



Abschlussprüfung Winter 2015/2016

Fachinformatiker für Anwendungsentwicklung
Dokumentation zur betrieblichen Projektarbeit

Entwicklung eines Datenbankvalidators

Tool zur Validierung einer Datenbank auf inhaltliche Restriktionen

Abgabetermin: Berlin, den 08.12.2015

Prüfungsbewerber:

Kai Sassnowski
Cauerstraße 25
10587 Berlin



Ausbildungsbetrieb:

synectic software & services gmbh
Weißenseer Weg 110
10369 Berlin

Inhaltsverzeichnis

Abbildungsverzeichnis	II
Tabellenverzeichnis	III
Listings	IV
Abkürzungsverzeichnis	V
1 Einleitung	1
1.1 Projektumfeld	1
1.2 Projektziel	1
1.3 Projektbegründung	2
1.4 Projektschnittstellen	2
1.5 Projektabgrenzung	2
2 Projektplanung	2
2.1 Projektphasen	2
2.2 Ressourcenplanung	3
2.3 Entwicklungsprozess	3
3 Analysephase	4
3.1 Ist-Analyse	4
3.2 Wirtschaftlichkeitsanalyse	5
3.2.1 „Make or Buy“-Entscheidung	5
3.2.2 Projektkosten	5
3.2.3 Amortisationsdauer	6
3.3 Nutzwertanalyse	6
3.4 Qualitätsanforderungen	7
3.5 Lastenheft/Fachkonzept	7
3.6 Zwischenstand	7
4 Entwurfsphase	8
4.1 Zielplattform	8
4.2 Architekturdesign	8
4.3 Entwurf der Benutzeroberfläche	9
4.4 Datenmodell	9
4.5 Geschäftslogik	9
4.6 Pflichtenheft/Datenverarbeitungskonzept	10
4.7 Zwischenstand	10
5 Implementierungsphase	11

5.1	Implementierung der Datenstrukturen	11
5.2	Implementierung des Kommandozeileninterfaces	11
5.3	Implementierung der Geschäftslogik	12
5.4	Zwischenstand	12
6	Abnahmephase	12
6.1	Zwischenstand	12
7	Einführungsphase	13
7.1	Zwischenstand	13
8	Dokumentation	13
8.1	Zwischenstand	13
9	Fazit	14
9.1	Soll-/Ist-Vergleich	14
9.2	Lessons Learned	14
9.3	Ausblick	15
	Eidesstattliche Erklärung	16
A	Anhang	i
A.1	Detaillierte Zeitplanung	i
A.2	Verwendete Ressourcen	i
A.2.1	Hardware	i
A.2.2	Software	ii
A.3	Lastenheft (Auszug)	ii
A.4	Kommandozeileninterface	iv
A.5	Komponentendiagramm	v
A.6	Aktivitätsdiagramm - Einlesen einer Regel	vi
A.7	Beispiel Hauptregeldatei	vii
A.8	Implementierung des Regel ADTs	vii
A.9	Implementierung des Komposit ADTs	viii
A.10	Pflichtenheft (Auszug)	viii

Abbildungsverzeichnis

1	Kommandozeileninterface	iv
2	Komponentendiagramm	v
3	Einlesen einer Regel	vi

Tabellenverzeichnis

1	Zeitplanung	3
2	Kostenaufstellung	5
3	Qualitätsanforderungen	7
4	Zwischenstand nach der Analysephase	7
5	Nutzwertanalyse Dateiformat	8
6	Zwischenstand nach der Entwurfsphase	10
7	Zwischenstand nach der Implementierungsphase	12
8	Zwischenstand nach der Abnahmephase	13
9	Zwischenstand nach der Einführungsphase	13
10	Zwischenstand nach der Dokumentation	14
11	Soll-/Ist-Vergleich	14

Listings

Listings/master.conf	vii
Listings/Rule.scala	vii
Listings/Composition.scala	viii

Abkürzungsverzeichnis

ADT	Algebraischer Datentyp (Algebraic Datatype)
API	Application Programming Interface
CI	Continuous Integration
DI	Dependency Injection
GUI	Graphical User Interface
HOCON	Human-Optimized Config Object Notation
JDBC	Java database connectivity technology
ORM	Object-relationship mapping
RDBMS	Relational database management system
SQL	Structured Query Language
TDD	Test Driven Development
UML	Unified Modeling Language
XML	Extensible Markup Language

1 Einleitung

1.1 Projektumfeld

Ausbildungsbetrieb ist die SYNECTIC SOFTWARE & SERVICES GMBH, im Folgenden als *synectic* bezeichnet. *synectic* ist ein mittelständisches Softwareunternehmen mit Sitz in Berlin. Zu den Produkten des Unternehmens gehören individuell anpassbare Softwarelösungen für die Arbeit im Bereich Case Management. Momentan beschäftigt das Unternehmen 21 Mitarbeiter.

Der Auftraggeber des Projektes ist die Entwicklungsabteilung des Produktes *syn6*.

Ziel von *syn6* ist es, Kunden neben der momentanen Desktoplösung auch eine moderne, webbasierte Lösung anzubieten.

1.2 Projektziel

Durch die Anforderungen an die Software *syn6* können die Daten innerhalb der Datenbank gewissen inhaltlichen Restriktionen unterliegen. Ein triviales Beispiel hierfür wäre, dass in einem bestimmten Feld nur Daten abgespeichert werden dürfen, die mit einem Großbuchstaben beginnen. Es sind allerdings auch komplexere Regeln möglich, die mehrere Felder aus mehreren Tabellen betreffen. Beispielsweise könnte eine Bedingung lauten, dass die Summe der Textlänge der Felder *A* und *B* die gleiche Länge wie das Feld *C* besitzen müssen.

Da sich das *syn6* noch in einem sehr frühen Entwicklungsstadium befindet, müssen momentan viele Daten noch händisch in die Datenbank eingetragen werden. Eine Prüfroutine vor dem tatsächlichen Abspeichern der Daten in die Datenbank ist dadurch nicht möglich.

Durch das manuelle Anlegen von Daten in der Datenbank ist, gerade bei komplexeren Regeln, die Chance, dass eine dieser inhaltlichen Regeln verletzt wird relativ hoch. Solche inkonsistenten Daten erschweren die Fehlersuche, da zuerst ermittelt werden muss, ob es sich um einen Fehler in der Programmlogik handelt oder ob fehlerhafte Daten vorliegen.

Ziel ist es, ein Konsolenbasiertes Tool zu entwickeln, was anhand von vordefinierten Regeln, in der Form von [SQL](#)-Abfragen, eine Datenbank auf inhaltliche Integrität überprüft. Diese Regeln müssen beliebig verschachtelbar sein. Das heißt, dass es möglich sein muss, die Ergebnismengen zweier oder mehrerer Regeln miteinander zu verknüpfen um so eine neue Regel erstellen zu können.

Das Ergebniss einer solchen Prüfung soll in einem leicht weiterverarbeitbaren Format ausgegeben werden um es bspw. in einem Monitoring System anzeigen zu lassen.

1.3 Projektbegründung

Das Hauptproblem des momentanen Validierungsprozesses ist das hohe Maß an manueller Arbeit. Da die Restriktionen teilweise sehr komplex sind und sich über mehrere Felder und Tabellen erstrecken können, ist die Chance, dass bei der manuellen Prüfung ein Fehler passiert sehr hoch. Außerdem ist dieser Prozess sehr Zeitintensiv.

Da es keinen automatisierten Weg zur Prüfung gibt, erfordern selbst kleinste Änderungen der Daten eine erneute manuelle Validierung der Datenbank. Dies kann sich massiv auf die Produktivität der Entwickler auswirken.

Des Weiteren ist es nur schwer möglich, das Ergebnis einer Prüfung weiterzuverwerten. Es existiert auch keine Möglichkeit einer Einbindung in das Continuous Integration ([CI](#)) System um die Validierung automatisch bei jedem Build auszuführen.

Diese Probleme sollen nun durch ein Tool zur automatischen Validierung der Datenbank gelöst werden. Ein weiterer Vorteil einer Softwarelösung ist es, dass sich das Tool auch problemlos auf andere Datenbanken und Projekte anwenden lässt.

1.4 Projektschnittstellen

Die Anwendung muss ausschließlich mit der zu validierenden Datenbank interagieren können. Hierfür muss der entsprechende Datenbanktreiber auf dem ausführenden System vorhanden sein.

Mit weiteren externen Systemen muss *nicht* interagiert werden.

1.5 Projektabgrenzung

Das Abschlussprojekt befasst sich nicht mit dem Weiterverarbeiten des Ergebnisses einer Prüfung. Die Integration in z. B. ein [CI](#)-System ist also nicht Bestandteil der Arbeit.

2 Projektplanung

2.1 Projektphasen

Insgesamt standen für die Umsetzung des Projekts 70 Stunden zur Verfügung. Vor Projektbeginn wurden diese auf die verschiedenen Phasen der Softwareentwicklung verteilt. Die grobe Zeitplanung inklusive der Hauptphasen lassen sich der Tabelle [1](#) entnehmen.

Eine detaillierte Übersicht der Phasen befindet sich im Anhang [A.1: Detaillierte Zeitplanung](#) auf Seite [i](#).

Projektphase	Geplante Zeit
Analysephase	7 h
Entwurfsphase	17 h
Implementierungsphase	33 h
Übergabe	1 h
Erstellen der Dokumentation	12 h
Gesamt	70 h

Tabelle 1: Zeitplanung

2.2 Ressourcenplanung

Eine vollständige Auflistung aller während der Umsetzung des Projekts verwendeten Ressourcen befindet sich im Anhang [A.2: Verwendete Ressourcen](#) auf Seite [i](#). Bei der Auswahl der verwendeten Software war es wichtig, dass hierdurch keine Zusatzkosten anfallen. Es sollte also Software verwendet werden, die entweder kostenfrei ist (z. B. Open Source), oder für die SYNECTIC bereits Lizenzen besitzt.

2.3 Entwicklungsprozess

Der Entwicklungsprozess definiert die Vorgehensweise, nach welcher das Projekt entwickelt wird. Für das Abschlussprojekt wurde sich für einen agilen Prozess nach dem Scrum Modell entschieden. Dies bot sich an, da das Projekt im Rahmen von *syn6* entwickelt wurde, welches ebenfalls einem agilen Softwareprozess folgt. Die Projektarbeit wurde als User Story innerhalb eines zweiwöchigen Sprints definiert und dem Autor zugewiesen.

Im Gegensatz zum Wasserfallmodell zeichnet sich ein agiler Entwicklungsprozess durch kurze Iterationszyklen mit häufiger Rücksprache mit dem Product Owner und den Stakeholdern aus. Hierdurch kann flexibel auf sich ändernde Anforderungen reagiert werden.

Das gesamte Projekt soll mithilfe von Test Driven Development ([TDD](#)), zu Deutsch *Testgetriebene Entwicklung*, umgesetzt werden. Das Kernprinzip von [TDD](#) besagt, dass die Softwaretests vor der tatsächlichen Implementierung geschrieben werden. Anders formuliert besagt [TDD](#), dass *keine Zeile Produktivcode* geschrieben werden darf, wenn es keinen dazugehörigen Test gibt der fehlschlägt. Dieser Prozess führt dazu, dass man sich bereits vor der Implementierung Gedanken über den Aufbau der einzelnen Komponenten machen muss. Dadurch haben die Tests nicht nur eine prüfende Funktion, sondern bestimmen maßgeblich die Architektur des Projekts mit.

Die Entwicklungsprozess des [TDD](#) wird auch als *red-green-refactor Loop* bezeichnet ([QUELLE](#)). Die einzelnen Phasen lassen sich wie folgt beschreiben:

1. Test schreiben, der fehlschlägt (red) Zuerst wird ein Test geschrieben, der die zu implementierende Funktionalität prüft. Auch wenn es redundant erscheint, sollte immer noch einmal sichergestellt

3 Analysephase

werden, dass der Test tatsächlich fehlschlägt. Dieser Schritt verifiziert, dass der Test sich auch tatsächlich korrekt verhält. Falls ein Test in dieser Phase bereits ohne Fehler durchläuft, ist er fehlerhaft, da noch keinerlei Implementierung existiert.

2. Minimum an Implementierung schreiben, sodass der Test erfolgreich durchläuft (green) Anschließend wird Produktivcode geschrieben. Wichtig hierbei ist, dass nur das *absolute Minimum* an Implementierung geschrieben wird, sodass der Test fehlerfrei durchläuft. Oft bedeutet das, dass zuerst hardkodiert die Erwartung des Tests zurückgegeben wird. Dies hat zwei Dinge zur Folge:

1. Ähnlich wie im ersten Schritt sichergestellt wurde, dass der Test fehlschlägt wenn keine Implementierung existiert, wird hierdurch festgestellt, dass der Test unter den korrekten Bedingungen ein positives Ergebnis zurückgibt.
2. Da erst weiter implementiert werden darf, sobald ein Test existiert der fehlschlägt, ist der Programmierer gezwungen weitere Tests zu schreiben, sodass das hardkodierte Ergebnis nicht mehr ausreichend ist.

3. Falls nötig Refaktorisieren des Codes (refactor) Falls benötigt wird in dieser Phase der Code refaktorisiert und restrukturiert. Da durch die ersten beiden Schritte eine vollständige Testsuite der momentan implementierten Funktionalität gewährleistet ist, wird sofort offensichtlich, wenn eine Änderung Fehler eingeführt hat.

Es ist durchaus möglich, dass diese Phase erst nach mehreren Durchläufen der ersten beiden Schritte notwendig wird.

3 Analysephase

3.1 Ist-Analyse

Da in dieser frühen Entwicklungsphase noch keine Entwicklertools existieren, muss der Großteil der Testdaten von Hand in der Datenbank angelegt werden. Hierdurch kann es passieren, dass bei der Konfiguration der teils sehr komplexen Zusammenhänge und Abhängigkeiten der Datensätze ein Fehler unterläuft. Dies hat zur Folge, dass inkonsistente Daten vorliegen, die zu Fehlern im Programmablauf führen können.

Wie in den Abschnitten [1.2 \(Projektziel\)](#) und [1.3 \(Projektbegründung\)](#) beschrieben, existiert kein automatisierter Weg eine Datenbank auf inhaltliche Integrität zu prüfen. Die Entwickler müssen die Datenbank manuell validieren. Dieser Prozess sieht in der Regel wie folgt aus:

1. Die betroffene Datenbank in einem [GUI-Programm](#) wie *pgAdmin* öffnen
2. Sämtliche betroffenen Felder und Tabellen manuell prüfen

3 Analysephase

3. Bei komplexeren Regeln muss zusätzlich noch geprüft werden, ob alle benötigten Einträge in den betroffenen Verbindungstabellen existieren

Dieser Prozess ist zeitaufwändig, monoton und gerade bei komplexeren Regeln sehr fehleranfällig. Die Fehlersuche wird hierdurch unnötig verlängert, da zuerst bestimmt werden muss, ob sich der Fehler in der Programmlogik oder in der Datenbank befindet.

3.2 Wirtschaftlichkeitsanalyse

Durch den sehr hohen zeitlichen Mehraufwand, der durch den momentan Prozess anfällt, ist eine Umsetzung des Projektes unbedingt nötig. In den folgenden Abschnitten wird erläutert, ob sich das Projekt auch aus wirtschaftlicher Sicht für das Unternehmen lohnt.

3.2.1 „Make or Buy“-Entscheidung

Da es sich bei der Problematik um ein sehr spezifisches Problem der Entwicklung von *syn6* handelt, konnten die Entwickler kein geeignetes Tool auf dem Markt finden, was allen benötigten Ansprüchen entspricht. Darum wurde sich dazu entschieden, das Projekt in Eigenentwicklung durchzuführen.

3.2.2 Projektkosten

Bei der Berechnung der Projektkosten müssen sowohl die Personalkosten, die durch die Entwicklung des Projektes anfallen, als auch die verwendeten Ressourcen berücksichtigt werden. Bei den Personalkosten muss weiterhin zwischen den Stundensätzen von Auszubildenden und Mitarbeitern unterschieden werden.

Der Stundensatz eines Mitarbeiters wird mit 37,50 € bemessen, der eines Auszubildenden mit 10 €. Für die Nutzung der Ressourcen¹ wird ein pauschaler Stundensatz von 15 € angenommen. Sämtliche Angaben stammen aus dem Controlling.

Aus diesen Werten ergeben sich die Projektkosten in Tabelle 2.

Vorgang	Zeit	Kosten pro Stunde	Kosten
Entwicklungskosten	70 h	10 € + 15 € = 25 €	1750,00 €
Fachgespräch	3 h	37,50 € + 15 € = 52,50 €	157,50 €
Code-Review	2 h	37,50 € + 15 € = 52,50 €	105,00 €
Abnahme	1 h	37,50 € + 15 € = 52,50 €	52,50 €
Projektkosten gesamt			2065,00 €

Tabelle 2: Kostenaufstellung

¹Räumlichkeiten, Arbeitsplatz, etc.

3.2.3 Amortisationsdauer

Die Automatisierung des Validierungsprozesses hat eine deutliche Zeitersparnis zur Folge.

Da es sich nicht um einen täglichen Prozess handelt, und die Dauer einer manuellen Validierung stark abhängig von der Komplexität der zu prüfenden Regel ist, ist es schwer eine Aussage über die tatsächliche Zeitersparnis zu treffen. Schätzungen der Entwickler lagen bei einer Einsparung von 15-60 Minuten pro Prüfung. Für die Berechnung der Amortisationsdauer wird davon ausgegangen, dass eine Prüfung einmal in der Woche nötig ist. Für die Zeitersparnis pro Prüfung werden als Mittelwert 37,5 Minuten angenommen.

Dies ergibt eine tägliche Ersparnis von

$$\frac{37,5 \text{ min/Woche}}{5 \text{ Tage/Woche}} = 7,5 \text{ min/Tag} \quad (1)$$

Bei einer Zeiteinsparung von 7,5 Minuten pro Tag für beide Backend Entwickler und 254 Arbeitstagen² im Jahr ergibt sich eine Zeiteinsparung von

$$2 \cdot 254 \frac{\text{Tage}}{\text{Jahr}} \cdot 7,5 \frac{\text{min}}{\text{Tag}} = 3810 \frac{\text{min}}{\text{Jahr}} \approx 63,5 \frac{\text{h}}{\text{Jahr}} \quad (2)$$

Dadurch ergibt sich eine jährliche Einsparung von

$$63,5 \text{ h} \cdot (37,5 + 15) \text{ €/h} = 3333,75 \text{ €} \quad (3)$$

Die Amortisationszeit beträgt also $\frac{2065,00 \text{ €}}{3333,75 \text{ €/Jahr}} \approx 0,6 \text{ Jahre} \approx 7 \text{ Monate}$.

3.3 Nutzwertanalyse

Neben den in 3.2.3 (Amortisationsdauer) aufgeführten wirtschaftlichen Vorteilen ergeben sich durch Realisierung des Projekts noch einige zusätzliche nicht-monitäre Vorteile.

Ein automatisierter Prüfprozess ermöglicht es, das Ergebnis einer Prüfung weiterzuarbeiten, um es z. B. in einer Weboberfläche oder einem Monitoring System anzeigen zu lassen. Außerdem ermöglicht es die Einbindung in das CI-System um die Datenbank kontinuierlich zu Prüfen.

Obwohl das Projekt im Rahmen von *syn6* entwickelt wurde, lässt es sich auch problemlos auf andere Projekte und Datenbanken Anwenden. Der tatsächliche Nutzen geht also über *syn6* hinaus.

²vgl. <http://www.schnelle-online.info/Arbeitstage/Anzahl-Arbeitstage-2015.html>

3.4 Qualitätsanforderungen

Die Qualitätsanforderungen an die Anwendung lassen sich Tabelle 3 entnehmen.

Qualitätsmerkmal	Definition
Abgabetermin einhalten	Der Abgabetermin vom 08.12.2015 ist einzuhalten.
Datenbankagnostisch	Die Anwendung muss unabhängig des verwendeten RDBMS lauffähig sein.
Konsolenbasiert	Die Anwendung muss über das Kommandozeileninterface bedienbar sein.
Wartbarkeit	Um die Wartbarkeit der Anwendung durch die Mitarbeiter von SYNECTIC zu gewährleisten, muss bei der Auswahl der Technologien darauf geachtet werden, dass diese in der Firma vertraut sind (z. B. Auswahl der Programmiersprache).

Tabelle 3: Qualitätsanforderungen

3.5 Lastenheft/Fachkonzept

- Auszüge aus dem Lastenheft/Fachkonzept, wenn es im Rahmen des Projekts erstellt wurde.
- Mögliche Inhalte: Funktionen des Programms (Muss/Soll/Wunsch), User Stories, Benutzerrollen

Beispiel Ein Beispiel für ein Lastenheft findet sich im Anhang [A.3: Lastenheft \(Auszug\)](#) auf Seite [ii](#).

3.6 Zwischenstand

Tabelle 4 zeigt den Zwischenstand nach der Analysephase.

Vorgang	Geplant	Tatsächlich	Differenz
1. Analyse des Ist-Zustands	3 h	2 h	-1 h
2. Wirtschaftlichkeitsanalyse	1 h	2 h	+1
3. Erstellen des Lastenhefts	3 h	3 h	

Tabelle 4: Zwischenstand nach der Analysephase

4 Entwurfsphase

4.1 Zielplattform

Wie in Abschnitt 1.2 (Projektziel) erwähnt, soll am Ende des Abschlussprojektes eine eigenständige Konsolenanwendung vorliegen. Die Anwendung muss vorerst nur auf Linuxsystemen lauffähig sein, da dies auch die Zielplattform von *syn6* ist.

Als Programmiersprache wurde Scala³ gewählt. Dies bot sich an, da sowohl der Autor, als auch die Backend-Entwickler SYNECTICS mit dieser Sprache bereits vertraut sind. Dies erleichtert später die Wartung des Projektes.

Für das Dateiformat der Regeldateien wurde die Human-Optimized Config Object Notation (HOCON)⁴ gewählt. Die Nutzwertanalyse, die der Entscheidung für das Dateiformat zugrunde liegt, lässt sich Tabelle 5 entnehmen. Sowohl für die Gewichtung als auch für die Bewertung der einzelnen Kriterien wurde eine Skala von 1 bis 5 verwendet⁵⁶.

Eigenschaft	Gewichtung	JSON	YAML	XML	CSV	HOCON
Scala Libraries	5	5	2	5	3	5
Implementierung	4	4	1	3	5	4
Lesbarkeit	4	4	5	2	1	5
Dateigröße	1	3	4	1	5	5
Multiline Support	4	2	5	5	1	5
Gesamt	18	68	58	66	48	86
Nutzwert		3,78	3,22	3,67	2,67	4,78

Tabelle 5: Nutzwertanalyse Dateiformat

4.2 Architekturdiseign

Um die Abhängigkeiten zwischen den einzelnen Programmkomponenten zu verwalten, wurde das Cake Pattern verwendet⁷. Beim Cake Pattern werden mithilfe von Scalas Traits Module erstellt, die dann als Mixin in anderen Komponenten verwendet werden können. Ein Vorteil gegenüber den meisten Dependency Injection (DI) Frameworks ist, dass hierbei bereits zur Kompilierungszeit festgestellt werden kann, ob alle benötigten Abhängigkeiten erfüllt sind.

Zur Kommunikation mit der zu testenden Datenbank wurde das *Slick* Framework verwendet⁸. Da die Anwendung nur reine SQL Queries ausführen muss, und daher kein Object-relationship mapping

³<http://www.scala-lang.org>

⁴<https://github.com/typesafehub/config/blob/master/HOCON.md>

⁵Gewichtung: 1 = unnötig, 5 = unbedingt erforderlich

⁶Bewertung: 1 = mangelhaft, 5 = sehr gut

⁷vgl. <http://lampwww.epfl.ch/~odersky/papers/ScalableComponent.pdf>

⁸<http://slick.typesafe.com/>

(ORM) benötigt wird, wird das Slick Framework hauptsächlich als dünner Java database connectivity technology (JDBC)-Wrapper verwendet.

Für das Parsen der Kommandozeilenparameter wurde die Library *scopt*⁹ verwendet.

4.3 Entwurf der Benutzeroberfläche

Da die Anwendung automatisiert ausgeführt werden soll, wurde auf eine grafische Benutzeroberfläche verzichtet. Stattdessen soll die Anwendung über die Kommandozeile gestartet werden können.

Wie bei Kommandozeilenprogrammen üblich, soll ein Hilfetext existieren, der die Syntax, sowie sämtliche Parameter, Optionen und Flags enthält.

Als einziger benötigter Parameter ist der Pfad zur Hauptdatei anzugeben. Zusätzlich soll es möglich sein, den Pfad anzugeben in dem sich die in der Hauptdatei referenzierten Regeldateien befinden und den Pfad der Ergebnisdatei anzupassen. Zuletzt soll ein Flag existieren, dass das Ergebnis nicht in eine Datei speichert, sondern in die Konsole ausgibt. Dies soll es ermöglichen, den Output einer Prüfung direkt in andere Skripte zu pipen um es z. B. direkt weiterzuverarbeiten.

4.4 Datenmodell

Im folgenden sollen die wichtigsten Komponenten des Datenmodells genannt und kurz erläutert werden. Ein vollständiges Komponentendiagramm befindet sich im Anhang [A.5: Komponentendiagramm](#) auf Seite [v](#).

Runner Der Runner ist dafür zuständig, auf Grundlage einer Konfigurationsdatei, einen vollständigen Testdurchlauf zu starten. Der Runner selbst besitzt relativ wenig eigene Funktionalität, sondern delegiert stattdessen die einzelnen Aufgaben, wie z. B. das Parsen der Regeln und die Ausgabe des Ergebnisses, an die zuständigen Komponenten.

Regel Eine Regel repräsentiert eine zu prüfende Bedingung. Es wird hierbei zwischen einfachen und Kompositregeln unterschieden. Beide Arten von Regeln besitzen einen Namen und eine Beschreibung. Eine simple Regel besteht zusätzlich aus einem [SQL-Query](#). Dieses Query stellt die Bedingung dar, die durch die Regel validiert werden soll. Eine Kompositregel besitzt kein eigenes Query, sondern stellt eine Verknüpfung aus mehreren anderen Regeln dar. Die verknüpften Regeln können wiederum Kompositregeln sein.

Reporter Der Reporter ist für die Ausgabe des Ergebnisses zuständig.

⁹<https://github.com/scopt/scopt>

4.5 Geschäftslogik

Der Ablauf einer Validierung lässt sich in vier Phasen unterteilen:

Einlesen der Hauptdatei Die Hauptdatei ist der Ausgangspunkt einer Prüfung. Sie enthält neben sämtlichen auszuführenden Regeln auch alle benötigten Daten für den Verbindungsaufbau mit der zu testenden Datenbank. Ein Beispiel einer solchen Datei findet sich im Anhang [A.7: Beispiel Hauptregeldatei](#) auf Seite [vii](#).

Erstellen der Regeln In dieser Phase werden sämtliche Regeln die in der Hauptdatei referenziert sind erstellt. Hierbei wird anhand des Regelnamens die dazugehörige Datei ermittelt und eingelesen. Anschließend wird die Regeldatei geparkt und das korrekte Regelobjekt erstellt. Im Anhang [A.6: Aktivitätsdiagramm - Einlesen einer Regel](#) auf Seite [vi](#) findet sich ein Aktivitätsdiagramm, dass diesen Vorgang beschreibt.

Evaluieren der Regeln Nun folgt die eigentliche Prüfung. Die Runner Komponente evaluiert sämtliche Regeln die im vorherigen Schritt erstellt wurden und erstellt daraus ein Ergebnisset.

Die Evaluation einer einfachen Regeln besteht daraus, dass das in der Regeln definierte [SQL-Query](#) abgesetzt wird. Bei einer Kompositregel werden sämtliche im Komposit definierten Regeln rekursiv evaluiert und anschließend je nach Kompositionsart miteinander verknüpft.

Ergebnisausgabe Der letzte Schritt ist die Ausgabe des Gesamtergebnisses. Der Reporter bekommt hierbei die gesamte Ergebnismenge übergeben und erstellt daraus eine [XML-Datei](#).

4.6 Pflichtenheft/Datenverarbeitungskonzept

- Auszüge aus dem Pflichtenheft/Datenverarbeitungskonzept, wenn es im Rahmen des Projekts erstellt wurde.

Beispiel Ein Beispiel für das auf dem Lastenheft (siehe Kapitel [3.5: Lastenheft/Fachkonzept](#)) aufbauende Pflichtenheft ist im Anhang [A.10: Pflichtenheft \(Auszug\)](#) auf Seite [viii](#) zu finden.

4.7 Zwischenstand

Tabelle [6](#) zeigt den Zwischenstand nach der Entwurfsphase.

Vorgang	Geplant	Tatsächlich	Differenz
1. Entwurf der Regelsyntax	3 h	1,5 h	-1,5 h
2. Entwurf des Kommandozeileninterfaces	2 h	1 h	-1 h
3. Definition des Ausgabeformats	4 h	4 h	
4. Ersellen eines UML-Komponentendiagramms der Anwendung	4 h	2 h	-2 h
5. Erstellen des Pflichtenhefts	4 h	5 h	+1 h

Tabelle 6: Zwischenstand nach der Entwurfsphase

5 Implementierungsphase

5.1 Implementierung der Datenstrukturen

Die zwei wichtigsten Datentypen, die das Herzstück der Anwendung bilden, sind Regeln und Kompositstrukturen, d.h. logische Verkettungen von Regeln.

Beide Datenstrukturen wurden als Algebraischer Datentyp ([ADT](#)) realisiert, genauer gesagt als *Summentyp*.

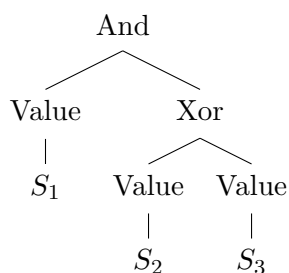
Eine Regel kann zwei mögliche Formen annehmen. Eine *SimpleRule*, d.h. eine einfache Regel, die nur aus Name, Beschreibung und dem auszuführenden [SQL-Query](#) besteht, oder eine *CompositeRule*, also eine zusammengesetzte Regel. Zusammengesetzte Regeln besitzen selbst keine [SQL-Abfrage](#), sondern enthalten stattdessen eine Verkettung von Regeln, wobei diese verketteten Regeln wiederum Kompositregeln sein können. Die implementierung des Regel [ADT](#) befindet sich im Anhang [A.8: Implementierung des Regel ADTs](#) auf Seite [vii](#).

Eine solche Verkettung von Regeln wird durch den *Composition* Datentyp abgebildet. Ein Komposit mehrere Formen annehmen, welches die Art der Verkettung repräsentiert. Die ersten vier Formen sind eine **AND**, **OR**, **XOR** oder **NAND** Verknüpfung. Eine Komposit stellt eine Baumstruktur dar, wobei jeder Zweig des Baums wiederum ein Komposit enthält. Die Blätter des Baumes enthalten den speziellen Typ **Value** der keine weitere Verkettung, sondern eine *SimpleRule* enthält.

Beispiel Gegeben sei der folgende Ausdruck, wobei S_n eine *SimpleRule* darstellt

$$S_1 \wedge (S_2 \oplus S_3)$$

Daraus würde sich also folgendes Komposit ergeben:



Die Implementierung des *Composition* Datentyps findet sich im Anhang [A.9: Implementierung des Komposit ADTs](#) auf Seite [viii](#).

5.2 Implementierung des Kommandozeileninterfaces

- Scopt Library (erstellt automatisch Hilfetext)
- Kommandozeileninterface macht nicht mehr als den Runner mit der CLI Config aufzurufen
- Screenshot des Hilfetextes im Anhang

5.3 Implementierung der Geschäftslogik

- Beschreibung des Vorgehens bei der Umsetzung/Programmierung der entworfenen Anwendung.
- Ggfs. interessante Funktionen/Algorithmen im Detail vorstellen, verwendete Entwurfsmuster zeigen.
- Quelltextbeispiele zeigen.
- Hinweis: Wie in Kapitel [1: Einleitung](#) zitiert, wird nicht ein lauffähiges Programm bewertet, sondern die Projektdurchführung. Dennoch würde ich immer Quelltextausschnitte zeigen, da sonst Zweifel an der tatsächlichen Leistung des Prüflings aufkommen können.

Beispiel Die Klasse `ComparedNaturalModuleInformation` findet sich im Anhang [??: ??](#) auf Seite [??](#).

5.4 Zwischenstand

Tabelle [7](#) zeigt den Zwischenstand nach der Implementierungsphase.

Vorgang	Geplant	Tatsächlich	Differenz
1. Implementierung des Kommandozeileninterfaces	2 h	3 h	+1 h
2. Implementierung des Regelparsers	13 h	16 h	+3 h
3. Implementierung des Testrunners	12 h	10 h	-2 h
4. Implementierung des Ergebnis Reporters	6 h	5 h	-1 h

Tabelle 7: Zwischenstand nach der Implementierungsphase

6 Abnahmephase

- Welche Tests (z. B. Unit-, Integrations-, Systemtests) wurden durchgeführt und welche Ergebnisse haben sie geliefert (z. B. Logs von Unit Tests, Testprotokolle der Anwender)?
- Wurde die Anwendung offiziell abgenommen?

Beispiel Ein Auszug eines Unit Tests befindet sich im Anhang ?? auf Seite ?. Dort ist auch der Aufruf des Tests auf der Konsole des Webserver zu sehen.

6.1 Zwischenstand

Tabelle 8 zeigt den Zwischenstand nach der Abnahmephase.

Vorgang	Geplant	Tatsächlich	Differenz
1. Abnahmetest der Fachabteilung	1 h	1 h	

Tabelle 8: Zwischenstand nach der Abnahmephase

7 Einführungsphase

- Welche Schritte waren zum Deployment der Anwendung nötig und wie wurden sie durchgeführt (automatisiert/manuell)?
- Wurden ggfs. Altdaten migriert und wenn ja, wie?
- Wurden Benutzerschulungen durchgeführt und wenn ja, Wie wurden sie vorbereitet?

7.1 Zwischenstand

Tabelle 9 zeigt den Zwischenstand nach der Einführungsphase.

Vorgang	Geplant	Tatsächlich	Differenz
1. Einführung/Benutzerschulung	1 h	1 h	

Tabelle 9: Zwischenstand nach der Einführungsphase

8 Dokumentation

- Wie wurde die Anwendung für die Benutzer/Administratoren/Entwickler dokumentiert (z. B. Benutzerhandbuch, [API-Dokumentation](#))?
- Hinweis: Je nach Zielgruppe gelten bestimmte Anforderungen für die Dokumentation (z. B. keine IT-Fachbegriffe in einer Anwenderdokumentation verwenden, aber auf jeden Fall in einer Dokumentation für den IT-Bereich).

Beispiel Ein Ausschnitt aus der erstellten Benutzerdokumentation befindet sich im Anhang ??: ?? auf Seite ??. Die Entwicklerdokumentation wurde mittels PHPDoc¹⁰ automatisch generiert. Ein beispielhafter Auszug aus der Dokumentation einer Klasse findet sich im Anhang ??: ?? auf Seite ??.

8.1 Zwischenstand

Tabelle 10 zeigt den Zwischenstand nach der Dokumentation.

Vorgang	Geplant	Tatsächlich	Differenz
1. Erstellen der Benutzerdokumentation	2 h	2 h	
2. Erstellen der Projektdokumentation	6 h	8 h	+2 h
3. Programmdokumentation	1 h	1 h	

Tabelle 10: Zwischenstand nach der Dokumentation

9 Fazit

9.1 Soll-/Ist-Vergleich

- Wurde das Projektziel erreicht und wenn nein, warum nicht?
- Ist der Auftraggeber mit dem Projektergebnis zufrieden und wenn nein, warum nicht?
- Wurde die Projektplanung (Zeit, Kosten, Personal, Sachmittel) eingehalten oder haben sich Abweichungen ergeben und wenn ja, warum?

¹⁰Vgl. ?

9 Fazit

- Hinweis: Die Projektplanung muss nicht strikt eingehalten werden. Vielmehr sind Abweichungen sogar als normal anzusehen. Sie müssen nur vernünftig begründet werden (z. B. durch Änderungen an den Anforderungen, unter-/überschätzter Aufwand).

Beispiel (verkürzt) Wie in Tabelle 11 zu erkennen ist, konnte die Zeitplanung bis auf wenige Ausnahmen eingehalten werden.

Phase	Geplant	Tatsächlich	Differenz
Entwurfsphase	19 h	19 h	
Analysephase	9 h	10 h	+1 h
Implementierungsphase	29 h	28 h	-1 h
Abnahmetest der Fachabteilung	1 h	1 h	
Einführungsphase	1 h	1 h	
Erstellen der Dokumentation	9 h	11 h	+2 h
Pufferzeit	2 h	0 h	-2 h
Gesamt	70 h	70 h	

Tabelle 11: Soll-/Ist-Vergleich

9.2 Lessons Learned

- Was hat der Prüfling bei der Durchführung des Projekts gelernt (z. B. Zeitplanung, Vorteile der eingesetzten Frameworks, Änderungen der Anforderungen)?

9.3 Ausblick

- Wie wird sich das Projekt in Zukunft weiterentwickeln (z. B. geplante Erweiterungen)?

Eidesstattliche Erklärung

Ich, Kai Sassnowski, versichere hiermit, dass ich meine **Dokumentation zur betrieblichen Projektarbeit** mit dem Thema

Entwicklung eines Datenbankvalidators – Tool zur Validierung einer Datenbank auf inhaltliche Restriktionen

selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, wobei ich alle wörtlichen und sinngemäßen Zitate als solche gekennzeichnet habe. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Berlin, den 08.12.2015

KAI SASSNOWSKI

A Anhang

A.1 Detaillierte Zeitplanung

Analysephase	7 h
1. Analyse des Ist-Zustands	3 h
1.1. Fachgespräch mit den Projektleitern	1 h
1.2. Prozessanalyse	2 h
2. Wirtschaftlichkeitsanalyse durchführen	1 h
3. Erstellen des Lastenhefts mit den Projektleitern	3 h
Entwurfsphase	17 h
1. Entwurf der Regelsyntax	3 h
1.1. Auswahl des Dateiformats	1 h
1.2. Definition der Syntax	2 h
2. Entwurf des Kommandozeileninterfaces	2 h
2.1. Aufruf in der Konsole	1 h
2.2. Benötigte Optionen und Flags	1 h
3. Definition des Ausgabeformats	4 h
3.1. Auswahl des Dateiformats	1 h
3.2. Ausgabesyntax des Ergebnisses	2 h
3.2. Erstellen einer Beispielausgabe	1 h
4. Erstellen eines UML-Komponentendiagramms der Anwendung	4 h
5. Erstellen des Pflichtenhefts	4 h
Implementierungsphase	33 h
1. Implementierung des Kommandozeileninterfaces	2 h
2. Implementierung des Regelparsers	13 h
2.1. Parsen von einfachen Regeln	3 h
2.2. Parsen von verschachtelten Regeln	10 h
3. Implementierung des Testrunners	12 h
4. Implementierung des Ergebniss Reporters	6 h
4.1. Programmierung der allgemeinen Reporter Schnittstelle	1 h
4.2. Programmierung des standard Reporters	4 h
Übergabe	1 h
1. Abnahme durch die Projektleiter	0,5 h
2. Einweisung in die Anwendung	0,5 h
Erstellen der Dokumentation	12 h
1. Erstellen der Benutzerdokumentation	2 h
2. Erstellen der Projektdokumentation	9,5 h
3. Programmdokumentation	0,5 h
3.1. Generierung durch ScalaDoc	0,5 h
Gesamt	70 h

A.2 Verwendete Ressourcen

A.2.1 Hardware

- Büroarbeitsplatz mit privatem MacBook Pro

A.2.2 Software

- OS X 10.11 El Capitan – Betriebssystem
- StarUML – Anwendung zum Erstellen von UML-Diagrammen
- Scala – Programmiersprache
- IntelliJ Community Edition – Entwicklungsumgebung Scala
- git – Verteilte Versionsverwaltung
- Gitlab – Selfhosted Repository Verwaltung
- texmaker – L^AT_EX Editor
- neovim – Moderne Distribution des Editors *vim*
- Postgresql – Datenbanksystem
- pgAdmin – Verwaltungswerkzeug für Postgres Datenbanken

A.3 Lastenheft (Auszug)

Es folgt ein Auszug aus dem Lastenheft mit Fokus auf die Anforderungen:

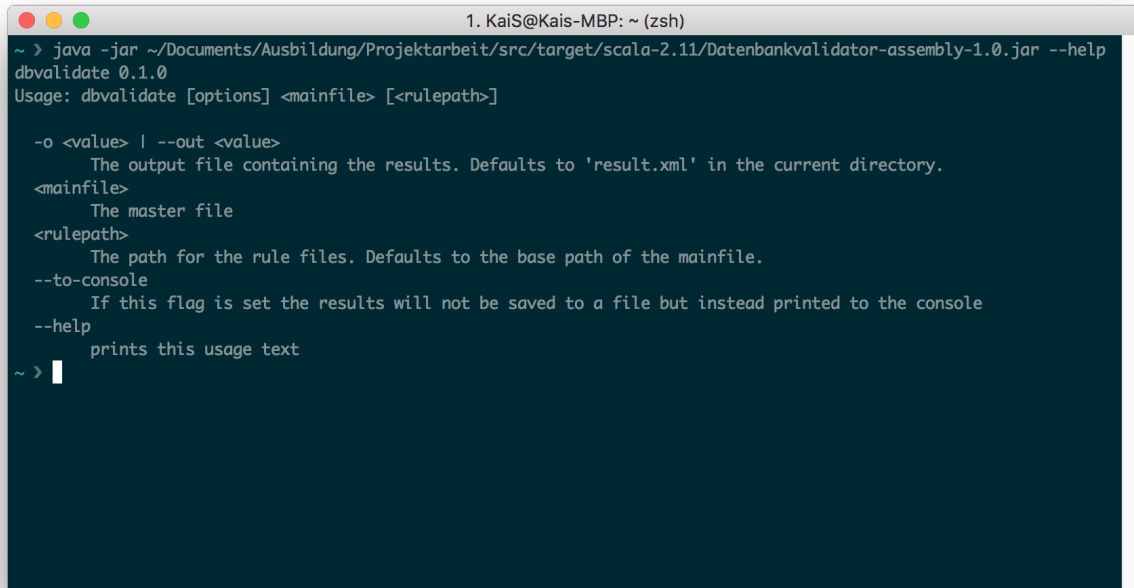
Die Anwendung muss folgende Anforderungen erfüllen:

1. Verarbeitung der Moduldaten
 - 1.1. Die Anwendung muss die von Subversion und einem externen Programm bereitgestellten Informationen (z.B. Source-Benutzer, -Datum, Hash) verarbeiten.
 - 1.2. Auslesen der Beschreibung und der Stichwörter aus dem Sourcecode.
2. Darstellung der Daten
 - 2.1. Die Anwendung muss eine Liste aller Module erzeugen inkl. Source-Benutzer und -Datum, letztem Commit-Benutzer und -Datum für alle drei Umgebungen.
 - 2.2. Verknüpfen der Module mit externen Tools wie z.B. Wiki-Einträgen zu den Modulen oder dem Sourcecode in Subversion.
 - 2.3. Die Sourcen der Umgebungen müssen verglichen und eine schnelle Übersicht zur Einhaltung des allgemeinen Entwicklungsprozesses gegeben werden.
 - 2.4. Dieser Vergleich muss auf die von einem bestimmten Benutzer bearbeiteten Module eingeschränkt werden können.
 - 2.5. Die Anwendung muss in dieser Liste auch Module anzeigen, die nach einer Bearbeitung durch den gesuchten Benutzer durch jemand anderen bearbeitet wurden.
 - 2.6. Abweichungen sollen kenntlich gemacht werden.
 - 2.7. Anzeigen einer Übersichtsseite für ein Modul mit allen relevanten Informationen zu diesem.

3. Sonstige Anforderungen

- 3.1. Die Anwendung muss ohne das Installieren einer zusätzlichen Software über einen Webbrowser im Intranet erreichbar sein.
- 3.2. Die Daten der Anwendung müssen jede Nacht bzw. nach jedem **SVN!**-Commit automatisch aktualisiert werden.
- 3.3. Es muss ermittelt werden, ob Änderungen auf der Produktionsumgebung vorgenommen wurden, die nicht von einer anderen Umgebung kopiert wurden. Diese Modulliste soll als Mahnung per E-Mail an alle Entwickler geschickt werden (Peer Pressure).
- 3.4. Die Anwendung soll jederzeit erreichbar sein.
- 3.5. Da sich die Entwickler auf die Anwendung verlassen, muss diese korrekte Daten liefern und darf keinen Interpretationsspielraum lassen.
- 3.6. Die Anwendung muss so flexibel sein, dass sie bei Änderungen im Entwicklungsprozess einfach angepasst werden kann.

A.4 Kommandozeileninterface



```
1. KaiS@Kais-MBP: ~ (zsh)
~ > java -jar ~/Documents/Ausbildung/Projektarbeit/src/target/scala-2.11/Datenbankvalidator-assembly-1.0.jar --help
dbvalidate 0.1.0
Usage: dbvalidate [options] <mainfile> [<rulepath>]

  -o <value> | --out <value>
      The output file containing the results. Defaults to 'result.xml' in the current directory.
  <mainfile>
      The master file
  <rulepath>
      The path for the rule files. Defaults to the base path of the mainfile.
  --to-console
      If this flag is set the results will not be saved to a file but instead printed to the console
  --help
      prints this usage text
~ > 
```

Abbildung 1: Kommandozeileninterface

A.5 Komponentendiagramm

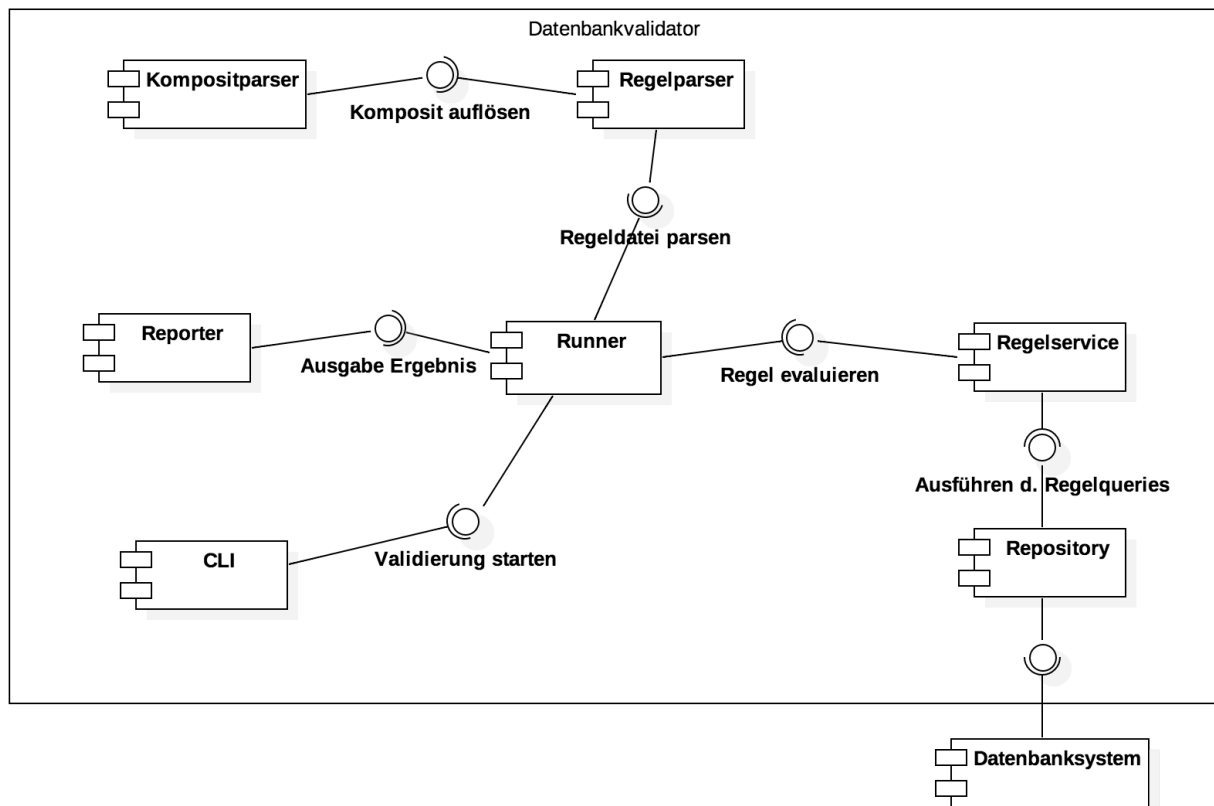


Abbildung 2: Komponentendiagramm

A.6 Aktivitätsdiagramm - Einlesen einer Regel

Das folgende Diagramm beschreibt das Einlesen einer Regel aus einer Regeldatei.

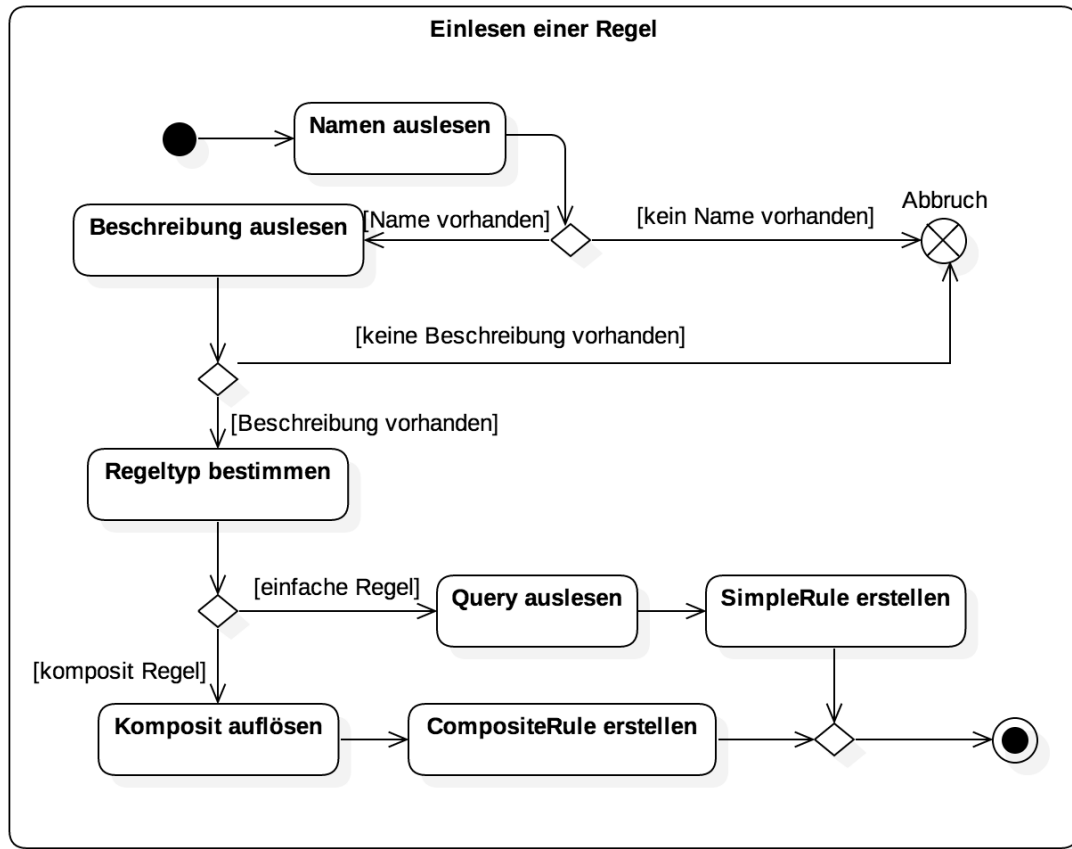


Abbildung 3: Einlesen einer Regel

A.7 Beispiel Hauptregeldatei

Ein Beispiel einer Hauptdatei. In der Hauptdatei werden sowohl die Datenbankverbindungsparameter beschrieben als auch sämtliche Regeln aufgelistet, die evaluiert werden sollen.

```

1 database {
2     driver = "org.sqlite.JDBC"
3     url = "jdbc:sqlite:/Users/KaiS/Projects/Abschlussprojekt/src/db.db"
4     user = ""
5     password = ""
6 }
7
8 rules = [
9     one_greater_than_two,
10    two_greater_than_one,
11    first_and_second,
12    first_or_second
13 ]

```

A.8 Implementierung des Regel ADTs

```

1 package Rule
2
3 /**
4  * An ADT representing a rule.
5  *
6  * A rule can either be a SimpleRule containing only the query through
7  * which it is defined, or a CompositeRule which itself does not contain
8  * a query, but a composition of rules.
9  */
10 sealed trait Rule {
11     def name: String
12     def description: String
13 }
14
15 case class SimpleRule(name: String, description: String, query: String) extends Rule
16 case class CompositeRule(name: String, description: String, composition: Composition) extends Rule

```

A.9 Implementierung des Komposit ADTs

```
1 package Rule
2
3 /**
4  * An ADT representing a composition of rules.
5  *
6  * Each branch of the composition is itself a composition. A composition
7  * containing only a single rule represents a leaf of the tree and is called
8  * a Value.
9  */
10 sealed trait Composition
11
12 case class And(left: Composition, right: Composition) extends Composition
13 case class Or(left: Composition, right: Composition) extends Composition
14 case class NAnd(left: Composition, right: Composition) extends Composition
15 case class XOr(left: Composition, right: Composition) extends Composition
16 case class Value(rule: Rule) extends Composition
```

A.10 Pflichtenheft (Auszug)

Zielbestimmung

1. Musskriterien

1.1. Modul-Liste: Zeigt eine filterbare Liste der Module mit den dazugehörigen Kerninformationen sowie Symbolen zur Einhaltung des Entwicklungsprozesses an

- In der Liste wird der Name, die Bibliothek und Daten zum Source und Kompilat eines Moduls angezeigt.
- Ebenfalls wird der Status des Moduls hinsichtlich Source und Kompilat angezeigt. Dazu gibt es unterschiedliche Status-Zeichen, welche symbolisieren in wie weit der Entwicklungsprozess eingehalten wurde bzw. welche Schritte als nächstes getan werden müssen. So gibt es z. B. Zeichen für das Einhalten oder Verletzen des Prozesses oder den Hinweis auf den nächsten zu tätigenden Schritt.
- Weiterhin werden die Benutzer und Zeitpunkte der aktuellen Version der Sourcen und Kompilate angezeigt. Dazu kann vorher ausgewählt werden, von welcher Umgebung diese Daten gelesen werden sollen.
- Es kann eine Filterung nach allen angezeigten Daten vorgenommen werden. Die Daten zu den Sourcen sind historisiert. Durch die Filterung ist es möglich, auch Module zu finden, die in der Zwischenzeit schon von einem anderen Benutzer editiert wurden.

1.2. Tag-Liste: Bietet die Möglichkeit die Module anhand von Tags zu filtern.

- Es sollen die Tags angezeigt werden, nach denen bereits gefiltert wird und die, die noch der Filterung hinzugefügt werden könnten, ohne dass die Ergebnisliste leer wird.

- Zusätzlich sollen die Module angezeigt werden, die den Filterkriterien entsprechen. Sollten die Filterkriterien leer sein, werden nur die Module angezeigt, welche mit einem Tag versehen sind.

1.3. Import der Moduldaten aus einer bereitgestellten **CSV!**-Datei

- Es wird täglich eine Datei mit den Daten der aktuellen Module erstellt. Diese Datei wird (durch einen Cronjob) automatisch nachts importiert.
- Dabei wird für jedes importierte Modul ein Zeitstempel aktualisiert, damit festgestellt werden kann, wenn ein Modul gelöscht wurde.
- Die Datei enthält die Namen der Umgebung, der Bibliothek und des Moduls, den Programmtyp, den Benutzer und Zeitpunkt des Sourcecodes sowie des Kompilats und den Hash des Sourcecodes.
- Sollte sich ein Modul verändert haben, werden die entsprechenden Daten in der Datenbank aktualisiert. Die Veränderungen am Source werden dabei aber nicht ersetzt, sondern historisiert.

1.4. Import der Informationen aus **SVN!** (**SVN!**). Durch einen „post-commit-hook“ wird nach jedem Einchecken eines Moduls ein **PHP!**-Script auf der Konsole aufgerufen, welches die Informationen, die vom **SVN!**-Kommandozeilentool geliefert werden, an **NatInfo!** übergibt.

1.5. Parsen der Sourcen

- Die Sourcen der Entwicklungsumgebung werden nach Tags, Links zu Artikeln im Wiki und Programmbeschreibungen durchsucht.
- Diese Daten werden dann entsprechend angelegt, aktualisiert oder nicht mehr gesetzte Tags/Wikiartikel entfernt.

1.6. Sonstiges

- Das Programm läuft als Webanwendung im Intranet.
- Die Anwendung soll möglichst leicht erweiterbar sein und auch von anderen Entwicklungsprozessen ausgehen können.
- Eine Konfiguration soll möglichst in zentralen Konfigurationsdateien erfolgen.

Produkteinsatz

1. Anwendungsbereiche

Die Webanwendung dient als Anlaufstelle für die Entwicklung. Dort sind alle Informationen für die Module an einer Stelle gesammelt. Vorher getrennte Anwendungen werden ersetzt bzw. verlinkt.

2. Betriebsbedingungen

Die nötigen Betriebsbedingungen, also der Webserver, die Datenbank, die Versionsverwaltung, das Wiki und der nächtliche Export sind bereits vorhanden und konfiguriert. Durch einen täglichen Cronjob werden entsprechende Daten aktualisiert, die Webanwendung ist jederzeit aus dem Intranet heraus erreichbar.