieser erstellt aus den Abhängigkeiten der Submodule zueinander, die Reihenfolge in der diese erstellt werden. Der „clean“ Befehl stellt dabei sicher, dass vorangegangene Produkte gelöscht und damit eine konfliktfreie Installation in das lokale Repository ausgeliefert wird [O’Brien2009].

3.1.5.3.4 Fazit: Maven

Das beschriebene Vorgehen erlaubt eine strikte Trennung zwischen Java- und Scala-Bereichen innerhalb der Anwendung, die auch getrennt voneinander erstellt werden. Gleichzeitig ist die nahtlose Integration in den Maven-Entwicklungszyklus gewährleistet. Die hier beschrieben Projektstruktur ist natürlich keine zwingende Vorlage. Sie kann aber als Beispielmuster für andere Projektstrukturen verwendet werden. Damit sind alle Ziele der Projektkonfiguration bezüglich der Scala-Kompatibilität mit Maven erfüllt.

3.1.6 Ergebnisse

3.1.6.1 Umsetzung einer modularen Java/Scala-Webanwendung

3.1.6.1.1 Probleme mit der Serialisierbarkeit

Bei der Implementierung der 3-Tier-Architektur traten Probleme mit der Akzeptanz der „java.io.Serializable“ Schnittstelle auf, welche zur Speicherung der Fachobjekte von diesen

implementiert werden muss. Im ersten Prototyp der Anwendung scheiterte der Versuch die Fachobjekte in der Business-Layer zu implementieren. Theoretisch stellt Scala die mächtigere Sprache dar, welche keine Probleme hat, Konzepte aus Java und somit auch Java-Frameworks zu übernehmen (Zugriff: Scala -> Java). In diesem Fall griff aber das Wicket-Framework auf Scala-Fachobjekte in der Business-Layer zu (Zugriff: Java -> Scala). Der Fehler äußerte sich in einer „java.io.NotSerializableException“, welche erst zur Laufzeit der Anwendung auftrat, obwohl die Fachobjekte die Schnittstelle „java.io.Serializable“ implementierten. Daraufhin wurde diese durch ihr Scala-Pendant „scala.Serializable“ ausgetauscht, was den Fehler jedoch nicht beseitigte. Aus diesem Grund fiel die Entscheidung, die Fachobjekte innerhalb der Datenbank-Schicht zu implementieren, welche in Java geplant und damit keine Probleme mit der Serialisierungsschnittstelle zu erwarten waren. Damit musste jedoch der Grundsatz der 3-Tier-Architektur gebogen werden, dass jede Schicht auf die darunterliegende zugreift. Die Presentation-Layer griff nun direkt, ohne durch die Business-Layer delegiert zu werden, auf die Datentypen in der DataBase-Layer zu.

Nachdem die Beispielanwendung fertig implementiert war, stieß der Verfasser dieser Arbeit auf einen Artikel [Wähner2012] indem beschrieben wurde, dass die Kapselung von Objekten durch Schnittstellen viele Probleme in heterogenen Projekten löst. Zu Testzwecken wurde die Klasse „BusinessAhpEvaluation“ als Domain-Objekt in der Scala-Schicht erstellt, welche von einem Trait „TBusinessAhpEvaluation“ gekapselt und die Implementierung der Serializable-Schnittstelle in der Business-Schicht erfolgt. Abbildung 32 verdeutlicht dabei das provisorische Vorgehen, welches den gewünschten Effekt erzielt, jedoch die Kapselung „AAhpEvaluation“ aushebelt, indem im konkreten Typ „BusinessAhpEvaluation“ der Super-Konstruktor-Aufruf von „AhpEvaluation“ erfolgt. Im Vergleich zu den übrigen Fachobjekten verbleiben diese in der Datenbank-Schicht und sind weiterhin durch die Erweiterung ihrer zugehörigen abstrakten Klasse gekapselt.

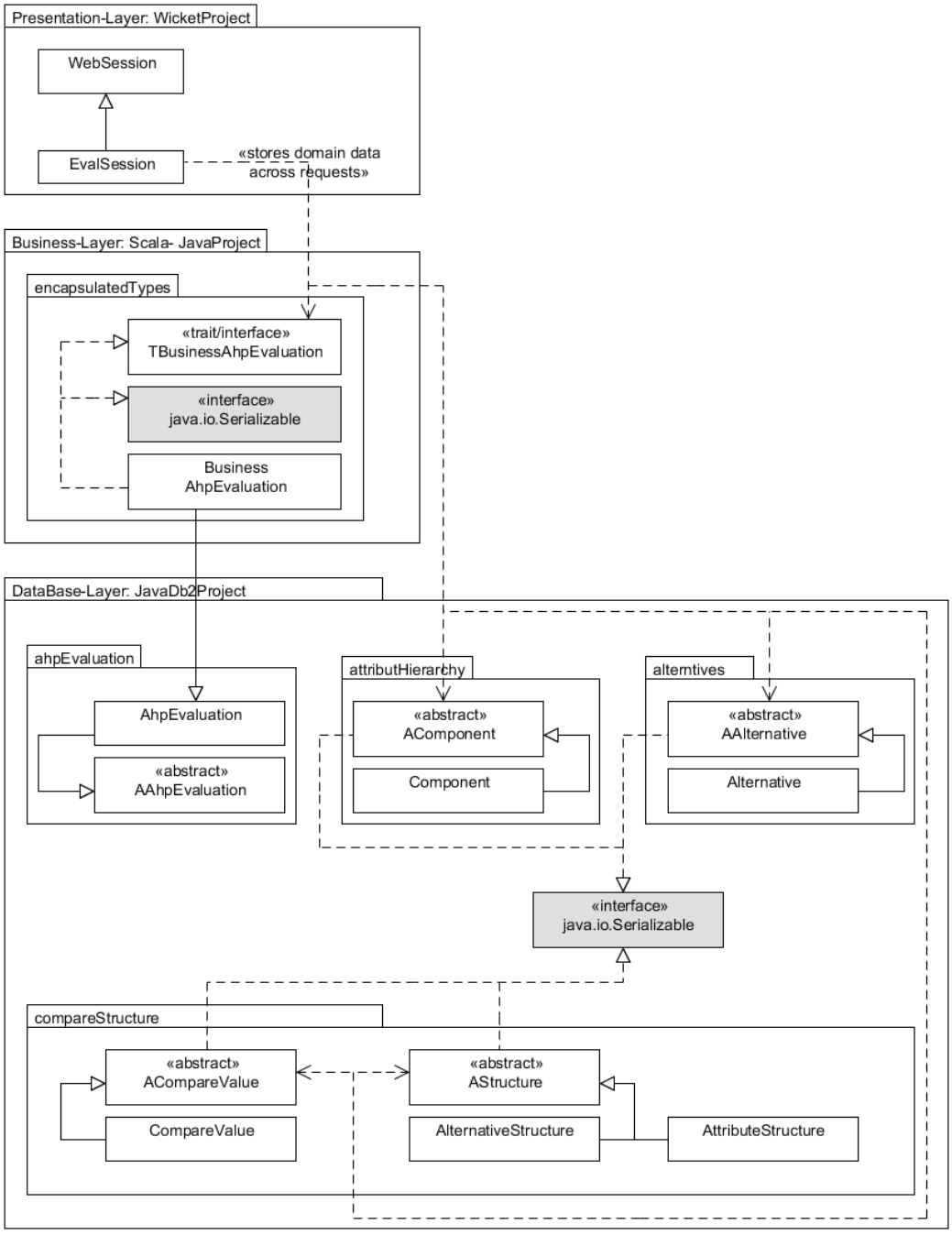


Abbildung 32 Kapselung des Fachobjektes „AhpEvaluation“ in „BusinessAhpEvaluation“ zum Test der Implementierung von „java.io.Serializable“ in der Scala-Business-Layer

Mit der Erscheinung neuer Fachliteratur wurde versucht eine konkrete Ursache für die Probleme mit der Serialisierung herauszufinden. So ergab die Recherche, dass die Scala Version 2.7 sehr große Probleme mit der Serialisierung aufwies, diese aber in der

nachfolgenden Version verbessert wurden. Aus diesem Grund wird auch der Einsatz der neuesten Scala-API dringend empfohlen. Weiter wurden Probleme mit anonymen Klassen geschildert, welche in vielen Fällen, jedoch nicht für Closures innerhalb von Objekten, vermieden werden können [Suereth2011].

3.1.6.1.2 Modulkapselung

Es stellt sich also die Frage: Wie kapselt der Entwickler ein Scala-Modul ab, welches problemlos in eine Java-Anwendung integriert werden kann? In Kapitel 3.2.1 wird aufgezeigt, wie sich ein Scala-Modul in einen Maven-Build einbinden lässt. Es folgt die Beschreibung von Traits, deren Theorie in Kapitel 2.2.1.14 zu entnehmen ist und die weiter als Schnittstelle zwischen beiden Sprachen betrachtet werden.
Enthält ein Trait rein abstrakte Methoden, also nur die Signaturen ohne Implementierung, und keine deklarierten oder definierten Felder, wird dieser in identischen Bytecode übersetzt, wie eine Schnittstelle aus Java [Odersky2010]. Diese Aussage kann auch überprüft werden, indem der erzeugt Bytecode eines Java-Interfaces und eines Scala-Traits miteinander verglichen werden. Listing 105 zeigt hierfür die Java-Schnittste „IJava“ und das Scala-Pendant „TScala“, welche in der Praxis natürlich nicht an gleicher Stelle definiert sein würden.

```
1 public interface IJava{
2         public void foo(int i);
3         public String bar(java.util.ArrayList<java.lang.Integer> integerList);
4 }
5
6 trait TScala{
7         def foo(i: scala.Int)
8         def bar(integerList: java.util.ArrayList[scala.Int]): String
9 }
```

Listing 105 Quelltext Vergleich: Java-Interface zu Scala-Trait

Nachdem die beiden Abstraktionen mit dem passenden Compiler übersetzt worden sind, werden die resultierenden *.class Dateien mit dem Java-Decompiler untersucht [Oracle Corp.2012b].

```
1 C:\...>javap -c IJava
2 Compiled from "IJava.java"
3 public interface IJava{
```

```
4 public abstract void foo(int);
5
6 public abstract java.lang.String bar(java.util.ArrayList);
7
8 }
9
10 C:\...>javap -c TScala
11 Compiled from "TScala.scala"
12 public interface TScala{
13 public abstract void foo(int);
14
15 public abstract java.lang.String bar(java.util.ArrayList);
16
17 }
```

Listing 106 Bytecode-Vergleich: Java-Interface zu Scala-Trait

Auf den ersten Blick erscheint das Ergebnis zufriedenstellend. Erst bei der Anwendung zeigen sich folgende Probleme auf. Wie schon erwähnt ist in Scala bekanntlich alles ein Objekt, diese Sicht ist für den Entwickler bequem. Tatsächlich behandelt Scala jedoch seinen Datentyp „scala.Int“ identisch zu Java, leistungsbedingt als 32bit Word. Das Prinzip des Boxing und Unboxing, welches in Java recht offen gehandhabt wird, ist in Scala vollkommen verborgen [Odersky2010]. Listing 107 demonstriert, wie in Java aus einem Ganzzahlenwert ein Wrapper-Objekt erstellt wird (Boxing) und die Umkehroperation (Unboxing) [Ullenboom2011].

```
1 int i          = 4711;
2 Integer j      = i;        // steht für  j = Integer.valueOf(i)     (Boxing)
3 int k          = j;        // steht für  k = j.intValue()           (Unboxing)
```

Listing 107 Demonstration des Boxing und Unboxing in Java, zwischen Primitiven und Objekten

Nur an den notwendigen Stellen wie der Ausführung einer Methode auf einem Ganzahlentyp - 1.toString() - , wird das Verfahren in Scala angewendet. Betrachtet wird nun mit diesen Erkenntnissen erneut das Listing 106. In Zeile 4 und 13 wurde aus beiden Sprachen, die Methode „foo“ und der Parameter-Datentyp „int“ im Bytecode erstellt. Problematischer ist der Einsatz von Primitives in generischen Parametern. Zeile 3 und 8 in Listing 105 zeigen beide eine typisierte Liste, jedoch mit unterschiedlicher generischer Parametrisierung. In Java muss der Parametertyp eines Generics, ein Objekt sein und kein Primitive. Aus diesem Grund wird die Zahl durch ein Objekt umhüllt (engl.: Wrap). Dies erklärt den Einsatz von Wrapper-Datentypen, wie „java.lang.Integer“, in Java. In Scala hingegen ist für den Compiler

alles ein Objekt, bzw. wird dazu gemacht. Der Bytecode beider Sprachen sieht jedoch identisch aus, da er diese Information nicht enthält, siehe Listing 106 Zeile 6 und 15. Der folgende Quelltext ruft aus Java heraus, die in Scala definierte Abstraktion „TScala“ auf und übergibt einen Wrapper-Typ als generischen Parameter.

```
1 TScala tscala = new ...          //Konkrete Klasse welche TScala implementiert
2 tscala.foo(5);
3 tscala.bar(new ArrayList<java.lang.Integer>);
```

Listing 108 Zugriff über den Trait „TScala“

Der Aufruf von Zeile 2 funktioniert problemlos, in Zeile 3 setzt jedoch die Interoperabilität der beiden Sprachen aus. Hier aber kann der Wrapper-Datentyp „java.lang.Integer“ nicht als Objekt behandelt werden, da in der „bar“-Methode des Scala-Traits, ein „scala.Int“ als generischer Typparameter steht, welcher intern als Java-Primitive umgesetzt wird (siehe Bytecodevergleich Listing 106 Zeile 4 und 13). Der Datentyp des übergebenen Parameters (Objekt) passt also nicht zur Typdefinition (Primitive) in der Methode „bar“.

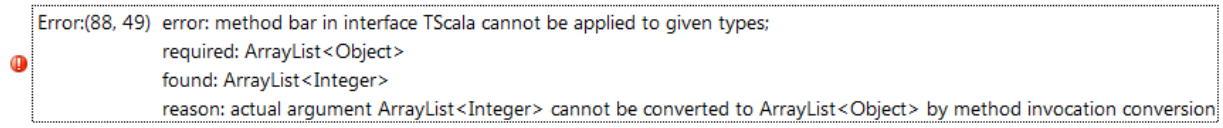


Abbildung 33 Fehler: Ein „java.lang.Integer“ kann nicht als Objekt verwendet werden

Um Probleme mit Primitives in generischer Parametrisierung zu umgehen, können Implicits definiert werden, deren Aufgabe das Umwandeln (engl. Casten) von Wrapper-Objekt in Scala-Primitive ist.

```
1 implicit def convertToScala(x: java.util.ArrayList[java.lang.Integer]) = {
2         x.asInstanceOf[java.util.ArrayList[scala.Int]]
3 }
```

Listing 109 Definition einer Implicit Methode, welche auf Scala-Seite Wrapper-Typen in Scala-Primitives castet

Wird nun innerhalb von Scala-Code auf ein Java-Interface zugegriffen, behandelt der Compiler das Wrapper-Objekt als Scala-Primitive, anstatt einen Fehler anzuzeigen [Suereth2011]. Umgekehrt, wenn der Zugriff von Java auf einen Scala-Trait erfolgt, kann die Typumwandlung in einer Methode des Traits definiert werden. Gecastet wird dann mittels dieser Methode, wie der folgende Java-Quelltext zeigt.

```
1 TScala tscala = new ...                    //Konkrete Klasse, welche TScala implementiert
2 tscala.bar(tscala.convertToScala(new ArrayList<java.lang.Integer>()));
```

Listing 110 Anwendung der Implicit-Methode auf Java-Seite

Die dritte Möglichkeit ist in beiden Sprachen für die generische Parametrisierung, ausschließlich Wrapper-Typen, wie „java.lang.Integer“, einzusetzen. Diese sind in der Java-API enthalten und können auf Scala-Seite importiert werden. Damit wird immer, auch in Scala, ein genetischer Typparameter als Objekt verwendet.

Bei der normalen Argumentübergabe hingegen wird der Einsatz von Primitives in beiden Sprachen als Parameter bevorzugt [Suereth2011]. Diese werden in identischen Bytecode übersetzt, wie man an der Methode „foo“ aus Listing 106 Zeile 4 und 13 sehen kann. Identisch dazu verhält sich die Übersetzung des Scala-Traits als auch des Java-Interfaces, die in ein „public interface“ übersetzt werden. Somit steht dem Entwickler eine gemeinsame Schnittmenge zwischen beiden Sprachen zur Verfügung, um Funktionalität und Daten zu kapseln und eine hybride Java-Scala-Anwendung zu entwerfen.

3.1.6.1.3 Fazit der Integration

Scala bietet eine Menge an Eigenschaften, welche in Java nicht vorhanden sind. Dazu zählen Closures, Traits, implizite Typ-Auflösung usw. Anders als vermutet, können aber nicht alle Merkmale von Java in Scala abgebildet werden. Dazu zählen statische Felder in Klassen, welche bereits erstellt werden, wenn die Klasse geladen wird. Diese sind ohne die Instanzierung der Klasse verfügbar, was in Scala nicht möglich ist, da jeder Wert ein Objekt ist. Dabei handelt es sich nur solange um ein theoretisches Problem, bis eine unverzichtbare Bibliothek genau auf diesen Umstand stößt. Weiterhin schwierig sind Abhängigkeiten zum Scala-Modul, wie die erwähnte Serialisierung oder Bibliotheken, welche Annotationen benutzen, da diese nicht in allen Fällen in identischen Bytecode übersetzt werden [Suereth2011].

Als Erkenntnis kann aus der Umsetzung der 3-Tier-WebAnwendung gewonnen werden, dass Java-Scala-Interoperabilität nicht wie beworben nahtlos ist. Treten Probleme auf, sind die Ursachen nicht trivialer Natur und müssen zeitaufwendig behoben werden. Mit Interfaces können komplexe Sprachfeatures von Scala gekapselt werden, die sich in separate Module

überführen lassen. Es wird aber aus beschriebenen Gründen dringend empfohlen, diese Module unidirektional einzubinden. Diese sollten eine spezifische Aufgabe entgegen nehmen, sie lösen und ein Ergebnis zurückliefern, ohne dabei von kritischen Anwendungsteilen abhängig zu sein.
Der Erfolg einer Scala-Modul-Einbindung ist maßgeblich von der Kompatibilität, der in Beziehung stehenden Module abhängig. Die frühzeitige Umsetzung eines Prototypen ist deshalb von größter Bedeutung.

3.1.6.2 Scala spezifische Anwendungsfälle

3.1.6.2.1.1 Parallelisierung mit dem Fork/Join-Framework

Mit Scala 2.9 ist ein neues Merkmal, die Parallel-Collections, in die Sprache eingeführt worden. Voraussetzung dafür ist ein JRE 1.6 oder besser ein Java 7, da die neuere Version das benötigte Fork/Join-Framework ohne zusätzliche Einbindung mitbringt. Dieses Framework ermöglicht die Parallelisierung von Algorithmen, ohne dass der Entwickler selbst sich um die Verwaltung von Threads kümmern muss. Voraussetzungen für den Einsatz sind, dass der Algorithmus parallelisierbar ist und dass die Rechenoperation aufwendig genug ist, um den Overhead der Lastverteilung zu kompensieren [Haase2011a].

Basis für die Parallelisierung bildet das Fork/Join-Framework, welches in Java direkt angesprochen wird. Verwaltet wird damit ein Thread-Pool, welcher erstellte Worker-Threads im „sleep“ Zustand verwaltet und bei Bedarf einsetzt. Hintergrund dieses Ansatzes ist die Vermeidung zusätzlichen Overheads bei der wiederkehrenden Allokation neuer Threads. Die zu verrichtende Aufgabe wird in der Abstraktion „Task“ definiert, welche sich selbst in neue Teilaufgaben „fork“ zerlegen lässt und abschließend das Gesamtergebnis „join“ zusammenführt [Prokopec2010]. Eine Task wird dabei vollständig in eine Queue eines beliebigen Threads platziert. Danach erfolgt die bereits beschriebene Zerlegung, bis das Problem sequentiell lösbar ist. Hat ein Thread seine Queue abgearbeitet, holt sich dieser eine weitere Aufgabe vom Ende des Queues eines anderen Threads. Damit entsteht eine unkomplizierte Lastverteilung, da selten ein Konflikt um die Ressource „Task“ entsteht [Haase2011a].

Betrachtet wird zunächst der Einsatz des Frameworks unter Java 7, um anschließend einen Vergleich mit der Scala-Implementierung zu ziehen. Als Beispiel wird die Matrixpotenzierung aus Kapitel 3.1.4.2.5 herangezogen, da sich deren Problemgröße flexibel skalieren lässt. Die zu berechnende Aufgabe wird innerhalb der „compute“ Methode einer Klasse definiert, die um die Klasse „RecursiveTask“ erweitert wird, wie das folgende Listing verdeutlicht [Eisele2011b].

```
1 public class ForkJoinTask extends RecursiveTask<ArrayList<ArrayList<BigDecimal>>> {
2
3    private static final int TRESHOLD = 6;
4    private int start;
5    private int end;
6
7        /* ctor */
8        public ForkJoinTask(     int start,
9                                 int end,
10                                ArrayList<ArrayList<BigDecimal>> inputStructure,
11                                ArrayList<ArrayList<BigDecimal>> resultStructure
12       ) {...}
13
14    @Override
15    public ArrayList<ArrayList<BigDecimal>> compute() {
16
17        if (end - start <= TRESHOLD) {
18
19            for (int i = start; i < end; i++) {
20                for (int j = 0; j < dimension; j++) {
21
22                    horizontalList = createHorizontalList(i);
23                    verticalList = createVerticalList(j);
24                    BigDecimal sum = new BigDecimal(0.0);
25
26                    for (int k = 0; k < dimension; k++) {
27                        sum = sum.add(horizontalList.get(k).multiply(verticalList.get(k)));
28                    }
29                    resultStructure.get(i).set(j, sum);
30                }
31            }
32            return resultStructure;
33
34        } else {
35
36            int mid = start  + (end - start) / 2;
37
38            ForkJoinTask t1 = new ForkJoinTask(start, mid, this.inputStructure,this.resultStructure);
39            ForkJoinTask t2 = new ForkJoinTask(mid, end, this.inputStructure,this.resultStructure);
40
41            t1.fork();
42            t2.compute();
```

```
43              t1.join();
44
45              return resultStructure;
46          }
47      }
48 }
```

Listing 111 Einsatz des Fork/Join-Frameworks zur Parallelisierung der Matrix-Potenzierung

Der zu verrichtende Algorithmus muss dazu in einer rekursiven, sprich funktionalen Form vorliegen. Der imperative Java-Entwickler muss an dieser Stelle seine gewohnte Denkweise verlassen und seine Lösung in eine Form mit Rekursionsanfang und Rekursionsschritten überführen. Dies ist aufgrund der Äquivalenz der Lambda-Kalkül zur Turing-Maschine für jede Art von Java-Algorithmus möglich [Piepmeyer2010]. Die eigentliche Aufgabe wird innerhalb des if{...}-Blockes (Listing 111, Zeile 17) implementiert und ausgeführt, wenn die Aufgabe klein genug ist. Bei der Aufgabe selbst handelt es sich um die Berechnung des Skalarproduktes, welches in der Business-Layer ausgeführt und in Kapitel 3.1.5.2.1.2 definiert ist. In Zeile 22 und 23 werden dafür die Zeilen- und Spaltenvektoren gebildet, um anschließend in Zeile 27 berechnet zu werden. Die internen Variablen „inputStructure“ und „resultStructure“ bilden dabei die Eingabe- und Ergebnisdatenmenge einer zweidimensionalen „ArrayList“ ab, auf denen die Berechnung der Matrixelemente abläuft. Der Grenzwert „THRESHOLD“ bestimmt, ab welcher Matrixgröße die Aufgabe auf mehrere Threads verteilt und somit parallel ausgeführt wird. In diesem Fall wird über die Dimension der Matrix skaliert. Bis zu einer Matrixgröße des Ranges sechs wird die Aufgabe sequentiell berechnet (siehe Zeile 3). Darüber wird die Berechnung über ihren Lauf-Index aufgeteilt, else{...}-Block (ab Zeile 36), und es werden neue Instanzen der Klasse selbst, mit verteilter Arbeit erstellt. Die Methode „fork“ erstellt einen neuen Subtask und startet diesen asynchron, während „join“ das Gesamtergebnis zusammenführt und sicherstellt, dass die Ausführung anhält, bis der Subtask abgeschlossen wurde [Eisele2011b].

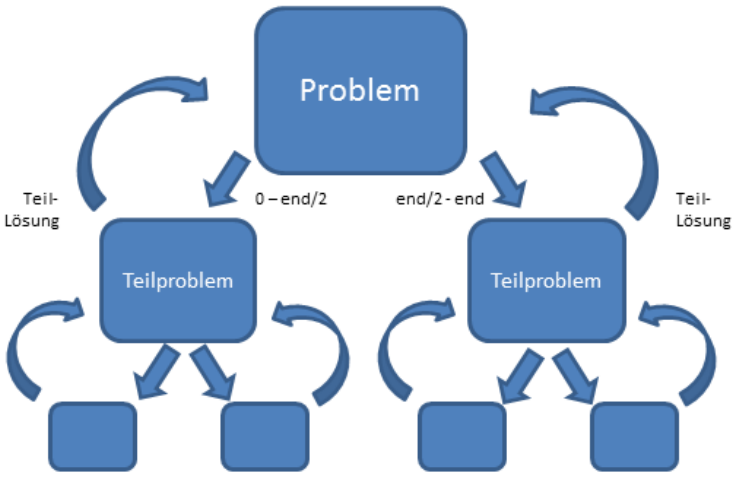


Abbildung 34 Fork/Join-Framework nach dem „divide and conquer“ Prinzip

Abbildung 34 stellt das angewandte Prinzip „Teile und herrsche“ (engl.: Divide And Conquer) plastisch dar [Eisele2011b]. Abschließend muss der beschriebene Thread-Pool erstellt, die Aufgabe übergeben und aufgerufen werden.

```
1 ForkJoinPool pool = new ForkJoinPool(Runtime.getRuntime().availableProcessors());
2 ForkJoinTask task = new ForkJoinTask(0,dimension,dataStructure,resultStructure);
3 pool.invoke(task);
```

Listing 112 Erstellung und Anwendung einer „ForkJoinTask“ Instanz

In Scala ist das Fork/Join-Framework direkt in die Parallel-Collections integriert. Die Handhabung der Tasks und der direkte Einfluss auf die Lastverteilung sind noch stärker gekapselt, als im vorhergehenden Beispiel. Betrachtet wird dafür die Erstellung und der Umgang mit Parallel-Collections, deren interne Arbeitsweise sowie die Anwendung auf das AHP-Beispiel, um einen direkten Vergleich zu ermöglichen.

```
1 1 val v1 = Vector(1,2,3,4,5,6,7,8)
2 2 val vp1 = v1.par
3 3 def process(i: Int) = println("process: " + i)
4
5 v1: scala.collection.immutable.Vector[Int] = Vector(1, 2, 3, 4, 5, 6, 7, 8)
6 vp1: scala.collection.parallel.immutable.ParVector[Int] = ParVector(1, 2, 3, 4, 5, 6, 7, 8)
7 process: (i: Int)Unit
```

Listing 113 Erstellung und Umwandlung von Collections in Scala 2.9

In Zeile 1 von Listing 113 wird ein Vektor mit den Werten 1 bis 8 erstellt, welcher anschließend in eine Parallel-Collection mit der Methode „par“ gewandelt wird. Die dritte Zeile definiert eine Funktion, welche weiter als Parameter benutzt wird. Um den Unterschied

zwischen „v1“ und „vp1“ zu verdeutlichen führen wir nun die Funktion „process“ auf jedem Element der beiden Collections aus.

```
1 v1.foreach(elem => process(elem))
2 process: 1
3 process: 2
4 process: 3
5 process: 4
6 process: 5
7 process: 6
8 process: 7
9 process: 8
```

```
1 vp1.foreach(elem => process(elem))
2 process: 7
3 process: 1
4 process: 3
5 process: 5
6 process: 4
7 process: 2
8 process: 8
9 process: 6
```

Listing 114 Anwendung der „foreach“ Methode

Wie zu erwarten, erfolgt die Bearbeitung der Elemente in „vp1“ willkürlich, da das Fork/Join-Framework das Thread-Scheduling, also die Reihenfolge der Bearbeitung, bestimmt. Parallel Collections sind mittels „Splitable“ Iteratoren umgesetzt, welche sich selbst teilen können. Dabei übernehmen sie einen Teil des Laufbereiches vom Ursprungs-Iterator. Die Operationen auf den Parallel-Collections sind als Aufgaben „Task“ definiert. Ebenso steht hier ein Pool aus Worker-Threads zur Verfügung, denen bei Bedarf ein Task zugewiesen wird. Dies lässt sich an der Methode „sum“ veranschaulichen, welche im Trait „TraversableOnce“ definiert und dem „Vector“ zur Verfügung steht [Suereth2011].

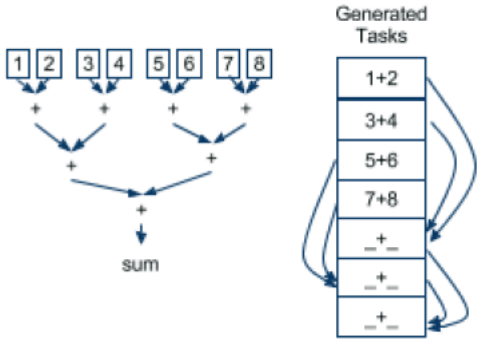


Abbildung 35 Aufteilung der Methode „sum“ in Aufgaben „Tasks“ und deren Zusammenführung

Nachdem nun der Zusammenhang des Verarbeitungsprinzips (Vergleiche Abbildung 34 mit Abbildung 35) zwischen Parallel-Collections und Fork/Join-Framework hergestellt wurde, kann dessen Anwendung auf das Beispiel der Matrix-Multiplikation erfolgen. Dazu wird das „for“ Statement, wie es aus Java bekannt ist, genauer betrachtet. Dieses wird intern in einen „foreach“ Methodenaufruf des List-Objektes umgesetzt, welches nicht zwangsweise eine

Collection sein muss [Prokopec2010]. Listing 115 verdeutlicht dabei wie der Rumpf (engl. Body) einer for-Schleife als Parameter eines Funktionsliterals (siehe Kapitel 2.2.2.2) angewendet wird.

```
1 for(x <- expr) body  wird übersetzt zu: expr foreach ( x => body )
```

Listing 115 Interne Umwandlung des „for“ Statements in Scala [Odersky2010]

Um über Zahlen, wie in gewöhnlichen for-Schleifen zu iterieren, wird deshalb eine Instanz der „Range“-Klasse erzeugt, welche die Ober- und Unter- Grenze des Iterationsbereichs sowie den Iterationsschritt selbst abbildet. Beispiele für die Erzeugung eines Laufbereiches (engl.: Range) sind „(1 to 5)“ oder mit ausgeschlossenem Intervall „(1 until 6)“. Analog existiert auch hier eine parallele Variante "ParRange". Diese Implementierung ermöglicht folgenden Einsatz.

```
1 for(i <- (1 to 8)) process(i)
2 process: 1
3 process: 2
4 process: 3
5 process: 4
6 process: 5
7 process: 6
8 process: 7
9 process: 8
```

```
1 for(i <- (1 to 8).par) process(i)
2 process: 1
3 process: 6
4 process: 7
5 process: 5
6 process: 8
7 process: 3
8 process: 2
9 process: 4
```

Listing 116 Anwendung der „par“ Methode auf eine for-Schleife

Die „split“-Methode innerhalb der ParRange-Klasse teilt den Iterationsbereich Start bis zur Hälfte und von der Hälfte bis zum Ende [Prokopec2010]. Der Algorithmus zur Berechnung der Matrix-Multiplikation kann dabei in seiner sequentiellen Form übernommen werden. Eine rekursive Anpassung, wie sie im Java 7 Beispiel zu sehen war, ist nicht nötig.

```
1 def multiplyMatrixParallel(): ScalaSquareMatrix = {
2         ...
3         for (i <- (0 until dimension).par) {
4           for (j <- (0 until dimension)) {
5
6                   val horizontalList: java.util.ArrayList[BigDecimal] = createHorizontalList(i)
7                   val verticalList: java.util.ArrayList[BigDecimal] = createVerticalList(j)
8                   resultStructure.get(i).set(j, calculateElementResult(horizontalList, verticalList))
9           }
10        }
11        ...
12 }
13
```

```
14 def calculateElementResult(
15                 horizontalList: java.util.ArrayList[BigDecimal],
16                 verticalList: java.util.ArrayList[BigDecimal]): BigDecimal = {
17
18       var sum = new BigDecimal("0.0")
19       for (i <- (0 until dimension)) {
20         sum = sum.add(horizontalList.get(i).multiply(verticalList.get(i)))
21       }
22       sum
23 }
```

Listing 117 Anwendung der „par“-Methode auf die Matrix-Multiplikation des AHPs

Identisch zum Java-Beispiel werden die Zeilen- und Spaltenvektoren in Zeile 6 und 7 erstellt und abschließend wird das Skalarprodukt in der Methode „calculateElementResult“ berechnet. Mit dem Aufruf der „par“ Methode in Zeile 3 ist der Algorithmus parallelisiert. Ein Leistungsvergleich beider Problemlösungen ist dem Kapitel 3.1.6.3.3.1 zu entnehmen.

Abschließend ist noch hinzuzufügen, dass vor der Umsetzung zuerst überlegt werden sollte, ob das Problem sich überhaupt parallelisieren lässt. Weiter ist festzustellen, ob auf dem Zielsystem freie Ressourcen in Form von unbeschäftigten Rechenkernen vorhanden sind. Ebenso ist vor dem Produktiveinsatz, eine Performancemessung sehr empfehlenswert, welche die tatsächliche Beschleunigung, im Vergleich zur sequentiellen Lösung der Aufgabe, sichtbar macht [Haase2011a].

3.1.6.2.1.2 Erweiterung bestehender Bibliotheken

Das Problem an fremden Bibliotheken ist der konkrete Zustand, in welchem diese ausgeliefert werden. Weicht die Anwendung der Bibliothek von der vorgesehenen Norm ab, entstehen in Java häufig „Helper“ Klassen mit statischen Methoden, die eine Konvertierung von einem Datentyp in einen anderen herbeiführen. Das folgende Beispiel zeigt die Umwandlung eines „java.util.Date“ in einen validen SQL-String für eine IBM-DB2 Datenbank, in der Methode „formatDate“ und die anschließende Anwendung.

```
1 //defintion
2 public class SQLHelper {
3       SQLHelper(){}
4       ...
5       public static String formatDate(java.util.Date date) {
6                 String res;
7                 if (date != null) {
8                           java.sql.Date sqlDat = new java.sql.Date(date.getTime());
```

```
9                       res = "'" + sqlDat + "'";
10              } else {
11                      res = "null";
12              }
13              return res;
14      }
15 }
16 ...
17 //call
18 String s = SQLHelper.formatDate( new java.util.Date() )
```

Listing 118 Implementierung einer Java-Helper-Klasse zum vereinfachten Umgang mit eingebundener Bibliothek

Andere Sprachen, wie C#, lassen beispielsweise das Hinzufügen, aber nicht die Modifikation über das Konzept der „extension methods“ zu. In Scala hat sich das Idiom „pimp my library“ für diese Problemstellung entwickelt, welches Implicites zur Lösung benutzt [Odersky2006]. Das folgende Listing zeigt die Umsetzung der Theorie aus Kapitel 2.2.2.3.

```
1 //definition
2 class RichDate(date: java.util.Date){
3       def formatDate():String = "'" + new java.sql.Date(date.getTime) + "'"
4 }
5 object RichDate{
6       implicit def asRichDate (d: java.util.Date) = new RichDate(d)
7 }
8 ...
9 //call
10 import RichDate._
11 val d1 = new java.util.Date()
12 d1: java.util.Date = Tue Mar 20 17:38:19 CET 2012
13
14 d1.formatDate
15 res0: String = '2012-03-20'
```

Listing 119 Hinzufügen der benötigten Funktionalität zu einer bestehenden Klasse aus externer Bibliothek

In diesem Fall wurde der Datentyp „java.util.Date“ um die Methode „formateDate“ ergänzt und kann direkt ohne Hilfsklassen aufgerufen werden [Haase2011b]. Mit diesem Idiom, lassen sich Bibliotheken nicht nur um Methoden erweitern, sondern dem Anwendungsfall spezifisch anpassen.

3.1.6.2.1.3 Erstellung eines Parsers

Besteht die Anforderung Daten aus einer Sprache/ Eingabequelle in eine strukturierte Form zu bringen, ergeben sich mehrere Möglichkeiten, diese zu lösen. Besitzt der Entwickler die Fachexpertise selbst einen Parser zu erstellen, bleibt diesem noch der Umsetzungsaufwand.

Weniger fortgeschrittene Entwickler greifen auf einen Parser-Generator wie Yacc, Bison oder ANTLR, der in Java verfasst ist, zurück. An dieser Stelle wird die Parser-Kombinator-Bibliothek vorgestellt, welche als interne Domain-Specific-Language (kurz DSL) in Scala implementiert und dadurch sehr gut integriert ist [Odersky2010].
Es wird ein Parser erstellt, welcher einen Input-String untersucht und als Ergebnis eine typisierte Liste mit Objekt-Tupeln ausgibt. Die Input-Daten des Beispiels könnten aus einer Datei stammen und bilden einen Studenten und sein Prüfungsergebnis ab, dessen Struktur dem folgenden Listing zu entnehmen ist:

```
1 class Student(val name:String, val matNr: Int ) {
2         override def toString() = name  + "(" + matNr + ")"
3 }
```

Listing 120 Beispielklasse: Student

Eine Anwendung des Parsers auf folgende Input-Daten könnte so aussehen:

Input: Mayr, Hubertus (123456): 95 Punkte Bak, Kacper (195054): 90 Punkte ...

Output: List((Mayr, Hubertus(123456),95), (Kacper, Bak(195054),90), ...)

Die Lösung erfolgt in zwei Schritten. Im ersten wird der Input-String in seine Bestandteile zerlegt und je Bestandteil ein passender Parser erstellt. Anschließend werden die Teil-Parser zu einem Gesamtparser verknüpft. Dies entspricht der typischen Vorgehensweise in der funktionalen Programmierung, Probleme in Teilfunktionen zu zerlegen und abschließend zusammenzuführen. Im letzten Schritt wird der Parser für alle Student-Instanzen innerhalb des Input-Strings angewendet.
Der Parser wird in der Klasse „ExamParser“ erstellt, die wiederum vom Trait „RegexParsers“ erweitert wird. Mit dieser Erweiterung ist es möglich, aus regulären Ausdrücken Parser zu erstellen. Dafür wird der Datentyp „Parser[T]“ verwendet, wobei „T“ für den Ergebnistyp des Parsers steht.

```
1 val int: Parser[Int] = """\d+""".r ^^ (_.toInt)
```

Listing 121 Implementierung des „int-Parsers“

Die Implementierung des ersten Parsers entsteht aus dem regulären Ausdruck „\d+“, welcher mehrere ganze Zahlen zulässt. Damit der Backslash nicht als Steuerszeichen

verarbeitet wird, werden zusätzliche Anführungszeichen gesetzt. Die Methode „r“ erstellt anschließend einen Parser vom Typ „Parser[String]“. Dieser muss abschließend mit dem „^^“-Operator in das passende Ausgabeformat, hier „Parser[Int]“ transformiert werden. Als Parameter nimmt der Operator eine Funktion „_.toInt“ auf, welche die Transformation beschreibt.

Nachdem ein Parser definiert wurde, erfolgt die Aufteilung eines Studenteneintrages im Input-String, bei der die Bezeichner der Teil-Parser angegeben sind.

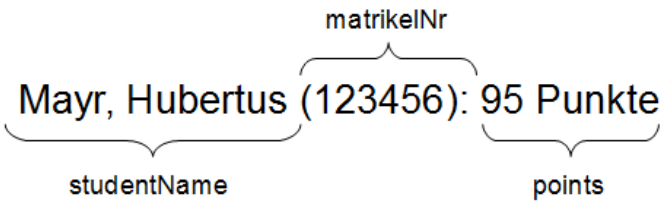


Abbildung 36 Aufteilung einer Student-Instanz des Input-Strings in Teil-Parser

Es ist sehr hilfreich den Kontext eines Eintrages in die Implementierung eines Parsers einzubeziehen. Betrachtet man den Namen des Studenten, endet dieser erst mit dem Anfang der Klammer. Der Name entspricht also allen Zeichen vor der Klammer. Dies bildet der Ausdruck „[^(]*“ ab, dessen Implementierung in Listing 122 erfolgt. Die endgültige Transformation mit „trim“ ändert zwar nichts am Rückgabewert, entfernt aber alle unnötigen Leerzeichen.

```
1 val studentName: Parser[String] = """[^(]+""".r ^^ (_.trim())
```

Listing 122 Implementierung des „studentName-Parsers“

Um die Matrikelnummer des Studenten zu untersuchen, bedarf es einer Definition, wie Parser-Ergebnisse miteinander verknüpft werden können. Dazu werden folgende drei Kombinatoren beschrieben.

a ~ b	Erzeugt einen neuen Parser, dieser parst zuerst a, dann b und gibt das Gesamtergebnis zurück.
a <~ b	Analog zu a ~ b, Rückgabe des Ergebnisses von a

a ~> b	Analog zu a ~ b, Rückgabe des Ergebnisses von b

Tabelle 35 Kombinatoren der Regex-Parser

Ein einfacher String mit einer öffnenden Klammer "(" erzeugt bereits einen Parser, für dieses Zeichen. Für die ganze Zahl innerhalb der Klammer wird der bereits definierte „int-Parser" genutzt. Abgeschlossen wird die Matrikelnummer durch eine schließende Klammer für die wieder ein passender Parser erstellt wird "): ". Der Doppelpunkt und das Leerzeichen werden mit in den Parser aufgenommen.

```
1 val matNr: Parser[Int] = "(" ~> int <~ "): "
```

Listing 123 Implementierung des „matNr-Parsers"

Da die Klammern nicht Bestandteil des Ergebnisses sein sollen, werden sie mittels entsprechender Kombinatoren ausgeschlossen (siehe Listing 123).
Zuletzt bleibt die Interpretation der Klausurpunkte eines Studenten.

```
1 val points: Parser[Int] = int <~ """\W*Punkte?\W*""".r
```

Listing 124 Implementierung des „points-Parsers"

Im regulären Ausdruck steht das „W*" für mögliche Leerzeichen und der Buchstabe „e" wird als Option „?" aufgenommen, falls der Student sich im Studiengang geirrt haben sollte.
Alle notwendigen Teil-Parser sind nun definiert und können zu einem Gesamt-Parser, der eine Student-Instanz interpretiert, kombiniert werden.

```
1 val oneExam: Parser[(Student,Int)] = studentName ~ matNr ~ points ^^{
2     case name~nr~points => (new Student(name,nr),points)
3 }
```

Listing 125 Implementierung des „oneExam-Parsers"

Bei der Funktion, welche die Transformation in den Rückgabewert vornimmt, handelt es sich um simples Pattern Matching (siehe Kapitel 2.2.2.4). Der Rückgabewert ist ein Tupel vom Typ (Student,Int).
Der letzte Schritt besteht darin, den „oneExam-Parser" für mehrere Studenten-Instanzen innerhalb des Input-Strings verfügbar zu machen. Dazu wird der „exam-Parser" definiert, welcher den „oneExam-Parser" verwendet und einen Seperator zwischen zwei Studenten, in

dem Fall das Leerzeichen, verwendet. Diese Art von Wiederholung mit Abgrenzung erfolgt durch die Methode „repsep“.

```
1 val exam: Parser[List[(Student, Int)]] = repsep(oneExam,"""\W*""".r)
```
Listing 126 Finaler Parser

Listing 127 zeigt den Aufruf des implementierten „ExamParsers“ und die Anwendung dessen in dem Objekt „ExamParserApp“, welche durch den Parser erweitert wird. Der Aufruf erfolgt durch die Methode „parseAll“, während mit „get“ auf den typisierten Rückgabewert des Parser zugegriffen werden kann. Der Input-String „examString“ zeigt drei Studenten, die wie man an der Konsolenausgabe erkennen kann, korrekt geparst werden [Braun2011].

```
1 //definition
2 class ExamParser extends RegexParsers {
3
4   val int: Parser[Int] = """\d+""".r ^^ (_.toInt)
5   val studentName: Parser[String] = """[^(]+""".r ^^ (_.trim())
6   val matNr: Parser[Int] = "(" ~> int <~ "): "
7   val points: Parser[Int] = int <~ """\W*Punkte?\W*""".r
8   val oneExam: Parser[(Student,Int)] = studentName ~ matNr ~ points ^^{
9     case name~nr~points => (new Student(name,nr),points)
10   }
11   val exam: Parser[List[(Student, Int)]] = repsep(oneExam,"""\W*""".r)
12 }
13 ...
14 //call
15 object ExamParserApp extends ExamParser {
16
17   def main(args: Array[String]) {
18     val examString =
19       "Mayr, Hubertus (123456): 95 Punkte " +
20       "Kacper, Bak (195054): 90 Punkte " +
21       "Michael, Doppelkorn (998877): 1 Punkt"
22     val resultList = parseAll(exam,examString).get
23     println(resultList)
24   }
25 }
26 ...
27 List((Mayr, Hubertus(123456),100), (Kacper, Bak(195054),95), (Michael, Doppelkorn(998877),1))
```
Listing 127 Vollständige Parser-Implementierung und dessen Einsatz

3.1.6.3 Java – Scala Vergleich

3.1.6.3.1 Quelltextzeilen

Für den Vergleich beider Sprachen wird eine geringere Anzahl an Quelltext-Zeilen in Scala erwartet. Diese Annahme beruht auf den Eigenschaften der Sprache, wie Semikolon- und Typ-Inferenz sowie der höheren Abstraktionsmöglichkeit durch objektfunktionale Programmierung.

Die Auswertung beider Schichten mit dem Werkzeug „STAN4J – Structure Analysis for Java“ ergab jedoch ein entgegengesetztes Bild. Der ELOC-Wert (Estimated Lines Of Code) der Scala-Business-Layer war nach der Evaluation des Tools doppelt so hoch (ELOC: 1667), wie der Wert der Java-BusinessLayer (ELOC: 769). Das Tool rechnet dabei selbstständig Import-Statements, Kommentare und Leerzeilen heraus [Odysseus Software2011a]. Da es sich bei dem Ergebnis um eine Schätzung (engl.: Estimate) handelt, erfolgt eine konkrete Gegenüberstellung beider Sprachen je Klasse und abschließendem Durchschnittswert, ohne jegliche Berücksichtigung der Kommentare, Import-Statements usw.

Klassenname:	**Java-Zeilenanzahl:**	**Scala-Zeilenanzahl:**
AhpCompareValueMatcher	115	67
FundamentalScale	23	21
BusinessAhpEvaluation	45	14
TBusinessAhpEvaluation	29	29
FinalWeightsCalculation	74	76
ForkJoinSquareMatrix + ForkJoinTask / ScalaSquareMatrix	295 + 117	264
RandomConsistencyVector	36	30
Validation	66	67
Durchschnitt	**800/8 = 100**	**568/8 = 71**

Tabelle 36 Gegenüberstellung der Quelltextzeilen der Java/Scala Business-Layer

Tabelle 36 veranschaulicht tatsächlich die signifikante Differenz zwischen Java- und Scala-Quelltextmenge. Die Position der Matrix ragt dabei besonders heraus, da für die identische Anforderung in Java zwei Klassen zur Parallelisierung benötigt werden.
Die Aussage „type less, do more“ [Wampler2008] kann in Scala-Projekten durchaus aufgehen. Jedoch handelt es sich dabei um einen willkommenen Nebeneffekt, dem durch seinen optionalen Charakter, kein besonders hohes Gewicht bemessen sollte. Statistisch wird bei der Quelltexterzeugung weniger Tipparbeit benötigt, deswegen können aber sehr abstrakte Scala-Konstrukte sich auch negativ auf die Wartung auswirken.

3.1.6.3.2 Kopplung zwischen Objekten

Ähnlich zu der LOC-Metrik verhält sich die Analyse der CBO-Metrik. Darin erreicht die Scala-Schicht einen höheren CBO-Wert (3,56) als die analoge Java-Business-Layer (3,34). Dies spricht grundsätzlich gegen eine Verwendung von Scala in der gesamten Steuerungs- und Funktions-Schicht. Auch hier ist ein zweiter Blick auf die Abhängigkeiten zwischen den Modulen möglich, der ein differentiertes Bild liefert.

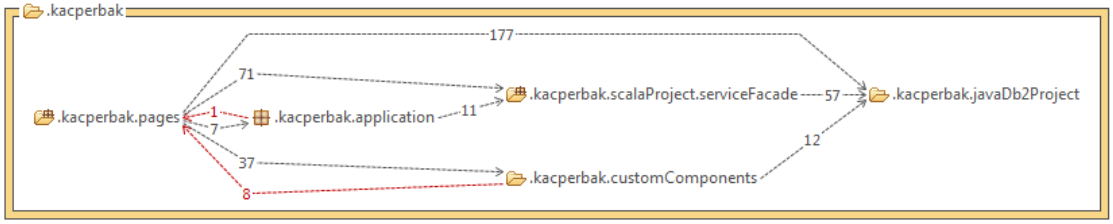


Abbildung 37 AHP-WebApp mit Scala-Business-Layer

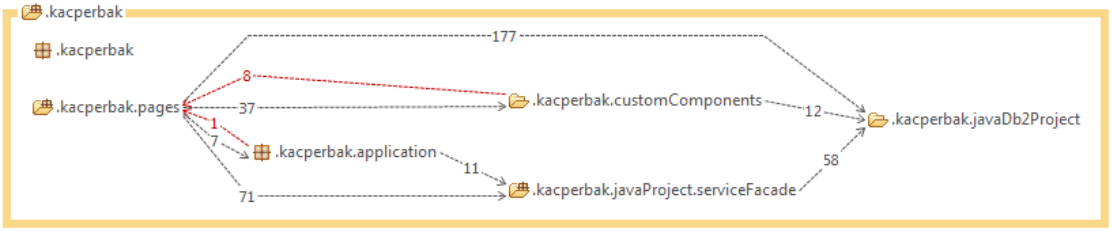


Abbildung 38 AHP-WebApp mit Java-Business-Layer

Die Abbildungen stellen die Projektstrukturen der Beispielanwendung mit Scala- und Java-Business-Layer dar. Die Pfeile zwischen den Modulen bilden die Beziehungen (Referenzen, Aufrufe, Enthält, ...) und deren Anzahl zwischen den Modulen ab. Jedoch wird bei dem Vergleich kein bedeutender Unterschied ersichtlich. Begründen lässt sich dieses Ergebnis mit dem Einsatz von Traits als Java-Schnittstellen. Durch deren analoge Verwendung kann eine identische Methodik zur Kapselung der Schichten, wie in Java erfolgen. Bei der Umsetzung dieses Ansatzes müssen allerdings wichtige Details der Interoperabilität beider Sprachen beachtet werden (siehe Kapitel 3.1.6.1.2).

Die Probleme und deren Lösung, welche bei der Umsetzung einer hybriden Java/Scala Anwendung entstehen können, steigern unter Umständen die Komplexität der gesamten Architektur. Die Umsetzung einer Anwendung mittels einer Programmiersprache, Java oder Scala, wird deshalb immer einen geringeren Aufwand bedeuten, da keine gemeinsame Schnittmenge zwischen zwei Sprachen erzeugt werden muss. Dies ist jedoch kein Scala spezifisches Problem, sondern eines der Polyglotten-Programmierung [Dewanto2012]. Als Folgerung sollte der Architekt einer Anwendung unterschiedliche Technologien nur dann vermischen, wenn deren Einsatz sehr viel Aufwand einspart oder alternativlos ist. Beispiele dafür können dem Kapitel 3.1.6.2 entnommen werden.

3.1.6.3.3 Performanz

3.1.6.3.3.1 Parallelisierung

In der Business-Layer (Kapitel 3.1.5.2.1.2) wurde auf die Berechnung der Matrix-Potenzierung des AHPs (Kapitel 3.1.4.2.5) eingegangen. Weiter wurden in Kapitel 3.1.6.2.1.1 zwei Möglichkeiten der Implementierung (Java/Scala) anhand des Fork/Join-Frameworks vorgestellt. Um die Leistungsfähigkeit beider Implementierungen vergleichen zu können, wurde eine Testanwendung „java-scala-matrix-test“ geschrieben, die einen Benchmark bereitstellt. Die folgenden zwei Abbildungen präsentieren die Ergebnisse. Die Diagramme stellen die verstrichene Zeit in Abhängigkeit zur Größe der Matrix dar. Verglichen wird die sequentielle Berechnung (Rechtecke), das Fork/Join-Framework in Java 7 (Kreise) und die Scala Parallel-Collections (Dreiecke).

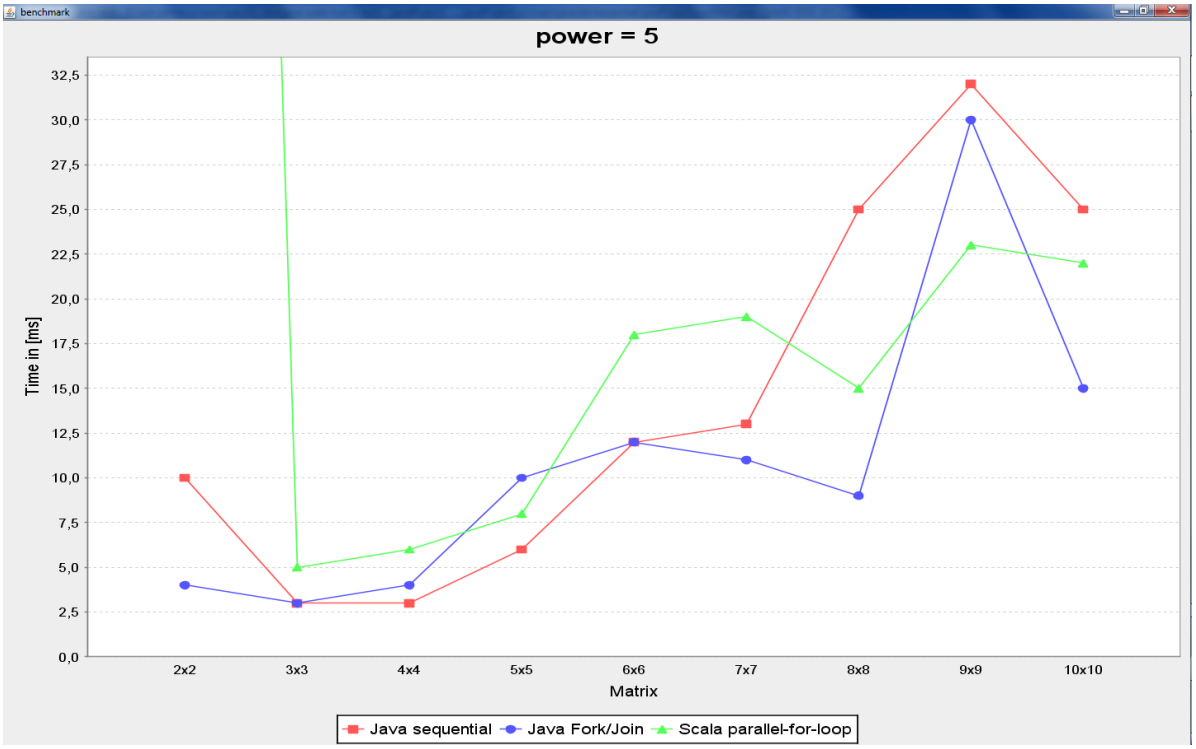


Abbildung 39 Benchmark der Parallelisierung mit der 5. Potenz

Abbildung 39 verdeutlicht, dass die Parallelisierung der Matrix frühestens ab einer 6x6 Matrix Sinn ergibt. Mit der fünften Potenz ist die Differenz zwischen zwei Ergebnissen eines AHP-Vergleiches, jedoch kleiner als 10^{-7}. Damit dient die Parallelisierung der AHP-Matrix,

dem rein akademischen Zweck, nämlich des Vergleiches zwischen Java- und Scala-Performance. Um den Aufwand für die Berechnung zu steigern, wird die Potenz mit der die Matrix mit sich selbst multipliziert wird, von fünf auf sieben erhöht.

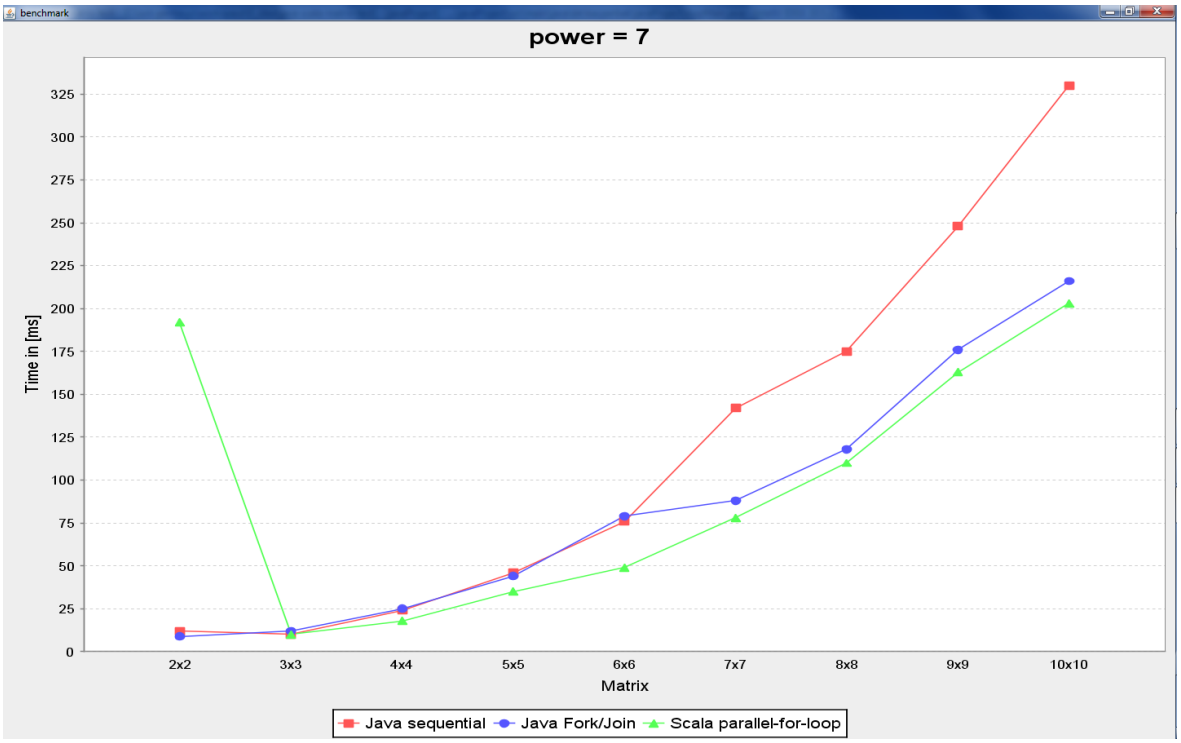


Abbildung 40 Benchmark der Parallelisierung mit der 7. Potenz

Zu erkennen ist, dass mit steigender Potenz die Differenz zwischen sequentieller und paralleler Abarbeitung steigt. Zur Verteidigung des Java Fork/Join-Frameworks muss an dieser Stelle der Grenzwert „THRESHOLD“ genannt werden. Dieser entspricht im Benchmark, dem Wert aus Listing 111, nämlich 6. Folglich setzt die Aufteilung der Aufgabe erst ab einer Matrixgröße > 6 ein, wie aus Abbildung 40 zu erkennen ist. Dieser Wert bildet somit ein Schlüsselkriterium bei der Feinabstimmung auf die Problemgröße. Hingegen ist aus beiden Diagrammen klar zu erkennen, dass die Initialisierung der Scala Parallel-Collection weit über der Zeit beider Alternativen liegt.

Diese Werte sind natürlich spezifisch für den Rechner des Autors. Dessen Kerndaten lauten:

- Windows Professional 64 Bit System mit 4 GigaByte RAM
- Intel i5 M520

- Runtime: JDK 1.7.01 mit Scala 2.9.1

Weiter ist darauf zu verwiesen, dass als interne Datenstruktur für die Matrix eine geschachtelte „java.util.ArrayList“ verwendet wurde, auch innerhalb der Scala-Implementierung. Der Grund für diese Entscheidung liegt bei einem Interview mit dem Twitter-Entwicklungsteam, welches sein Backend von Ruby auf Scala erfolgreich migriert hat. Intern verwendet Twitter weiterhin die Java-Collections, da deren Performance wesentlich besser ist, als der gegenwärtige Stand des Scala-Pendants [Venners2009]. Als sehr positiv kann dafür die reibungslose Verwendung dieser Java-Collection in Scala bezeichnet werden, was für die Integration von Scala spricht.

3.1.6.3.3.2 Übersetzung

Einer der Hauptnachteile von Scala ist die unverhältnismäßig lange Übersetzungszeit von Quelltext in Java-Bytecode. Eine der Hauptursachen dafür ist die Initialisierung des Compilers, welcher alle in das Projekt eingebundenen Java-Archive auf deren Inhalt prüft [Odersky2010]. Weitere Merkmale, wie Typinferenz sowie Implicits, haben ihren Preis, der spätestens an diesem Punkt eingefordert wird. Als Beispiel dienen hier die Maven-Erstellungszeiten.

```
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Summary:
[INFO]
[INFO] DA-App ..................................... SUCCESS [0.670s]
[INFO] javaDb2Project ............................. SUCCESS [3.204s]
[INFO] scalaProject ............................... SUCCESS [16.634s]
[INFO] wicketProject .............................. SUCCESS [5.732s]
[INFO] ------------------------------------------------------------------------
```

Abbildung 41 Erstellungszeiten der Beispielanwendung „AHP-WebApp“ mit Scala-Business-Layer

```
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Summary:
[INFO]
[INFO] DA-App-Java ................................ SUCCESS [0.705s]
[INFO] javaDb2Project ............................. SUCCESS [3.272s]
[INFO] javaProject ................................ SUCCESS [1.572s]
[INFO] wicketProject .............................. SUCCESS [6.185s]
[INFO] ------------------------------------------------------------------------
```

Abbildung 42 Erstellungszeiten der Beispielanwendung „AHP-WebApp“ mit Java-Business-Layer

Aus dem Vergleich der Abbildung 41 und Abbildung 42 wird ersichtlich, dass der Scala-Compiler trotz weniger Klassen und Quelltext, siehe LOC Vergleich - Kapitel 3.1.6.3.1 - ,

wesentlich mehr Zeit für die Erstellung des Java-Bytecodes benötigt.
Für diesen Umstand wurde die Anwendung „fsc“ (Fast-Scala-Compiler) entwickelt, welche im Hintergrund als Task gestartet wird und immer das vom Entwickler erstellte Delta an Änderungen am Quelltext übersetzt. Innerhalb der Kommandozeilenentwicklung kann der Übersetzungsprozess damit wesentlich beschleunigt werden [Seeberger2011]. Eine Überführung dieses Fast-Compilers in IDE’s und Build-Werkzeuge ist für den Entwicklungszyklus absolut entscheidend. Für die IntelliJ IDEA Entwicklungsumgebung besteht diese Art der Integration für Scala-Projekte, deren Konfiguration Abbildung 43 und Abbildung 44 zu entnehmen ist [Fatin2011].

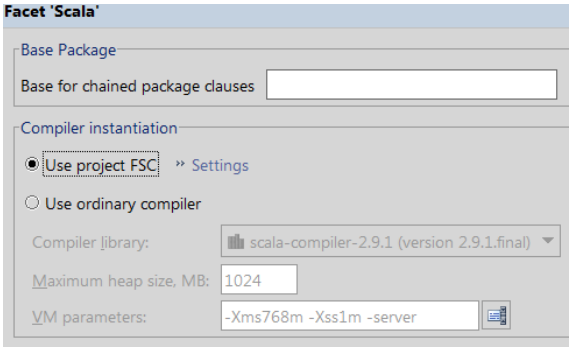


Abbildung 43 Auswahl des Scala-Compilers in der IntelliJ IDEA IDE

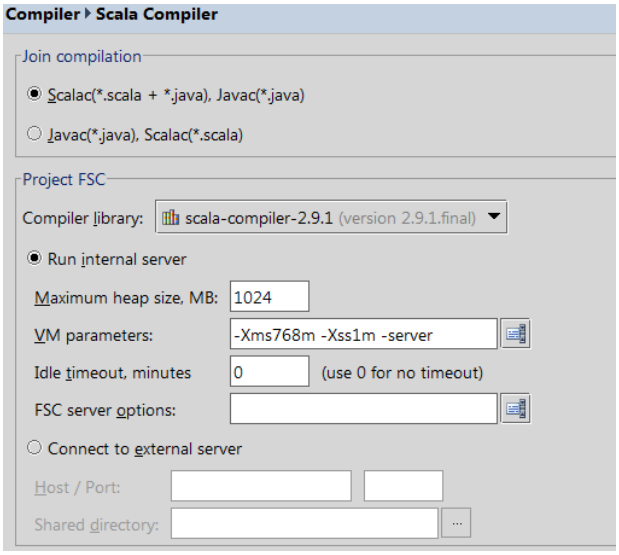


Abbildung 44 Konfiguration des Scala-Compilers in der IntelliJ IDEA IDE

Weiter besteht fsc-Unterstützung für SBT und Maven. In Maven kann der „fsc“ mittels „mvn scala:cc –Donce=true“ Befehl ausgeführt werden [Scala Tools2010]. Eine erfolgreiche Anwendung, spezifisch auf die Business-Layer, schlug jedoch fehl.

3.1.6.4 Beste IDE für modulare Java/Scala Projekte

Nach der detaillierten Evaluation der Entwicklungsumgebungen im AHP Kapitel folgt nun das Fazit der Bewertung. Für gemischte Java/Scala Projekte, deren Entwicklung modulbasiert in einer IDE stattfinden soll, kommt gegenwärtig nur IntelliJ IDEA in Frage. Als einzige stabile

IDE konnte eine ähnlich gute Unterstützung erfahren werden, wie es der Java-Entwickler von seiner Standard-IDE gewohnt ist. Auf dem zweiten Platz folgt Eclipse, da mit dieser IDE ohne Maven-Unterstützung auch stabil gearbeitet werden kann. Hinzukommt, dass in Zukunft das Scala-Plugin direkt vom Scala-Entwickler-Team unterstützt wird [Typesafe Inc.2012]. Damit geht der Trend für Eclipse in die richtige Richtung, auch wenn gegenwärtig auf der Website der Scala-IDE mehr versprochen wird, als tatsächlich implementiert ist. Als letztes folgt Netbeans, da zwischen einem sehr einfachen Setup und tadelloser Maven-Unterstützung, bis hin zur vollkommenen Unbrauchbarkeit durch den Verlust von Projektbestandteilen, wurden alle Zwischenstadien, während der Testphase durchlaufen wurden, ohne endgültige Stabilität zu erreichen.

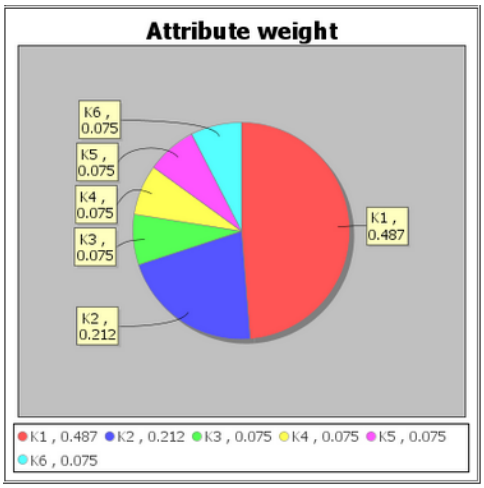


Abbildung 45 Ergebnisse der Evaluierung: Beste Java/Scala-IDE aus der Beispielanwendung AHP-WebApp: Lokale Attributgewichte

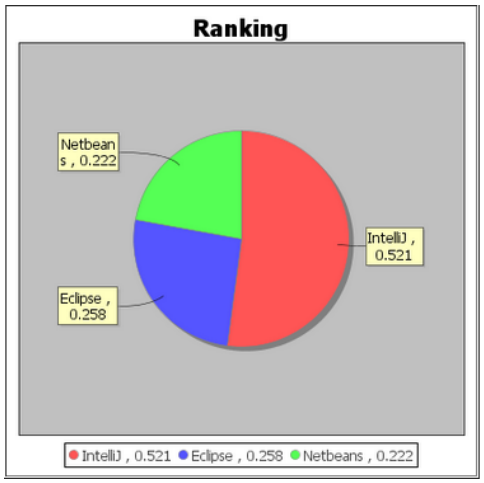


Abbildung 46 Ergebnisse der Evaluierung: Beste Java/Scala-IDE aus der Beispielanwendung AHP-WebApp: Ergebnisrangfolge

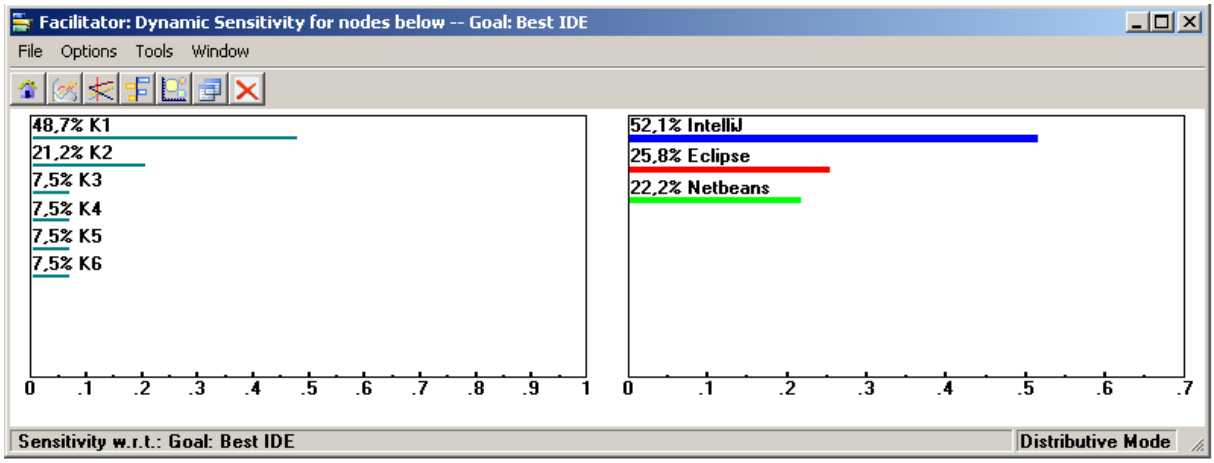


Abbildung 47 Ergebnisse der Evaluierung: Beste Java/Scala-IDE aus dem kommerziellen Produkt: Expert Choice

3.2 Toolsupport

3.2.1 Einsatz von Maven unter Scala

Die Bedeutung und Vorteile des Build-Werkzeugs Maven wurden bereits im Kontext der Beispielanwendung im Kapitel 3.1.5.3 behandelt. Dieser Abschnitt widmet sich der Konfiguration eines Scala-Moduls, welches mit Maven verarbeitet werden kann. Getestet wurde diese Konfiguration unter einer Maven 3.0.3 Installation mit einem JDK1.6_30.

Analog zu Java-Modulen wird ein Scala-Modul erstellt, dessen Projektkonfigurationsdatei das „maven-scala-plugin“ enthalten muss [Scala Tools2010]. Innerhalb des Moduls ist die standardisierte Projektstruktur anzupassen und innerhalb der pom.xml anzugeben:

Java-Projektstruktur:	**Scala-Projektstruktur:**
src/main/java	src/main/scala
src/test/java	src/test/scala

Tabelle 37 Änderung der Projektstruktur für Scala-Module in Maven

Eine vorhergehende Scala-Installation ist nicht nötig. Empfehlenswert ist aber die Scala-Sprach-Bibliothek als Abhängigkeit im „dependency“ Tag anzugeben, um eine Verbesserung der Kompatibilität der IDE‘s zu erreichen. Sowohl für das Plugin, als auch für die Bibliothek sind die Repositories, also die Maven-Bezugsquellen, anzugeben. Während der Entwicklung der Beispielanwendung fiel der besonders hohe Speicherbedarf der JVM auf. Eclipse beispielsweise empfiehlt sein Scala Plugin zwischen 256MB (Minimum: Xms) und 1024MB (Maximum: Xmx) zu konfigurieren, wobei die Grenze nach oben offen ist [ScalaIDE wiki2012]. Für die Entwicklung der AHP-WebApp Beispielanwendung wurden die vorgegebenen Werte des „maven-scala-plugin“ verwendet, die aus den „jvmArgs“ Tags aus Listing 128 zu entnehmen sind.

```
1 <project ...>
2       ...
3       <parent>
4               <artifactId>DA-App</artifactId>
5               <groupId>de.kacperbak</groupId>
6               <version>0.6.0</version>
7       </parent>
8       <artifactId>scalaProject</artifactId>
9       <groupId>de.kacperbak.scalaProject</groupId>
10      <packaging>jar</packaging>
11      <name>scalaProject</name>
12      ...
13      <repositories>
14              <repository>
15                      <id>scala-tools.org</id>
16                      <name>Scala-tools Maven2 Repository</name>
17                      <url>http://scala-tools.org/repo-releases</url>
18              </repository>
```

```
19      </repositories>
20      ...
21      <pluginRepositories>
22              <pluginRepository>
23                      <id>scala-tools.org</id>
24                      <name>Scala-tools Maven2 Repository</name>
25                      <url>http://scala-tools.org/repo-releases</url>
26              </pluginRepository>
27      </pluginRepositories>
28      ...
29      <build>
30      <sourceDirectory>src/main/scala</sourceDirectory>
31      <testSourceDirectory>src/test/scala</testSourceDirectory>
32      ...
33      <plugins>
34              <plugin>
35                      <groupId>org.scala-tools</groupId>
36                      <artifactId>maven-scala-plugin</artifactId>
37                      <version>2.14</version>
38                      <executions>
39                              <execution>
40                                      <goals>
41                                              <goal>compile</goal>
42                                              <goal>testCompile</goal>
43                                      </goals>
44                              </execution>
45                      </executions>
46                      <configuration>
47                              <scalaVersion>2.9.1</scalaVersion>
48                              <jvmArgs>
49                                      <jvmArg>-Xms64m</jvmArg>
50                                      <jvmArg>-Xmx1024m</jvmArg>
51                              </jvmArgs>
52                      </configuration>
53              </plugin>
54      </plugins>
55      </build>
56      ...
57      <dependencies>
58              <dependency>
59                      <groupId>org.scala-lang</groupId>
60                      <artifactId>scala-library</artifactId>
61                      <version>2.9.1</version>
62              </dependency>
63      ...
64      </dependencies>
65  </project>
```

Listing 128 Auszug aus der pom.xml des Submodules ScalaProject in der Beispielanwendung: AHP-WebApp

Natürlich besteht für Scala auch ein eigenes Erstellungswerkzeug, genannt Simple-Built-Tool (kurz SBT). Dieses wird auch in einschlägiger Literatur zur Entwicklung empfohlen [Seeberger2011]. Aus Gründen der Kompatibilität zu bestehenden Technologien (IDE's,

JUnit, ...), vor allem aber aus der Erfahrung nicht zu viele neue Technologien auf einmal zu verwenden, entschied sich der Autor dieser Arbeit für die etablierte Technologie: Maven. Für alle Anwendungsfälle der Beispielanwendung konnte eine Lösung mit Maven realisiert werden. Diese Erfahrung blieb dem Startup-Unternehmen „Yammer – The Enterprise Social Network“ nicht erspart. Nach einer langen frustrierenden Phase mit SBT wechselte das Entwicklerteam zu Apache Maven [Colebourne 2011]. Die Ursache dieses Fehlgriffes liegt wohl in der Intention von SBT, primär ein Tool für die Scala-Entwicklung zu sein. Das praktische Interesse an Scala, geht aber in Richtung gemischter Java/Scala Projekte.

3.2.2 Scala - Entwicklung in der Kommandozeile

Analog zu Java wird der Scala Quelltext mittels des Scala-Compilers, „scalac“, in Java-Bytecode übersetzt, welcher auf der JVM ausgeführt werden kann. Der Übersetzungsvorgang einer Datei namens „File.scala“ erfolgt analog zum Java-Compiler „javac“.

```
1 c:\TEMP\test>scalac File.scala
```
Listing 129 Anwendung des Scala-Compilers

Daneben gibt es auch einen Compiler-Deamon „fsc“, welcher einmal gestartet im Hintergrund immer das neu entstandene Delta übersetzt. Um übersetzte Programme nun starten zu können, erfolgt der Aufruf von „scala“.

```
1 c:\TEMP\test>scala File
2 Hello Scala World
```
Listing 130 Aufruf von Scala-Programmen

Viel interessanter ist aber das Starten der „scala“ Anwendung ohne Parameter.

```
1 c:\TEMP\test>scala
2 Welcome to Scala version 2.9.1.final (Java HotSpot(TM) Client VM, Java 1.6.0_30).
3 Type in expressions to have them evaluated.
4 Type :help for more information.
```
Listing 131 Start der interaktiven Scala-Konsole

Damit wird eine interaktive Scala-Konsole gestartet, welche nicht nur Ausdrücke sofort verarbeitet, sondern den Kontext der bisherigen Anwendung mit einbezieht. So ist der

Zugriff auf die „main()“ Methode, die jedes lauffähige Programm aufweisen muss, des File-Objektes aus „File.scala“ möglich [Seeberger2011].

```
1 scala> val result = 1 + 1
2 result: Int = 2
3
4 scala> File.main(null)
5 Hello Scala World
```

Listing 132 Scala REPL

Hierbei handelt es sich um eine Read-Evolve-Print-Loop, wie sie bereits aus anderen funktionalen Programmiersprachen bekannt ist. Sie dient als Grundlage zur Evaluierung von Scala Ausdrücken und ist besonders in der Lernphase von Scala von großem Nutzen. Auf die Eingabeaufforderung „scala>“ wurde in den Listings dieser Arbeit verzichtet.

Für simple Experimente unterstützt die Scala-REPL den Entwickler durch Autovervollständigung mit Betätigung der Tabulator-Taste. Komplexere Zusammenhänge sollten hingegen innerhalb eines Editors erstellt und in die Konsole überführt werden. Dazu bietet diese die Befehle „:paste“, welcher die Kopie des Inhaltes aus der Zwischenablage mit STRG+V zulässt und „:load <Dateiname>“, welcher die angegebene Scaladatei in den Konsolenkontext lädt [Braun2011].

Um die Grundlagen für das Scalastudium zu vervollständigen, benötigt der Entwickler nur noch einen Editor, der die Scala-Syntax unterstützt. Hier kann auf das Unternehmen Typesafe verwiesen werden. Dieses setzt sich das Ziel, Scala kommerziell mit Beratung und Schulungen zu fördern sowie den Einstieg in Scala zu vereinfachen. Dazu bietet es den „Typesafe Stack“ an, welcher neben der Programmiersprache Scala, diverse Frameworks, Dokumentationen und Plugins für die gängigsten Editoren anbietet [Typesafe Inc.2012].

Abschließend soll noch ergänzt werden, dass die „scala“ Anwendung unter einer 64 Bit JVM nicht fehlerfrei funktionierte. Dieses Problem kann mit einer parallelen 32 Bit JDK Installation umgangen werden, wobei darauf geachtet werden muss, dass die Umgebungsvariablen „PATH“ und „JAVA_HOME“ auf die 32 Bit Verzeichnisse verweisen.

3.2.3 Scala - Entwicklung mit IDE - Unterstützung

Es folgt die Installationsbeschreibung der in dieser Arbeit evaluierten IDE's. Details zu Vor- und Nachteilen aller drei IDE's sind dem AHP-Vergleich aus Kapitel 3.1.4 zu entnehmen. Ausgangsbasis für die nachfolgende Betrachtung ist ein modulares Java/Scala Projekt, welches innerhalb der Kommandozeile mit Maven fehlerfrei übersetzt werden kann.

3.2.3.1 Installation der IntelliJ IDE

Die von Jetbrains entwickelte Java-Entwicklungsumgebung IntelliJ IDEA steht als kostenlose und kommerzielle Version auf der Herstellerseite zur Verfügung. Für den Einsatz innerhalb von Entwicklungsprojekten wird empfohlen, die kommerzielle Version zu verwenden. Diese bietet neben HTML/CSS Integration sehr gute Unterstützung für Enterprise-Webserver und etablierte Web- und Persistenzframeworks, wie Spring und Hibernate [JetBrains s.r.o.2011]. Voraussetzung für eine Installation ist ein installiertes Java-Development-Kit (kurz JDK) der Version 1.6 oder höher. Nach der erfolgreichen Installation kann über den Plugin Manager das Scala-Plugin installiert werden. Dazu muss ein IDEA-Repository nach dem Begriff „Scala" durchsucht werden. Mit einem Rechtsklick auf den gefundenen Eintrag erfolgt die Installation des Plugins. Nachdem die Installation erfolgreich abgeschlossen ist, kann ein neues Projekt erstellt und ein bestehendes Maven Projekt importiert werden. Sollten Referenzen auf die einzelnen Projektbestandteile nicht sofort erkannt werden, so ist ein manueller Reimport über das zugehörige Maven-Menü möglich.

3.2.3.2 Installation der Eclipse IDE

Die Eclipse IDE gibt es in sehr vielen unterschiedlichen Ausprägungen. Um eine ausreichend kompatible Version zu beziehen, ist die Seite des „Scala-IDE for Eclipse" Plugins [Dragos2012] aufzusuchen und sich über die passende IDE zu informieren. Beachtet werden muss außerdem, dass die IDE selbst in einer 32 und 64 Bit Version angeboten wird, die nur mit einer entsprechenden Java-Laufzeitumgebung kompatibel ist. In dieser Arbeit wurde die „Java IDE for EE Developers" in der Version 3.6.2 (Helios) verwendet, um auf Kompatibilität zum Enterprise Umfeld zu testen. Es werden nun zwei Möglichkeiten der Installation vorgestellt. Die Erste beinhaltet die Maven-Unterstützung, welche sich aber relativ instabil in

der Bedienung mit zunehmender Projektgröße verhält. Dies ist auf die geringe Reife und unterschiedliche Herstellungsquellen der Plugins zurückzuführen. Die zweite Möglichkeit verzichtet auf die Maven-Unterstützung, bietet aber eine stabile Variante der Entwicklung an.

Voraussetzung für die Installation von Eclipse ist ein bereits installiertes JDK der Version 1.6. Nachdem Eclipse heruntergeladen und entpackt wurde, ist in der „eclipse.ini“ der absolute Pfad des JDK anzugeben. Dies ist notwendig, damit Eclipse keine höhere Java Version auf dem System verwendet, zu der es gegenwärtig nicht kompatibel ist.

```
1...
2 --launcher.XXMaxPermSize
3 256M
4 -vm
5 C:/Program Files/Java/jdk1.6.0_30/bin
6 -showsplash
7 org.eclipse.platform
8...
```

Listing 133 Konfiguration der eclipse.ini

Der Eintrag in der „eclipse.ini“ sollte der Form aus Listing 133 entsprechen, wobei nur der hervorgehobene Text relevant ist. Um sicherzustellen, dass Eclipse auch das JDK 1.6 verwendet, ist die Überprüfung der Installationsdetails empfehlenswert.

```
1 Aktion in der Menüleiste:
2 Fensterpfad zu den Installationsdetails: Help > About Eclipse > Installation Details
3 Der Wert "java.version=1.6.0_30" sollte enthalten sein.
```

Listing 134 Überprüfung der Installationsdetails in Eclipse

Nun kann eine fehlerfreie Installation der notwendigen Plugins über den Installationsmanager erfolgen. Die Plugins sollten alle über die Online-Installation erfolgen, da nur diese die korrekten Einstellungen und internen Abhängigkeiten richtig konfiguriert. Für diesen Vorgang werden die URL-Quellen der Plugins benötigt. Die URL‘s sind innerhalb von Eclipse über den „Install new Software“ – Dialog zu verwenden.

PluginName	URL
Scala-IDE for Scala 2.9.x (stable) [Dragos2012]	http://download.scala-ide.org/releases-29/stable/site
M2E – Maven Integration for Eclipse [Bentmann2012]	http://download.eclipse.org/technology/m2e/releases
m2eclipse-scala [Bernard2012c]	http://alchim31.free.fr/m2e-scala/update-site/

Tabelle 38 Eclipse Plugin URL's

Anmerkung zum “Scala-IDE for Eclipse” Plugin: Die Version der verwendeten Scala-Bibliothek muss zwingend zur Plugin-Version der IDE passen. Im Testprojekt wurde beispielsweise in der Maven Projektkonfigurationsdatei des Scala Projektes die Scala Bibliothek 2.9.1 als Abhängigkeit eingetragen und dies auch dem „maven-scala-plugin“ ersichtlich gemacht.

```
1 <build>
2       <plugin>
3               <groupId>org.scala-tools</groupId>
4               <artifactId>maven-scala-plugin</artifactId>
5               <version>2.14</version>
6
7               <!-- Scala SDK version -->
8               <configuration>
9                       <scalaVersion>2.9.1</scalaVersion>
10              </configuration>
11
12              ...
13      </plugin>
14 </build>
15
16 <dependencies>
17      <!-- Scala SDK version -->
18      <dependency>
19              <groupId>org.scala-lang</groupId>
20              <artifactId>scala-library</artifactId>
21              <version>2.9.1</version>
22      </dependency>
23      ...
24 <dependencies>
```

Listing 135 Explizite Angabe der verwendeten Scala Version passend zur „Scala IDE for Eclipse“ for Scala 2.9.x

3.2.3.2.1 Installation der Eclipse IDE mit Maven Unterstützung

Nachdem alle Plugins erfolgreich installiert wurden, kann ein bestehendes Maven-Projekt dem „eclipse-workspace“ hinzugefügt werden. Ab diesem Punkt wird ersichtlich, dass die Plugins nicht aufeinander abgestimmt sind. Das „m2eclipse-scala“ Plugin ermöglicht zwar den Einsatz von Maven ohne Fehlermeldungen, scheitert aber bei der Konfiguration der Projektart (Java oder Scala). Diese Einstellung muss von Hand vorgenommen werden.

```
1 Aktionen im "Package Explorer":
2 Scala-Projekt markieren > Configure > Add Scala Nature
```
Listing 136 Einstellung der Projektart in Eclipse

Damit der Scala-Compiler auch für das Scala-Modul verfügbar wird, ist es notwendig, die Scala-Library dem Modul bekannt zu machen. Anschließend erfolgt eine Aktualisierung der Projektkonfiguration, um die Änderungen in der IDE wirksam zu machen.

```
1 Aktionen im "Package Explorer":
2 Haupt-Projekt markieren >rechts-Klick > Maven > Update Projekt Configuration
```
Listing 137 Aktualisierung der Projektkonfiguration in Eclipse

Als letzter Schritt muss die „Clean...“ Aktion in Eclipse aufgerufen werden.

```
1 Aktion in der Menüleiste:
2 Projekt > Clean...
```
Listing 138 Bereinigung des Projektes unter Eclipse

Nach diesen Schritten sollten alle Projekte korrekt angezeigt und valider Scala-Quelltext vom Compiler nicht als Fehler erkannt werden. Mit zunehmender Projektgröße und dem Ausführen von Maven-Aktionen wird es nötig, die Schritte „Clean...“ und „Update Projekt Configuration“ erneut auszuführen, um den stabilen Zustand zu erhalten [Bernard2012a].

3.2.3.2.2 Installation der Eclipse IDE mit Maven aus der Kommandozeile

Ein modulares Java/Scala Projekt kann auch ohne Maven Unterstützung aus der IDE entwickelt werden. Dabei werden die Maven Befehle direkt in der Kommandozeile auf Ebene des Hauptprojektes ausgeführt, während die Entwicklung der einzelnen Module in der IDE ausgeführt wird. Zu installieren ist in diesem Fall ausschließlich das „Scala IDE for Eclipse“ Plugin. Diese Variante ist im Vergleich zu vorherigen wesentlich aufwendiger. Ein weiterer Nachteil ist die Notwendigkeit, dass IDE spezifische Einstellungen innerhalb der

Maven-Projektkonfigurationsdatei des Scala Projektes angepasst werden müssen. Damit wird neben dem „Scala-IDE for Eclipse“ Plugin, welches von einer bestimmten Scala-Bibliothek abhängig ist, eine weitere Abhängigkeit hinzugefügt.

Die nachfolgende Konfiguration des Maven-Eclipse-Plugins wird in den „build“ Bereich der pom.xml des Scala Projektes eingefügt. An dieser Stelle ist anzumerken, dass die Modulaufteilung nach Sprachen anzuwenden ist.

```
1  <plugin>
2    <groupId>org.apache.maven.plugins</groupId>
3    <artifactId>maven-eclipse-plugin</artifactId>
4    <version>2.8</version>
5    <!-- see http://maven.apache.org/plugins/maven-eclipse-plugin/eclipse-mojo.html for more
information -->
6    <configuration>
7      <downloadSources>true</downloadSources>
8      <downloadJavadocs>true</downloadJavadocs>
9      <projectnatures>
10       <projectnature>org.scala-ide.sdt.core.scalanature</projectnature>
11       <projectnature>org.eclipse.jdt.core.javanature</projectnature>
12     </projectnatures>
13     <buildcommands>
14       <buildcommand>org.scala-ide.sdt.core.scalabuilder</buildcommand>
15     </buildcommands>
16     <classpathContainers>
17       <classpathContainer>org.scala-ide.sdt.launching.SCALA_CONTAINER</classpathContainer>
18       <classpathContainer>org.eclipse.jdt.launching.JRE_CONTAINER</classpathContainer>
19     </classpathContainers>
20     <excludes>
21       <exclude>org.scala-lang:scala-library</exclude>
22       <exclude>org.scala-lang:scala-compiler</exclude>
23     </excludes>
24     <sourceIncludes>
25       <sourceInclude>**/*.scala</sourceInclude>
26       <sourceInclude>**/*.java</sourceInclude>
27     </sourceIncludes>
28   </configuration>
29 </plugin>
```

Listing 139 Konfiguration des Maven-Eclipse-Plugins

Anschließend erfolgt die Generierung der Eclipse-spezifischen Projektdateien „.classpath“, „.project“ mit folgendem Befehl auf dem Wurzelverzeichnis des Hauptprojektes.

```
1 mvn eclipse:clean eclipse:eclipse
```

Listing 140 Befehl zum Erstellen der Eclipse-spezifischen Projektdateien

Im letzten Schritt muss noch überprüft werden, ob die Eclipse IDE auf das lokale Maven-Repository verweist, damit es heruntergeladene Abhängigkeiten des Projektes erkennt.

```
1 Aktion in der Menüleiste:
2 Window > Prefrences > Java > Build Path > Classpath Variable
```

Listing 141 Überprüfung der M2_REPO Variable innerhalb von Eclipse

Die „M2_REPO“ Variable sollte vorhanden sein und ihr Wert auf den Benutzerpfad des Maven-Repository verweisen. Abschließend kann das Hauptprojekt über „Existing Projects into Workspace“ in Eclipse importiert werden.
Im Vergleich zur Installation mit Maven-Unterstützung werden die Subprojekte voneinander entkoppelt in den Workspace importiert. Die beschriebenen Schritte sind einmalig beim Anlegen des Projektes vorzunehmen, gewährleisten aber eine stabile Nutzung des „Scala-IDE for Eclipse“ Plugins.
Alle Operationen, welche den Anwendungslebenszyklus betreffen, müssen aber auf dem Hauptprojektverzeichnis in der Kommandozeile mit Maven-Befehlen ausgelöst werden. [Bernard2012b].

3.2.3.3 Installation der Netbeans IDE

Netbeans kann von der offiziellen Website bezogen werden [Oracle Corp.2012a]. Die IDE installiert als einzige eine Java-Lautzeitumgebung mit, weshalb hinterher die Umgebungsvariablen „PATH“und „JAVA_HOME“ überprüft werden sollten, ob deren Werte zur installierten Scala-Version kompatibel sind. Die Installation verläuft reibungslos und stellt durch seinen interaktiven Installer keine große Hürde dar. Das zugehörige Scala Plugin muss von der Plugin-Entwicklungsseite manuell heruntergeladen werden [Dcaoyuan2012] und über den IDE internen Plugin Manager ausgewählt und installiert werden. Ähnlich zu IntelliJ IDEA unterstützt Netbeans, Maven von Beginn an. Leider gibt es keine Möglichkeit eine manuelle Aktualisierung der Projektreferenzen vorzunehmen. Die Auswirkung dieses Umstandes kennzeichnen sich durch Compiler-Fehler, verursacht durch nicht erkannte Projektbestandteile.

3.2.4 Testen mit JUnit

Testen ist ein essentieller Bestandteil der Softwareentwicklung. Größere Projekte werden selten als monolithische Blöcke entwickelt, sondern setzen sich aus Modulen zusammen. Die Gesamtfunktionalität eines Produktes entsteht durch das Zusammenspiel dieser Module. Grundvoraussetzung dafür ist, dass einzelne Softwarebausteine die Funktion ausführen, welche von ihnen erwartet wird. Dies schafft Vertrauen in erstellte Module und erleichtert die Fehlersuche durch das Ausschließen von möglichen Fehlerquellen.

Die Erstellung aussagekräftiger Testfällen ist nicht Gegenstand dieser Arbeit. Von großem Interesse ist aber die Möglichkeit, inwieweit sich Scala-Quelltext zur gegenwärtigen Zeit effizient testen lässt. Für Scala stehen zwei neue Frameworks bereit, ScalaTest und specs, um Unit-Tests umzusetzen, jedoch ist deren IDE Unterstützung unterschiedlich ausgeprägt [Seeberger2011]. Um hier auf eine weitere Unterscheidung zu verzichten, ist der kleinste gemeinsame Nenner vorzuziehen. Dieser ist das aus Java etablierte JUnit-Framework, welches von allen wichtigen Entwicklungsumgebungen Unterstützung erfährt. Für den Entwicklungslebenszyklus einer Anwendung ist die Integration der Testfälle von großer Bedeutung. Deshalb ist die Kompatibilität des Testframeworks zu Maven als Projektkonfigurationstool sehr wichtig und im Falle von JUnit auch gegeben.

Damit im Projekt einzelne Methoden bis hin zu gesamten Projektmodulen getestet werden können, wird die Projektstruktur der Quelltexte auf eine Projektstruktur der Testfälle gespiegelt. Vor allem um Modultests mit Maven vornehmen zu können, ist die explizite Angabe dieser Pfade in der Projektkonfigurationsdatei anzugeben (siehe Kapitel 3.2.1). Weiter ist für das Testen mit Maven das „Surefire“ Plugin mit in die Projektkonfigurationsdatei aufzunehmen, welches die Ziele für JUnit-Tests innerhalb der Testphase setzt [Apache2012c]. Für jede hinzukommende Klasse im „src“ Quelltextstruktur wird eine Testklasse mit dem Suffix „Test“ in die Teststruktur eingefügt. Wie aus Abbildung 48 und Abbildung 49 zu entnehmen ist, lässt sich diese Methode für Java- und Scala-Projekte anwenden.

```
my-app
|-- pom.xml
`-- src
    |-- main
    |   `-- java
    |       `-- com
    |           `-- mycompany
    |               `-- app
    |                   `-- App.java
    `-- test
        `-- java
            `-- com
                `-- mycompany
                    `-- app
                        `-- AppTest.java
```

Abbildung 48 Maven Projektstruktur anhand eines Java-Projektes [Apache2012b]

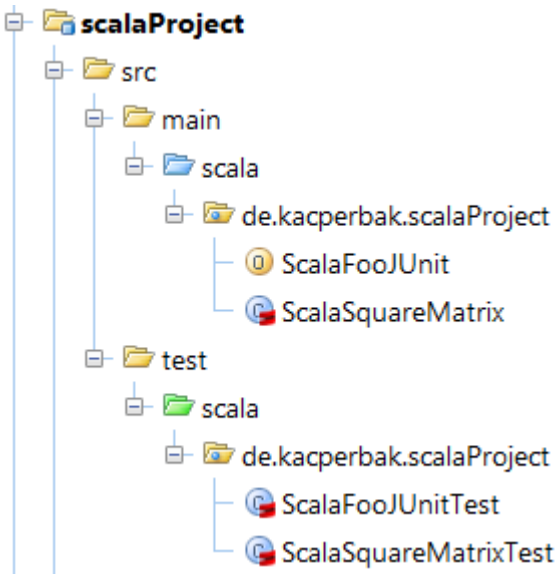


Abbildung 49 Scala Projektstruktur aus der "java-scala-matrix-test" Anwendung

Der Compiler prüft den Quelltext auf korrekte Syntax, während der Unit-Test die semantische Prüfung gewährleistet. Dazu erstellt der Entwickler Test-Methoden in denen eine spezifische Funktionalität getestet wird. Die Einleitung der Testmethode beginnt mit der Annotation „@Test“. Innerhalb der Methode werden Vorbedingungen erstellt, unter denen die Methode positiv oder negativ getestet werden soll. Dazu gehört sowohl die Ressourcen-Allokation, als auch die Erstellung von Testdaten, welche als Eingabedaten des Testes dienen. Dem Entwickler stehen dabei eine große Auswahl an „Assert“ Methoden aus dem JUnit-Framework zur Verfügung. Der Programmierer wählt dem Testfall entsprechend, die passende Methode aus [Hunt2003].

Getestet wird das Beispielobjekt „ScalaFooJUnit“. Von der Liste innerhalb des Objektes wird eine „java.lang.IndexOutOfBoundsException“ erwartet, sobald über den allokierten Speicherbereich zugegriffen wird. Dieses Verhalten testet die Methode „testRange“ innerhalb der Klasse „ScalaFooJUnitTest“ aus Listing 142.

```
1 package de.kacperbak.scalaProject
2
3 object ScalaFooJUnit {
4   val list = List(1,2,3)
5   def add(x:Int, y:Int):Int = x + y
6 }
```

Listing 142 Objekt: ScalaFooJUnit aus der Beispielanwendung: "java-scala-matrix-test"

Tritt die erwartete „IndexOutOfBoundsException“ nicht auf, so ist das Ausbleiben eines Fehlverhaltens ebenfalls als Fehler zu interpretieren, da es sich um keinen kontrollierten Zustand handelt. In diesem Fall gilt der Test als nicht bestanden und der Unit-Test macht mit einer „java.lang.AssertionErrorException“ darauf aufmerksam.

Der zweite Testfall aus Listing 143 überprüft das korrekte Verhalten der Methode „add“, indem Eingabedaten an diese übergeben werden und auf ein zu erwartendes Ergebnis geprüft wird (siehe Listing 143, Zeile 14).

```
1 package de.kacperbak.scalaProject
2
3 import org.junit.Assert._
4 import org.junit.Test
5
6 class ScalaFooJUnitTest {
7
8   @Test (value = classOf[java.lang.IndexOutOfBoundsException])
9   def testRange():Unit = {
10      for(i <- (0 until 4)) println("[" + i + "]:" + ScalaFooJUnit.list(i))
11   }
12
13   @Test
14   def testAdd(){
15     val expected = 5;
16     val actual = ScalaFooJUnit.add(2,3);
17     assertEquals(expected,actual);
18   }
19 }
```

Listing 143 Klasse: ScalaFooJUnitTest aus der Beispielanwendung: "java-scala-matrix-test"

Das Ergebnis eines Unit-Tests ist bestanden oder nicht bestanden. Dies ermöglicht eine schnelle Aussage über die Art des Fehlers, über den Namen der Testmethode und eine schnelle Eingrenzung der Ursache.

Die Ausführung von Unit-Tests wurde mit der IntelliJ IDEA Entwicklungsumgebung erfolgreich umgesetzt. Sowohl das Erstellen, als auch die Praktizierung werden sehr gut unterstützt. Ebenso ist das Starten von Testfällen über die parametrisierbare „Maven Build Configuration“ möglich. Dazu muss der Parameter „test“ übergeben werden. Bei der Ausführung werden alle Projektmodule übersetzt und getestet [O’Brien2009].

4 Fazit

Wie bereits am theoretischen Teil dieser Arbeit zu erkennen ist, bietet die Multiparadigmensprache Scala eine sehr große Breite an Möglichkeiten Probleme zu lösen. Für den routinierte Java Entwickler bedeutet das in erster Linie die OOP Konzepte von Scala zu verstehen und anzuwenden. Mit diesem Wissen ließ sich in weniger LOC die Funktions-Schicht der Beispielanwendung erstellen, siehe Kapitel 3.1.6.3.1. Auch wenn diese keine herausragend hohe Komplexität, im Vergleich zu echten Projekten aufweist, ist ein klarer Trend zu weniger Quelltext sichtbar. Weiter überzeugen die in der Sprache integrierten Entwurfsmuster, welche einen einheitlichen als auch schnellen Zugang zu bewährten Lösungsansätzen anbieten. Die Kombination von OOP und FP kann zu sehr einfachen Lösungen von komplexen Aufgaben führen, wie der Parallelisierungsvergleich in Kapitel 3.1.6.2.1.1 zwischen Java und Scala veranschaulicht hat. Zudem lassen sich Probleme in Teilprobleme einfach zerlegen und zu einer Gesamtlösung zusammensetzen, wie der Parser-Anwendungsfall in Kapitel 3.1.6.2.1.3 sehr gut verdeutlicht. All diese neue Macht muss der Programmierer aber auch beherrschen können. Der einarbeitungsaufwand in Scala ist als hoch anzusehen. Die OOP dient als Einstieg in Scala, da das funktionale Kapitel hier nur angeschnitten wurde. Nicht umsonst, besteht eine Dokument zur Klassifizierung der Scala-Skills nach welchem sich diese Arbeit auf dem zweiten Level - „intermediate application programmer“ von sechs bewegt [EPFL2012c]. Fortgeschrittene Konzepte wie Typ-Parametrisierung führen zu Themen wie Ko- und Contravarianz. Das Thema Monaden wurde aus dieser Arbeit vollkommen herausgehalten, das heißt aber nicht, dass dessen Anwendung in Scala nicht möglich ist. Ein Beispiel dafür ist die Bibliothek „scalaz“, welche die Anwender der „reinen“ funktionalen Programmierung bedient. Aus dieser stammt auch folgendes Negativbeispiel, wie weit man Scala-Code treiben kann [scalaz2012].

```
/** Cofree recursion */
trait CofreeRec_[F[_],A] {
  val extract: A
  val out: F[CofreeRec_[F,A]]
}
object CofreeRec_ {
  def apply[F[_],A](
    a: A,
    v: => F[CofreeRec_[F,A]]
  ) : CofreeRec_[F,A] =
  new CofreeRec_[F,A] {
    val extract = a
    val out = v
  }
  def unapply[F[_],A](v: CofreeRec_[F,A]) = Some((v.extract, v.out))
  implicit def unwrap[F[_],A](v: CofreeRec_[F,A]) = (v.extract, v.out)

  implicit def CofreeRec_Traverse[T[+_]:Traverse]
    : Traverse[({type λ[α]=CofreeRec_[T, α]})#λ] =
  new Traverse[({type λ[α]=CofreeRec_[T, α]})#λ] {
    def traverse[F[_]: Applicative, A, B](f : A => F[B], t : CofreeRec_[T,A]) : F[CofreeRec_[T,B]] = {
       (f(t.extract) <**> t.out.traverse(traverse(f,_)))((a, o) => CofreeRec_(a,o))
    }
  }
}
```

Abbildung 50 Beispiel aus der funktionalen „scalaz“ Bibliothek, entnommen aus: Recursion.scala

Ohne sehr gute und tiefe Kenntnisse der Scala Programmiersprache ist ein Begreifen solchen Quelltextes unmöglich. Weiter steigert der Einsatz von mächtigen Werkzeugen, wie „Implicits“, die Komplexität um eine weitere Dimension. Aus Sicht des Autors besteht damit die Gefahr, dass Programmcode erstellt wird, welcher selbst für den Verfasser, ohne ausreichende Dokumentation, mit der Zeit unverständlich wird. Diese Komplexität macht Wartungsarbeiten in fremden Programmteilen sehr aufwendig.

Die versprochenen 100% Kompatibilität hin zu Java konnte nicht bestätigt werden. Die Umsetzung der Beispielanwendung hat ergeben, dass nicht in jeder Situation eine uneingeschränkte Java/Scala-Symbiose zwischen den Sprachen herrscht. Zwar kann mittels Interfaces/Traits-Kapselung eine Lösung für das Problem gefunden werden, jedoch muss ein Architekt die Details beider Sprachen sehr gut kennen, um mögliche Fehlerquellen zu identifizieren. Weitere Inkompatibilitäten traten bei der Implementierung der „java.io. Serializable“ Schnittstelle auf, welche durch einen Kompromiss in der Architektur gelöst wurden. Mögliche Fehlerquellen können auch durch Annotationen entstehen, welche in sehr vielen Java-Frameworks eingesetzt werden. Da nicht alle Frameworks sich an Standards für

die Umsetzung von Annotationen halten, kann nicht pauschal davon ausgegangen werden, dass ein bestimmtes Java-Framework auch mit Scala funktionieren wird.

Vorstellbar ist der Einsatz von Scala in autarken Modulen in denen Scala-spezifische Vorteile, wie eine besonders einfache Lösung eines Problems mittels FP oder ein komplexes Problem das parallelisiert werden soll. Das Scala-Modul sollte keinen kritischen Bereich der Anwendung darstellen, ohne diesen Kernanwendungsfälle nicht funktionieren. Die Abhängigkeit zu diesem Modul sollte unidirektional sein, sprich das Modul darf keine Abhängigkeit zu weiteren Modulen in fremden Sprachen aufweisen. In diesem Fall ist auch der IDE-Support auf einem angemessenen Level, da keine Vermischung zwischen Sprachen stattfindet und in „reinem“ Scala entwickelt werden kann.

Das Scala-Ökosystem ist noch sehr jung. Dies merkt man besonders an der Unausgereiftheit der IDE-Plugins, vor allem in Verbindung mit Java und Maven. Zwar konnte eine stabile Konstellation an Tools (IntelliJ, Maven, JUnit) für einen flüssigen Entwicklungslebenszyklus gefunden werden, allerdings kompensierte diese Suche jegliche Art von Zeitersparnis, welche mit der Anwendung von Scala auftraten. An vielen Stellen ist noch deutlich erkennbar, dass es sich um Open Source im Wandel handelt. Dies betrifft die Sprache selbst als auch die Tools dafür. Die besten Werkzeuge stellten demnach kommerzielle Unternehmen wie, JetBrains mit seinem hervorragenden Scala-Plugin dar. Nun da auch Scala unter dem Unternehmen Typesafe weiterentwickelt wird, geht zumindest der Trend in die richtige Richtung. Solange jedoch Probleme wie binäre Inkompatibilität zwischen den Scala-Versionen bestehen und alle Komponenten immer mit der gleichen Compilerversion übersetzt werden müssen, um untereinander zu funktionieren [Lindpere2012], bleibt der Einzug von Scala in das Enterprise-Umfeld aus.

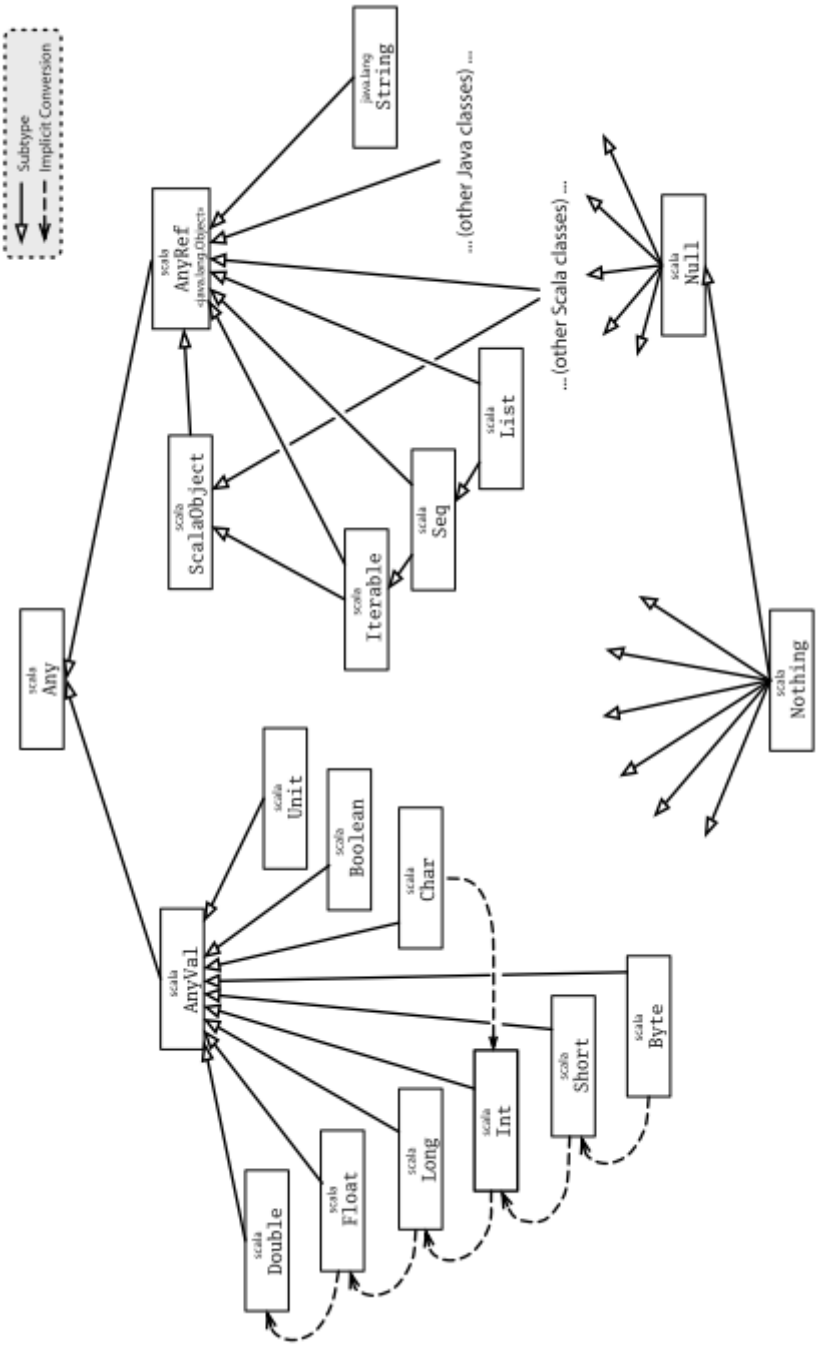
Abgesehen von den technischen Hürden, verlangt Scala dem Entwickler eine hohe Motivation ab sich die vielen neuen Konzepte beizubringen und auch Erfahrungen damit zu sammeln. Zum gegenwärtigen Zeitpunkt, ist deshalb nicht damit zu rechnen viele Scala-Experten anwerben zu können. Dies spiegelt sich auch an der gegenwärtigen Literatur wider.

Bücher über Scala selbst existieren bereits in deutscher Sprache, während Praxisbücher oder „BestPractice“ Werke gerade erst in englischer Sprache entstehen [Suereth2011].

Mit zunehmender Verbreitung können sich bestimmt mehr Unternehmen, als Startups, wie Twitter, den Einsatz von Scala vorstellen, da sowohl Webframeworks wie Lift [WorldWide Conferencing2012], als auch Persistenzlösungen, wie Circumflex ORM [Okunskiy2012], in Scala entstehen. Somit löst sich auch das Problem, der Kompatibilität zwischen den JVM-Sprachen, langsam aber sich von alleine. Das eine Ablösung von Java durch Scala in den nächsten 5 Jahren stattfindet, ist sehr unwahrscheinlich. Eine ergänzende Koexistenz wie bei C und C++ hingegen schon.

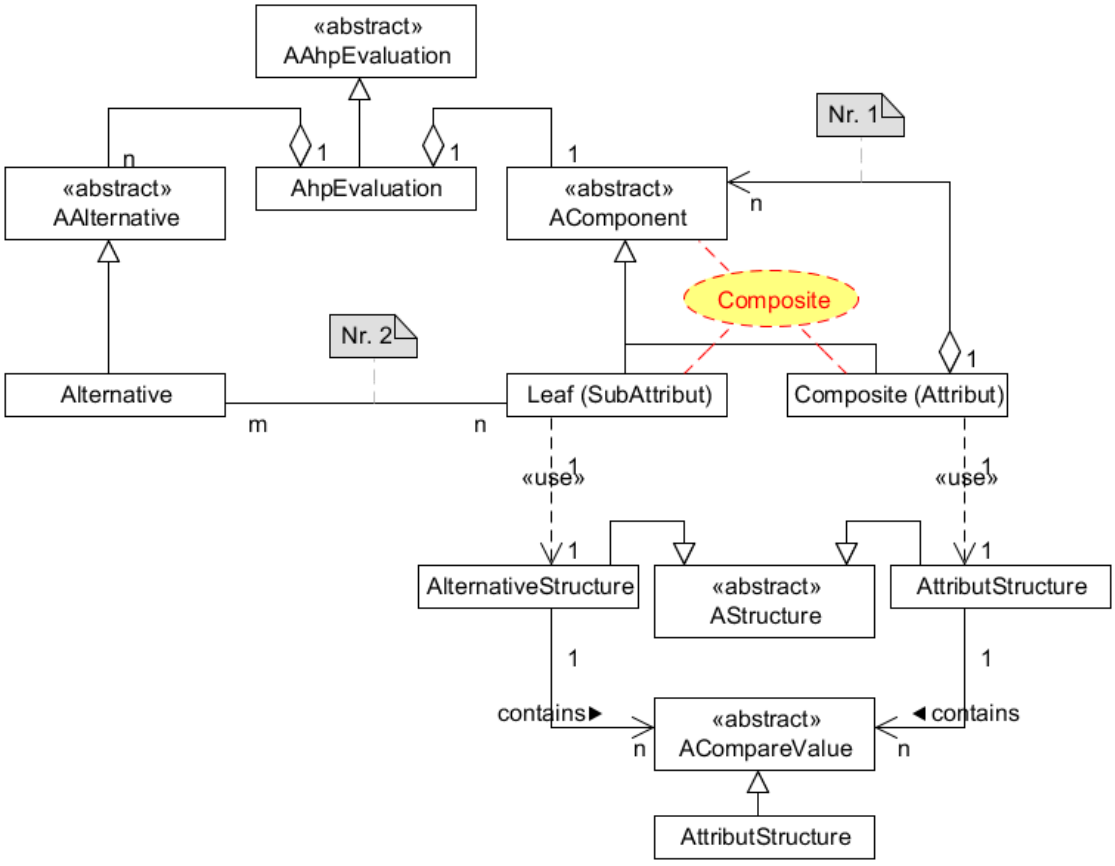
5 Anhang

5.1 Klassendiagramm der Scala-Typhierarchie

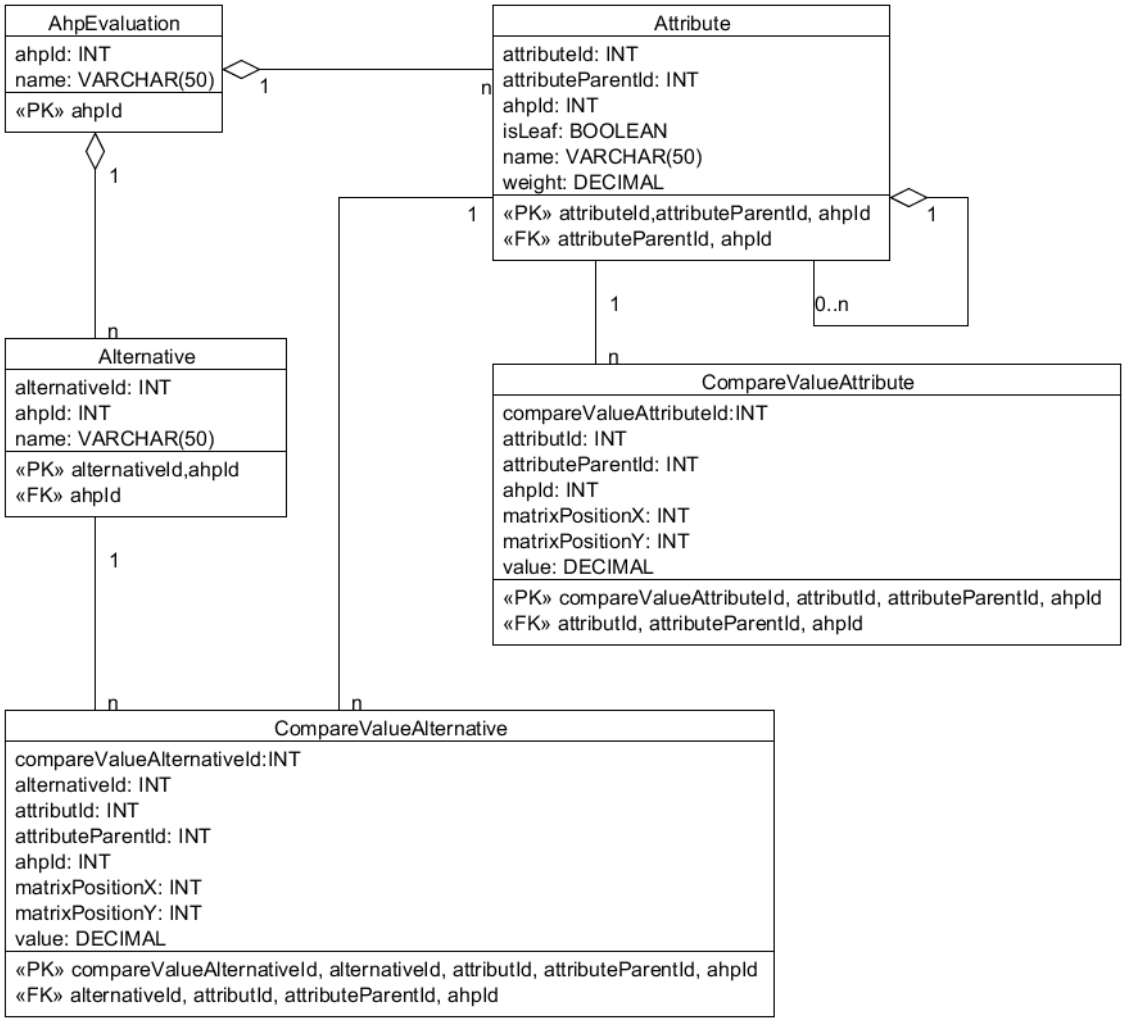


[Odersky2010]

5.2 Objektmodel der Beispielanwendung



5.3 Datenbankschema der Beispielanwendung



5.4 Inhalte des beigefügten Datenträgers

Der beiliegnde Datenträger enthält die in dieser Arbeit genannten Projekte

Name in der Arbeit:	**Name auf dem Datenträger:**
„AHP-WebApp“ mit Scala-Funktions-Schicht	„DA-App“
„AHP-WebApp“ mit Java-Funktions-Schicht	„DA-App-Java“
Benchmark zum Vergleich der Parallelisierung	"java-scala-matrix-test"

6 Abbildungsverzeichnis

Abbildung 1 Programmiersprachen Trends [Google Inc.2011] 11

Abbildung 2 Einsatzzwecke der unterschiedlichen JVM-Sprachen 15

Abbildung 3 Mögliche Aufgaben entsprechend der Einsatz-Schicht.....15

Abbildung 4 Objekte im RAM 25

Abbildung 5 Objekte im RAM 25

Abbildung 6 Kapselung mittels Traits anstelle eines Java-Interface 46

Abbildung 7 Vererbung mit Traits 49

Abbildung 8 Folgen der Vererbung mit Traits 49

Abbildung 9 Vererbungshierarchie der Klasse „Dog“ aus Listing 60 52

Abbildung 10 Verbreitung der Java-IDE’s [Redaktion JAXenter2011a] (bearbeitet).....74

Abbildung 11 Schwächen von Scala [CamelCaseCon2011] 74

Abbildung 12 Schichtenarchitektur der Beispielanwendung 76

Abbildung 13 Modifizierter Ablauf des AHPs für eine Webanwendung 80

Abbildung 14 Elementhierarchie des AHPs: Beste Java/Scala IDE 82

Abbildung 15 Allgemeiner Aufbau der Vergleichsmatrix 84

Abbildung 16 Beispielobjekte 84

Abbildung 17 Allgemeine Darstellung der Normalisierung von Saaty 87

Abbildung 18 Beispielrechnung der globalen Attributgewichte für K1.1, K1.2 und K.6.2 112

Abbildung 19 Anwendungsfalldiagramm der Beispielanwendung AHP-WebApp 114

Abbildung 20 Bedienmaske zur Bearbeitung der Attribut-Hierarchie 115

Abbildung 21 Bedienmaske zur Wertung der Paarvergleiche 116

Abbildung 22 Bedienmaske zur Berechnung und Ergebnisdarstellung der Attributgewichte 117

Abbildung 23 Ausschnitt aus der Präsentations-Schicht der AHP-WebApp. Die grau hervorgehobenen Klassen entstammen dem Wicket-Framework 119

Abbildung 24 Funktions- und Steuerungs-Schicht der Beispielanwendung: AHP-WebApp ..122

Abbildung 25 Skalierbarkeit der Berechnung innerhalb der AHP-WebApp Beispielanwendung 124

Abbildung 26 Prioritäten mit k = 2.....124

Abbildung 27 Prioritäten mit k = 3.....124

Abbildung 28 Paket-Diagramm der Datenbank-Schicht (DataBase-Layer) der Beispielanwendung AHP-WebApp.....125

Abbildung 29 Dreistufiges AHP Model [Jayaswal2006].....126

Abbildung 30 Maven standard Verzeichnisstruktur [O'Brien2009] 129

Abbildung 31 Maven Lifecycle [O'Brien2009] 130

Abbildung 32 Kapselung des Fachobjektes „AhpEvaluation" in „BusinessAhpEvaluation" zum Test der Implementierung von „java.io.Serializable" in der Scala-Business-Layer 135

Abbildung 33 Fehler: Ein „java.lang.Integer" kann nicht als Objekt verwendet werden.....138

Abbildung 34 Fork/Join-Framework nach dem „divide and conquer" Prinzip.....143

Abbildung 35 Aufteilung der Methode „sum" in Aufgaben „Tasks" und deren Zusammenführung.....144

Abbildung 36 Aufteilung einer Student-Instanz des Input-Strings in Teil-Parser.....149

Abbildung 37 AHP-WebApp mit Scala-Business-Layer 154

Abbildung 38 AHP-WebApp mit Java-Business-Layer 154

Abbildung 39 Benchmark der Parallelisierung mit der 5. Potenz.....155

Abbildung 40 Benchmark der Parallelisierung mit der 7. Potenz.....156

Abbildung 41 Erstellungszeiten der Beispielanwendung „AHP-WebApp“ mit Scala-Business-Layer.....157

Abbildung 42 Erstellungszeiten der Beispielanwendung „AHP-WebApp“ mit Java-Business-Layer.....157

Abbildung 43 Auswahl des Scala-Compilers in der IntelliJ IDEA IDE 158

Abbildung 44 Konfiguration des Scala-Compilers in der IntelliJ IDEA IDE.....158

Abbildung 45 Ergebnisse der Evaluierung: Beste Java/Scala-IDE aus der Beispielanwendung AHP-WebApp: Lokale Attributgewichte 160

Abbildung 46 Ergebnisse der Evaluierung: Beste Java/Scala-IDE aus der Beispielanwendung AHP-WebApp: Ergebnisrangfolge.....160

Abbildung 47 Ergebnisse der Evaluierung: Beste Java/Scala-IDE aus dem kommerziellen Produkt: Expert Choice 160

Abbildung 48 Maven Projektstruktur anhand eines Java-Projektes [Apache2012b].....172

Abbildung 49 Scala Projektstruktur aus der "java-scala-matrix-test" Anwendung.....172

Abbildung 50 Beispiel aus der funktionalen „scalaz“ Bibliothek, entnommen aus: Recursion.scala 175

7 Formelverzeichnis

Formel 1 Fakultät 19

Formel 2 Volumen eines Zylinders 59

Formel 3 Quadrierung der Beispiel-Evaluationsmatrix P 87

Formel 4 Maximaler Eigenwert 109

Formel 5 CI: Konsistenzindex 109

Formel 6 CR: Konsistenzverhältnis 109

Formel 7 Konsistenzprüfung am Beispiel des Flächenvergleiches - 1. Schritt 110

Formel 8 Konsistenzprüfung am Beispiel des Flächenvergleiches - 2. Schritt 111

Formel 9 Berechnung des Konsistenzindex 111

Formel 10 Verhältnisbildung und Vergleich mit Grenzwert 0,1 111

Formel 11 Quadratische Matrix 123

Formel 12 Steigender Rechenaufwand, demonstriert am Skalarprodukt 123

8 Quellenverzeichnis

[Apache2011a] Apache Software Foundation : Create a Wicket Quickstart - with Maven, Apache Wicket, Zugriff am: 1. November 2011, http://wicket.apache.org/start/quickstart.html

[Apache2012b] Apache Software Foundation : Maven Getting Started Guide - How do I make my first Maven project?, Apache Maven Project, Zugriff am: 4. März 2012 , http://maven.apache.org/guides/getting-started/index.html#What_is_Maven

[Apache2012c] Apache Software Foundation : Maven Surefire Plugin, Apache Maven Project, Zugriff am: 7. März 2012, http://maven.apache.org/plugins/maven-surefire-plugin/

[Apache2012d] Apache Software Foundation : Maven Compiler plugin - Compiling Sources Using A Different JDK, Zugriff am: 8. März 2012, http://maven.apache.org/plugins/maven-compiler-plugin/examples/compile-using-different-jdk.html

[Beaton2011] Beaton, W. u.a. : Was ist Eclipse?, August 2011, JAXenter Portal für Java - Enterprise Architekturen - SOA, Zugriff am: 30. November 2011, http://it-republik.de/jaxenter/artikel/Was-ist-Eclipse-Teil-3-4165.html

[Bentmann2012] Bentmann, B. u.a. : Maven Integration (m2e), Zugriff am: 15. Februar 2012, http://www.eclipse.org/m2e/

[Bernard2012a] Bernard, David : Using Maven with the Scala IDE for Eclipse With M2Eclipse, Scala IDE for Eclipse wiki, Zugriff am: 5. März 2012, http://www.assembla.com/wiki/show/scala-ide/With_M2Eclipse

[Bernard2012b] Bernard, David : Using Maven with the Scala IDE for Eclipse with Maven CLI, Scala IDE for Eclipse wiki, Zugriff am: 5. März 2012, http://www.assembla.com/wiki/show/scala-ide/With_Maven_CLI

[Bernard2012c] Bernard, David : github - m2eclipse-scala, Zugriff am: 15. Februar 2012, https://github.com/sonatype/m2eclipse-scala

[Braun2011] Braun, Oliver : Scala Objektfunktionale Programmierung, Hanser, 2011

[CamelCaseCon2011] camelcasecon : Auswertung der Scala-Umfrage 2011, Zugriff am: 2. Dezember 2011, http://www.camelcasecon.de/pdf/Auswertung_der_Scala_Umfrage.pdf

[Codehaus 2011] Codehaus Groovy Community : groovy: A dynamic language for the Java

platform, Zugriff am: 2. November 2011, http://groovy.codehaus.org/

[Colebourne 2011] Colebourne, Stephen: Real life Scala feedback from Yammer, 29. November 2011, Stephen Colebourne's blog, Zugriff am: 12. März 2012, http://blog.joda.org/2011/11/real-life-scala-feedback-from-yammer.html

[Dashorst2008] Dashorst, M. / Hillenius, E. : WICKET in Action, Manning Publications, 2008

[Dcaoyuan2012] Dcaoyuan : nbscala-2.9.x-0.9 - NetBeans Plugin detail, Zugriff am: 15. Februar 2012, http://plugins.netbeans.org/plugin/38999/nbscala-2-9-x-0-9

[Dewanto2012] Dewanto, Lofi : Warum Polyglot Programming nicht praxistauglich ist, 26. März 2012, heise developer, Zugriff am: 10. März 2012, http://www.heise.de/developer/artikel/Warum-Polyglot-Programming-nicht-praxistauglich-ist-1479542.html

[Dragos2012] Dragos, I. u.a. : Scala IDE for Eclipse, Zugriff am: 15. Februar 2012, http://scala-ide.org/index.html

[EC2012] Expert Choice Inc. : Expert Choice 11.5 15-Day Trial Version, Zugriff am: 20. Februar 2012, http://www.expertchoice.com/academic-program/free-trial

[Eilebrecht2010] Eilebrecht, K. /Starke, G. : Patterns kompakt – Entwurfsmuster für effektive Software-Entwicklung, 3. Auflage, Spektrum AKADEMISCHER VERLAG, 2010

[Eisele2011a] Eisele Markus : Was ist neu in Java 7 Teil 1 - Produktivität, 7. Juli 2011, heise developer, Zugriff am: 1. November 2011, http://www.heise.de/developer/artikel/Was-ist-neu-in-Java-7-Teil-1-

Produktivitaet-1274360.html

[Eisele2011b] Eisele Markus : Was ist neu in Java 7 Teil 2 – Performance, 7. Juli 2011, heise developer, Zugriff am: 1. November 2011, http://www.heise.de/developer/artikel/Was-ist-neu-in-Java-7-Teil-2-Performance-1288272.html

[EPFL2012a] École Polytechnique Fédérale de Lausanne (EPFL) : Scala comes to .Net, scala-lang.org, 2012, Zugriff am: 11. März 2012, http://www.scala-lang.org/node/10299

[EPFL2012b] École Polytechnique Fédérale de Lausanne (EPFL) : Scala License, scala-lang.org, 2012, Zugriff am: 29. Februar 2012, http://www.scala-lang.org/node/146

[EPFL2012c] École Polytechnique Fédérale de Lausanne (EPFL) : Scala levels: beginner to expert, application programmer to library designer, scala-lang.org, 2012, Zuriff am 27. April 2012, http://www.scala-lang.org/node/8610

[Evans2011] Evans, B. / Verburg, M. : ORACLE Java magazine – By and for the Java community, 11/12 2011, Polyglot Programming on the JVM, S.51

[Fatin2011] Fatin, Pavel : Real FSC support, IntelliJ Scala plugin blog, 5. Oktober 2011, Zugriff am: 29. März 2012, http://blog.jetbrains.com/scala/2011/10/05/real-fsc-support/

[Google Inc.2011] Google Inc. : google insights for search (beta), Zugriff am: 1. November 2011, http://www.google.com/insights/search/#cat=0-5-31-732&q=groovy%2Cscala%2Ckotlin%2Cclojure%2Cgosu&cmpt=q

[Grechenig2010] Grechenig, T. u.a. : Softwaretechnik - Mit Fallbeispielen aus realen Entwicklungsprojekten, Pearson Studium, 2010

[Haase2011a] Haase, Arno : Java magazin, 07/2011, Neues in Scala – Was steckt im

neuen Release?, S.39 – 42

[Haase2011b] Haase, Arno : Scala Bytes: Pimp My Library, Dezember 2011, JAXenter Portal für Java - Enterprise Architekturen - SOA, Zugriff am: 16. Februar 2012, http://it-republik.de/jaxenter/artikel/Scala-Bytes-Pimp-My-Library-4254.html

[Hickey2011] Hickey, Rich : clojure.org, Zugriff am: 2. November 2011, http://clojure.org/

[Hilyard2007] Hilyard, J. / Teilhet, S. : C# 3.0 Cookbook, Third Edition, O'Reilly Media , 2007

[Hunt2003] Hunt A. / Thomas D. : Pragmatic Unit Testing - in Java with JUnit, The Pragmatic Bookshelf, 2003

[Jayaswal2006] Jayaswal, B./ Patton, P. : Design for Trustworthy Software: Tools, Techniques, and Methodology of Developing Robust Software, Prentice Hall, 2006

[JetBrains s.r.o.2011] JetBrains s.r.o. : IntelliJ IDEA :: Best Java IDE to do more high-quality code in less time, Zugriff am: 26. Februar 2012, http://www.jetbrains.com/idea/

[Klaeren2007] Klaeren, H. / Sperber M. : Die Macht der Abstraktion – Einführung in die Programmierung, 1. Auflage, 2007

[Lindpere2012] Lindpere, Erkki : Scala: Sink or Swim? Part 2, 21. März 2012, zeroturnaround.com Blog, Zugriff am: 27. April 2012, http://zeroturnaround.com/blog/scala-sink-or-swim-part-2/

[Lütters2008] Lütters, H./ Staudacher, J. : Marketing Review St. Gallen, Februar 2008, Wirksame Unterstützung: Strategische Kontrolle mit dem Analytic Hierarchy Process

[Meixner2002] Meixner, O./Haas R. : Computergestützte Entscheidungsfindung – Expert Choice und AHP - innovative Werkzeuge zur Lösung komplexer Probleme, Wirtschaftsverlag Carl Ueberreuter, Frankfurt/Wien, 2002

[Neumann2011] Neumann, Alexander : Oracle verspricht Revolution für Java 8, heise developer, 4. Oktober 2011, Zugriff am: 2. Novemver 2011, http://www.heise.de/developer/meldung/Oracle-verspricht-Revolution-fuer-Java-8-1353558.html

[O'Brien2009] O'Brien T. u.a. : Maven – The Definitive Guide , Edition 0.8, O'REILY, 2009

[Odersky2006] Odersky, Martin : Ideas, Languages, and Programs - Pimp my Library, 2006, artima developer, Zugriff am: 20. März 2012, http://www.artima.com/weblogs/viewpostP.jsp?thread=179766

[Odersky2010] Odersky M. u.a : Programming in Scala Second Edition – A comprehensive step by step guide, Artima Press, 2010

[Odysseus Software2011a] Odysseus Software GmbH : STAN - Structure Analysis for Java - Dokumentation

[Odysseus Software2011b] Odysseus Software GmbH : STAN - Structure Analysis for Java, Zugriff am: 2. Dezember 2011, http://stan4j.com/

[Okunskiy2012] Okunskiy, B. u.a. : Circumflex ORM, Zugriff am: 27. April 2012, http://circumflex.ru/projects/orm/index.html

[Oracle Corp.2012a] Oracle Corporation, Welcome to NetBeans, Zugriff am 5. März 2012, http://netbeans.org/

[Oracle Corp.2012b] Oracle Corporation, javap - The Java Class File Disassembler, Java SE Documentation, Zugriff am: 7. März 2012, http://docs.oracle.com/javase/1.5.0/docs/tooldocs/windows/javap.htm

l

[Papula2011] Papula, Lothar : Mathematik für Ingenieure und Naturwissenschaftler Band 2 – Ein Lehr- und Arbeitsbuch für das Grundstudium, 10. Auflage, vieweg, 2001

[PENTASYS AG2012] PENTASYS AG, PENTASYS AG :: Home, Zugriff am: 7. April 2012, http://www.pentasys.de/

[Piepmeyer2010] Piepmeyer, Lothar : Grundkurs funktionale Programmierung mit Scala, Hanser, 2010

[Pollak2009] Pollak, David : Beginning Scala - Learn the powerful Scala functional-object language in a fun , interactive way, Apress, 2009

[Popp2009] Popp, Günther : Konfigurationsmanagement mit Subversion, Maven und Redmine - Grundlagen für Softwarearchitekten und Entwickler, 3. überarbeitete und erweiterte Auflage, dpunkt.verlag, 2009

[Prokopec2010] Prokopec, A. u.a. : A Generic Parallel Collection Framework, 2010, Zugriff am: 2. Januar 2012, http://infoscience.epfl.ch/record/150220/files/pc.pdf

[Redaktion JAXenter2011a] Redaktion JAXenter: Wie beliebt ist Java?, JAXenter Portal für Java - Enterprise Architekturen - SOA, Zugriff am: 1. November 2011, http://it-republik.de/jaxenter/news/Wie-beliebt-ist-Java-054341.html

[Redaktion JAXenter2011b] Redaktion JAXenter : Scala und die "Trittbrettfahrer": Kotlin, Ceylon, Gosu, 12. August 2011, JAXenter Portal für Java - Enterprise Architekturen - SOA, Zugriff am: 18. August 2011, http://it-republik.de/jaxenter/news/Scala-und-die-Trittbrettfahrer-Kotlin-Ceylon-Gosu-060062.html

[Saaty2006] Saaty, L. Thomas : Fundamentals of Decision Makingand Priority Theory

with the Analytic Hierarchie Process, Second Edition, 2006

[Saaty2008] Saaty L. Thomas : Decision making with the analytic hierarchy process, Int. J. Services Sciences Vol. 1, No. 1, 2008, Zugriff am: 1. Oktober 2011, http://www.colorado.edu/geography/leyk/geog_5113/readings/saaty_2008.pdf

[Scala Tools2010] Scala Tools : maven-scala-plugin, 2010, Zugriff am: 12. März 2012, http://scala-tools.org/mvnsites/maven-scala-plugin/index.html

[ScalaIDE wiki2012] Scala Ide for Eclipse wiki, Zugriff am: 12. März 2012, http://www.assembla.com/spaces/scala-ide/wiki/Setup

[scalaz2012] scalaz : Type Classes and Pure Functional Data Structures for Scala, Zugriff am: 27. April 2012, https://github.com/scalaz/scalaz/blob/master/core/src/main/scala/scalaz/Recursion.scala

[Schwichtenberg2010] Dr. Schwichtenberg, H. u.a. : iX - Magazin für professionelle Informationstechnik, April 2010, .NET 4 vs. Java EE 6, S.62-72

[Seeberger2011] Seeberger H. / Roelofsen R. : Durchstarten mit Scala, entwickler.press, 2011

[Steger2009] Steger, Nikolaus : Datenbanken und Informationssysteme 2, Sommersemester 2009, Hochschule Kempten

[Suereth2011] Suereth, Joshua : Scala in Depth, Manning Publications, Manning Early Access Program version 11, 2011

[TIOBE2011] TIOBE Software BV : TIOBE Programming Community Index for November 2011, Zugriff am: 1. November 2011, http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

[Typesafe Typesafe Inc., Typesafe — Stack: Overview, Zugriff am: 6. März 2012,

Inc.2012] http://typesafe.com/stack

[Ullenboom2011] Ullenboom, Christian : Java ist auch eine Insel - Das umfassende Handbuch, 9. Auflage, Galileo Computing, 2011

[Venners2009] Venners, Bill : Twitter on Scala - A Conversation with Steve Jenson, Alex Payne, and Robey Pointer, 3. April 2009, artima developer, Zugriff am: 2. Dezember 2011, http://www.artima.com/scalazine/articles/twitter_on_scala.html

[Vigdorchik2012] Vigdorchik, E. u.a. : JetBrains Plugin Repository, Zugriff am: 12. Februar 2012, http://plugins.intellij.net/plugin/?id=1347

[Wähner2012] Wähner, Kai : Babylonische Vielfalt - Sprachen für die JVM im Zusammenspiel mit Java, 24. Februar 2012, heise Developer, Zugriff am: 24. Februar 2012, http://www.heise.de/developer/artikel/Sprachen-fuer-die-JVM-im-Zusammenspiel-mit-Java-1442046.html

[Wampler2008] Wampler, D. / Payne, A. : Programming Scala, O'Reilly Media, 2008, Zugriff am: 10. April 2012, http://ofps.oreilly.com/titles/9780596155957/TypeLessDoMore.html

[Weber1993] Weber, Dr. Karl : Mehrkriterielle Entscheidungen, R.Oldenbourg Verlag GmbH, München, 1993

[WorldWide Conferencing2012] WorldWide Conferencing : Lift Web Framework, Zugriff am: 27. April 2012, http://liftweb.net/

9 Quelltextverzeichnis

Listing 1 Erstellung einer unveränderbaren Variable 16

Listing 2 Zuweisung eines neuen Wertes an eine unveränderbare Variable 16

Listing 3 Erstellung einer booleschen Variable 17

Listing 4 Zuweisung einer Fließkommazahl an eine boolesche Variable 17

Listing 5 Erstellung der Klasse „Person“ und einer Instanz „p“ 18

Listing 6 Erstellung einer Liste mit zwei Personen 18

Listing 7 Sortieren der Liste 18

Listing 8 Implementierung der Fakultät unter Java 19

Listing 9 Implementierung der Fakultät in Scala 20

Listing 10 Aufruf der erweiterten „!“ Methode aus Scala 21

Listing 11 Anwendung des „using“ Schlüsselwortes unter C# .NET 22

Listing 12 Definition der Methode „using“, welche als Kontrollstruktur verwendet werden kann 22

Listing 13 Anwendung der „using“ Methode als Kontrollstruktur 23

Listing 14 Scala-Iteration mit und ohne „break“ Kontrollstruktur 23

Listing 15 Erstellung und Verwendung einer Klasse 24

Listing 16 Initialisierende Klassen mit konstantem, öffentlichen Feld 25

Listing 17 Initialisierende Klasse mit variablem, öffentlichen Feld 26

Listing 18 Initialisierende Klasse mit variablem, öffentlichen Feld in Java 26

Listing 19 Initialisierende Klasse mit konstantem, nicht öffentlichen Feld 26

Listing 20 Klassenfelder mit fehlenden Standardwerten 27

Listing 21 Klassenfelder mit zugewiesenen Standardwerten 27

Listing 22 Klassendefinition mit privaten, aber mit initialisierten Standardwerten 28

Listing 23 Klassendefinition mit privaten, aber initialisierten Feldern 28

Listing 24 Klassendefinition mit Properties 29

Listing 25 Anwendung von Scala Properties 29

Listing 26 Anwendung von Java Getter- und Setter-Methoden 30

Listing 27 Definition einer Methode 30

Listing 28 Definition einer Prozedur in Scala 31

Listing 29 Konstante als Rückgabewert 31

Listing 30 Definition und Methodenanwendung 31

Listing 31 Operator-Notation 32

Listing 32 Demonstration der links- und rechtsassoziativen Methoden 32

Listing 33 Anwendung der „Auxiliary Constructors“ 33

Listing 34 Sperrung des Primär-Konstruktors 34

Listing 35 Initialisierung mit „named and default arguments“ 34

Listing 36 Aufruf eines „static“ Members in Java 35

Listing 37 Definition und Anwendung eines Singleton-Objektes 36

Listing 38 Scala Singleton-Object mit „main“ Methode als Einstiegspunkt 36

Listing 39 Anwendung des Singleton-Object als Begleitobjekt einer Klasse 37

Listing 40 Vererbung mit „extends“ 38

Listing 41 Beschränkung der Vererbung mit „final“ 38

Listing 42 Überschreiben der „toString“ Methode in Java und Scala 39

Listing 43 Fehlermeldung bei fehlendem „override“ Schlüsselwort 39

Listing 44 Abstrakte Klassendefinition 40

Listing 45 Beispiele für das Überschreiben der Member der Basisklasse 40

Listing 46 Übergabe des „name“ Parameters von „SaintBernard“ an „Dog“ 41

Listing 47 Verarbeitungsreihenfolge der Konstruktoren 41

Listing 48 Java-Polymorphie 41

Listing 49 Polymorphie-Vergleich von Attributen in Java und Scala 42

Listing 50 Iteration mit Primitives und Objekten 43

Listing 51 Polymorpher Zugriff auf eine Liste mit Werte- und Referenztypen 44

Listing 52 Erstellung einer leeren Liste 45

Listing 53 Anwendung eines Traits zur Kapselung von Funktionalität 46

Listing 54 Definierte Member innerhalb eines Traits 47

Listing 55 Erweiterung einer Klasse um zwei Traits 47

Listing 56 Erstellung einer Anonyme-Klasse mittels eines Traits 48

Listing 57 Vererbung mit Traits 49

Listing 58 Fehlende gemeinsame Basistypen 50

Listing 59 Unterschiedliche Basistypen 50

Listing 60 Linearisierungsbeispiel 51

Listing 61 Erstellung eines Memberkonfliktes 51

Listing 62 Funktionen und Prozedur 54

Listing 63 Verknüpfung von Funktionen 54

Listing 64 Anwendung atomarer Funktionen 55

Listing 65 Imperative und rekursive Berechnung einer Summe 55

Listing 66 Ein Funktionsliteral 56

Listing 67 Erstelllung eines Funktionswertes (Objektes) aus einem Funktionsliteral 57

Listing 68 Definition von Funktionswerten mit expliziter Typangabe 57

Listing 69 Anwendung der Funktionen auf eine Collection 58

Listing 70 Anwendung des „_“ Platzhalters 58

Listing 71 Definition von lokalen Funktionen 59

Listing 72 Erstellung einer Funktion aus einer Methode 60

Listing 73 Abgrenzung einer Funktion von einer Closure 60

Listing 74 Zuweisung einer freien Variable innerhalb einer Closure 61

Listing 75 Verknüpfung von Funktionen 61

Listing 76 Beispielobjekt mit abstrahierbarem Algorithmus 62

Listing 77 Anwendung des Beispielobjektes 63

Listing 78 Beispielobjekt mit abstrahiertem Algorithmus der for-Schleife 63

Listing 79 Abstrakte Klasse mit generischer Methode „matcher“ in Java 64

Listing 80 Implementierung der generischen Methode „matcher“ im Untertyp: „EvenNumbersIntervalMatcher“ 64

Listing 81 Anwendung des „template method“ Entwurfsmuster 65

Listing 82 Erstellung einer Kontrollstruktur 66

Listing 83 Erstellung einer Kontrollstruktur in Curry-Form 66

Listing 84 Anwendung von „using“ in Curry-Form 67

Listing 85 Typsichere Definition des Schlüsselwortes „using“ 67

Listing 86 Demonstration der Rechtsassoziativität 67

Listing 87 Betrachtung rechtsassoziativer Aufrufe 68

Listing 88 Implizite Typumwandlung 68

Listing 89 Definition des „Implicits“ im Begleit-Objekt der Klasse „Dog“ 69

Listing 90 Mustererkennung anhand von Konstanten 70

Listing 91 Mustererkennung anhand des Objekttypes.....71

Listing 92 Anwendung der Mustererkennung mit unterschiedlichen Objekttypen.....72

Listing 93 Anwendung der Mustererkennung mit Guards 72

Listing 94 Mustererkennung mit Funktionen 73

Listing 95 Anwendung einer Funktion mit Mustererkennung.....73

Listing 96 Definition einer „wicket:id“ innerhalb eines Anker-Tags im HTML-Dokument: Index.html 118

Listing 97 Verknüpfung der „wicket:id“ mit korrespondierender Java-Klasse: Index.java 118

Listing 98 Implementierung der NxN-Matrix in der Java-Klasse: CompareAlternatives.java 121

Listing 99 Auszug aus der pom.xml des Hauptprojektes der Beispielanwendung: AHP-WebApp 128

Listing 100 Wechsel in das Projektverzeichnis und Maven-Befehl zur Auslieferung der fertigen Artefakte in das lokale Repository [O’Brien2009] 129

Listing 101 pom.xml des Wurzelprojektes: DA-App (alias AHP-WebApp) 131

Listing 102 pom.xml der Webanwendung: WicketProject 132

Listing 103 Auszug Maven-Projektkonfiguration: AHP-WebApp 132

Listing 104 Erstellung der Beispielanwendung AHP-WebApp mittels Maven 133

Listing 105 Quelltext Vergleich: Java-Interface zu Scala-Trait.....136

Listing 106 Bytecode-Vergleich: Java-Interface zu Scala-Trait 137

Listing 107 Demonstration des Boxing und Unboxing in Java, zwischen Primitiven und Objekten 137

Listing 108 Zugriff über den Trait „TScala“ 138

Listing 109 Definition einer Implicit Methode, welche auf Scala-Seite Wrapper-Typen in Scala-Primitives castet 138

Listing 110 Anwendung der Implicit-Methode auf Java-Seite.....139

Listing 111 Einsatz des Fork/Join-Frameworks zur Parallelisierung der Matrix-Potenzierung 142

Listing 112 Erstellung und Anwendung einer „ForkJoinTask“ Instanz 143

Listing 113 Erstellung und Umwandlung von Collections in Scala 2.9 143

Listing 114 Anwendung der „foreach“ Methode.....144

Listing 115 Interne Umwandlung des „for“ Statements in Scala [Odersky2010].....145

Listing 116 Anwendung der „par“ Methode auf eine for-Schleife 145

Listing 117 Anwendung der „par“-Methode auf die Matrix-Multiplikation des AHPs.....146

Listing 118 Implementierung einer Java-Helper-Klasse zum vereinfachten Umgang mit eingebundener Bibliothek 147

Listing 119 Hinzufügen der benötigten Funktionalität zu einer bestehenden Klasse aus externer Bibliothek 147

Listing 120 Beispielklasse: Student 148

Listing 121 Implementierung des „int-Parsers“ 148

Listing 122 Implementierung des „studentName-Parsers“ 149

Listing 123 Implementierung des „matNr-Parsers“ 150

Listing 124 Implementierung des „points-Parsers“ 150

Listing 125 Implementierung des „oneExam-Parsers“ 150

Listing 126 Finaler Parser 151

Listing 127 Vollständige Parser-Implementierung und dessen Einsatz 151

Listing 128 Auszug aus der pom.xml des Submodules ScalaProject in der Beispielanwendung: AHP-WebApp 162

Listing 129 Anwendung des Scala-Compilers 163

Listing 130 Aufruf von Scala-Programmen 163

Listing 131 Start der interaktiven Scala-Konsole 163

Listing 132 Scala REPL 164

Listing 133 Konfiguration der eclipse.ini 166

Listing 134 Überprüfung der Installationsdetails in Eclipse 166

Listing 135 Explizite Angabe der verwendeten Scala Version passend zur „Scala IDE for Eclipse“ for Scala 2.9.x 167

Listing 136 Einstellung der Projektart in Eclipse 168

Listing 137 Aktualisierung der Projektkonfiguration in Eclipse 168

Listing 138 Bereinigung des Projektes unter Eclipse 168

Listing 139 Konfiguration des Maven-Eclipse-Plugins 169

Listing 140 Befehl zum Erstellen der Eclipse-spezifischen Projektdateien.....169

Listing 141 Überprüfung der M2_REPO Variable innerhalb von Eclipse.....170

Listing 142 Objekt: ScalaFooJUnit aus der Beispielanwendung: "java-scala-matrix-test".....173

Listing 143 Klasse: ScalaFooJUnitTest aus der Beispielanwendung: "java-scala-matrix-test"
.....173

10 Tabellenverzeichnis

Tabelle 1 AHP Skala nach Thomas L. Saaty (bearbeitet) [Saaty2006].....83

Tabelle 2 Aufstellung der Evaluationsmatrix: P.....84

Tabelle 3 Vollständig verglichene Evaluationsmatrix: P.....85

Tabelle 4 Paarvergleichsmatrix mit dem Ziel: Beste Java/Scala IDE.....90

Tabelle 5 Paarvergleichsmatrix mit dem Ziel: K1 IDE technische Merkmale.....91

Tabelle 6 Paarvergleichsmatrix mit dem Ziel: K2 Maven Unterstützung.....92

Tabelle 7 Paarvergleichsmatrix mit dem Ziel: K3 Quelltextbearbeitung.....93

Tabelle 8 Paarvergleichsmatrix mit dem Ziel: K4 Navigation.....94

Tabelle 9 Paarvergleichsmatrix mit dem Ziel: K5 Refactoring.....96

Tabelle 10 Paarvergleichsmatrix mit dem Ziel: K6 Fehlerdiagnose und Analyse.....97

Tabelle 11 Paarvergleichsmatrix mit dem Ziel: K1.1 Stabilität.....98

Tabelle 12 Paarvergleichsmatrix mit dem Ziel: K1.2 Gemischte Java/Scala Projekte.....99

Tabelle 13 Paarvergleichsmatrix mit dem Ziel: K1.3 Java 7 Unterstützung.....99

Tabelle 14 Paarvergleichsmatrix mit dem Ziel: K1.4 Plugins.....100

Tabelle 15 Paarvergleichsmatrix mit dem Ziel: K2.1 Projektimport und Bytecode-Erstellung 100

Tabelle 16 Paarvergleichsmatrix mit dem Ziel: K2.2 IDE unabhängiges Maven-Projekt.....101

Tabelle 17 Paarvergleichsmatrix mit dem Ziel: K2.3 Projekterweiterung um weiteres Modul 101

Tabelle 18 Paarvergleichsmatrix mit dem Ziel: K3.1 Fehlerhervorhebung 102

Tabelle 19 Paarvergleichsmatrix mit dem Ziel: K3.2 Quelltextvervollständigung.....102

Tabelle 20 Paarvergleichsmatrix mit dem Ziel: K3.3 Simultane Bearbeitung unabhängiger Projekte.....103

Tabelle 21 Paarvergleichsmatrix mit dem Ziel: K3.4 Syntaxhervorhebung.....103

Tabelle 22 Paarvergleichsmatrix mit dem Ziel: K4.1 Gehe zur Deklaration 104

Tabelle 23 Paarvergleichsmatrix mit dem Ziel: K4.2 Finden aller Referenzen 104

Tabelle 24 Paarvergleichsmatrix mit dem Ziel: K4.3 Anzeige der Klassenstruktur 105

Tabelle 25 Paarvergleichsmatrix mit dem Ziel: K4.4 Anzeige der Typhierarchie 105

Tabelle 26 Paarvergleichsmatrix mit dem Ziel: K5.1 Umbenennen von Elementen 106

Tabelle 27 Paarvergleichsmatrix mit dem Ziel: K5.2 Verschieben von Elementen 106

Tabelle 28 Paarvergleichsmatrix mit dem Ziel: K5.3 Automatische Verwaltung von Abhängigkeiten 107

Tabelle 29 Paarvergleichsmatrix mit dem Ziel: K5.4 Automatische Formatierung des Quellcodes 107

Tabelle 30 Paarvergleichsmatrix mit dem Ziel: K6.1 Funktionierende Haltepunkte.....108

Tabelle 31 Paarvergleichsmatrix mit dem Ziel: K6.2 Interaktive REPL Konsole 108

Tabelle 32 Zufälliger Konsistenzverhältnisindex [Saaty2006] 109

Tabelle 33 Ermittlung der globalen Alternativgewichte.....112

Tabelle 34 Ermittlung der Alternativrangfolge.....113

Tabelle 35 Kombinatoren der Regex-Parser.....150

Tabelle 36 Gegenüberstellung der Quelltextzeilen der Java/Scala Business-Layer 153

Tabelle 37 Änderung der Projektstruktur für Scala-Module in Maven 161

Tabelle 38 Eclipse Plugin URL's 167

Erklärung

gemäß §31, Abs. (7) der Rahmenprüfungsordnung für die Hochschulen in Bayern (RaPO) in der jeweils gültigen Fassung.

Ich versichere, dass ich diese Diplomarbeit selbstständig angefertigt, nicht anderweitig für Prüfungszwecke vorgelegt, alle benutzten Quellen und Hilfsmittel angegeben, sowie wörtliche und sinngemäße Zitate gekennzeichnet habe.

Kempten, den 30. April 2012 Unterschrift _____

Ermächtigung

Hiermit ermächtige ich die Hochschule Kempten zur Veröffentlichung der nachstehenden Kurzzusammenfassung meiner Arbeit, z.B. auf gedruckten Medien oder auf einer Internetseite.

Kempten, den 30. April 2012 Unterschrift _____

Kurzzusammenfassung

Mit Unterstützung der PENTASYS AG, wurde die objektfunktionale Programmiersprache Scala auf der Java Plattform ausgewertet. Neben der Abgrenzung des objektorientierten und funktionalen Paradigmas wurden Entwicklungswerkzeuge zur Editierung, Erstellung von gemischten Java/Scala Anwendungen evaluiert. Dies geschah mit dem Analytischen Hierarchieprozess, der als 3-Schicht-Architektur mit Scala-Funktions- und Steuerungs-Schicht umgesetzt wurde. Analog dazu fand eine Java Implementierung dieser Schicht statt, um die Komplexität und den Entwicklungsaufwand beurteilen zu können. Weiter wurden scala-spezifische Anwendungsfälle ausgewertet, in denen Scala gegenüber Java vorzuziehen ist. Auch wenn Scala die wesentlich mächtigere Programmiersprache darstellt, trüben die nicht 100% Interoperabilität mit Java, der Reifegrad der Werkzeuge und der hohe Einarbeitungsaufwand das Gesamtbild. Von einer kommenden Ablösung durch Scala kann gegenwärtig keine Rede sein, jedoch von einer sehr mächtigen Erweiterung.