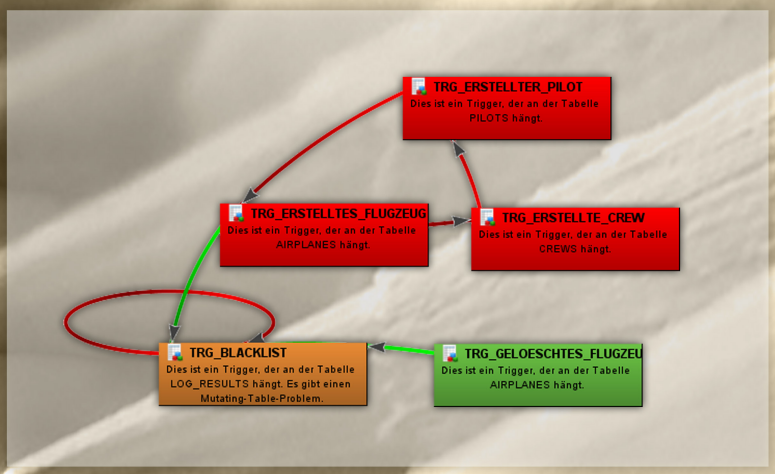
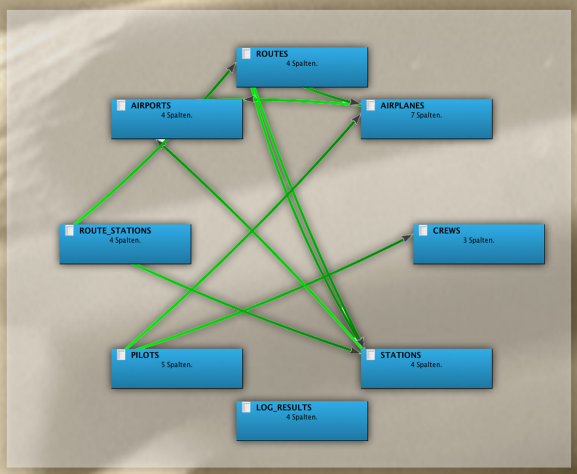
Eingereichte Diplomarbeit (August 2009)

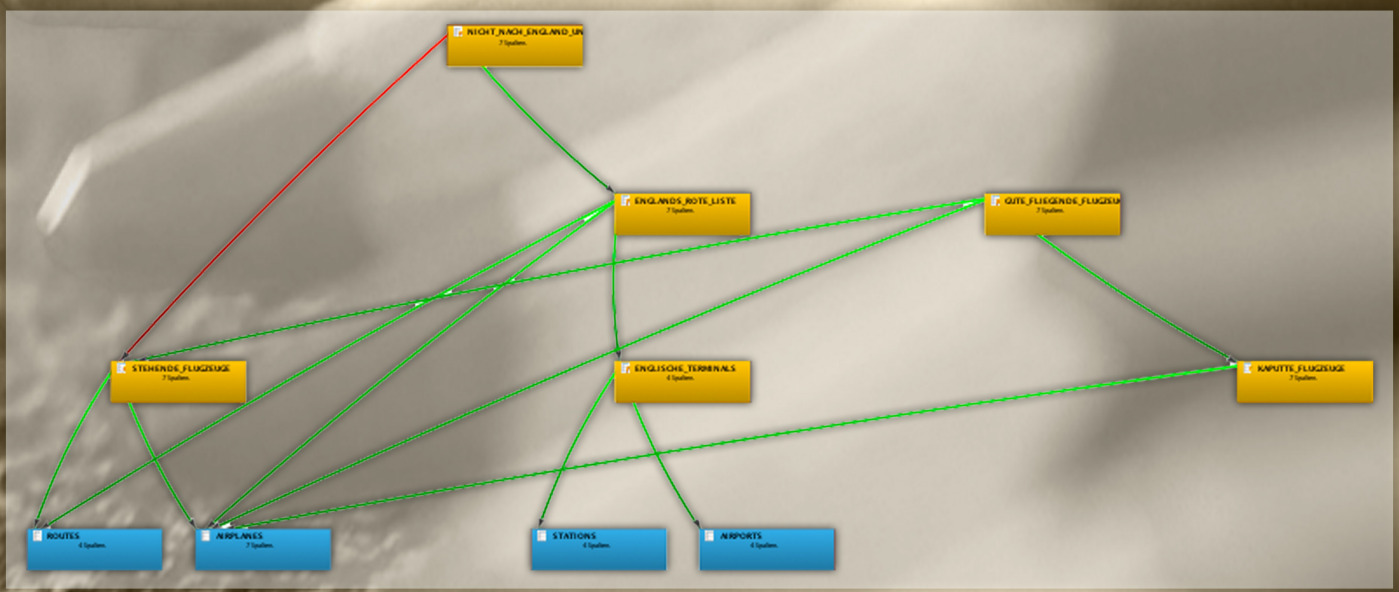
Fachhochschule Köln
Cologne University of Applied Sciences

Visualisierung der Abhängigkeiten von Datenbankobjekten

Andre Kasper
Jan Philipp







Diplomarbeit an der Fachhochschule Köln, Campus Gummersbach, an der Fakultät für Informatik und Ingenieurwissenschaften

Visualisierung der Abhängigkeiten von Datenbankobjekten

Diskussion, Bewertung und Realisierung der wechselseitigen und rekursiven Abhängigkeiten zwischen Tabellen, Views und Triggern in einem Datenbanksystem.

Die entwickelte Software „visualDependencies for databases“ ist ab sofort unter der Internetadresse `http://drop.io/visualDependencies` verfügbar.

Abgabe:	05. August 2009
Note:	**sehr gut (1,0)**
Letzte Korrektur:	21. August 2009 (Veröffentlichung)
Letzte Änderung:	10. Juni 2010 (Auszeichnungen)

Die Autoren

Jan Philipp	`ak1709@googlemail.com`
Andre Kasper	`jan@philipp-online.de`

Auszeichnungen

Opitz Consulting Innovationspreis 2009, 3. Platz

Der Innovationspreis[1] (3. Platz) für Informatik wurde am 11. Dezember 2009 von der Firma Optiz Consulting in Gummersbach verliehen. Optiz lobte diesen Preis zum ersten Mal aus.

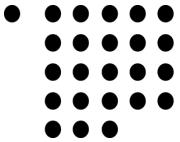
Presse: oberberg-aktuell.de[2].

Univention Absolventenpreis 2010, 1. Platz

Mit dem 1. Platz des Absolventenpreis 2010[3] der Firma Univention wurde die Diplomarbeit auf der Berliner Messe „LinuxTag 2010“[4] prämiert. Der am 09. Juni 2010 vergebene Preis belohnte die Diplomarbeit unter anderem unter dem Gesichtspunkt von innovativen Open Source Software. Univention lobte diesen Preis zum dritten Mal aus.

Presse: pro-linux.de[5], linux-magazin.de[6], heise.de/open[7], golem.de[8].

[1] http://www.opitz-consulting.com/unternehmen/opitz_consulting_innovationspreis.php
[2] http://www.oberberg-aktuell.de/index.php?id=144&tx_ttnews%5Btt_news%5D=103535
[3] http://www.univention.de/preistraeger10.html
[4] http://www.linuxtag.org/2010
[5] http://www.pro-linux.de/news/1/15766/univention-zeichnet-abschlussarbeiten-aus.html
[6] http://www.linux-magazin.de/NEWS/Linuxtag-2010-Absolventenpreis-belohnt-Datenbank-Visualisierung
[7] http://www.heise.de/open/meldung/LinuxTag-2010-Univention-Absolventenpreis-vergeben-1018038.html
[8] http://www.golem.de/1006/75687.html



Fachhochschule Köln
Cologne University of Applied Sciences

Visualisierung der Abhängigkeiten von Datenbankobjekten

Am Institut für Informatik
an der Fachhochschule Köln
Campus Gummersbach

im Studiengang Allgemeine Informatik

zur
Erlangung des Grades Diplom-Informatiker (FH)
eingereichte

D i p l o m a r b e i t

vorgelegt von

Andre Kasper und Jan Philipp

Erster Prüfer:	Prof. Dr. Heide Faeskorn-Woyke
Zweiter Prüfer:	Dr. Andreas Behrend

Gummersbach, im August 2009

Inhaltsverzeichnis

Abbildungsverzeichnis 7

Tabellenverzeichnis 10

Abkürzungs- und Symbolverzeichnis 11

1 Danksagung 12

2 Einleitung 15

3 Aufgabenbeschreibung 18
- 3.1 Relevante Objekte in der Datenbank . . . 18
- 3.2 Ansichten . . . 20
- 3.3 Details zu einer Abhängigkeit . . . 20
- 3.4 Datenbanksystem . . . 21
- 3.5 Vorhandene funktionale Anforderungen . . . 21
- 3.6 Vorhandene nicht-funktionale Anforderungen . . . 23
- 3.7 Beispiel . . . 24

4 Diskussion und Vergleich vorhandener Softwareprodukte 25
- 4.1 Nicht-funktionale Anforderungen . . . 26
- 4.2 Funktionale Anforderungen . . . 26
- 4.3 Übersicht . . . 27
- 4.4 Vergleich . . . 27
  - 4.4.1 DbVisualizer . . . 28
  - 4.4.2 Microsoft Office Access . . . 30
  - 4.4.3 MySQL Workbench . . . 31
  - 4.4.4 Oracle SQL Developer . . . 33
  - 4.4.5 SQL Developer . . . 35
  - 4.4.6 Toad . . . 38
- 4.5 Fazit . . . 40

5 Vergleich und Diskussion von Frameworks zur Entwicklung 42
- 5.1 SQL-Parser . . . 43
  - 5.1.1 Nicht-funktionale Anforderungen . . . 44
  - 5.1.2 Funktionale Anforderungen . . . 45

5.1.2.1 FROM-Komponente 46
5.1.2.2 WHERE-Komponente 46
5.1.2.3 HAVING-Komponente 46
5.1.2.4 PL/SQL 46
5.1.3 Übersicht 47
5.1.4 Vergleich 47
5.1.4.1 GSP - General SQL Parser 48
5.1.4.2 SQL Query Parser 50
5.1.4.3 SQLJEP 50
5.1.4.4 SqlParser von Zoran Milakovic 51
5.1.4.5 ZQL-Parser 51
5.1.4.6 Parsen eines Triggers in PL/SQL 52
5.1.5 Fazit 53
5.2 Graphenframeworks 54
5.2.1 Nicht-funktionale Anforderungen 55
5.2.2 Funktionale Anforderungen 55
5.2.3 Übersicht 57
5.2.4 Vergleich 57
5.2.4.1 Graphviz 58
5.2.4.2 JGraph 60
5.2.4.3 JUNG 62
5.2.4.4 prefuse 64
5.2.4.5 Andere Frameworks 64
5.2.4.6 JavaFX 65
5.2.5 Fazit 66

6 Implementierung **68**
6.1 Funktionen 69
6.1.1 Technischer Hinweis 70
6.1.2 Anforderungen 70
6.1.3 Spezialfunktionen 72
6.2 Architektur 72
6.2.1 Übersicht 72
6.2.2 Muster 73
6.2.2.1 Observer und Listener 73
6.2.2.2 Model-View-Controller 74
6.2.2.3 Allgemeine Erzeugermuster 75

6.2.2.4 Data Access Object 77
6.2.2.5 Flyweight 78
6.2.2.6 Strategiemuster 79
6.2.2.7 Andere Muster 80
6.2.3 Externe Frameworks, APIs und Werkzeuge 80
6.2.3.1 Persistenz 80
6.2.3.2 Layout 83
6.2.4 Allgemeiner Aufbau 84
6.2.5 Kapselung der konkreten Fremdkomponenten 85
6.2.6 Metadata 86
6.2.7 Serialisierte XML-Schemata 87
6.3 Analyse und Parsen 89
6.3.1 Aufbau 90
6.3.2 Die Metadaten-Fabrik 91
6.3.3 Die Metadaten 92
6.3.4 Analyse 93
6.3.5 Parsen 97
6.3.5.1 Parsen einer View 98
6.3.5.2 Parsen eines Triggers 101
6.4 Graphen 102
6.4.1 Übersicht zur API 103
6.4.2 Das JUNG2-System 104
6.4.3 Typisierter Graph 105
6.4.4 Unterschiedliche Ansichten von Graphen 105
6.4.5 Individuelle Visualisierung 106
6.4.6 Layout 106
6.4.6.1 Basis 106
6.4.6.2 Definition 107
6.4.6.3 Rekursionsproblematik 107
6.4.6.4 Erweiterungen 108
6.4.6.5 Bestimmen der Knoten-Level 108
6.4.6.6 Laufzeit 109
6.4.7 Zentrale Komponenten 110
6.4.7.1 VisualizationViewer 110
6.4.7.2 Renderer 111
6.4.7.3 GraphMouse 111

6.4.8 Aufbau . . . 112
6.4.9 Fazit . . . 113

7 Vorstellung und Übersicht der entwickelten Software 114
7.1 Anzeige der Verbindungsübersicht . . . 115
7.2 Anzeige der Viewabhängigkeiten . . . 116
7.3 Anzeige der Triggerabhängigkeiten . . . 117
7.4 Anzeige des Entity-Relationship-Diagramms . . . 117

8 Testen 119
8.1 Exkurs: Die Psychologie des Testens . . . 121
8.2 Testfälle für die Auswahl eines Parsers . . . 122
8.2.1 Parsen der einzelnen SQL-Anweisungselemente . . . 122
8.2.2 Parsen von PL/SQL . . . 123
8.2.3 Fazit . . . 124
8.3 Statisches Testen . . . 124
8.3.1 FindBugs/PMD . . . 125
8.3.1.1 Datenflussanalyse . . . 126
8.3.1.2 Kontrollflussanalyse . . . 128
8.3.1.3 Prüfung auf Einhaltung der Konventionen und Standards 130
8.3.2 Fazit . . . 131
8.4 Dynamisches Testen . . . 131
8.4.1 Code-Coverage . . . 132
8.4.2 JUnit . . . 134
8.4.3 Fazit . . . 135
8.5 Weitere Testverfahren . . . 136
8.5.1 Vorstellung der einzelnen Testverfahren . . . 136
8.5.1.1 Crashtest . . . 137
8.5.1.2 Installationstest . . . 137
8.5.1.3 Interoperabilitätstest . . . 137
8.5.1.4 Internet-Abschalttest . . . 137
8.5.1.5 Oberflächentest . . . 138
8.5.1.6 Smoketest . . . 139
8.5.1.7 Wiederinbetriebnahmetest . . . 139
8.5.2 Fazit . . . 139

9 Fazit 141

9.1 Zusammenfassung 141
9.2 Ausblick 143

Literaturverzeichnis **146**

A Anhang – Bilder **154**

B Anhang – Programmcodes **179**

C Anhang – Testfallspezifikationen und Auswertungen **192**

D Anhang – Backus-Naur-Formen **195**

Abbildungsverzeichnis

3.1 Beispiel einer View-Hierarchie mit drei Ebenen 20
3.2 Konzeptzeichnung für den Startbildschirm, angelehnt an Programmen wie Oracle SQL Developer 22
3.3 Konzeptzeichnung für den Inhalt einer Datenbankverbindung 23
4.1 DbVisualizer – grafische Anzeige der Abhängigkeite 29
4.2 DbVisualizer – tabellarische Anzeige der Abhängigkeiten 30
4.3 Microsoft Office Access – ER-Diagramm 31
4.4 MySQL Workbench – EER-Diagramm 33
4.5 Oracle SQL Developer – Abhängigkeiten 35
4.6 SQL Developer – Diagramm 37
4.7 TOAD – Entity-Relationship-Diagramm 39
4.8 TOAD – Abhängigkeiten einer View 40
4.9 TOAD – Abhängigkeiten aller Views 40
5.1 Graphendarstellung mit Satellitenfenster 62
6.1 UML-Anwendungsfalldiagramm der gesamten Anwendung 71
6.2 Screenshot des Fortschrittsbalkens mit Detailinformationen des Status 74
6.3 MVC Architektur als UML2-Diagramm 75
6.4 Die *einfache Fabrik* als UML2-Diagramm 76
6.5 Die Entitäten als UML2-Diagramm 77
6.6 Formulardialog – Neue Verbindung anlegen 84
6.7 Die grobe Architektur als UML2-Diagramm 84
6.8 Die Klasse MetaDataFactory als UML-Diagramm 91
6.9 Beispiel einer Darstellung von Graphen des *JUNG2*-Frameworks 104
7.1 Logo und Ladebild der Anwendung *visualDependencies* 114
A.1 ER-Diagramm der Tabellen aus dem Beispielschema „Flughafen & Flugzeuge“ 154
A.2 ER-Diagramm der Views und Trigger aus dem Beispielschema „Flughafen & Flugzeuge“ 155
A.3 Früher Prototyp einer View-Hierarchie mit *JGraph*. Die Objekte wurden nachträglich repositioniert. 156
A.4 Beispielhafter Screenshot des Programmes *JGraphpad Pro Diagram Editor*. . 156
A.5 Screenshot 1/2 einer *JUNG*-Anwendung zur Demonstrationszwecken (in JUNG enthalten). 157
A.6 Screenshot 2/2 einer *JUNG*-Anwendung zur Demonstrationszwecken (in JUNG enthalten). 158

A.7 Fortgeschrittener Prototyp einer View-Hierarchie inklusive eines Satelliten (*JUNG*). Die Darstellung nutzt ein alternatives Layout zu Testzwecken. 159
A.8 Fortgeschrittene Swing-Mockups für das Visualisieren der Datenbankobjekte . 159
A.9 Screenshot einer *prefuse*-Anwendung zur Demonstrationszwecken (Anwendung auf [prefuse 09] verfügbar). 160
A.10 Interner Ablauf des Analyseprozesses im Metadata-Paket 161
A.11 Die JUNG-Architektur im Überblick 162
A.12 Anwendung mit Willkommensbildschirm 163
A.13 Anwendung mit Übersicht der View-Hierarchie 164
A.14 Anwendung mit Anzeige einer negativen View-Abhängigkeit 164
A.15 Anwendung mit Anzeige eines einfachen ERDs 165
A.16 Anwendung mit Anzeige der Verbindungsübersicht 165
A.17 Anwendung mit Anzeige der Verbindungsübersicht – Triggerdefinition 166
A.18 Anwendung mit Anzeige der Triggerabhängigkeiten 166
A.19 SQL Developer – Übersicht 167
A.20 SQL Developer – Verbindungen 167
A.21 SQL Developer – Editor 168
A.22 SQL Developer – Procedure-Viewer 168
A.23 Oracle SQL Developer – Übersicht 169
A.24 Oracle SQL Developer – Verbindung 169
A.25 Oracle SQL Developer – Editor 170
A.26 Oracle SQL Developer – Procedure Editor 170
A.27 Oracle SQL Developer – Neue Prozedur 171
A.28 DbVisualizer – Verbindungsübersicht 171
A.29 DbVisualizer – Neue Verbindung 171
A.30 DbVisualizer – Editor 172
A.31 DbVisualizer – Objektdaten-Anzeige 172
A.32 DbVisualizer – Objekt-Anzeige 172
A.33 MySQL Workbench – Übersicht 173
A.34 MySQL Workbench – Tabellen anlegen 173
A.35 MySQL Workbench – Schemaübersicht 173
A.36 Microsoft Office Access – Übersicht 174
A.37 Microsoft Office Access – Tabellen anlegen 174
A.38 Microsoft Office Access – Tabellenübersicht 174
A.39 Microsoft Office Access – Bericht 175
A.40 Microsoft Office Access – Eingabeformular 175

A.41 TOAD – Startansicht 175
A.42 TOAD – SQL Editor 176
A.43 TOAD – SQL Modeler 176
A.44 TOAD – Schema Browser 176
A.45 Testergebnis – JUnit-Testfälle 177
A.46 Testergebnis – Code-Coverage 177
A.47 Testergebnis – Code-Coverage als HTML-Ausgabe 178
A.48 Testergebnis – PMD Konsolen-Ausgabe 178
A.49 Testergebnis – FingBugs Konsolen-Ausgabe 178

Tabellenverzeichnis

1 Lizenzkosten General SQL Parser 49
2 Aufbau der Struktur des Metadaten-Paketes 90
3 Übersicht der Paketstruktur der Graphenimplementierung – Erstellen . . . 112
4 Übersicht der Paketstruktur der Graphenimplementierung – Darstellen . . 113
5 Datenflussanalyse – Verwendung einer Variablen 127
6 Datenflussanalyse – Anomalien 127

Abkürzungs- und Symbolverzeichnis

API	Application Programming Interface
BLOB	Binary Large Object
CLOB	Character Large Object
CSV	Character Separated Values
DBMS	Datenbankmanagementsystem
DDL	Data Definition Language
DML	Data Manipulation Language
DQL	Data Query Language
EERD	Extended Entity Relationship Diagram
ERD	Entity-Relationship-Diagramm
GPL	General Public License
GUI	Graphical User Interface
HMTL	Hypertext Markup Language
HSQLDB	HyperSQL DataBase
IT	Informationstechnik
JAR	Java Archive
JDBC	Java Database Connectivity
JNI	Java Native Interface
JUNG	Java Universal Network/Graph Framework
JVM	Java Virtual Machine
LGPL	Lesser General Public License
LPG	Lexer Parser Generator
MDI	Multiple Document Interface
MVC	Model-View-Controller
PL/SQL	Procedural Language/SQL
POJO	Plain Old Java Object
SQL	Structured Query Language
UML	Unified Modeling Language
URL	Uniform Resource Locator
XML	Extensible Markup Language
YAML	YAML Ain't Markup Language

1. Danksagung

An dieser Stelle danken wir allerherzlichst Frau Prof. Dr. Heide Faeskorn-Woyke für ihre aufopfernden Bemühungen und stetige Unterstützung vor und während der Diplomarbeit.

Ein besonderer Dank geht an Sebastian Janzen der „IT Jungs“, der uns freundlicherweise mit technischen Ressourcen in Form eines Versionskontrollsystems unterstützt hat.

Last but not least danken wir Sir Timothy John Berners-Lee, dem Hirten Kaldi und der Firma CollabNet für die Erfindungen Internet, Kaffee und Subversion. Was wären wir nur ohne sie.

Danksagung Andre Kasper

Meinem Vater, Dietmar Kasper, möchte ich diese Diplomarbeit widmen. Er verstarb am 19.07.2002. In ewigem Gedenken möchte ich ihm diese Arbeit zuwenden.

In erster Linie möchte ich meiner Mutter, Rosemarie Kasper, und ihrem Lebenspartner, Günter Völkel, danken. Ich konnte mich nicht nur während der Diplomarbeit auf die Beiden verlassen, sondern auch während meines gesamten Studiums. Sie waren immer für mich da und haben mich unterstützt, ob finanziell oder geistig. Ich konnte jederzeit mit meinen Problemen zu ihnen kommen und wusste direkt, dass ich mich auf meine Mutter und Günter verlassen kann. Deshalb gilt mein ganz besonderer Dank diesen beiden Menschen.

Ein weiterer ganz besonderer Dank geht an meine beiden engsten und besten Freunde Philipp Emme und Sascha Frank. Während dieser Arbeit habe ich gemerkt, dass es teilweise doch sehr anstrengend sein kann, wenn etwas nicht so läuft, wie man es gerne hätte. Und ich konnte zu jeder Tages- und Nachtzeit mit einer Krise zu ihnen kommen und sie haben mir Kraft gegeben, um weiterzumachen. Die Hilfe, die mir diese beiden Menschen entgegen bringen, geht weit über das normale Maß hinaus.

Ebenfalls möchte ich mich an dieser Stelle bei einem weiteren sehr guten Freund bedanken, Martin Hesseler. Als diese Arbeit gerade zur Hälfte fertiggestellt wurde, hatte mich meine Kraft so langsam verlassen und ich hatte einen völligen Aussetzer, sobald es an das Schreiben ging. Zu diesem Zeitpunkt hat mich Martin zu einer Pause überredet und es war genau das, was ich gebraucht habe um mit voller Kraft wieder ans Werk zu gehen.

Ein weiterer, ganz besonderer Dank geht an Jan Philipp, mit dem ich diese Diplomarbeit zusammen geschrieben und das Programm entwickelt habe. Jan ist immer ruhig geblieben, auch wenn ich teilweise etwas Panik bekommen habe. Er hatte immer ein offenes Ohr und ist auf mich eingegangen, auch wenn das manchmal nicht gerade leicht war. Ohne ihn wäre diese Arbeit nicht zu dem geworden, was sie jetzt ist.

Weiterhin möchte ich mich hier noch bei einigen Personen bedanken, die mich während dieser Arbeit unterstützt und mir immer die Daumen gedrückt haben. Diese Leute haben mir durch ihren Zuspruch immer wieder Mut gemacht (die Reihenfolge der Namen erfolgt alphabetisch und hat keine besondere Bedeutung): Julian Anders, Christiane Grünloh, Daniela Mann, Alexander Polz, Petra Riemer, Sandra Sartory, Martin Tubandt und Daniel Walker.

Danksagung Jan Philipp

An dieser Stelle möchte ich meinen Eltern danken, die mich im gesamten Studium nicht nur finanziell unterstützt, sondern mir auch bei alltäglichen Dingen zur Seite gestanden haben. Ohne sie hätte ich sicher nicht mein Wunschstudium mit dem Einsatz, Erfolg und einer Diplomarbeit abschließen können.

Ebenfalls danke ich meinen Großeltern für eine ständige Unterstützung, und sei es nur das Ausleihen der Autoschlüssel – das hat vieles sehr viel angenehmer gemacht.

Sehr bedanken möchte ich mich bei meinem Kommilitonen Andre Kasper. Die Anfertigung der Diplomarbeit hat nicht nur Spass gemacht, sondern war auch lehrreich und hat durch gegenseitige Anregungen und Ideen einen qualitativen Mehrwert erhalten.

Ein weiterer Dank geht an meinen zukünftigen Arbeitskollegen Peter Bekiesch, der uns sowohl mit zahlreichen Performance- & Effizienztipps als auch Oracle spezifischen Informationen gut geholfen hat.

Ich möchte mich außerdem bei all denen bedanken, die mir bei Formulierungen, Korrekturen oder sonstigen Kleinigkeiten zur Seite standen: Stanimira Gocheva, Benjamin Philipp und Michael Philipp.

2. Einleitung

In der heutigen Zeit ist Software in allen Lebenssituationen präsent. Mit dem Einzug der digitalen Welt werden auch die Softwaresysteme immer komplexer; sie leisten, kontrollieren und speichern immer mehr. Mit der Funktionsvielfalt und der Komplexität steigen auch die Anforderungen an die Speicherstrukturen. Es müssen nicht nur viele Daten gespeichert werden – die abgelegten Daten müssen auch wiedergefunden, aggregiert, umsortiert, gefiltert oder in Teilen verändert werden. Seit über 30 Jahren werden dafür vorwiegend relationale Datenbanksysteme verwendet[9].

Bereits heute sind viele der Softwarelösungen mit integrierten Datenbanken ausgestattet. Von den offensichtlichen Datenbanken in Webanwendungen oder der Literaturfunktion im Textverarbeitungsprogramm abgesehen, besitzen sogar vereinzelte Anwendungen wie Internet-Browser, E-Mail-Clients oder Hardwaregeräte wie Handys und Smartphones eigene, autonome Datenbanksysteme. Zurzeit sind dies oft rein speicherorientierte Datenbanklösungen auf der Basis von HSQLDB oder SQLite. Mit neuen funktionalen Erweiterungen und fortschreitender Technik werden diese Systeme künftig immer komplexer.

Schon in den 90er Jahren wurden vermehrt die so genannten *Data-Warehouses* entwickelt, welche je nach Unternehmen eine oft unvorstellbare Menge an Daten vorhalten. Da diese Daten per Definition aus unterschiedlichen Quellen stammen, müssen sie bei der späteren Verwendung häufig mehrfach gefiltert, aggregiert, reorganisiert und verändert werden: So müssen etwa bestimmte Daten wechselseitig ergänzt werden; manche Daten dürfen aus rechtlichen, persönlichen oder sonstigen speziellen Interessen nur bestimmten Abteilungen eines Unternehmens zugänglich gemacht werden; wieder andere Daten hängen unter genau spezifizierten Bedingungen voneinander ab. Die Datenbanksysteme lösen diese Probleme, indem sie virtuelle Sichten (sprich: Views) auf die Datenbestände (sprich: Tabellen) bereitstellen. Diese werden durch Datenbankanwender angelegt und je nach Einsatzziel mit entsprechenden Benutzerrechten verknüpft. Bei hoch komplexen Strukturen werden diese Sichten wiederum miteinander verknüpft und verschachtelt. Das Endresultat ist eine komplexe Vermischung von Verknüpfungen, Vermengung, Sortierung und Filterung der Daten. Dies sind die Grundfunktionalitäten der Structured Query Language (SQL) und damit auch die Basis der Views.

Zur Optimierung der Abfrageperformanz und des Datenaufkommens können zwischengespeicherte Sichten auch für oft angeforderte Anfragen dienen. Je nach Komplexität der unterliegenden Daten kann eine geeignete Projektion und Selektion die Geschwindigkeit

[9] Nur relationale Datenbanksysteme sind im Rahmen dieser Diplomarbeit von Bedeutung.

zulasten der redundanten Speicherung optimieren[10].

Welche konkrete Relevanz hat nun aber das Hinzufügen, Löschen oder Ändern eines Datums in einer Tabelle? Wann und wo treten Veränderungen welcher Art auf?

Um der steigenden Komplexität der Datenbanken gerecht zu werden, helfen oft automatische, interne Aufträge (Trigger) im Datenbanksystem. Die Trigger definieren ein Stück Programmcode und führen ihn zu einem bestimmten Zeitpunkt und unter einer bestimmten Bedingung aus. Der Zweck eines Triggers ist es, die Datenintegrität aufrecht zu erhalten oder in Verknüpfung mit materialisierten Views für konsistente Daten zu sorgen. Allerdings ist es nicht so ohne Weiteres ersichtlich, ob die Ausführung eines Triggers erfolgreich sein kann, wenn dadurch weitere Trigger ebenfalls aktiv werden. Erst zur Laufzeit, also nach dem statischen Analysieren und gegebenenfalls auch Parsen des Triggers, kann ein Fehler auftreten. Damit stößt man jedoch schnell an die Grenzen des Machbaren. Dies ist auch ein Grund, warum beispielsweise Oracle trotz der beschriebenen Problematik anstandslos das Anlegen von Triggern akzeptiert, welche aber zu rekursiven Abläufen führen oder das *Mutating Table*-Problem auslösen könnten.

In der Regel behilft man sich mit einer Umgehung der Problematik. Beispielsweise kann man das *Mutating-Table*-Problem durch das Aufteilen in zwei Triggereinheiten mit einer temporären Tabelle lösen. Allerdings wird dadurch die Komplexität der Abhängigkeitsstruktur nochmals verstärkt: Was passiert konkret, wenn man ein Datum einer Tabelle oder View ändert? Kann man das Auftreten des Fehlers vorher schon erkennen? Lassen sich potenzielle Zyklen und Rekursionen im Ablauf im Vorfeld erkennen?

Es gibt eine Reihe von visuellen Werkzeugen für Datenbanksysteme oder Anwendungen, die einige grafische Zusatzfunktionen anbieten. Nahezu alle beschränken sich dabei auf die Darstellung der Fremdschlüsselbeziehungen und zielen daher auf die referentielle Integrität. Für die Entwicklung eines Datenbankmodells und auch für das Verständnis des Modells ist eine solche Darstellung oft hilfreich und deshalb auch sehr sinnvoll. Hingegen verschafft es dem Datenbankadministrator oder einem interessierten Anwender nicht die gewünschte Transparenz, um die komplexen, wechselseitigen Abhängigkeiten der Views und Tabellen nachvollziehen zu können.

Im Rahmen dieser Diplomarbeit wird eine Software entwickelt, welche die Abhängigkeiten[11] von Objekten einer Datenbank transparent für den Anwender darstellt. Die Tabellen, Trigger und Views werden dazu zunächst analysiert und anschließend mittels Graphen visualisiert. Eine Software, welche hilfreich bei der Erkennung komplexer Viewabhängig-

[10] In der Informatik wird ein derartiges Speicherverfahren für temporäre Daten „Caching“ genannt. Der Umfang eines Caches ist abhängig vom Datenbanksystem, Hardware und speziellen Anforderungen.

[11] Im Folgenden verwenden wir „Abhängigkeit“ als Synonym von „Assoziation“.

keiten und rekursiver Triggeraktivitäten ist, gibt es weder mit der analytischen noch der grafischen Komponente.

Ein Teil der Arbeit beschäftigt sich zusätzlich mit einer Evaluation und Bewertung verschiedener vorhandener Frameworks und Werkzeuge, die für die Entstehung der Software von Bedeutung sind. Außerdem wird bewertet, inwieweit die Software erweiterungsfähig ist.

Die Diplomarbeit entstand aus einer Idee von Dr. Andreas Behrend und wurde in Zusammenarbeit mit der Universität Bonn erstellt. Die Autoren Andre Kasper und Jan Philipp haben diese Arbeit zusammen an der Fachhochschule Köln, Campus Gummersbach, erstellt.

3. Aufgabenbeschreibung

Es wird eine Anwendung entwickelt, die die komplexen Abhängigkeiten zwischen einzelnen Objekten in einer Datenbank visuell darstellt.

Die Anwendung stellt dabei im Wesentlichen vier Funktionalitäten zur Verfügung. Davon werden zwei als Grundfunktionalitäten bezeichnet. Insbesondere geht es dabei einerseits um Views, die jeweils Abhängigkeiten zu anderen Tabellen oder Views in der Datenbank besitzen. Dabei werden diese Abhängigkeiten weiter differenziert untersucht. Außerdem werden die vorhandenen Trigger der Datenbank analysiert, die auf den Tabellen liegen. Mit letzterer Funktionalität wird sowohl eine Übersicht der zu einer Tabelle verfügbaren Trigger erzeugt als auch die Beziehungen der Triggern untereinander visualisiert. Im letzten Schritt werden in Triggern die impliziten Aufrufe weiterer Trigger erkannt. Die beiden anderen Funktionalitäten ergeben sich aus den bereits gesammelten Daten. So sollen die Objekte einer Datenbankverbindung in einer Baum- oder Listendarstellung angezeigt werden[12]. Die letzte Funktionalität ist eine vereinfachte Darstellung der Fremdschlüsselbeziehungen in Form eines *Entity-Relationshipship-Diagramms*.

Mit diesen Funktionalitäten soll ein hilfreiches Werkzeug bei der visuellen Betrachtung und Untersuchung von Datenbankschemata erstellt werden.

Zusätzlich wird evaluiert, welche (weiteren) Funktionalitäten hierbei technisch möglich sind. Dabei kommen für die Graphen relevante Funktionsabgrenzung und das vorab durchgeführte, grafische Prototyping[13] so genannte Mockups[14] zum Einsatz.

Im Kontext einer grafischen Anwendung sind Mockups frühe Konzeptzeichnungen und -modelle, die in abstrakter und visueller Form die Funktionalitäten des Produktes beschreiben. Auf Details, wie das korrekte Layout, kann – muss aber nicht – in einem Mockup Rücksicht genommen werden. Sowohl das Design als auch die korrekte Beschriftung oder Platzierung der einzelnen Objekte sind in der Regel von nachrangiger Bedeutung.

3.1. Relevante Objekte in der Datenbank

Ein Schema eines Datenbankmanagementsystems beinhaltet eine Fülle von Informationen, die man als *Objekte* in der Datenbank ansehen kann.

[12] vgl. ähnliche hierarchischen Darstellungen in Tools wie SQL Developer (s. Seite 35, Abschnitt 4.4.5 4.4.5)

[13] Bezeichnet die Herstellung und/oder Verwendung so genannter Prototypen. Ein Prototyp ist eine frühe, unfertige Konzeptversion eines Produkts. Neben Testszenarien werden Prototypen auch in einer frühen Evaluation verwendet.

[14] Die Mockups wurden mit dem Werkzeug Balsamiq Mockups (`www.balsamiq.com`) erstellt.

Im Kontext des Themas „Visualisierung der Abhängigkeiten“ sind folgende Objekte von Bedeutung[15]:

Tabellen sind elementar notwendig, da sowohl Views als auch Trigger sich immer auf konkrete Tabellen beziehen. Jedes Tabellenobjekt wird durch Name und Schema definiert.

Views gehören zur ersten Grundfunktionalität. Ihre Existenz ist die Grundlage für das Vorhandensein der verschiedenen Abhängigkeiten. Entsprechend ihrer äußerlichen Ähnlichkeit zu Tabellen werden auch Viewobjekte durch einen Namen und ein Schema definiert. Zusätzlich spezifiziert eine Definition, eine Structured Query Language (SQL)-Anfrage, den Umfang der abgebildeten Daten.

Trigger sind die zweite Grundfunktionalität der Anwendung. Dabei zeigen potenzielle Trigger auf Tabellen oder Views. Jedes Triggerobjekt besteht neben einem Namen und dem eigentlichen Ausführungscode aus diversen Daten, etwa Ausführungszeitpunkt oder -bedingung. Jeder Trigger beinhaltet einen Procedural Language/SQL (PL/SQL)-Codeblock.

Funktionen und Prozeduren sind benannte und abgelegte PL/SQL-Codeblöcke und können direkt über andere Codeblöcke oder SQL-Anweisungen ausgeführt werden. Wie jeder andere PL/SQL-Codeblock können auch Funktionen und Prozeduren weitere SQL-Queries ausführen.

Integritätsbedingungen und Assertions sind abgelegte Regeln und Integritätsüberwachungen, die bei Änderung der Daten deren Integrität sicherstellen. Eine Integritätsbedingung (Constraints) kann beispielsweise durch die Definition von *Cascade*[16] weitere, indirekte Anweisungen ausführen.

Diese Arbeit wird zunächst nur die Tabellen, Views und Trigger in einem Datenbankschema berücksichtigen. Die Hinzunahme weiterer Objekte wie Funktionen oder Prozeduren ist zu diesem Zeitpunkt nicht von weiterer Relevanz[17].

[15] In der Auflistung werden die Objekte aus Sicht des Datenbanksystems Oracle beschrieben. Etwaige Unterschiede, beispielweise im verwendeten SQL-Sytax, können je nach Datenbankhersteller variieren.

[16] Der Begriff Kaskadieren bedeutet hier, dass das Löschen eines Datensatzes auch das Löschen anderer Datensätze zur Folge hat. Im Sinne der Datenintegrität sind dies zum Beispiel Daten, die mit dem ursprünglich gelöschten Datensatz unmittelbar verknüpft sind (auch Master- und Detaildatensatz genannt).

[17] s. Seite 143, Abschnitt 9.2 (9.2)

3.2. Ansichten

Die Anwendung stellt neben der Ansicht technischer Elemente drei zentrale Ansichten zur Verfügung: die Viewabhängigkeiten, die Triggerabhängigkeiten und das vereinfachte Entity-Relationship-Diagramm.

Die Übersicht der Viewabhängigkeiten wird in einem Graph als Baum oder Netzwerk dargestellt. Jeder Knoten ist dabei eine Tabelle oder eine View und besitzt ein Attribut Level, welches der Höhe von unten nach oben entspricht[18]. Algorithmisch gesehen ist das der längste Pfad der Kinder des Knotens plus 1.

Die erkannten Trigger der Datenbank werden in der zweiten Ansicht dargestellt. Dabei werden durch die Abhängigkeiten die impliziten, potenziellen Aufrufe anderer Trigger sichtbar gemacht. Soweit wie möglich werden Besonderheiten der Trigger aufgezeigt. Da die Informationen über zyklische Abfolgen oder eines möglichen Triggers mit *Mutating-Table*-Problem verfügbar sind, könnten diese im Graphen dargestellt werden.

In der letzten Ansicht wird ein Entity-Relationship-Modell der erkannten Tabellen angezeigt. Die Abhängigkeiten bestehen dabei aus den Fremdschlüsselbeziehungen der Tabellen untereinander.

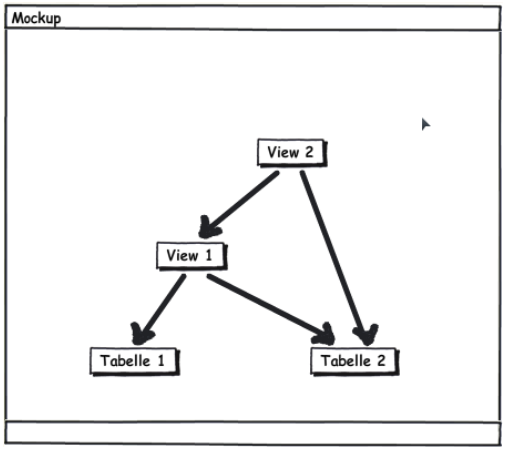


Abbildung 3.1: Beispiel einer View-Hierarchie mit drei Ebenen

3.3. Details zu einer Abhängigkeit

Die Abhängigkeit zwischen einer View und einer ihr untergeordneten Tabelle oder View muss feiner untersucht und analysiert werden: Unter anderem ist zu entscheiden, ob es sich um eine positive oder eine negative Abhängigkeit handelt.

[18] s. Seite 108, Abschnitt 6.4.6.5 (6.4.6.5)

Positive Abhängigkeiten sind solche, in welchen der *WHERE*-Bedingungsteil der *SELECT*-Anfrage eine Enthaltungsmenge durch Operatoren wie = oder `LIKE` spezifiziert. Neutrale *SELECT*-Anweisungen sind immer positiv.

Negative Abhängigkeiten sind solche, in welchen der *WHERE*-Bedingungsteil explizit Daten ausschließt – beispielsweise mittels einem `NOT EXISTS` verbunden mit einer *Unterabfrage*.

Ein üblicher Verwendungszweck für negative Abhängigkeiten findet sich in praxisnahen Situationen wieder, in welchen bestimmte Daten nur gefiltert weitergegeben werden. Beispielsweise kann es wichtig sein, aus Gründen des Datenschutzes oder der Übersichtlichkeit wegen, bestimmte Datensätze aus einer Sicht auszublenden. Vor allem große und komplexe Datenbanksysteme enthalten oft einzelne Tabellen mit unterschiedlich sensiblen Daten. Dabei kann jedes Datum eine unterschiedliche Relevanz und Sicherheitsbestimmung je Benutzer besitzen. Das Verknüpfen von Daten ist eine positive Abhängigkeit, das Ausschließen bestimmter Daten ist dagegen eine negative Abhängigkeit.

Es sollen zunächst keine multiplen Kanten[19] im Graphen erstellt und angezeigt werden. Das bedeutet, dass mehrere gleichartige (positive oder negative) Abhängigkeiten zu einer zusammengefasst werden. Falls es sowohl eine positive als auch eine negative Abhängigkeit gibt, dann werden beide als eine negative Abhängigkeit interpretiert. Damit wird ein Darstellungsproblem des Graphens gelöst, weil selten mehrere Kanten pro Knotenpaar übersichtlich dargestellt werden können.

3.4. Datenbanksystem

Primär wird die Anwendung für das Datenbanksystem *Oracle 11g* entwickelt. Dabei muss darauf geachtet werden, dass die Architektur sinnvoll gewählt wird, damit zu einem späteren Zeitpunkt andere Datenbanksysteme oder Features ergänzt werden können. Dazu zählt ebenfalls eine Einschätzung, welchen Aufwand etwa eine Integration von MySQL hat[20].

3.5. Vorhandene funktionale Anforderungen

Die typischen Entwicklungsschritte bei der Erstellung einer Software beinhalten am Anfang, dass die funktionalen Anforderungen gemäß der Aufgabenbeschreibung zu ermit-

[19] Mit multiplen Kanten sind jene Kanten gemeint, die den gleichen Start- und Zielknoten im Graphen haben.
[20] s. Seite 90, Abschnitt 6.3.1 (6.3.1)

teln sind. Dies ist wichtig, damit von vorne herein klar ist, was die Software leisten soll – aber auch, was sie nicht leisten muss.

Dieser übliche Weg, aus der Aufgabenbeschreibung die Anforderungen herzuleiten und daraus die Software und die Testfälle abzuleiten, stellt sich in diesem Falle aber als eine besondere Herausforderung dar.

Zwar sind die funktionalen Anforderungen gleichzeitig auch Verbindlichkeiten für die eigentliche Entwicklung, aber sie sind gleichzeitig auch Bestandteil dieser Arbeit und müssen somit selbst erst ermittelt werden.

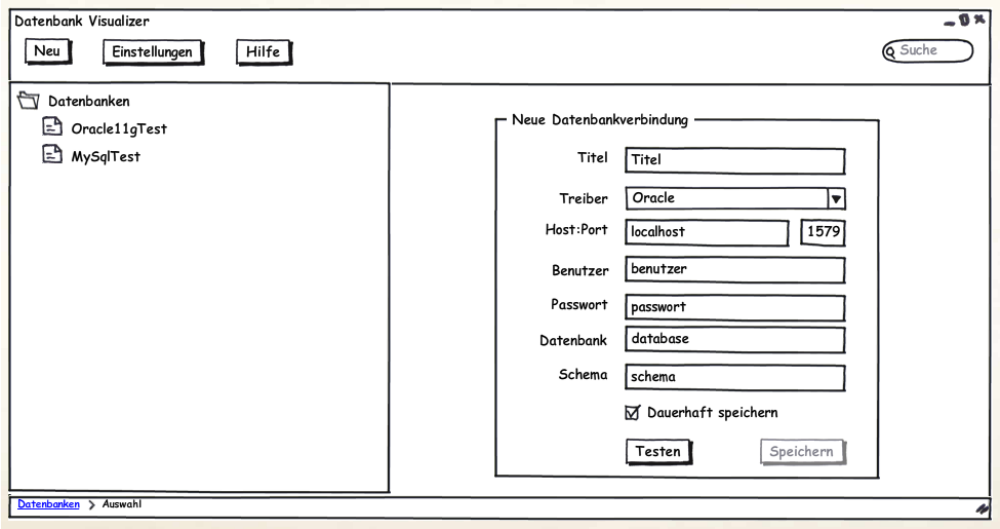


Abbildung 3.2: Konzeptzeichnung für den Startbildschirm, angelehnt an Programmen wie Oracle SQL Developer

Die weiter oben angeführte Aufgabenbeschreibung[21] deckt nicht alle technischen Details vollständig ab. Daher werden die darüber hinaus zu definierenden Anforderungen an die Anwendung bei Bedarf erläutert[22].

Zu Beginn dieser Diplomarbeit wurde die grobe Richtung der Diplomarbeit festgelegt. Die weiteren Details über die Ermittlung und Darstellung von Abhängigkeiten sind jedoch ebenfalls ein Teil dieser Arbeit. Die Evaluation und die Machbarkeitsstudien von möglichen Funktionalitäten und Ideen, die im Vorfeld und während der Durchführung entstanden, finden sich daher zusammengefasst im Kapitel Graphen[23]. Dies betrifft die Themenschwerpunkte Parser und Graphenvisualisierung. Durch Gespräche und Austausch von Konzeptvorschlägen, Mockups, Ideen und Anregungen entstanden somit nach und nach die hinreichend spezifizierten Anforderungen.

[21] s. Seite 18, Abschnitt 3 (3)
[22] s. Seite 69, Abschnitt 6.1 (6.1)
[23] s. Seite 55, Abschnitt 5.2.2 (5.2.2)

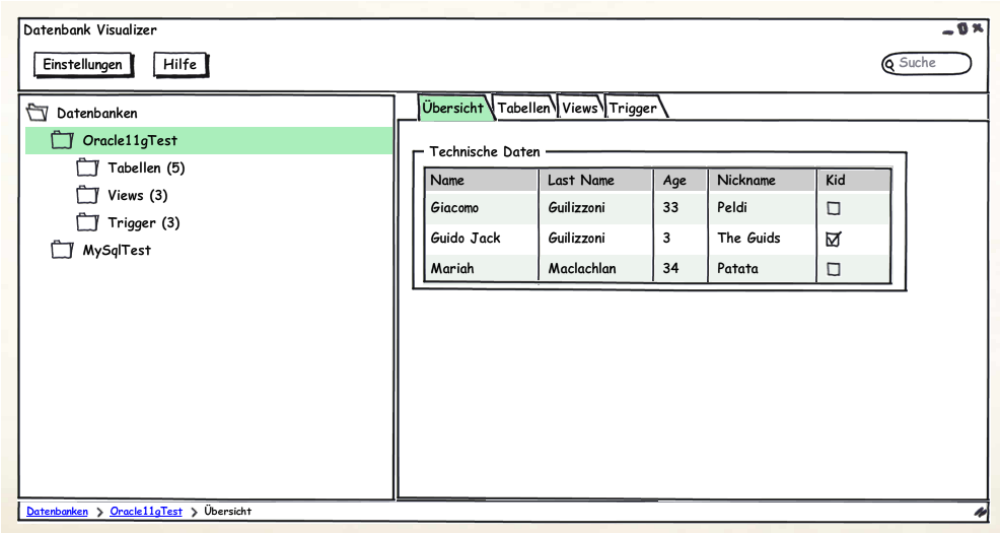


Abbildung 3.3: Konzeptzeichnung für den Inhalt einer Datenbankverbindung

Die eigentliche, konkrete Darstellung der Graphen muss erst über konkretisierte Anforderungen definiert werden. Konzeptionell wird eine Art Visualisierung mit Hilfe eines Baumes oder Netzwerkes geschaffen. In welcher Art und Weise dies aber konkret realisiert werden kann, muss zunächst in einem weiteren Schritt[24] analysiert werden.

Deswegen wird zunächst in einem ersten Schritt der aktuelle Markt untersucht. Die Vor- und Nachteile der einzelnen Programme werden bewertet und dienen unter Umständen als Inspiration für die eigentlichen Funktionalitäten. Eine weiterführende Vorstellungen der Funktionen findet sich im nächsten Kapitel[25].

Aus den vorgenannten Gründen ist eine testgetriebene Entwicklung ungeeignet. Für eine solche Entwicklungsmethode sollten die genauen Anforderungen bereits vor der eigentlichen Entwicklung bekannt sein, da diese die Basis für die Funktionstest sind. Die Entscheidung fällt daher gegen eine testgetriebene Entwicklung, um ein aufwändiges Konfigurations- und Änderungsmanagement zu sparen. Folglich bietet sich ein Testen nach dem V-Modell für dieses Projekt an[26].

3.6. Vorhandene nicht-funktionale Anforderungen

Bereits im ersten Gespräch wurde einstimmig beschlossen, dass die Entwicklung in Java stattfinden soll. Außerdem entschieden wir uns, als Untersuchungsobjekt eine vorhandene *Oracle*-Datenbank (aktuelle Version 11g) zu nutzen.

[24] s. Seite 106, Abschnitt 6.4.6 (6.4.6)
[25] s. Seite 69, Abschnitt 6.1 (6.1)
[26] s. Seite 119, Abschnitt 8 (8)

3.7. Beispiel

Als Motivation liegt dieser Diplomarbeit ein selbst entwickeltes Datenbankschema bei. Das Beispielschema „Flughafen und Flugzeuge“ besteht aus acht Tabellen, zehn Views und sieben Triggern und ist im Anhang als SQL für eine Oracle-Datenbank verfügbar[27]. Es beinhaltet diverse Fremdschlüsselbeziehungen[28] und verschiedene Ansichten auf bestimmte Problemsituationen oder Informationen[29].

Jeder *Airport* kann mehrere *Airplanes* und *Stations* (Landebahnen/Terminals) besitzen. Ein Flugzeug steht genau mit einer *Route* in Beziehung. Zusammen mit der *Many-to-Many*-Assoziation *Route_Station* bilden *Route* und *Airplane* die einzelnen Flugpunkte einer Flugroute. Jedes *Airplane* kann einen oder mehrere *Pilots* besitzen, welche wiederum in *Crews* zusammengeschlossen sind. Die Entität *Log_Results* ist eine reine technische Protokolleinheit und beinhaltet keine referentielle Integrität.

Im Zusammenhang der Darstellung der View-Hierarchien bestehen teilweise komplexen Definitionen und Abhängigkeiten zwischen einzelnen Views. Dies ist im Kontext der beispielhaften Präsentation zu berücksichtigen.

Der Trigger *TRG_Erstelltes_Flugzeug* wird im SQL-Script erst nach dem Einfügen der Beispieldatensätze erstellt, weil mit Hilfe dieses Triggers eine Rekursion von Triggern provoziert werden kann.

Einige Darstellungen des Schemas mit der entwickelten Software befinden sich im Anhang[30].

[27] s. Anhang B, S. 182, Listing 26 (26)
[28] s. Anhang A, S. 154, Abb. A.1 (A.1)
[29] s. Anhang A, S. 155, Abb. A.2 (A.2)
[30] s. Seite 114, Abschnitt 7 (7)

4. Diskussion und Vergleich vorhandener Softwareprodukte

Ein Teil dieser Arbeit ist es, die bereits vorhandenen Programme zu untersuchen und zu bewerten[31]. Die Evaluierung der vorhandenen Softwareanwendungen dient als Grundlage, um entscheiden zu können, ob eine derartige Software bereits vorhanden ist. Weiterhin dienen die einzelnen Anwendungen als Inspiration für die Eigenentwicklung.

Aus den Anforderungen an die Softwareanwendung ergeben sich vier Grundfunktionen[32]. Die einzelnen Datenbankobjekte eines Schemas werden dabei zunächst in der Hauptansicht in tabellarischer Form dargestellt. Diese Ansicht dient zur Übersicht, um einen ersten Eindruck über den Umfang der Datenbankobjekte zu erhalten. Die drei Arten von Visualisierungen beschäftigen sich mit den Abhängigkeiten der einzelnen Objekte untereinander. Dabei hat jede Ansicht ihren eigenen Schwerpunkt und stellt verschiedene Abhängigkeiten einzelner Datenbankobjekte unterschiedlich dar[33].

Abhängigkeiten der Views Die einzelnen Views besitzen Abhängigkeiten zu Tabellen oder anderen Views. Diese Abhängigkeiten beruhen auf der *SELECT*-Anweisung einer View, wobei eine Abhängigkeit positiv oder negativ sein kann[34]. Positive und negative Abhängigkeiten werden unterschiedlich dargestellt.

Abhängigkeiten der Trigger Die Trigger werden, anders als Funktionen oder Prozeduren, durch einen Data Manipulation Language (DML)-Befehl automatisch angestoßen und benötigen keinen expliziten Aufruf. In einem Datenbanksystem können die Trigger durch weitere DML-Befehle im *BODY (Rumpf)* auf andere Tabellen wiederum weitere Trigger anstoßen. Dadurch entstehen Abhängigkeiten der Trigger untereinander[35].

Abhängigkeiten der Tabellen Die Tabellen können in einem Datenbankschema auch Abhängigkeiten untereinander in Form von Fremdschlüsselbeziehungen besitzen. Diese Form der Abhängigkeit dient in erster Instanz als Hilfe bei einem Entwurf eines Datenbankschemas. Sie kann aber auch später zum Einsatz kommen, um ungewünschte oder fehlende Abhängigkeiten zu korrigieren.

Im Wesentlichen wird eine Visualisierung in Form eines ER-Diagramms angeboten, um die Abhängigkeiten der Tabellen untereinander darzustellen. Diese Art der grafischen An-

[31] s. Seite 23, Abschnitt 3.5 (3.5)
[32] s. Seite 18, Abschnitt 3 (3)
[33] s. Seite 20, Abschnitt 3.2 (3.2)
[34] s. Seite 20, Abschnitt 3.3 (3.3)
[35] s. Seite 26, Abschnitt 4.2 (4.2)

zeige ist Bestandteil von vielen Softwareanwendungen und häufig die einzig verfügbare Art der visuellen Darstellung.

Bei der Suche nach einer Softwareanwendung, die Visualisierungen neben der bereits erwähnten Abhängigkeit der Tabellen anbietet, wird man nur sehr schwer fündig. Es gibt eine Vielzahl von Anwendungsprogrammen für Datenbanken, die zum Großteil auf diese besondere Anforderung verzichten. Sie dienen dabei eher zur Unterstützung administrativer Aufgaben und Datenbankabfragen, wobei die Ergebnisse in tabellarischer Form ausgegeben werden.

Diese Produkte sind zum größten Teil kostenlos und frei verfügbar, andere erfordern eine Registrierung beim Hersteller. Unter anderem gibt es aber auch kostenpflichtige Programme, wie zum Beispiel Toad[36]. Die verschiedenen Programme werden hier kurz vorgestellt und anhand der Anforderung[37] evaluiert und bewertet. Sie dienen im späteren Verlauf dieser Arbeit als Inspiration.

4.1. Nicht-funktionale Anforderungen

Das Programm, dass in dieser Diplomarbeit beschrieben wird, wird für eine Oracle-Datenbank[38] entwickelt. Dabei werden die Versionen 9i, 10g und 11g genauer untersucht. Außerdem sollte das Programm prinzipiell auch Datenbanksysteme anderer Hersteller unterstützen bzw. eine Unterstützung anbieten. Eine Plattformformunabhängigkeit oder für die jeweiligen Betriebssysteme angepasste Version soll verfügbar sein.

4.2. Funktionale Anforderungen

Das Programm soll eine Fähigkeit zur Visualisierung besitzen. In diesem Zusammenhang sollen drei Ansichten realisiert werden, die die teilweise komplexen Abhängigkeiten der Datenbankobjekte als Graph darstellen.

Die erste zu erfüllende Funktionalität ist die Anzeige der Abhängigkeiten von Views zu den Tabellen oder zu anderen Views. Es soll außerdem zwischen negativen und positiven Abhängigkeiten differenziert werden[39].

Die zweite Funktionalität ist die Visualisierung von Triggern und den Abhängigkeiten der Trigger untereinander. In einem Datenbankschema besteht die Möglichkeit, dass ein

[36] s. Seite 38, Abschnitt 4.4.6 (4.4.6)
[37] vgl. Seite 18, Abschnitt 3 (3)
[38] vgl. Seite 23, Abschnitt 3.6 (3.6)
[39] s. Seite 20, Abschnitt 3.3 (3.3)

Trigger einen anderen Trigger anstößt und einen Kreislauf bildet. Eine solche Triggerrekursivität würde die Datenbank verlangsamen oder eventuell sogar zum Absturz bringen. Um dies zu verhindern, sollen auch genau diese Kreisläufe grafisch aufgezeigt werden, um dem Benutzer eine solche Schwachstelle in der Datenbank aufzuzeigen.

Ebenfalls werden die Tabellenabhängigkeiten durch Fremdschlüsselbeziehungen in der Form eines vereinfachten Entity-Relationship-Diagramms (ERD) dargestellt. Diese Ansicht stellt ausschließlich die Abhängigkeiten der Tabellen untereinander dar ohne eine Anzeige der Views. So können die Abhängigkeiten der Tabellen übersichtlicher dargestellt werden.

Durch diese drei Sichten von Abhängigkeiten erhält der Anwender eine Übersicht über die angelegten und verwendeten Objekte eines Schemas. Weitere Objekte wie Funktionen, Prozeduren und ähnliche werden zunächst nicht angezeigt. Allerdings kann die Anwendung um diese Datenbankobjekte erweitert werden[40], da die Architektur dafür ausgelegt wird.

4.3. Übersicht

Die verschiedenen Softwareprogramme werden anhand der geforderten Funktionalitäten untersucht und evaluiert. Auch Programme, die kostenlos zur freien Verfügung stehen, weisen eine gute Dokumentation in Form von Tutorials[41] und Bedienungsanleitungen auf. Auch anhand dieser guten Dokumentation lässt sich schnell entscheiden, ob eine Softwareanwendung den Anforderungen entspricht.

Die Evaluierung zeigt auf, dass die meisten Softwareprodukte auf Visualisierungen komplett verzichten oder die Abhängigkeiten beispielsweise in tabellarischer Form darstellen. Sie dienen eher als Werkzeug in Form eines Editors, um die Datenbankobjekte ansprechen und verändern zu können.

4.4. Vergleich

Bei dem Vergleich werden die verschiedenen Anwendungen gegen die Anforderung evaluiert. Ein Bestandteil der Anforderung ist, dass die Anwendung eine Fähigkeit zur visuellen Darstellung besitzen soll. Bei der Visualisierung der Abhängigkeiten soll zwischen

[40] s. Seite 143, Abschnitt 9.2 (9.2)

[41] Eine schriftliche Gebrauchsanleitung, die mit Hilfe von Beispielen Schritt für Schritt erklärt, wie man mit einem Computerprogramm bestimmte Ergebnisse erzielt.

positiven und negativen Abhängigkeiten unterschieden werden[42]. Die vorgestellten Softwareprogramme werden ebenfalls auf diese Funktionalität untersucht, vorausgesetzt, eine grafische Darstellung ist vorhanden.

Einige hier vorgestellte Programme besitzen keine Funktion zur Visualisierung. Allerdings bieten sie Darstellungen über die vorhandenen Datenbankobjekte und teilweise auch eine Anzeige über deren Abhängigkeit. Daher werden auch diese Programme hier erwähnt und vorgestellt.

4.4.1. DbVisualizer

Der DbVisualizer[43] ist eine Softwareanwendung zur Unterstützung administrativer Aufgaben. Außerdem dient der DbVisualizer zur Abfrage und Visualisierung der Objekte eines Schemas in einer Datenbank. Er kann Verbindungen zu vielen verschiedenen Datenbanken herstellen (Oracle, Sybase, SQL Server, PostgreSQL, DB2, Mimer, Neoview, MySQL, Informix, JavaDB/Derby)[44]. Nicht mitgelieferte Datenbanktreiber können über ein Menü geladen werden. Dadurch bietet der DbVisualizer Unterstützung für alle Hersteller, die über einen Java Database Connectivity (JDBC)-Treiber verfügen. Es sind Versionen für die Betriebssysteme Windows, Linux und Mac OS X vorhanden.

Um eine Verbindung zu einer Datenbank aufzubauen, verfügt der DbVisualizer über ein modales Dialogfenster[45]. Um eine Verbindung anzulegen, können dort alle benötigten Einstellungen vorgenommen werden. Es besteht die Möglichkeit, die Verbindungseinstellungen im Vorfeld zu testen. Nach dem erfolgreichen Speichern einer Verbindung werden die Tabellen, Views, Prozeduren und Funktionen ausgelesen und in einem Übersichtsfenster[46] angezeigt. Dabei fällt auf, dass der DbVisualizer komplett auf die Anzeige anderer Datenbankobjekte wie Trigger verzichtet.

Der DbVisualizer verfügt über einen Editor[47], mit dem einzelne Structured Query Language (SQL)-Statements oder aber auch ganze SQL-Skripte gegen die Datenbank abgesetzt werden können. Bei einer Abfrage werden die entsprechenden Ergebnisse in tabellarischer Form im unteren Bereich des Editors ausgegeben. Eine mögliche Fehlermeldung, wie zum Beispiel eine falsche Syntax in einen Statement, wird ebenfalls angezeigt.

[42] s. Seite 20, Abschnitt 3.3 (3.3)
[43] s. [Minq Software AB 09b]
[44] vgl. [Minq Software AB 09a]
[45] s. Anhang A, S. 171, Abb. A.29 (A.29)
[46] s. Anhang A, S. 171, Abb. A.28 (A.28)
[47] s. Anhang A, S. 172, Abb. A.30 (A.30)

Die einzelnen Datenbankobjekte in der Verbindungsübersicht lassen sich anklicken, wodurch sich ein Fenster öffnet, in dem Informationen zu dem ausgewählten Objekt angezeigt werden[48]. So werden die vorhandenen Spalten, die eingefügten Daten, Integritätsbedingungen (zum Beispiel Primär- oder Fremdschlüssel) und weitere Informationen dargestellt.

Sobald eine Verbindung ausgewählt wurde, wird eine Übersicht der ausgelesenen Datenbankobjekte in der sogenannten „Object View“[49] angezeigt. Somit erhält der Benutzer eine erste Übersicht über die vorhandenen Objekte, die äquivalent zu der Übersicht bei den Verbindungen ist. Neben dem standardmäßig ausgewählten Reiter „Tables“ besteht die Auswahlmöglichkeit „References“, die nur als Option angezeigt wird, wenn ein komplettes Schema und nicht eine Tabelle oder View in der Verbindungsübersicht ausgewählt wurde.

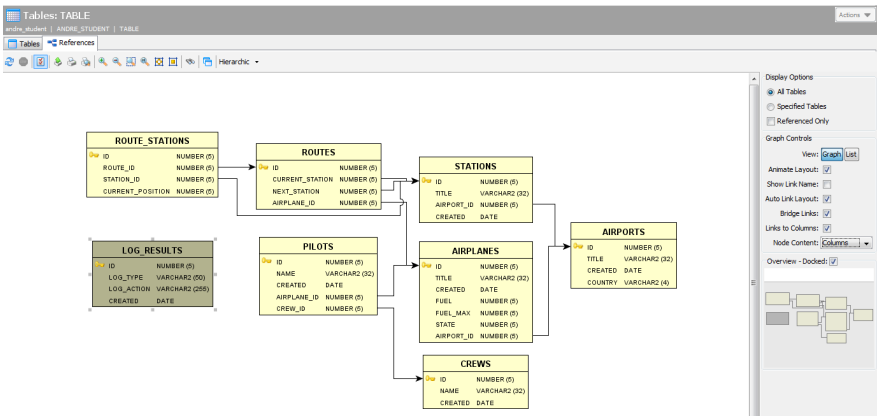
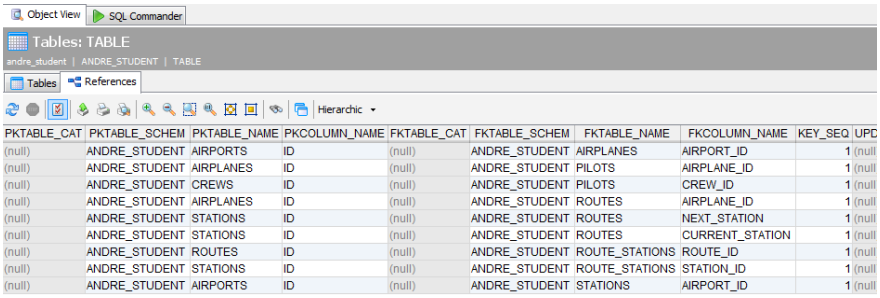


Abbildung 4.1: DbVisualizer – grafische Anzeige der Abhängigkeite

In der zweiten Ansicht („*References*“) werden die Abhängigkeiten der Schema-Tabellen mit Fremdschlüsselbeziehungen dargestellt[50]. Dabei werden nur die Tabellen in Form eines Entity-Relationship-Diagramms (ERD) ohne Views angezeigt. Die Fremdschlüsselbeziehungen werden dabei grafisch gut dargestellt. Es besteht die Option, sich die Fremdschlüsselbeziehungen in tabellarischer Form anzeigen zu lassen[51]. Diese beiden Sichten sind die einzigen, über die der DbVisualizer verfügt. Ohne die Anzeige der Views lassen sich allerdings keine Abhängigkeiten der Views untereinander oder zu den entsprechenden Tabellen erkennen. Eine Visualisierung von Triggern besteht nicht, so dass auch hier keine Abhängigkeiten oder Rekursionen erkennbar dargestellt werden.

[48] s. Anhang A, S. 172, Abb. A.31 (A.31)
[49] s. Anhang A, S. 172, Abb. A.32 (A.32)
[50] s. S. 29 Abb. 4.1 (4.1)
[51] s. S. 30 Abb. 4.2 (4.2)



PKTABLE_CAT	PKTABLE_SCHEM	PKTABLE_NAME	PKCOLUMN_NAME	FKTABLE_CAT	FKTABLE_SCHEM	FKTABLE_NAME	FKCOLUMN_NAME	KEY_SEQ	UPD
(null)	ANDRE_STUDENT	AIRPORTS	ID	(null)	ANDRE_STUDENT	AIRPLANES	AIRPORT_ID	1	(null
(null)	ANDRE_STUDENT	AIRPLANES	ID	(null)	ANDRE_STUDENT	PILOTS	AIRPLANE_ID	1	(null
(null)	ANDRE_STUDENT	CREWS	ID	(null)	ANDRE_STUDENT	PILOTS	CREW_ID	1	(null
(null)	ANDRE_STUDENT	AIRPLANES	ID	(null)	ANDRE_STUDENT	ROUTES	AIRPLANE_ID	1	(null
(null)	ANDRE_STUDENT	STATIONS	ID	(null)	ANDRE_STUDENT	ROUTES	NEXT_STATION	1	(null
(null)	ANDRE_STUDENT	STATIONS	ID	(null)	ANDRE_STUDENT	ROUTES	CURRENT_STATION	1	(null
(null)	ANDRE_STUDENT	ROUTES	ID	(null)	ANDRE_STUDENT	ROUTE_STATIONS	ROUTE_ID	1	(null
(null)	ANDRE_STUDENT	STATIONS	ID	(null)	ANDRE_STUDENT	ROUTE_STATIONS	STATION_ID	1	(null
(null)	ANDRE_STUDENT	AIRPORTS	ID	(null)	ANDRE_STUDENT	STATIONS	AIRPORT_ID	1	(null

Abbildung 4.2: DbVisualizer – tabellarische Anzeige der Abhängigkeiten

4.4.2. Microsoft Office Access

Das Datenbankmanagementsystem (DBMS) Access aus dem Microsoft Office-Paket[52] ist eine Softwarelösung, die auch unerfahrenen Benutzern Schritt für Schritt hilft, eine Datenbank zu erstellen. Beim Starten der Anwendung kann zwischen dem Öffnen einer vorhandenen oder dem Erstellen einer neuen Datenbank ausgewählt werden[53]. Es gibt bereits einige Vorlagen, die dem Benutzer direkt fertige Tabellen und Abhängigkeiten erstellen. Dieses Schema kann allerdings problemlos erweitert und den eigenen Bedürfnissen angepasst werden. Es besteht auch die Möglichkeit, eine komplett leere Datenbank anzulegen.

Nachdem eine Datenbank angelegt wurde, wird ein neues Fenster geöffnet. Dieses Fenster ist zunächst leer oder es wird eine Übersicht über die vorhandenen Tabellen angezeigt[54], falls eine vorhandene Datenbank geöffnet wurde. In diesem Fenster lassen sich auch neue Tabellen anlegen oder aber auch die vorhandenen Tabellen editieren[55]. So lassen sich neue Spalten anlegen, neue Daten eintragen oder auch die Tabellen wieder löschen. Die Primärschlüssel, Fremdschlüssel oder weitere Integritätsbedingungen können direkt beim Erstellen oder Editieren einer Spalte erstellt werden.

Die Access-Datenbank von Microsoft verfügt über keine Views, Trigger, Funktionen, Prozeduren oder ähnliches. Es können sogenannte Macros definiert werden, die dann explizit aufgerufen werden (ähnlich einer Funktion). Nach dem Anlegen der Tabellen können Berichte erstellt werden, die den aktuellen Tabelleninhalt wiedergeben[56]. Es gibt vorgefertigte Berichte, die alle Spalten anzeigen. Diese Berichte lassen sich aber auch editieren, so dass bestimmte Spalten eventuell nicht angezeigt oder Spalten einer anderen Tabelle eingefügt werden. Sie dienen zur Übersicht der vorhandenen Daten und können

[52] s. [Microsoft Corp. 09b]
[53] s. Anhang A, S. 174, Abb. A.36 (A.36)
[54] s. Anhang A, S. 174, Abb. A.38 (A.38)
[55] s. Anhang A, S. 174, Abb. A.37 (A.37)
[56] s. Anhang A, S. 175, Abb. A.39 (A.39)

ausgedruckt werden[57].

Die Anwendung gibt dem Benutzer eine direkte Möglichkeit zum Erstellen eines Formulars[58]. Mit diesen Formularen können die Daten später ohne weitere SQL-Kenntnisse eingegeben werden. Dabei kann zwischen vorgefertigten Formularen ausgewählt werden, die die vorhandenen Tabellen und Spalten benutzen, oder es werden eigene Formulare erstellt[59]. Die vorgefertigten Formulare lassen sich auch editieren, so dass die zusätzlich gewünschten Formularinhalte eingefügt oder die unerwünschten gelöscht werden können.

Microsoft Access bietet eine Möglichkeit zur Visualisierung der Tabellen in Form eines ER-Diagramms[60]. Dabei werden die Fremdschlüsselbeziehungen zwischen den Tabellen dargestellt. In dieser Ansicht können die entsprechenden Abhängigkeiten auch modelliert werden. Weitere Arten der visuellen Darstellung bietet die Access-Datenbank nicht an.

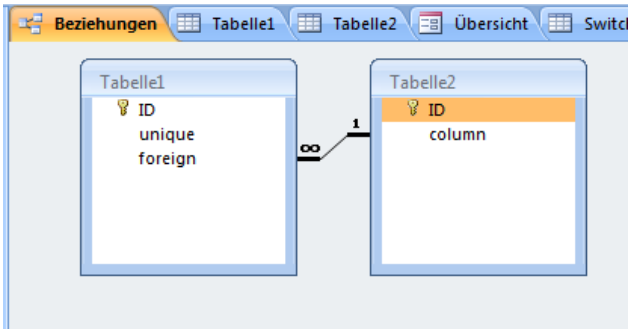


Abbildung 4.3: Microsoft Office Access – ER-Diagramm

Das DBMS Access ist nur für Windows verfügbar und erfüllt somit nicht die Anforderung der Plattformunabhängigkeit.

4.4.3. MySQL Workbench

Die MySQL Workbench ist eine Schemadesignsoftware, die frei erhältlich[61] und auf den Plattformen Windows, Linux und Mac OS X lauffähig ist. Abbildung A.33 (Anhang A) veranschaulicht den prinzipiellen Aufbau der MySQL Workbench. Die Hauptansicht zeigt nach dem Start nur das Hintergrundgitter, das ausgeschaltet werden kann. Es besteht nun die Möglichkeit, entweder ein vorhandenes Modell zu laden oder mit dem Entwurf

[57] vgl. [Asal 09]
[58] s. Anhang A, S. 175, Abb. A.40 (A.40)
[59] vgl. [Asal 09]
[60] s. S. 31 Abb. 4.3 (4.3)
[61] s. [Sun Microsystems 09e]

eines neuen zu beginnen. Alternativ besteht die Möglichkeit, ein vorhandenes Schema zu untersuchen und ein Modell zu erstellen (Reverse Engineering).

Die MySQL Workbench ist ein Designwerkzeug und arbeitet mit dem Extended Entity Relationship (EER)-Modell[62] zur Erstellung kleiner und großer Modelle[63]. Dabei bietet sie zwei Funktionen an, den Import eines vorhandenen Schemas (Reverse Engineering) oder das Erstellen und Erzeugen der Skripte eines neuen Schemas. Bei dem Reverse Engineering beherrscht die MySQL Workbench eine Zweiwegesynchronisation. Es besteht die Möglichkeit eines Updates des Modells nach Änderungen im Schema und umgekehrt. Eventuell auftretende Konflikte müssen allerdings manuell gelöst werden. Dabei wird einer der beiden Seiten (Modell oder existierendes Schema) Vorrang gewährt.

In der Hauptansicht lassen sich neue Datenbankobjekte anlegen, wobei die Funktionalität auf Tabellen, Views und Prozeduren beschränkt ist[64]. Benötigte Trigger können bei dem Erstellen einer Tabelle angelegt und auf dieser Tabelle definiert werden. Ein späteres Bearbeiten der Tabellen und Views ist auch möglich, wodurch die Trigger auch erst im späteren Verlauf erstellt werden können. Bei dem Anlegen einer Tabelle können die Spalten und Integritätsbedingungen wie Primär- oder Fremdschlüssel direkt per Mausklick angelegt werden. Die erstellten Objekte werden im Hauptfenster zur Übersicht angezeigt.

Die zweite mögliche Ansicht[65], die die MySQL Workbench zur Verfügung stellt, ist das EER-Diagramm. Bei dem Import (Reverse Engineering) eines vorhandenen Schemas werden die Objekte direkt angezeigt. Es lassen sich aber auch neue Integritätsbedingungen wie Fremdschlüsselbeziehungen definieren, wobei die Änderungen im Modell per Knopfdruck an das Schema übertragen werden können. Bei dem Anlegen eines neuen Schemas besteht die Möglichkeit, die in der ersten Hauptansicht erstellten Objekte in einem Menü auszuwählen und mit der Maus in diese Ansicht zu ziehen.

In diesem EER-Diagramm werden ausschließlich Tabellenbeziehungen über die Fremdschlüssel angezeigt. Weitere Möglichkeiten einer Visualisierung werden nicht angeboten. So lässt sich grafisch nicht erkennen, welche Views von anderen Views oder Tabellen abhängen. Die Trigger werden nicht grafisch dargestellt, wodurch sich auch hier keine Abhängigkeiten oder Rekursionen erkennen lassen.

[62] Ein Extended Entity Relationship-Modell beinhaltet zusätzlich Aggregationen sowie Generalisierung und Spezialisierung
[63] vgl. [Sun Microsystems 09f]
[64] s. Anhang A, S. 173, Abb. A.34 (A.34)
[65] s. S. 33 Abb. 4.4 (4.4)

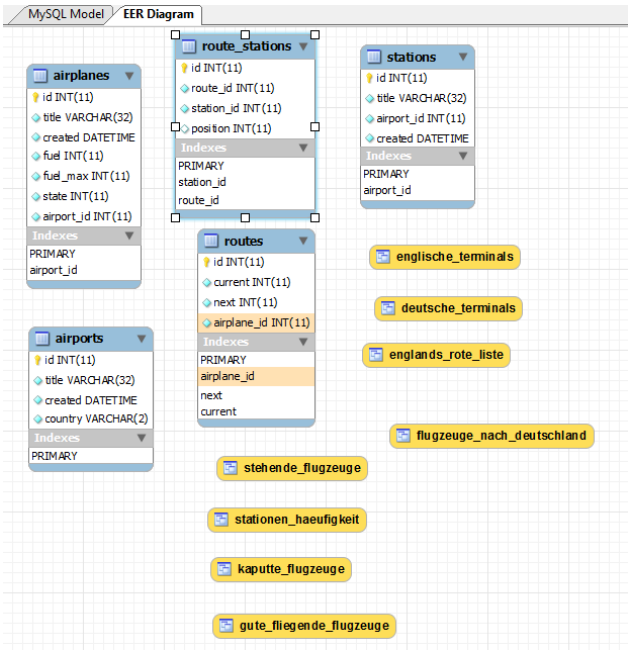


Abbildung 4.4: MySQL Workbench – EER-Diagramm

4.4.4. Oracle SQL Developer

Der Oracle SQL Developer[66] ist eine Anwendung zur Unterstützung administrativer Aufgaben. Diese Anwendung wurde ebenfalls in Java implementiert und ist somit plattformunabhängig. Allerdings ist der SQL Developer auf eine Oracle-Datenbank beschränkt und kann nicht für andere Datenbankhersteller verwendet werden.

Die Hauptkomponenten des Oracle SQL Developers in der grafischen Oberfläche sind:

1. Menü und Werkzeugleiste
2. Datenbank Navigator mit Kontextmenü
3. Konsole zur Ausgabe und Fehleranzeige
4. SQL-Editor
5. Verschiedene Editoren für „Procedural Language/SQL(PL/SQL)“

Der Datenbank Navigator ermöglicht eine Übersicht über alle angelegten Verbindungen[67]. Er bietet zu jeder Verbindung eine Anzeige über alle angelegten Tabellen, Views, Funktionen, Prozeduren, Trigger und viele weitere Datenbankobjekte. Die einzelnen Datenbankobjekte lassen sich über ein Kontextmenü verändern und löschen. Weiterhin lassen sich auch neue Objekte direkt anlegen.

[66] s. [Oracle 07]
[67] s. Anhang A, S. 169, Abb. A.23 (A.23)

Der Oracle SQL Developer ermöglicht das Bearbeiten, Anlegen und Löschen einer Verbindung über ein modales Dialogfenster. In diesem Dialogfenster kann die Verbindung getestet werden, es gibt eine Option zum Speichern des Passwortes und weitere Auswahlmöglichkeiten (Verbindung als Datenbankadministrator zum Beispiel). Die Verbindung kann direkt hergestellt werden und es gibt ein Hilfemenü[68].

Weiterhin verfügt der Oracle SQL Developer über ein Editor-Fenster. In diesem Fenster können Structured Query Language (SQL)-Statements ausgeführt werden, wobei die Ergebnisse bei einer Data Query Language (DQL)-Abfrage in tabellarischer Form im unteren Bereich ausgegeben werden[69]. In diesem Fenster lassen sich auch neue Datenbankobjekte anlegen, wie zum Beispiel Tabellen, Views, Funktionen und vieles mehr. Eine mögliche Fehlermeldung wird direkt in dem Reiter „Script Output“ angezeigt, wobei auch eine Meldung bei Erfolg ausgegeben wird.

Bestehende Datenbankobjekte, die in Procedural Language/SQL (PL/SQL) angelegt werden, können über einen speziellen Editor direkt bearbeitet und kompiliert werden. Um diesen Editor aufzurufen genügt ein Doppelklick auf das entsprechende Datenbankobjekt in der Verbindungsübersicht[70]. Es werden zwei neue Fenster geöffnet, wobei der entsprechende Quellcode des Datenbankobjektes in beiden Fenstern angezeigt wird. Das erste Fenster dient ausschließlich zur Anzeige des Quellcodes und das Objekt kann dort nicht bearbeitet werden. Im zweiten Fenster[71] gibt es diese Option zusätzlich und auch eine Funktion zum Kompilieren des PL/SQL-Codes. Handelt es sich bei diesem Datenbankobjekt um eine Prozedur oder Funktion, kann diese direkt ausgeführt werden. Eine zusätzliche Debugging-Funktion ist ebenfalls vorhanden.

Neue Datenbankobjekte, die in PL/SQL geschrieben sind, können auch direkt über ein Kontextmenü in der Verbindungsübersicht[72] angelegt werden. Dabei öffnet sich ein Dialogfenster, das alle benötigten Einstellungen wie Name des Objektes, Rückgabewerte (bei einer Funktion) oder das Erstellen der Übergabeparameter erlaubt[73].

Der Oracle SQL Developer verzichtet, bezogen auf die funktionalen Anforderungen, komplett auf Visualisierungen. Mögliche Abhängigkeiten zwischen den einzelnen Datenbankobjekten werden in tabellarischer Form ausgegeben[74]. So werden die Fremdschlüsselbeziehungen bei Tabellen wie auch die involvierten Tabellen oder Views zu einer an-

[68] s. Anhang A, S. 169, Abb. A.24 (A.24)
[69] s. Anhang A, S. 170, Abb. A.25 (A.25)
[70] s. Anhang A, S. 169, Abb. A.23 (A.23)
[71] s. Anhang A, S. 170, Abb. A.26 (A.26)
[72] s. Anhang A, S. 169, Abb. A.23 (A.23)
[73] s. Anhang A, S. 171, Abb. A.27 (A.27)
[74] s. S. 35 Abb. 4.5 (4.5)

deren View angezeigt. Auch die entsprechenden PL/SQL-Objekte wie Trigger oder Funktionen, die auf einer Tabelle definiert sind, werden in dieser Form dargestellt. Dabei können diese Trigger und Funktionen auf weitere Tabellen oder Views zugreifen. Die hieraus entstehenden Abhängigkeiten zu anderen Tabellen oder Views können durch diese tabellarische Ausgabe nur schwer erkannt werden.

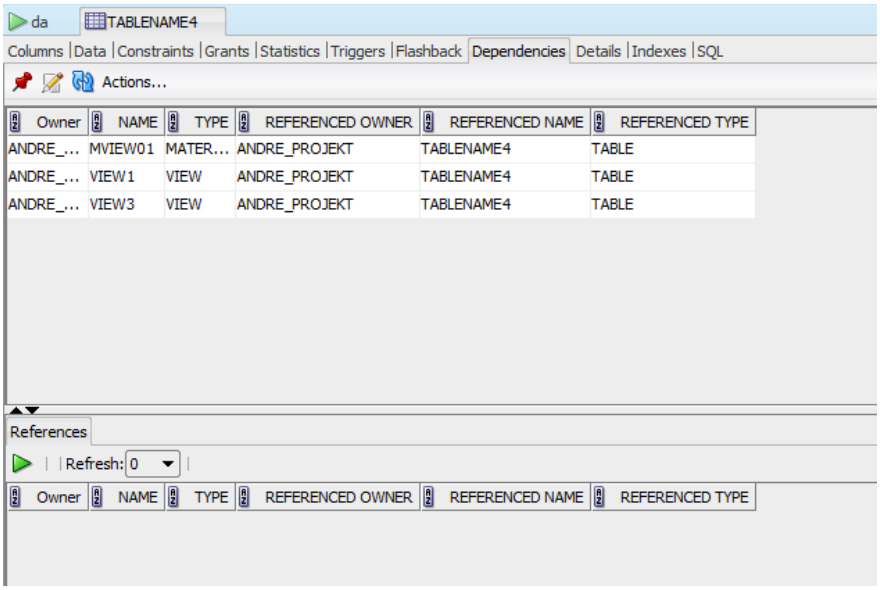


Abbildung 4.5: Oracle SQL Developer – Abhängigkeiten

4.4.5. SQL Developer

Der SQL Developer[75] ist ein SQL Client. Er dient zur Abfrage und Visualisierung von Datenbankobjekten und unterstützt administrative Aufgaben. Da die Anwendung komplett in Java implementiert wurde, ist sie auf allen Plattformen lauffähig, die die entsprechende Java Laufzeitumgebung (Version 1.4 oder höher) unterstützen. Das Tool unterstützt alle Datenbanksysteme, die über einen JDBC-Treiber verfügen, wie zum Beispiel Oracle, MySQL, DB2 oder PostgreSQL.

Um mit dem SQL Developer eine Verbindung zu einer Datenbank herzustellen, wird diese Verbindung zunächst angelegt. Die benötigten Einstellungen einer Verbindung können in einem Dialogfenster erstellt werden[76]. Damit der SQL Developer auf eine Datenbank zugreifen kann, muss ein passender JDBC-Treiber installiert sein. Die entsprechenden JDBC-Treiber werden in der Regel auf den Internetseiten des jeweiligen Datenbankherstellers kostenfrei zum Download angeboten.

[75] s. [Borchers 07]
[76] s. Anhang A, S. 167, Abb. A.20 (A.20)

Die Hauptkomponenten des SQL Developers in der grafischen Oberfläche sind[77]:

1. Menü und Werkzeugleiste
2. Datenbanknavigator mit Kontextmenü
3. Konsole zur Ausgabe und Fehleranzeige
4. SQL-Editor
5. Verschiedene Anzeigewerkzeuge zur Darstellung der Datenbankobjekte im Multiple Document Interface (MDI)

Der Datenbanknavigator stellt alle aktuell geöffneten Datenbankverbindungen in einer Baumansicht dar und erlaubt den Zugriff auf einzelne Datenbankobjekte wie zum Beispiel Tabellen, Views und Benutzer. Der Navigator verfügt über eine Werkzeugleiste, mit der die wichtigsten Funktionen aufgerufen werden können. Es besteht zu dem ein Kontextmenü, über das man Zugriff auf den vollständigen Funktionsumfang erhält. Am unteren Rand wird der Pfad der aktuellen Selektion im Baum in einer Statuszeile angezeigt.

Der SQL Developer verfügt über zwei Editoren[78], den sogenannten *Statement Editor* und den *Skript Editor*. Der *Statement Editor* dient zur Ausführung einzelner SQL-Statements. Der Inhalt des Editors wird komplett zur Ausführung an die Datenbank gesendet. Die Ergebnisse der gesendeten Abfragen werden als Tabelle im unteren Teil des Editors angezeigt. Im Gegensatz dazu dient der *Skript Editor* zur Ausführung ganzer SQL-Skripte. Der Inhalt des Editors wird dabei in einzelne Befehle zerlegt, die jeweils an die Datenbank gesendet werden.

Weiterhin besitzt der SQL Developer eine Anzeige von Binary Large Objects (BLOB). Mit dieser Funktion können beliebige Binärdateien dargestellt werden. Bilder in gängigen Formaten werden direkt angezeigt oder gespeichert. Mit der Anzeige für Character Large Objects (CLOB) können gespeicherte Texte angezeigt werden. Befindet sich das Dokument im XML-Format, wird der Text automatisch mit einer Syntax-Hervorhebung angezeigt.

Für das Bearbeiten und Kompilieren von Prozeduren oder Funktionen ist der *Stored Program Editor* zuständig[79]. Dabei wird der komplette Inhalt des Editors zum Kompilieren an die Datenbank gesendet.

Der SQL Developer bietet zudem eine Visualisierung von Datenbankobjekten, die für das Projekt besonders interessant ist. Diese Funktion stellt den Inhalt eines Schemas

[77] s. Anhang A, S. 167, Abb. A.19 (A.19)
[78] s. Anhang A, S. 168, Abb. A.21 (A.21)
[79] s. Anhang A, S. 168, Abb. A.22 (A.22)

ähnlich einem erweiterten Entity-Relationship-Diagramm (EERD) dar[80].

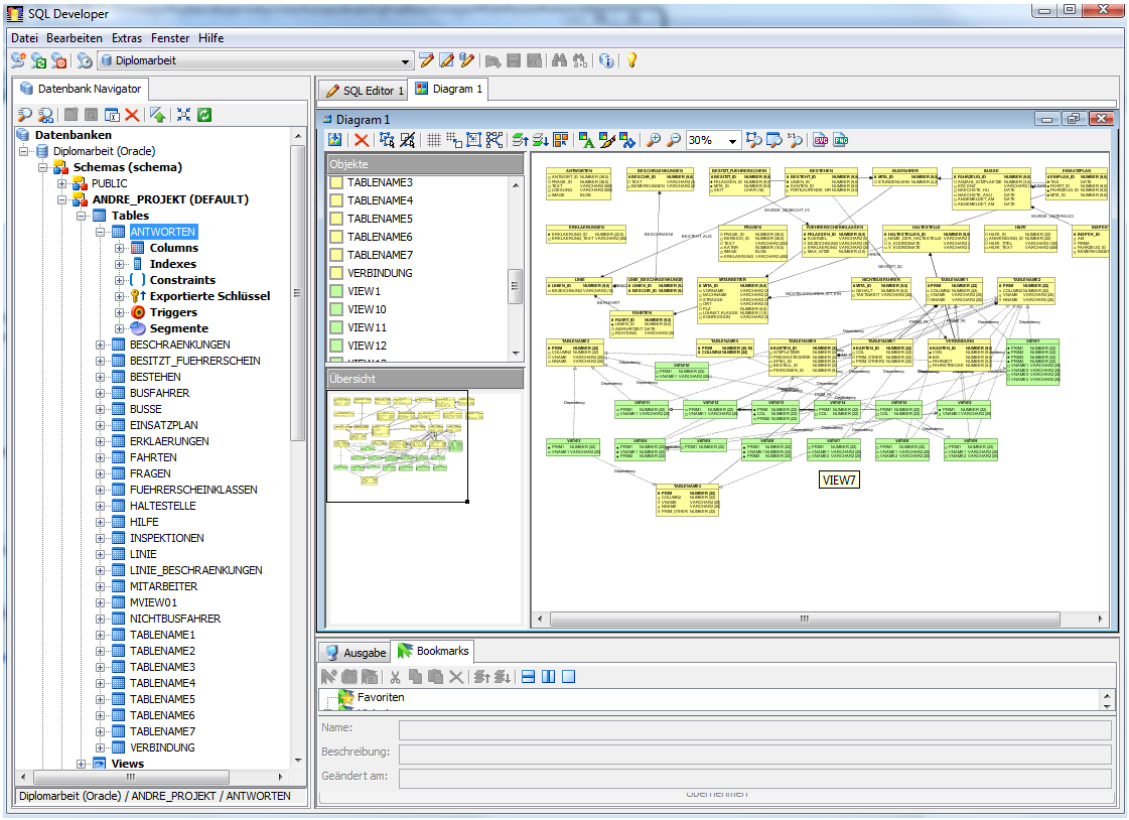


Abbildung 4.6: SQL Developer – Diagramm

In diesem ERD werden die Tabellenbeziehungen anhand der Fremdschlüssel angezeigt sowie die Beziehungen zu einer View. Beide Beziehungsarten werden in einem Diagramm dargestellt, wodurch die Ansicht bei größeren Schemata sehr unübersichtlich wird. Die Fremdschlüsselbeziehungen der Tabellen untereinander werden ohne Kardinalität und Optionalität angezeigt. Der SQL Developer zeigt keinen Unterschied zwischen einer identifizierenden oder einer nicht-identifizierenden Beziehung an. Allerdings werden die Attribute der einzelnen Beziehungen angezeigt.

Bei den Abhängigkeiten von einer View zu einer Tabelle oder einer anderen View wird nicht zwischen negativen und positiven Abhängigkeiten unterschieden. Ein Pfeil verdeutlicht, dass eine Beziehung zwischen diesen beiden Datenbankobjekten vorhanden ist. Allerdings wird dadurch nicht ersichtlich, in welcher Form diese Abhängigkeit besteht.

Eine Visualisierung über die vorhandenen Trigger wurde nicht implementiert. So lässt sich aus grafischer Sicht nicht erkennen, welcher Trigger zu welcher Tabelle gehört und ob es eine Rekursivität unter den Triggern gibt.

[80] s. S. 37 Abb. 4.6 (4.6)

4.4.6. Toad

Die Softwareanwendung Toad[81] steht für „Tool for Oracle Application Developer“ und wurde zunächst für den Datenbankhersteller Oracle entwickelt. Mittlerweile[82] ist Toad auch in anderen Versionen und somit auch für weitere Datenbankhersteller verfügbar (zum Beispiel MySQL oder DB2). Diese Anwendung ist ein SQL Client und dient zur Unterstützung administrativer Tätigkeiten. Allerdings ist Toad, wie auch das DBMS Access[83], nur für Windows verfügbar und erfüllt nicht die Anforderung der Plattformunabhängigkeit.

Um Toad für Oracle in Betrieb zu nehmen muss auf dem System entweder eine Oracle Datenbank installiert oder zumindest die Oracle Client Version vorhanden sein. Beim Starten der Anwendung überprüft Toad das Vorhandensein einer Installation von Oracle und gibt eine Meldung an den Benutzer. Das Anlegen einer Verbindung ist ohne installierte Version von Oracle in einer der beiden Formen nicht möglich und der Betrieb von Toad kann nicht weiter ausgeführt werden.

Nach dem Starten der Anwendung erscheint direkt ein modaler Dialog mit einer Verbindungsübersicht[84]. In dieser Ansicht werden die bereits angelegten Datenbankverbindungen angezeigt und es kann eine neue Verbindung angelegt werden. Die benötigten Einstellungen, um eine Verbindung zu einer Datenbank herzustellen, können in diesem Fenster erledigt werden. Im Gegensatz zu den anderen hier aufgeführten Softwareanwendungen kann eine neue Verbindung nicht getestet werden. Nach dem Auswählen einer Verbindung gelangt man in die Hauptansicht der Anwendung.

Die Softwareanwendung Toad besitzt eine Vielzahl an Funktionen, von denen einige hier vorgestellt werden. Eine Funktion dient dabei zur Abfrage, Veränderung, Einfügen und Löschen von Daten. Der SQL Editor[85] verarbeitet einzelne SQL-Anweisungen oder ganze SQL-Skripte. Sollte ein Fehler auftreten, wird eine entsprechende Meldung im unteren Bereich des Editors ausgegeben und der Anwender kann entsprechend reagieren. Es werden auch die Ausgaben der Abfragen von Daten in diesem Bereich in tabellarischer Form dargestellt.

Weiterhin verfügt Toad über den SQL Modeler[86], mit dem sich die `SELECT`-Anweisungen erstellen lassen. Dabei bietet Toad eine Auswahl mit den vorhandenen Datenbankobjekten wie Tabellen oder Views an. Durch Klicken auf die entsprechende Anweisungskom-

[81] s. [Quest Software 09]
[82] Stand Juli 2009
[83] s. Seite 30, Abschnitt 4.4.2 (4.4.2)
[84] s. Anhang A, S. 175, Abb. A.41 (A.41)
[85] s. Anhang A, S. 176, Abb. A.42 (A.42)
[86] s. Anhang A, S. 176, Abb. A.43 (A.43)

ponente (`FROM`- oder `WHERE`-Klausel, `GROUP BY` und weitere vorhandene Möglichkeiten) kann die gewünschte `SELECT`-Anweisung interaktiv erstellt werden. Mit dieser Funktion bietet Toad auch unerfahrenen Datenbankanwendern eine Möglichkeit, die teils komplexen `SELECT`-Anweisungen zu erstellen und auszuführen.

Mit dem Schema Browser[87] bietet Toad eine Übersicht über alle in der Datenbank vorhandenen Objekte. Die einzelnen Objekte werden dabei nach ihrem Typ unterschieden (zum Beispiel Tabelle oder View) und jeweils zu einer Liste zusammengefasst. Durch die Auswahl eines Objektes wird eine nähere Beschreibung des Objektes angezeigt. So bekommt der Anwender beispielsweise bei einer Tabelle die Spaltendefinitionen, Integritätsbedingungen, vorhandene Trigger der Tabelle, Rechte und noch einige weitere Daten angezeigt.

Eine Visualisierung, die von Toad implementiert wurde, ist die Darstellung der Abhängigkeiten durch Fremdschlüsselbeziehungen[88]. Dabei liest die Funktion ein vorhandenes Schema aus und zeigt die Abhängigkeiten der Tabellen untereinander in einem Entity-Relationship-Diagramm (ERD) an.

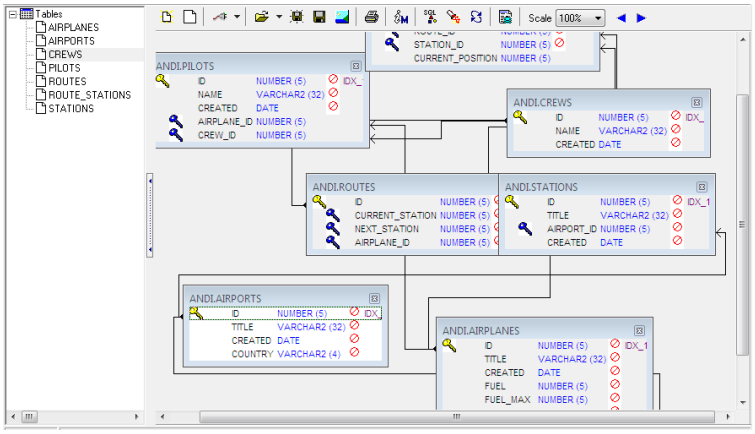


Abbildung 4.7: TOAD – Entity-Relationship-Diagramm

Eine grafische Ausgabe weiterer Abhängigkeiten findet nicht statt. Allerdings werden die Abhängigkeiten der Views zu den Tabellen oder anderen Views in tabellarischer Form dargestellt[89]. Die Anzeige, welche Views von der vorhandenen View abhängen, kann über einen weiteren Reiter eingesehen werden.

[87] s. Anhang A, S. 176, Abb. A.44 (A.44)
[88] s. S. 39 Abb. 4.7 (4.7)
[89] s. S. 40 Abb. 4.8 (4.8)

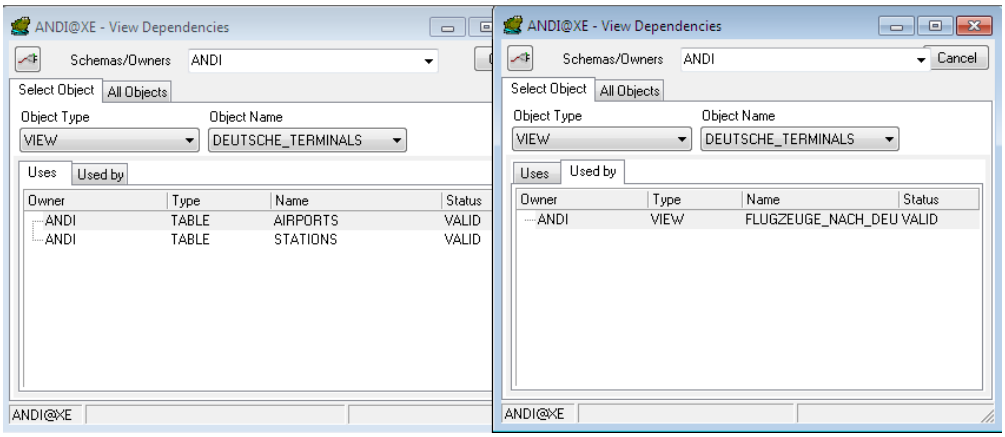


Abbildung 4.8: TOAD – Abhängigkeiten einer View

In diesem Reiter besteht außerdem die Möglichkeit der Anzeige aller Abhängigkeiten der verfügbaren Views[90]. In dieser Ansicht werden auch indirekte Abhängigkeiten der Views dargestellt. Da eine View auf einer oder mehreren anderen Views basieren kann, ermöglicht diese Funktion das Ermitteln der Ursprungstabellen.

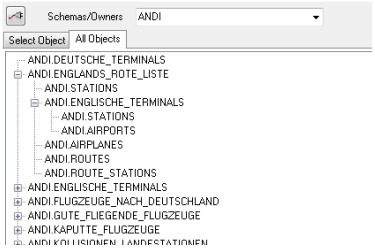


Abbildung 4.9: TOAD – Abhängigkeiten aller Views

4.5. Fazit

Neben den hier vorgestellten Softwareanwendungen gibt es noch weitere, teils kostenpflichtige Anwendungen, die alle einen ähnlichen Funktionsumfang bieten. Wir haben während der Evaluierung der Softwareprogramme festgestellt, dass ein Großteil der Anbieter gänzlich auf Visualisierungen irgendeiner Art verzichtet. Diese Anwendungen dienen daher eher als Unterstützung administrativer Aufgaben zum Erstellen, Bearbeiten und Löschen von Datenbankobjekten, der Vergabe von Rechten oder aber auch zur Analyse der Datenbank.

Somit ist die wesentliche Aufgabe, die unsere Anwendung leisten soll (die Visualisie-

[90] s. S. 40 Abb. 4.9 (4.9)

rung) nicht erfüllt[91]. Wenn eine Visualisierung implementiert wurde, dann wurde sie auf ein ERD oder ein EERD beschränkt. Die Abhängigkeiten der Views zu Tabellen oder anderen Views wurde in keiner hier vorgestellten Anwendung realisiert. Daher werden auch keine positiven und negativen Abhängigkeiten dargestellt. Die Trigger werden teilweise gar nicht angezeigt, auch nicht in der Datenbankobjekt-Übersicht. Somit kann der Benutzer nicht erkennen, ob ein Trigger durch einen DML-Befehl eventuell einen weiteren Trigger anstößt. So können auch keine Rekursionen der Trigger untereinander wahrgenommen werden.

Keine der hier vorgestellten Anwendungen erfüllt somit die gestellten Anforderungen[92] und Funktionen. Auch haben die Recherchen ergeben, dass eine Anwendung, wie sie gefordert ist, momentan[93] nicht auf dem Markt existiert. Durch diese Recherchen können wir sicherstellen, dass unsere Anwendung eine neue Form der Visualisierungen im Bereich der Datenbankanwendungen darstellt und damit eine Lücke in diesem Bereich schließt.

[91] s. Seite 20, Abschnitt 3.2 (3.2)
[92] s. Seite 18, Abschnitt 3 (3)
[93] Stand Juli 2009

5. Vergleich und Diskussion von Frameworks zur Entwicklung

Neben dem Vergleich und der Diskussion vorhandener Softwareprodukte[94] werden im Rahmen dieser Arbeit noch zwei weitere, wichtige Frameworks evaluiert und bewertet. Für diese beiden Komponenten der Anwendung, das Parsen und die grafische Anzeige der Abhängigkeiten, gibt es bereits einige fertige Implementierungen.

Diese beiden Komponenten bilden den Grundstein der Anwendung. Das Parsen der Structured Query Language (SQL)-Anweisungen wird benötigt, um die teilweise komplexen positiven und negativen Abhängigkeiten der Views untereinander und zu den Tabellen aufzuschlüsseln. Weiterhin muss der Parser die von SQL unterschiedliche Procedural Language/SQL (PL/SQL) verstehen und verarbeiten können, um die Abhängigkeiten der Trigger untereinander zu entschlüsseln. Ein gutes Graphenframework wird benötigt, um genau diese Abhängigkeiten geeignet visuell darzustellen. Die Auswahl dieser Komponenten ist also entscheidend für die Qualität der zu entwickelnden Software.

Für den Parser ist es wichtig, dass die Abhängigkeiten genau erkannt werden. Nur so kann sichergestellt werden, dass die Software voll funktionsfähig ist. Werden die SQL- beziehungsweise PL/SQL-Anweisungen falsch oder sogar nur teilweise geparst, so werden die Abhängigkeiten falsch oder eventuell gar nicht dargestellt.

Ebenfalls ist die Auswahl eines guten Graphenframeworks wichtig. Es reicht nicht aus, mit dem Parser die Abhängigkeiten richtig auszulesen und zu entschlüsseln, wenn sie später grafisch nicht korrekt dargestellt werden können. Nur so kann sichergestellt werden, dass der Anwender die teilweise komplexen Abhängigkeiten erkennen kann.

Zunächst gilt es abzuschätzen, ob eine Eigenentwicklung einer bestehenden Lösung vorzuziehen ist. Dafür wird der Umfang einer eigenen Entwicklung abgeschätzt. Im Gegensatz dazu wird betrachtet, wie aufwendig es ist, eine fertige Lösung zu integrieren und auf die Aufgabenstellung hin anzupassen.

Für die Auswahl eines Graphenframeworks bietet es sich an, Prototypen zu erstellen und zu evaluieren. Diese Prototypen sind leicht zu erstellen und dienen als Grundlage für die Bewertung dieses Frameworks. Zu beachten ist dabei unter anderem, dass die einzelnen Knoten (in diesem Fall die zu visualisierenden Datenbankobjekte) im Graphen keine Baumstruktur aufweisen, sondern eventuell sogar rekursive Kanten besitzen. Es können auch einzelne Knoten ohne jegliche Verbindung entstehen. Das Graphenframework muss durch ein besonderes Layout die Fähigkeit besitzen, auch solche Graphen anzuzeigen.

[94] s. Seite 25, Abschnitt 4 (4)

Auch für den Parser bietet es sich an, jeweils einen Prototypen zu erstellen und zu bewerten. Allerdings findet bei dem Parser keine grafische Ausgabe statt. Um entscheiden zu können, wie gut ein Parser funktioniert, müssen konkrete Testfälle entwickelt werden[95]. Anhand der Testauswertung kann evaluiert werden, ob dieser Parser den Anforderungen entspricht.

Für beide Frameworks besteht außerdem die Anforderung, dass die Anwendung erweiterbar sein soll[96]. Es sollen später neue Funktionen und weitere grafische Ausgaben erfolgen. Außerdem soll die Anwendung später auch für weitere Datenbankhersteller erweitert werden. Durch andere Datenbankdialekte entstehen besondere Anforderungen an den Parser[97]. Daher werden diese beiden Frameworks auch auf ihre Erweiterbarkeit und Wartbarkeit hin untersucht, um zu vermeiden, dass die Frameworks später ausgetauscht werden müssen. Um eine wesentliche Mehrarbeit bei Erweiterungen zu vermeiden, sollte im Vorfeld auf diese Anforderung eingegangen werden.

5.1. SQL-Parser

Die Auswahl des Structured Query Language (SQL)-Parsers ist bedeutend für die Anwendung, um die Abhängigkeiten der Datenbankobjekte exakt zu erkennen. Der Parser muss zwei verschiedene Arten von Objekten parsen können. Das sind die in der Datenbank gespeicherten virtuellen Sichten (Views), die in den SQL-Bereich fallen, und die Trigger, die in Procedural Language/SQL (PL/SQL) geschrieben sind. Da sich SQL von PL/SQL in Syntax und Semantik unterscheidet, muss der Parser beides erkennen und verarbeiten können.

Alle Informationen über die Abhängigkeiten der Views zu den Tabellen oder der Views untereinander werden beim Erstellen einer View in der *SELECT*-Anweisung gespeichert. Diese Anweisung gibt Auskunft über die bestehenden Abhängigkeiten. Zunächst gibt der Teil der *FROM*-Klausel Auskunft über alle involvierten Tabellen und Views. Weiterhin klärt die *WHERE*-Bedingung, in welcher Form die Tabellen und Views involviert sind. Hieraus ergeben sich negative und positive Abhängigkeiten zu den Tabellen und Views[98]. Eine Oracle-spezifische Rekursion durch eine `CONNECT BY`-Komponente lässt dabei nicht auf weitere Abhängigkeiten schließen, da sich die Rekursion auf die Tabellenebene bezieht. Die Tabellen, von denen eine View abhängt, werden bereits durch das Parsen der `FROM`- beziehungsweise der `WHERE`-Komponente erkannt und gespeichert.

[95] s. Seite 122, Abschnitt 8.2 (8.2)
[96] s. Seite 18, Abschnitt 3 (3)
[97] s. Seite 45, Abschnitt 5.1.2 (5.1.2)
[98] s. Seite 20, Abschnitt 3.3 (3.3)

Um die Anhängigkeiten grafisch darstellen zu können, werden also genau diese Informationen der Abhängigkeiten benötigt. Die Informationen der Projektion (*SELECT*-Anweisung) sind für die Bestimmung der Abhängigkeiten irrelevant und werden nicht geparst.

Trigger sind Datenbankobjekte, die in den PL/SQL-Bereich fallen. „... PL/SQL ist eine *ORACLE*-spezifische prozedurale Erweiterung von SQL, die auf ADA[99] basiert...“[100]. Mit PL/SQL lassen sich Funktionen und Prozeduren realisieren, die durch einen entsprechenden Aufruf oder durch ein bestimmtes Ereignis ausgelöst werden. Trigger werden automatisch, ohne expliziten Aufruf, durch ein Ereignis ausgelöst und ausgeführt.

Bei dem Parsen eines Triggers ergibt sich die Schwierigkeit der vollkommen unterschiedlichen Syntax von PL/SQL, die der Parser erkennen muss. Es sind andere Schlüsselwörter zu beachten, die der Parser richtig interpretieren muss und dementsprechend die involvierten Tabellen und Views erkennt. Da nicht zuletzt PL/SQL wesentlich umfangreicher als SQL ist, stellt dies eine weitere schwer zu erfüllende Anforderung an die Grammatik des Parsers. Um entscheiden zu können, welche Tabellen oder Views durch diesen Trigger involviert sind, muss der Rumpf des Triggers geparst und die entsprechenden Tabellen oder Views gefiltert werden.

Es gibt einige, bereits fertig implementierte Parser auf dem Markt. Für die Entscheidung, ob ein fertiger Parser einer Eigenentwicklung vorzuziehen ist, werden verschiedene Parser auf ihre Funktionalität getestet[101].

5.1.1. Nicht-funktionale Anforderungen

Da die Anwendung in Java entwickelt wird[102], sollte der verwendete Parser in Java implementiert sein oder zumindest über eine gute Integrationsmöglichkeit verfügen. Dies erweist sich als schwierig, weil es nur wenige SQL-Parser gibt, die diesem Kriterium entsprechen. Einige Parser wurden in anderen Programmiersprachen implementiert wie C oder C++. Die Integrationsmöglichkeiten sind daher entweder gar nicht oder nur sehr eingeschränkt vorhanden.

Einige Parser benötigen eine explizite Installation. Dies ist für die Anwendung nicht möglich, da sie alleine und ohne weitere Hilfsprogramme laufen soll. Die Anwendung soll auf jeder Plattform laufen, auf der Java ab der Version 1.6 installiert ist.

[99] s. [Wikipedia 09b]
[100] [Faeskorn-Woyke 07], S. 348
[101] s. Seite 122, Abschnitt 8.2 (8.2)
[102] vgl. Seite 23, Abschnitt 3.6 (3.6)

Die Anwendung wird in erster Linie für eine Oracle-Datenbank entwickelt. Da sich die SQL-Dialekte der einzelnen Datenbankhersteller unterscheiden, ergibt sich die Anforderung an den Parser, dass er den Oracle spezifischen SQL-Dialekt interpretieren kann und keine Programmausnahme wirft. Der Parser muss alle Varianten zum Erstellen einer View unter Oracle ohne Fehlermeldung parsen können.

Da die Anwendung auch für weitere Datenbanken erweiterbar sein soll, ergibt sich hieraus die zweite Schwierigkeit bei der Auswahl eines geeigneten Parsers. Durch die verschiedenen SQL-Dialekte der einzelnen Datenbankhersteller, sollte der Parser nicht nur den Oracle spezifischen SQL-Dialekt erkennen können. Es erscheint zunächst nicht praktikabel, für jedes Datenbanksystem einen eigenen Parser zu benutzen, sondern es sollte ein Parser für zumindest alle gängigen Datenbankhersteller benutzbar sein.

5.1.2. Funktionale Anforderungen

Der Parser muss zwei unterschiedliche Anweisungen erkennen und verarbeiten können. Dies ist sowohl die *SELECT*-Anweisung einer View als auch der *BODY (Rumpf)* eines Triggers.

Für die Views wird ein Parser benötigt, der die wesentlichen Komponenten der *SELECT*-Anweisung interpretieren und auslesen kann. Eine *SELECT*-Anweisung besteht in ihrer Grundform aus sechs Komponenten, von denen drei für die Anwendung wichtig sind und geparst werden müssen. Die Komponenten *SELECT*, *GROUP BY* und *ORDER BY* sind irrelevant, da sie keine Auskunft über die involvierten Tabellen oder Views und ihre positiven oder negativen Abhängigkeiten geben. Daher wird an dieser Stelle nicht weiter auf diese Komponenten eingegangen.

Um die Trigger parsen zu können, wird ein Parser benötigt, der PL/SQL interpretieren kann. Die Information, bei welchem Ereignis (*INSERT*, *UPDATE* oder *DELETE*) der Trigger ausgeführt wird, lässt sich aus dem Data Dictionary erfragen. Weitere Bestandteile der Informationen aus dem Data Dictionary sind die Bedingung beziehungsweise *WHEN*-Klausel (der Trigger wird nur nach erfolgreicher Überprüfung ausgeführt) sowie auch der Modus (Ausführung vor oder nach einem Ereignis). Die eigentliche Aufgabe des Parsers beschränkt sich also auf den Aktionsteil des Triggers, der sich im Rumpf befindet. Da PL/SQL Oracle spezifisch ist, muss dieser Teil des Parsens gegebenenfalls komplett ausgetauscht oder überarbeitet werden[103], wenn man Trigger in einem anderen SQL Dialekt wie MySQL parsen möchte.

[103] s. Seite 89, Abschnitt 6.3 (6.3)

5.1.2.1. FROM-Komponente Die *FROM*-Komponente ist von zentraler Bedeutung. Es werden die beteiligten Tabellen und Views aufgeführt, wodurch eine spätere Zuteilung der positiven und negativen Abhängigkeiten erfolgen kann. Außerdem werden für die Tabellen und Views teilweise Aliase vergeben. Dies ist besonders für das Parsen der anderen Komponenten von Relevanz, da die dort aufgeführten Bedingungen den entsprechenden Tabellen oder Views zugeordnet werden müssen. Anstelle eines Tabellen- oder Viewnamens kann auch hier eine Unterabfrage stehen, die wiederum eine weitere *FROM*-Komponente enthält. Auch dies muss richtig erkannt und verarbeitet werden.

5.1.2.2. WHERE-Komponente Durch das Parsen dieser Komponente wird ersichtlich, ob die Tabellen und Views aus der *FROM*-Komponente positiv oder negativ von dieser View abhängig sind. Der Parser sollte anhand jeder einzelnen Bedingung entscheiden können, ob sich daraus eine positive oder negative Abhängigkeit ergibt und er muss diese Bedingung der entsprechenden Tabelle oder View zuordnen können. Die verschiedenen Syntaxen einer Bedingung muss er erkennen können. Ein Outer-Join kann beispielsweise in der *FROM*-Komponente stehen. Allerdings gibt es in Oracle auch die Möglichkeit, die Join-Beziehung durch ein Plus-Zeichen an der entsprechenden Spalte kenntlich zu machen. Der Parser muss also auf beide Syntaxen reagieren können. Ebenfalls, wie in der *FROM*-Kausel, besteht auch hier die Möglichkeit einer Unterabfrage, in der andere Tabellen oder Views mit entsprechenden Abhängigkeiten aufgeführt werden. Dies muss ebenfalls erkannt und entsprechend geparst werden.

5.1.2.3. HAVING-Komponente Die *HAVING*-Komponente bezieht sich ausschließlich auf eine Gruppierungsfunktion und kann daher nur nach einer *GROUP BY*-Komponente erfolgen. Die *GROUP BY*-Komponente ist für die Anwendung unwichtig, da sie keine Relevanz für eine Abhängigkeit hat. Allerdings besteht die Möglichkeit, in der *HAVING*-Komponente eine Unterabfrage auszuführen. Dort können wiederum andere involvierte Tabellen/Views aufgeführt sein und neue Abhängigkeiten entstehen. Daher ist auch das Parsen dieser Komponente für das Erkennen der Abhängigkeiten relevant.

5.1.2.4. PL/SQL PL/SQL hat im Gegensatz zu SQL mehr Schlüsselwörter. Es können verschiedene Data Definition Language (DDL)-, Data Manipulation Language (DML)- oder auch Data Query Language (DQL)-Befehle ausgeführt werden. Ebenfalls sind weitere PL/SQL-Anweisungen möglich. So ist nicht nur wie bei einer View ein *SELECT* möglich, sondern auch *INSERT*, *UPDATE* und *DELETE*. Es gibt neue Befehle wie zum Beispiel `BEGIN` und `END`, die Transitionsvariablen `:NEW` und `:OLD`, aber auch Bedingungsblöcke

mit `IF/THEN/ELSE` oder Schleifenblöcke wie `LOOP/WHILE/FOR` sind möglich und müssen erkannt werden. Außerdem muss unterschieden werden, ob auf eine Tabelle oder View in Form einer *SELECT*-Anweisung lesend zugegriffen wird oder ob die entsprechende Tabelle oder View in Form einer DML-Anweisung manipuliert wird. Wichtig sind vor allem die DML-Anweisungen, da durch diese Anweisungen eventuell andere Trigger indirekt ausgeführt werden. Diese impliziten Triggeraktivitäten können zu rekursiven Verkettungen führen.

5.1.3. Übersicht

Die nachfolgend vorgestellten SQL-Parser werden auf die vorgestellten Anforderungen getestet. Leider gibt es nur sehr wenige Parser-Implementierungen, die den Anforderungen im Vorfeld genügen. Fast alle Parser sind entweder nicht in Java implementiert oder können den Oracle spezifischen SQL-Dialekt nicht interpretieren.

Anderseits gibt es Parser, die diesen Anforderungen entsprechen, die aber nicht Open Source oder frei verfügbar sind. Kostenpflichtige, kommerzielle Produkte können für die Anwendung nicht eingesetzt werden[104], da sie kostenfrei bleiben soll. Einige Parser, die zunächst gute Ergebnisse erzielen, werden im Folgenden vorgestellt.

Der Parser darf auch nicht als komplett eigene Anwendung laufen und muss integrierbar sein. Einige Parser sind nur als eigenes Programm lauffähig und bieten keine Unterstützung einer Integration. Diese Parser sind wegen dieser besonderen Eigenschaft ausgeschlossen.

Die Dokumentationen der einzelnen Parser sind unvollständig. Dadurch kann nicht vorab entschieden werden, ob ein Parser den Anforderungen entspricht. Es sind einige Testfälle nötig, um den Parser gegen die Anforderungen zu evaluieren und um zu entscheiden, ob der Parser für das Projekt geeignet ist[105].

5.1.4. Vergleich

Hier werden nun einige Parser mit ihren Funktionen vorgestellt. Die vorgestellten Parser werden gegen die Anforderung hin überprüft und beurteilt. Im Vergleich auf, dass ein Großteil der getesteten Parser sehr große Schwächen im Umgang mit Unterabfragen

[104] am Beispiel von GSP, s. Seite 48, Abschnitt 5.1.4.1 (5.1.4.1)
[105] s. Seite 122, Abschnitt 8.2 (8.2)

aufweisen. Auch Unterabfragen, die laut Oracle-Spezifikation möglich sind, muss der Parser erkennen und verarbeiten können. Diese Funktion muss somit komplett implementiert worden sein.

Die Parser werden auf diesen wichtigen Aspekt hin sehr genau untersucht und geprüft. Anhand dieses Kriteriums sowie die Ergebnisse und Auswertungen der Testfälle wird entschieden, ob ein Parser eingesetzt werden kann.

In einem späteren Abschnitt[106] werden noch die Schwierigkeiten beim Finden eines geeigneten Parsers für den PL/SQL-Teil (Trigger parsen) beschrieben. Da PL/SQL von den Parsern wenig oder meist gar nicht unterstützt wird, wird in einem eigenen Abschnitt hierauf eingegangen.

5.1.4.1. GSP - General SQL Parser

Es gibt einige mehr oder weniger gute SQL-Parser. Diese Parser sind zum Teil kostenpflichtig oder erfüllen nicht die Anforderungen, da sie in PHP, C oder C++ implementiert werden. Ein besonderer Parser wird an dieser Stelle vorgestellt. Er ist zwar nicht in Java implementiert worden und ist kostenpflichtig, allerdings wird vom Hersteller demnächst[107] eine Java-Version in Aussicht gestellt. Außerdem weist dieser Parser sehr gute Eigenschaften auf.

Der *General SQL Parser*[108] kann nicht nur den Oracle spezifischen Datenbankdialekt verarbeiten, sondern ist auch für fast alle gängigen Datenbanken einsetzbar. Er unterstützt beispielsweise MySQL, DB2, Informix, Sybase und viele weitere. GSP bietet eine Syntaxanalyse, kann mit Unterabfragen umgehen und erstellt eine umfangreiche Analyse über involvierte Tabellen und Spalten. Positive und negative Abhängigkeiten werden somit erkannt und angezeigt. Der Parser erstellt einen Binärbaum, der für die weitere Verarbeitung der SQL-Anweisung benutzt werden kann.

Der Parser liest den SQL-String ein und teilt die einzelnen Wörter in Tokens ein. Es können auch direkt mehrere SQL-Anweisungen hintereinander eingelesen werden. Dazu kann ein Delimiter (*Trennzeichen*) zum Beenden eingegeben werden. Der Parser erstellt dann eine zusammengehörige Liste aller eingelesenen Tokens. Anhand des eingestellten Datenbankdialektes erstellt der Parser einen *Parse-Baum*[109], solange es keinen Syntax-Fehler gibt. Der erstellte Baum wird intern umgeformt, die Wurzel ist die entsprechende SQL-Anweisung (*SELECT*, *INSERT* und so weiter). Wenn eine involvierte Tabelle gefunden wird, wird eine entsprechende Meldung ausgegeben. Eine spezielle Klasse wandelt

[106] s. Seite 52, Abschnitt 5.1.4.6 (5.1.4.6)
[107] Stand Juli 2009
[108] s. [Gudu Software 09a]
[109] Der Parse-Baum entspricht einem Binär-Baum mit Tokens der einzelnen Anweisungselemente.

den erstellten Baum in ein XML-Dokument um. Es kann aber auch eine eigene Klasse implementiert werden, die einen Baum erstellt, der den eigenen, angepassten Bedürfnissen entspricht. So kann die SQL-Anweisung zum Beispiel mit einer Syntax-Hervorhebung ausgegeben werden.

Auch bei fehlerhaften SQL-Anweisungen bricht der Parser das Programm nicht ab, sondern ignoriert die entsprechende Anweisung, gibt dafür eine Fehlermeldung aus und arbeitet die weiteren SQL-Anweisungen ab. Selbst wenn in einer Unterabfrage eine fehlerhafte Anweisung steht, umgeht der Parser diese Unterabfrage und liest nur die Hauptabfrage aus.

Der GSP verfügt über eine kurze Anleitung[110] mit einer Beschreibung über die Arbeitsweise des Parsers und wie er funktioniert. Weiterhin gibt es noch eine ausführliche, in mehrere Dokumente aufgeteilte Anleitung[111], die alle über den Webauftritt von GSP eingesehen werden können.

Dieser Parser ist kostenpflichtig und bisher nicht in Java implementiert. Die Preise für die Lizenzierung ergeben sich wie folgt:

Einzellizenz mit 30-Tage Support	USD	79.95
Einzellizenz mit einjährigem Support	USD	119.95
Einzellizenz mit zweijährigem Support	USD	149.95
Einzellizenz Supportverlängerung für ein Jahr	USD	50.00
Projektlizenz mit 30-Tage Support	USD	139.95
Projektlizenz mit einjährigem Support	USD	219.95
Projektlizenz mit zweijährigem Support	USD	299.95
Projektlizenz Supportverlängerung für ein Jahr	USD	100.00

Tabelle 1: Lizenzkosten General SQL Parser

Wie der Tabelle 1 zu entnehmen ist, würden im Rahmen dieser Arbeit für diesen Parser Kosten in Höhe von USD 139.95 entstehen, sofern kein weiterer Support benötigt wird. Es gibt eine kostenfreie Testperiode, die aber für die Anwendung nicht einsetzbar ist, da die Anwendung auch nach Ablauf der Testzeit voll einsetzbar bleiben soll. Eine kostenfreie Testversion ist für Java verfügbar[112]. Diese Version bleibt jedoch nur so lange gültig, bis der Parser in Java komplett implementiert wurde. Weiterhin ist dieser Parser als selbstständige Software lauffähig und nicht integrierbar. Aus den beiden genannten Gründen kann er für die Anwendung nicht eingesetzt werden.

[110] vgl. [Gudu Software 09c]
[111] vgl. [Gudu Software 09b]
[112] Stand Juli 2009

5.1.4.2. SQL Query Parser Der SQL Query Parser[113] gehört zu dem „DTP SQL Development Tools Project“ von Eclipse[114]. Der Parser ist ein generierter Parser und benötigt zur Erstellung den Lexer Parser Generator (LPG)[115]. Anhand eines Satzes von grammatikalischen Regeln kann der Parser in Java erstellt werden[116].

Der Parser nimmt DML- und DQL-Anweisungen entgegen und parst diese Informationen. Er generiert daraus ein sogenanntes „SQL Query Model“, falls die SQL-Anweisung syntaktisch korrekt ist. Der Parser kann verschiedene datenbankspezifische Dialekte bearbeiten, die dem Parser beim Erstellen in Form von grammatikalischen Regeln mit übergeben werden. Somit ist dieser Parser unabhängig vonm Datenbankhersteller. Der Parser kann überprüfen, ob eine Anweisung syntaktisch und semantisch korrekt ist.

Ohne Probleme kann der Parser Unterabfragen erkennen und gibt die entsprechenden Spalten korrekt zurück. Die SQL-Anweisung wird formatiert ausgegeben. Der Parser kann für alle Spalten den entsprechenden Typ zurückgeben. Diese Informationen werden allerdings an dieser Stelle nicht benötigt und sind für die Entscheidung eines Parsers unwichtig.

Der Parser kann die *FROM*-Komponente erkennen und parsen, allerdings findet keine Zuordnung zwischen Tabellen- oder Viewnamen und einem eventuell vergebenen Alias statt. Daher kann auch im weiteren Verlauf keine Zuordnung der entsprechenden Abhängigkeit erfolgen. Der Parser ist dementsprechend für die Anwendung unbrauchbar.

5.1.4.3. SQLJEP SQLJEP[117] ist eine Java API, um SQL-Anweisungen oder auch SQL ähnliche Anweisungen zu parsen und evaluieren. Unter anderem besitzt dieser Parser ein erweitertes `ResultSet` mit mehr Möglichkeiten zur Anfrageverarbeitung als das `ResultSet` aus dem Paket von `java.sql`[118]. Es werden Mengenoperationen und boolesche Ausdrücke komplett unterstützt. Hier wird aber nicht näher auf diese Funktionalität eingegangen, da sie für die Anwendung irrelevant ist.

Dieser Parser bietet keine Unterstützung in Form einer Syntaxanalyse, allerdings kann er durch seine einfache Arbeitsweise alle bekannten Datenbankdialekte erkennen und verarbeiten. Der Parser liest eine SQL-Anweisung aus und verarbeitet sie. Jedes eingelesene Wort wird als sogenanntes Token bezeichnet. Der Parser ermöglicht so die Abfrage der benutzten Schlüsselwörter, der involvierten Tabellen und Views aus der *FROM*-

[113] s. [Eclipse Foundation 09a]
[114] s. [Eclipse Foundation 09b]
[115] s. [SourceForge 09]
[116] vgl. [Eclipse Foundation 06]
[117] s. [Gaidukov 07]
[118] vgl. [Sun Microsystems 08b]

Komponente sowie eine Analyse der *WHERE*-Komponente, um negative und positive Abhängigkeiten erkennen zu können. Die Grammatik für den Parser muss allerdings noch komplett implementiert werden. Dadurch ist dieser Parser aber auch unabhängig von diversen Datenbankdialekten.

5.1.4.4. SqlParser von Zoran Milakovic Ein weiterer Parser von Zoran Milakovic[119] wurde als Open Source Java Klasse geschrieben und ist beispielsweise in der Open Source Code Search Engine von koders.com[120] zu finden. Der Parser steht unter der GNU Lesser General Public License (LGPL) und kann angepasst oder erweitert werden. Dieser Parser verfügt über keine Dokumentation, ausser den kommentierten Stellen im Quellcode.

Dieser Parser erkennt die *FROM*- und *WHERE*-Komponente und kann sehr gut mit Unterabfragen umgehen. Es findet auch eine Zuordnung von Tabellen- beziehungsweise Viewnamen und deren eventuell vorhandenem Alias statt. Der Parser hat dementsprechend die Möglichkeit, zu erkennen, welche Tabellen und Views involviert sind und welche Spalten für die Abhängigkeit herangezogen werden. Hinzu kommt, dass keine Programmausnahmen auftreten, wodurch die Anwendung auch in einem Fehlerfall stabil weiterläufen kann.

Ein großer Nachteil entsteht daraus, dass der Parser bei einer Unterabfrage zwar keine Fehlermeldung ausgibt und das Programm auch korrekt weiterläuft, allerdings werden die Tabellennamen aus der ursprünglichen *FROM*-Komponente überschrieben. Das bedeutet, dass nach einer Unterabfrage die Tabellen- und Viewnamen der Hauptabfrage nicht mehr zur Verfügung stehen und somit keine Zuordnung der Abhängigkeit erfolgen kann.

Weiterhin ermöglicht der Parser die Ausgabe der beteiligten Spalten, doch gibt es keine Möglichkeit eine Information darüber zu erhalten, wie diese Spalten involviert sind. So wird ein mögliches *MINUS* oder *NOT EXISTS* zwar erkannt, aber auch hier besteht keine Möglichkeit, diese für die Anwendung wichtige Information zu bekommen. Wegen dieser Nachteile kann der Parser nicht verwendet werden.

5.1.4.5. ZQL-Parser Der ZQL-Parser bietet als einer der wenigen Parser eine sehr gute Dokumentation[121] an. Der Parser ist kein Open Source Projekt, aber er ist kostenfrei und kann somit in jedem Projekt eingesetzt werden.

[119] s. [Milakovic 03]
[120] s. [Black Duck Software 08]
[121] vgl. [Gibello 08a]

Der ZQL-Parser[122] liefert für die Anwendung auf den ersten Blick noch gute Ergebnisse. Die ersten Tests verliefen durchaus positiv. Der Parser erkennt sehr gut alle beteiligten Spalten aus der *SELECT*-Anweisung. Die *WHERE*-Bedingung wird gut aufgeschlüsselt. Es werden alle *AND*-Bedingungen richtig verknüpft und geklammert. Sobald ein *OR* (oder) auftaucht, wird auch an dieser Stelle die Klammer richtig gesetzt. Der ZQL-Parser liefert eine extra Komponente für die Analyse der *WHERE*-Bedingung, die für die Anwendung besonders wichtig ist, um positive und negative Beziehungen erkennen zu können. Es wurden auch Unterabfragen in der *WHERE*-Bedingung erkannt und verarbeitet. Erste Tests für das Parsen der *FROM*-Bedingung verliefen auch sehr positiv[123].

Dieser Parser schafft es allerdings nicht, eine Unterabfrage in der *FROM*-Klausel zu verarbeiten. An dieser Stelle entsteht eine `ParseException`[124], die für diesen Parser entwickelte wurde, da der Parser an dieser Stelle einen Tabellen- oder Viewnamen erwartet. Da Unterabfragen aber möglich sind, sollte das Programm an dieser Stelle nicht abbrechen.

Eine mögliche Lösung für das Problem wäre es, an dieser Stelle die entstandene Programmausnahme abzufangen und an eine andere Methode weiterzuleiten. Dort könnte mit einem anderen oder einem eigenen Parser an dieser entsprechenden Stelle die *FROM*-Klausel weiterverarbeitet werden. Der große Nachteil ist, dass dann mindestens zwei Parser benötigt werden. Außerdem soll das Programm an dieser Stelle keine Programmausnahme werfen, da diese Unterabfragen erlaubt sind. Wegen dieser großen Schwachstelle kann der ZQL-Parser in der vorliegenden Diplomarbeit nicht eingesetzt werden.

5.1.4.6. Parsen eines Triggers in PL/SQL

Das Problem beim Parsen eines Triggers besteht darin, dass Trigger unter Oracle in PL/SQL implementiert werden. Alle bisher vorgestellten Parser können allerdings PL/SQL nicht interpretieren und beenden an dieser Stelle das Parsen mit einen Fehler. Schon ein Semikolon, welches in PL/SQL dazu benutzt wird, um einen Befehl zu beenden, stört den Parser in seinem Ablauf.

Die Suche nach einem Parser, der auch PL/SQL interpretieren kann, erweist sich als schwierig bis unmöglich. Es sind keine fertigen Lösungen für dieses Problem vorhanden. Auch kostenpflichtige Parser bieten keinerlei Unterstützung in diesem Bereich.

Als Lösung bietet sich der SQLJEP-Parser[125] an. Durch seine einfache Implementie-

[122] s. [Gibello 08b]
[123] s. Seite 122, Abschnitt 8.2.1 (8.2.1)
[124] Programmfehler, Programmausnahme
[125] s. Seite 50, Abschnitt 5.1.4.3 (5.1.4.3)

rung muss die Grammatik dieses Parsers zwar neu geschrieben werden, aber der Parser erweist sich dadurch sehr anpassungsfähig. Leider hat auch dieser Parser trotz der offenen Grammatik mit Teilen der PL/SQL-Syntax Probleme, wie zum Beispiel mit einem Semikolon. Ein Semikolon wird zum beenden eines Ausdrucks benutzt und dient daher als Steuerzeichen. Um dies zu umgehen werden durch reguläre Ausdrücke Teile des Trigger-Bodys ersetzt und erst dann geparst[126]. So kann auch hier sichergestellt werden, dass der Parser vernünftige Ergebnisse liefert.

5.1.5. Fazit

Wir haben verschiedene Parser getestet und auf ihre Funktionalität hin überprüft. Die Dokumentation der frei verfügbaren Parser ist entweder nicht vorhanden oder nur kurz und unübersichtlich gehalten. Diese Parser werden meist für eine einzelne Problemstellung entwickelt und anschließend als Open Source zur Verfügung gestellt. Die Entwicklung geschieht dabei zum größten Teil von nur einem Entwickler, worauf die schlechte Dokumentation zurück zu führen ist.

Die kommerziellen Parser hingegen sind in der Regel sehr gut dokumentiert. Es sind sowohl die Programmdokumentation vom Quellcode als auch Anwendungsbeispiele und Tutorials verfügbar. Weiterhin sind sie sehr leicht anpassbar und erweiterbar. Wir haben uns gegen ein kommerzielles Produkt entschieden, damit unsere Anwendung kostenfrei bleibt. Außerdem konnten auch die kommerziellen Produkte gegenüber den frei verfügbaren keine wesentlich besseren Ergebnisse liefern, die einen erhöhten Kostenaufwand rechtfertigen würden. Dies war ein weiterer Grund für die Entscheidung gegen einen kommerziellen Parser.

Fast alle getesteten Parser weisen an einer oder mehreren Stellen Probleme auf. Es ist wichtig für unsere Anwendung, dass ein Parser sowohl die *SELECT*-Anweisung der View wie auch den Triggerrumpf fehlerfrei parsen kann. Nur so können wir gute Ergebnisse erzielen und die Abhängigkeiten grafisch korrekt darstellen. Das Parsen ist ein Hauptbestandteil der Anwendung und muss daher schnell, effizient und problemlos funktionieren.

Letztendlich haben wir uns wegen der großen Vorteile für den SQLJEP[127] als Parser entschieden. Dieser Parser liest problemlos alle bekannten Datenbankdialekte ein und gibt die entsprechend geparsten Anweisungen zur weiteren Verarbeitung zurück. Nach kleineren Anpassungen durch reguläre Ausdrücke[128] kann SQLJEP auch für das Par-

[126] s. Seite 123, Abschnitt 8.2.2 (8.2.2)
[127] s. Seite 50, Abschnitt 5.1.4.3 (5.1.4.3)
[128] Reguläre Ausdrücke können durch syntaktische Regeln komplizierte Textersetzungen durchführen.

sen in PL/SQL eingesetzt werden. Die Grammatik und somit die Syntax der verwendeten Datenbank wird von uns implementiert. Auf Basis von dem Parser SQLJEP wird eine Eigenentwicklung mit erweiteter Funktionalität umgesetzt, um die Anforderungen zu erfüllen.

Nur so können wir sicherstellen, dass die SQL- beziehungsweise PL/SQL-Anweisungen korrekt geparst und verarbeitet werden, ohne das Programm zum Absturz zu bringen. Unsere Anwendung kann daher auch leicht für einen anderen Datenbankhersteller erweitert werden und es sind nur kleinere grammatikalische Anpassungen notwendig.

5.2. Graphenframeworks

Für die Wahl eines geeigneten Frameworks ist es zunächst erstrangig zu ermitteln, welche Eigenschaften wichtig und welche weniger wichtig sind. Dabei muss nicht nur darauf geachtet werden, welche Anforderungen derzeit an die Anwendung gestellt werden, sondern auch, welche später hinzukommen könnten. Die Aufgabenbeschreibung ergibt[129], dass die Anwendung zu einem späteren Zeitpunkt um weitere Funktionen erweitert werden könnte. Daher sollten potenzielle Funktionserweiterungen erwogen und entsprechend gewichtet bewertet werden.

Im Kontext der visuellen Graphen beschreiben die Funktionserweiterungen oft den detaillierten Umfang einer Abhängigkeit oder einer Relation zwischen zwei Objekten. So wäre es denkbar, dass auch rekursive Viewabhängigkeiten dargestellt werden sollen. Dabei muss das Framework nicht nur die Darstellung dieser zyklischen Graphen sicherstellen, sondern die darunter liegenden Algorithmen sollten bestenfalls auch eine entsprechende, zyklische Erkennung besitzen. Für die Darstellung der einzelnen Objekte ist es wichtig, dass diese sehr individuell erstellt und bearbeitet werden können. Es ist denkbar, dass zu einem späteren Zeitpunkt weitere Objekte (zum Beispiel Prozeduren und Funktionen) hinzugefügt oder weitere Zusatzinformationen an vorhandenen Objekten ergänzt werden. Das Framework sollte derartige Erweiterungen so gering wie möglich limitieren.

Die Arbeit von Tobias Kloss[130] enthält eine Auflistung möglicher Bewertungspunkte für die Bereiche Layout und Graphenframeworks. Nicht alle Punkte sind auch für diese Arbeit relevant. Eventuelle Überschneidungen werden entsprechend gekennzeichnet.

Schon in der Aufgabenbeschreibung[131] als auch in den Anforderungen[132] wurde deut-

[129] s. Seite 18, Abschnitt 3 (3)
[130] vgl. [Tobias Kloss 05], S. 27ff
[131] s. Seite 18, Abschnitt 3 (3)
[132] s. Seite 69, Abschnitt 6.1 (6.1)

lich, dass eine visuelle Darstellung in Graphen innerhalb des Programmes zu realisieren ist. Der Benutzer soll in der Lage sein, mit dem Graphen zu interagieren. Damit wird ein Schwerpunkt auf die Interaktion der Darstellung gelegt, welcher deswegen bei der Bewertung auch eine wichtige Rolle spielen muss.

Auf das seit Ende 2008 verfügbare JavaFX als Alternative zu einer reinen Javalösung wird in einem kurzem Abschnitt ebenfalls eingegangen[133].

5.2.1. Nicht-funktionale Anforderungen

Da die Anwendung in Java entwickelt wird, sollte das Graphenframework entweder auch in Java programmiert sein oder eine gute und solide Integration in bestehende Javaumgebungen unterstützen. In der Praxis zeigt sich allerdings, dass diese recht weiche Anforderung in der Umsetzung bereits daran scheitert, dass eine Integration von Nicht-Java-Software in Java-Software selten gut funktioniert.

Dies liegt oft daran, dass eine solche Brücke zwischen Java und Nicht-Java entweder mittels Plattform spezifischen Java Native Interfaces[134] (JNI) realisiert oder über externe Programme und die Kommandozeile implementiert wird. In beiden Fällen ist die Implementierung stark von anderen Modulen abhängig, erschweren einen problemlosen Umgang der Java-Anwendung und widersprechen damit dem Grundgedanken der Betriebssystemabhängigkeit von Java. Deswegen muss das Framework auch auf solche Problemsituationen hin bewertet werden.

5.2.2. Funktionale Anforderungen

Im eigentlichen Sinne wird zunächst ein Framework benötigt, welches einen Graphen, beispielsweise einen Baum mit View-Objekten, visuell darstellt. Diese Anforderungen lassen sich wiederum in Einzelpunkte aufteilen. Beispielsweise nennt Kloss[135] dafür einige Gründe, die auch hier wichtig sind: das automatische Layouten von Graphen, die Individualisierung von Knoten und Kanten, Plattformunterstützung (Java)[136], Verfügbarkeit und Aktualität. Weniger wichtig sind dabei vorhandene Algorithmen, Zugriffsverfahren und -geschwindigkeit, da die verwendeten Graphen in einem gewissen Komplexitätsrahmen

[133] s. Seite 65, Abschnitt 5.2.4.6 (5.2.4.6)
[134] s. [Sun Microsystems 06]
[135] vgl. [Tobias Kloss 05]
[136] vgl. Seite 55, Abschnitt 5.2.1 (5.2.1)

bleiben. Beim heutigen Stand der Technik[137] und im Kontext dieser Desktopanwendung sind die Zugriffsverfahren der API weniger entscheidend als die Realisierung der Darstellungstechnik selber[138]. Die dargestellten Graphen sind zwar einerseits nicht hochkomplex, andererseits muss aber jedes Objekt einzeln zeitnah gezeichnet werden.

Im Detail ergeben sich dabei eine Reihe von Anforderungen an ein Graphenframework.

Model Das eigentliche Modell (*model*) eines Graphen $G = (V, E)$[139] und auch die grafischen Repräsentanten der einzelnen Knoten und Kanten, welche die relevanten Informationen beinhalten. Das Framework muss in der Lage sein, mit den Graphenvarianten Baum, Wald und Hierarchie umzugehen[140]. Das *model* wird von der Geschäftslogik der Anwendung genutzt, um den Graphen zu erstellen und Graphenoperationen anzuwenden.

(Automatisches) Layout Das Layout ist die Komponente im Framework, welche für geeignete Darstellung, Positionierung sowie Art und Weise der Kanten sorgt. Dabei ist nicht jedes *model* mit jedem Layouter kompatibel und geeignet darstellbar. Ein Layout ist zwingend notwendig, damit der Graph auch als solcher zu erkennen ist.

(Individuelles) Layout Zusätzlich zu den automatischen Layouts sollte es auch möglich sein, einen eigenen Layouter zu implementieren und ihn einzusetzen[141].

Renderer Der Renderer ist die Komponente im Framework, welche das eigentliche Zeichnen übernimmt. Bestenfalls lässt sich der Renderer für Knoten und Kanten komplett frei individualisieren. Beispiele dafür sind: Farbgebung, Formgestaltung, Beschriftung.

Interaktionen Bei einer grafischen Visualisierung ist es naheliegend, dass der Benutzer der Anwendung mit dem Graphen interagieren kann. Dies können zum Beispiel sein: Verschieben von Objekten, kontextsensitive Popupmenüs, Veränderung des Zoomfaktors, Bildschirmfotos.

Aktualität und Verfügbarkeit Kostengünstige und frei verfügbare Softwarelösungen sind vorzuziehen. Dabei muss aber beachtet werden, ob das jeweilige Werkzeug von einem Entwicklungsteam auch aktuell gepflegt wird. Deuten Anzeichen darauf hin, dass ein Projekt eingestellt wurde oder in letzter Zeit keine Aktualisierungen erhielt, sollte dies in die Bewertung einfließen.

[137] Für diese Desktopanwendung sind auch mittelklassische Computersysteme (Stand 2009) mehr als ausreichend genug.
[138] vgl. Seite 57, Abschnitt 5.2.3 (5.2.3)
[139] $G = (V, E)$, wobei V die Menge der Knoten und E die Menge der Kanten definiert
[140] vgl. Seite 18, Abschnitt 3 (3)
[141] vgl. Seite 108, Abschnitt 6.4.6.5 (6.4.6.5)

Algorithmen Vorhandene Algorithmen sind ein klassisches *nice to have*-Kriterium; selbst wenn derzeit keine Notwendigkeit ersichtlich ist, dass allgemeine oder spezielle Algorithmen aus der Graphentheorie benötigt werden, so bereichern sie dennoch das Framerwork für spätere Zwecke. Sie sind aber nicht zwingend erforderlich und lassen sich jederzeit mit entsprechendem Aufwand selber entwickeln.

5.2.3. Übersicht

Es gibt sehr viele Java-Frameworks, die Graphen jeglicher Art und Weise darstellen können. Im Folgenden werden die Frameworks *Graphviz*, *JGraph* (mit *JGraph X* und *JGraph Layout Pro*), *JUNG* und *prefuse* vorgestellt. Fast alle sind komplett kostenfrei verfügbar und wurden unter einer Open Source Lizenz veröffentlicht. Einige wenige wie etwa die Gruppe um *JGraph* werden gemischt verteilt: Der wesentliche Teil *JGraph* ist zwar kostenfrei, aber hilfreiche, spezielle Erweiterungen wie *JGraph Layout Pro* sind entweder für den kommerziellen Gebrauch kostenpflichtig oder sogar immer kostenpflichtig.

Alle rein kommerziellen Frameworks, die im Rahmen der Evaluation betrachtet wurden, finden sich im Schluss[142]. Da es keine Anforderung gibt, das Softwareprodukt ausschließlich mit kostenfreien Frameworks und Werkzeugen zu entwickeln, kann der Preis die Verfügbarkeit nicht direkt beeinflussen. Allerdings ist es sinnvoll, den Preis in Bezug zu der gelieferten Qualität und zum Leistungsumfang zu setzen.

Bis auf *Graphviz* sind alle vorgestellten Graphenframeworks in Java entwickelt. Die Sonderrolle von *Graphviz* wird später[143] erläutert.

Es folgt nun eine Übersicht und ein Vergleich einiger bekannter Frameworks. Aus Gründen der Funktionsvielfalt einiger Frameworks wird bewusst auf die Vorstellung sämtlicher Funktionen verzichtet. Nur die für diese Anwendung unmittelbar oder eventuell relevante Funktionen werden erwähnt.

5.2.4. Vergleich

Die Frameworks werden mit Hilfe der oben genannten Anforderungen qualitativ bewertet. Für die Anwendung ist es wichtig, dass die zentralen Anforderungen an die Visualisierung erfüllt werden. Außerdem ist es sinnvoll, auf spätere Erweiterungsmöglichkeiten hinzuweisen – im Wesentlichen ist dies eine indirekte Bewertung über den strukturellen Aufbau der Framework-API und der Architektur des Frameworks im Gesamteindruck.

[142] s. Seite 64, Abschnitt 5.2.4.5 (5.2.4.5)
[143] s. Seite 58, Abschnitt 5.2.4.1 (5.2.4.1)

Im Prozess der Evaluation fielen insbesondere Schwächen hinsichtlich der verfügbaren Dokumentation, Manuals und Tutorials auf. Während zwar ein Teil der Frameworks mit durchaus nennenswerten Referenzen ihre Brauchbarkeit unter Beweis stellen, ist das vorliegende Material meist sehr dünn und teilweise auch veraltet. Aus diesem Grund fließt die Qualität und auch die Quantität der Dokumentation und Anwendungshilfen in die Bewertung mit ein. Zwar ist zu berücksichtigen, dass es sich teils um freie oder kostenlose Frameworks handelt, die häufig durch ein Hobby-Projektteam gepflegt werden. Andere Projekte zeigen aber, dass dies nicht die Regel sein muss. Da man als anwendender Entwickler eines externen Frameworks auf die mitgelieferten Hilfestellungen angewiesen ist, besitzen jene eine nicht zu verachtende wichtige Rolle bei der Umsetzung des Problems.

Für die Verwendung eines Frameworks sind mindestens zwei Arten von Dokumentationen wichtig: Einerseits die Erklärung, wie das Framework strukturell aufgebaut ist und welche Möglichkeiten gegeben sind. Dies könnte beispielsweise in mehreren Tutorials, Übungen, Erklärungen oder UML-Diagrammen zur Verfügung gestellt werden. Andererseits wird eine beigelegte und ordentliche Javadoc[144] erwartet. Unter einer ordentlichen Javadoc versteht man eine API-Dokumenation, die erklärt, welche Funktion eine Klasse, eine Methode oder jeder einzelne Parameter hat.

Bei der Darstellung eines Graphen muss beachtet werden, dass es verschiedene technische Varianten der Speicherstrukturen gibt. Da ein Graph unter Umständen sehr groß werden kann, ist die Verwendung der Entwurfsmuster *Flyweight* verbunden mit der *Delegation* zu empfehlen[145].

5.2.4.1. Graphviz Das Programmpaket Graphviz[146] hat sowohl in der IT-Branche[147] als auch in wissenschaftlichen Arbeiten eine gute Reputation. Es besteht aus einer Komposition verschiedener Open Source Programme und kann vielfältige, graphen-typische Datenmengen visualisieren. Für die Zwecke dieser Anwendung ist die Teilanwendung *dot* von Relevanz, da sich diese mit der Darstellung von Bäumen und Netzwerken beschäftigt. Das gesamte Graphviz-Paket und die meisten Zusatzprogramme sind kostenfrei verfügbar.

Graphviz liest einen Graphen über eine DOT-Datei ein – dies ist eine deskriptive Sprache zur Definition eines visuellen Graphens[148]. Dabei verfügt Graphviz über eine ganze Reihe von Visualisierungsprogrammen, wovon *dot* wiederum nur eins von vielen ist.

[144] API-Dokumentationsstandard für Java
[145] s. Seite 73, Abschnitt 6.2.2 (6.2.2)
[146] s. [Graphviz 09] (Homepage)
[147] allgemein verwendete Bezeichnung für die Branche der Informationstechniken und -technologien
[148] s. [Wikipedia 09c]

Durch geeignete, mitgelieferte Layouts werden die Knoten und Kanten in der bestmöglichen Art und Weise positioniert und angeordnet. Graphviz bedient sich beispielsweise des Algorithmus von Sugiyama zur Minimierung von Kantenschnitten[149].

Da Graphviz komplett in C++ entwickelt wurde, gibt es dementsprechend nur eine umfangreiche C++-Bibliothek. Eine Java-Option ist die Verwendung der Graphviz- Kommandozeile[150]. Daraus ergeben sich aber Folgeprobleme: Zwar ist es einfach zu gestalten, einen Kommandozeilenaufruf in Java zu realisieren, aber die gesamte Anwendung ist damit aus Sicht Javas sehr starr und wenig individuell. Zudem lässt sich mit dem Graphen nicht interagieren, da über die Kommandozeile stets nur ein statisches Bild erstellt wird.

Systemtechnisch betrachtet ergibt sich ein weiteres Problem. Während die Anwendung in Java entwickelt wird und damit prinzipiell plattformunabhängig auf allen Systemen funktioniert, würde sie unmittelbar von einer vorhandenen Graphviz-Installation abhängig sein. Dabei kommt erschwerend hinzu, dass die Installationen auf den verschiedenen Betriebssystemen unterschiedlich vorgenommen werden[151]. Tatsächlich gibt es Javaprodukte in Kombination mit Graphviz wie den alternativen Javadoc-Doclet-Generator APIviz[152]. Die Entwicklung erfolgt aber mit der beabsichtigen Einschränkung, dass das Produkt ohne eine installierte Graphviz-Instanz beim Anwender nicht (wie gewünscht) funktioniert. Damit ergibt sich das Hauptargument gegen Graphviz: eine fehlende und saubere Integration in Java seitens Graphviz.

Es gibt zwei unterschiedliche Ansätze und Optionen, die eine Integration in Java ermöglichen würden; beide scheitern jedoch an den vorgegebenen Anforderungen.

Das Grappa-Framework[153] stellt eine Verbindung von Java zu Graphviz her und verfügt über eine kleine, kompakte API. Jedoch ist dieses Projekt seit Jahren eingestellt. Es wird nicht mehr weitergepflegt[154] und es fehlen viele wichtige und allgemeine Graphenfunktionalitäten. Nicht zuletzt aufgrund der alten Codebasis und fehlenden Funktionen ist ein erfolgreiches und nachhaltiges Einsetzen von Grappa nicht empfehlenswert.

Einen weiteren Ansatz zur Integration ist der DOT-Viewer idot[155]. Dieses Werkzeug ist eine Kombination von Graphviz' *dot* und einer Darstellung durch das Framework *prefuse*[156]. Dieses Tool besteht infolgedessen sowohl aus den Stärken als auch aus den

[149] vgl. [Gansner 09], S. 3
[150] s. [Graphviz-Team 04] (Übersicht)
[151] Das Programm wird unter den verschiedenen Betriebssystemen an unterschiedlichen Orten und in unterschiedlicher Weise installiert.
[152] s. [Lee 09] (Homepage)
[153] s. [Graphviz 09] (Homepage)
[154] s. [Graphviz 09] (Homepage)
[155] s. [Vinni 09] (Homepage)
[156] vgl. Seite 64, Abschnitt 5.2.4.4 (5.2.4.4)

Schwächen von Graphviz' *dot* und dem Framework *prefuse*.

Obwohl Graphviz keineswegs die Vorgaben erfüllen kann, kann es dennoch ein gutes Werkzeug zur Visualisierung seinn. Deshalb wird der Anwendung eine Zusatzoption mitgeliefert, die den aktuellen Graphen als DOT-Datei exportiert.

5.2.4.2. JGraph Ebenfalls sehr bekannt im Java-Umfeld ist das Graphenframework *JGraph*. Ähnlich wie *prefuse*[157] kann es mit einer großen Liste von Referenzen[158] glänzen. Dabei ist festzustellen, dass auch einige kommerzielle Anwendungen auf JGraph setzen. Einige bekannte Firmen und Hochschulen haben bereits JGraph eingesetzt[159].

Zwar sind in JGraph individuelle Objektrenderer möglich, wie die Referenzen auf der Homepage bestätigen. Allerdings ist die Dokumentation dürftig, denn die Javadoc[160] ist knapp gehalten und in Teilen auch unvollständig. Ebenso sind auch Tutorials schwer zu erhalten, welche nur die Handhabung des Frameworks erklären. Eine Ausnahme bietet das Tutorial von Gaudenz Alder[161]. Die Aussagekraft ist allerdings als gering einzuschätzen, weil es bereits einige Jahre alt ist und für die Vorgängerversionen mit mittlerweile veralteter API erstellt wurde.

JGraph bietet keine für diese Anwendung relevanten automatischen Layouts an, weil diese Bestandteil des kommerziellen Teils *Layout Pro* sind. Damit müssen alle notwendigen Techniken des Layoutens entweder komplett selber implementiert oder erworben werden. Die mitgelieferten Layouts können zumindestens für die Darstellung der View-Hierarchie nicht unverändert genutzt werden, da die Darstellung der Levels nicht korrekt ist.

Im Gegensatz zu den anderen hier vorgestellten Frameworks besitzt die Datenstruktur bei JGraph ein ganz anderes Muster. Bei JGraph werden alle Knoten- und Kantenobjekte als Zellen im Graphenmodell angesprochen. Dieser alternative Ansatz verkompliziert die Verwendung der API für die Zwecke dieser Anwendung in unnötiger Weise.

JGraph X ist vom gleichen Hersteller und der indirekte Nachfolger von JGraph. Die Erfahrungen, die bei der Entwicklung von JGraph über die Jahre gemacht wurden, flossen bei der Neuentwicklung von JGraph X direkt ein. Dabei benutzt das Framework weite Teile eines weiteren externen Frameworks[162] des gleichen Herstellers. Prinzipiell scheint

[157] s. Seite 64, Abschnitt 5.2.4.4 (5.2.4.4)
[158] s. [JGraph 09c] (Screenshots)
[159] s. [JGraph 09a] (Kunden)
[160] s. [Alder 08]
[161] s. [Alder 03]
[162] s. [JGraph 09d]

JGraph X bzw. mxGraph ein gutes Framework zu sein, es fehlen jedoch jegliche Hilfestellungen. Mit erheblichen Aufwand gelang die Erstellung eines eigenen, individuellen Renderers. Das entstandene Produkt wurde jedoch den Anforderungswünschen nicht gerecht, da die entsprechenden Informationen nicht ausreichend genug waren[163].

Ein besonderer Nachteil ist, dass JGraph für jedes Objekt in einem Graphen ein entsprechendes Pendant für die Darstellung in Swing erzeugt. Da alle Objekte vorgehalten werden müssen, bedeutet dies einen zusätzlichen Speicher- und Verwaltungsaufwand. Im Vergleich dazu löst das Framework JUNG[164] dieses Problem wesentlich eleganter (mehr dazu im nächsten Abschnitt). Bei der Begutachtung von JGraph X wurde positiv festgestellt, dass dieses Problem (siehe JGraph) vom Hersteller mittlerweile berücksichtigt wurde.

Des Weiteren zeigt der mit dem JGraph-Framework entwickelte *JGraphpad Pro Diagram Editor* eindrucksvoll, welche Funktionalitäten das Framework zu bieten hat. Dabei ist dieser Diagrammeditor bereits für sich alleine ein sehr umfangreiches Produkt. Er wird wie JGraph selber unter einer Open Source Lizenz zur Verfügung gestellt. Dabei begnügt sich der Funktionsumfang nicht nur mit einfachen Formen und Kanten, sondern er enthält diverse unterschiedliche Kantenendvarianten (beispielsweise für *Entity-Relationship-Diagramme*) und Knotenformen. Der Screenshot in Abbildung A.4 präsentiert nur einen kleinen Teil der Fähigkeiten. Leider zeigt auch die separat erhältliche Javadoc[165], dass sie einen nicht ausreichenden Dokumentationsumfang besitzt. Daher muss auch für eine spätere Verwendung die gesamte Struktur in entsprechender Feinarbeit inspiziert werden. Gleichsam ist der Editor in der Form funktional komplett überdimensioniert.

Die preisliche Struktur und Lizenzierungen von JGraph sehen wie folgt aus:

JGraph und JGraphPad sind kostenfrei nutzbar und werden unter der Less General Public License (LGPL) vertrieben.

Layout Pro kostet in einer Einzellizenz 389 EUR. Bei Mehrbedarf gibt es entsprechende Mengenrabatte[166].

Allgemeiner Support wird mit ab 189 EUR im Jahr berechnet. Daneben gibt es einen (kostenlosen) Community-Support in den üblichen Formen Forum, Mailing Liste, Tracker sowie einem Fragenkatalog.

Die kommerziellen Produkte sind für einen begrenzten Zeitraum frei verfügbar und testbar.

[163] s. Seite 66, Abschnitt 5.2.5 (5.2.5)
[164] s. Seite 62, Abschnitt 5.2.4.3 (5.2.4.3)
[165] s. [JGraph 05]
[166] s. [JGraph 09b]

5.2.4.3. JUNG Das Framework JUNG ist ein seit 2003 existierendes Java-Framework zum Modellieren, Anwenden und Visualisieren von Graphen. Man kann mit diesem Framework sowohl objektbasierte Graphen erstellen als auch diverse mitgelieferte Graphenalgorithmen nutzen. Speziell für diesen Kontext interessant ist die Möglichkeit, Graphen auch zu visualisieren. Dabei bedient sich das Framework einiger anerkannter Entwicklungsmuster wie dem Strategiemuster um eine Vielzahl an Detailerweiterungen zu erlauben. Damit sind vielfältige Möglichkeiten gegeben, die Visualisierung ganz individuell zu gestalten.

Die aktuelle Version 2.0 wurde im April 2009 veröffentlicht[167] – im Weiteren wird die Abkürzung JUNG sowohl für Version 2.0 als auch die Vorgängerversionen 1.x genutzt, wobei sich in der Regel auf die neuste Version bezogen wird. Etwaige Ausnahmen werden gekennzeichnet.

Das JUNG-Framework wird unter der Open Source Lizenz Berkeley Software Distribution (BSD) distributiert.

Das Framework ist komplett objektorientiert in Java entwickelt und verwendet Generics[168] als Platzhaltertypen der Geschäftsobjekte. Durch die nahtlose Integration in Swing ist eine Realisierung eines Graphen in eine grafische Oberfläche mit geringem Aufwand möglich.

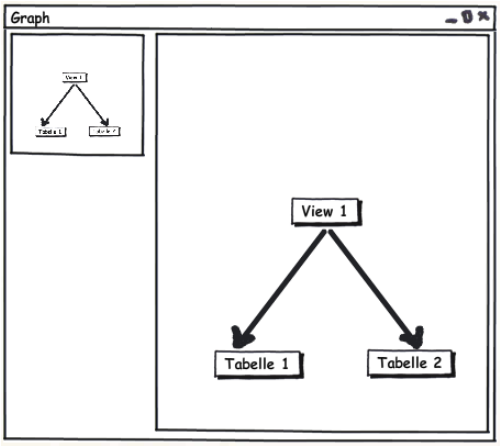


Abbildung 5.1: Graphendarstellung mit Satellitenfenster

Außerdem verfügt das Framework über einige fertige, automatische Layouts (zum Beispiel *TreeLayout* und *Circlelayout*) und mitgelieferten Standardalgorithmen aus der Graphentheorie. Zusätzlich werden einige fertige Module mitgeliefert, mit deren Hilfe sich

[167] vgl. [JUNG Dev. Team 09a] (Homepage)

[168] Java Generics sind ausschließliche Annotationen für den Java-Compiler und dienen der Typsicherheit von Objekten.

das Aussehen von Knoten oder Kanten beeinflussen lässt. Zu den Modulen gehören auch spezielle Mausmodule, um Zoomen über Scrollrad oder andere Aktionen zu ermöglichen.

Im Gegensatz zu JGraph löst JUNG das Problem von vielen Swing-Objekten elegant mit einer einzigen Rendererkomponente[169]. Gleichzeitig aber erlaubt es weiterhin eine Vielzahl von Erweiterungen durch das Strategiemuster[170].

Erwähnenswert, und *nice to have*, ist die Funktion „Satellitenfenster“: JUNG erlaubt das Kopieren und gleichzeitige Darstellen eines visualisierten Graphen. Damit ist es möglich, einen kleinen Satelliten zu erzeugen, welcher für eine bessere Übersicht bei großen Diagrammen sorgt. Eine vergleichbare Funktionalität findet man bei Grafikprogrammen. Natürlich arbeitet diese Funktion perfekt mit dem ebenfalls verfügbaren Zoom-Feature zusammen.

Im Anhang präsentieren einige mitgelieferte Beispiele die Möglichkeiten des Frameworks[171].

Die Dokumentation zu JUNG ist äußerst bescheiden. Das gesamte Frameworksystem ist zwar sehr modular aufgebaut und lässt sich mit eigenen Erweiterungen kombinieren. Die dadurch entstandene Komplexität wird jedoch in keiner Dokumentation und keinem Tutorial gerecht. Eine Javadoc ist zwar verfügbar[172], ist jedoch auch nach zwei Jahren Entwicklung nicht fortschrittlich ausgebaut worden. Teilweise existieren Lücken oder es fehlen komplette Teile der API-Dokumentation.

Das JUNG-Entwicklerteam hat erst im April diesen Jahres die neue Version JUNG 2.0 veröffentlicht[173]. Bereits seit zwei Jahren ist eine Vorabtestversion verfügbar, dennoch existiert bis heute nur ein kompaktes Tutorial[174], welches weder auf die Struktur des Frameworks, noch auf die eigentliche Verwendung im Detail eingeht – hierbei insbesondere im Kontext der Visualisierung.

Während die Verwendung der grundsätzlichen Graphen-API bei entsprechenden Vorkenntnissen überschaubar und verständlich ist, ist die Verwendung der Visualisierungsklassen schwer zugänglich. Auch der technische Bericht über das Framework[175] wurde das letzte Mal im August 2006 aktualisiert und bietet nur einen mäßigen Überblick über die Grundfunktionen.

[169] vgl. Seite 78, Abschnitt 6.2.2.5 (6.2.2.5)
[170] vgl. Seite 80, Abschnitt 6.2.2.7 (6.2.2.7)
[171] s. Seite 154, Abschnitt A (A)
[172] s. [JUNG Dev. Team 09d] (Javadoc)
[173] s. [JUNG Dev. Team 09a] (Homepage)
[174] s. [JUNG Dev. Team 09b] (Tutorial)
[175] s. [O'Madadhain 06]

Mehr Details zur visuellen Darstellung bietet beschränkt die ebenfalls knapp gehaltene Übersicht[176], die in einer sehr kompakten Art und Weise die verschiedenen Plugins für *JUNG 1.7* erklärt.

Als Grundlage für die *best practice* können die mitgelieferten Beispiele dienen. Jedoch ersetzen diese in keiner Weise eine vernünftige Dokumentation.

5.2.4.4. prefuse Das *prefuse Visualization Toolkit*[177] ist ebenfalls ein Java-Framework und besitzt einen Besonderheitscharakter, da es neben Java auch Visualisierungsmöglichkeiten in Flash und ActionScript ermöglicht. Nach den Referenzen[178] wird es bei äußerst interaktiven und großen Graphen eingesetzt.

Das prefuse-Framework wird unter der Open Source Lizenz Berkeley Software Distribution (BSD) kostenfrei und offen zur Verfügung gestellt.

Prefuse bietet nach Bildern der Homepage viel Potenzial: Die Graphen sind äußerst vielfältig, sind optisch ansprechend und lassen sich kreativ erweitern.

Mitgeliefert werden eine Reihe von automatischen Layoutern wie etwa der Fruchterman-Reingold-Algorithmus zur Minimierung der Kantenschnitte ästhetischer Graphen. Außerdem gibt es bereits fertige Java-Swing-Komponenten, die in die bestehende Anwendung eingebunden werden können. Dazu gehören neben Graphenfenstern auch Kontrollelemente für Tastatur- und Maussteuerung.

Leider ist auch prefuse seit über zwei Jahren in einer Entwicklungsphase. Es sind bis jetzt weder aussagekräftige Tutorials oder Erklärungen zur Verwendung noch eine stabile und aktuelle Version[179] vorhanden. Zwar ist die Javadoc zu prefuse in Teilen relativ umfangreich und solide formuliert, aber die vorhandenen Materialien übermitteln nur das Grundverständnis für die allgemeine API. Die Verwendung der Visualisierungsklassen wird nicht weiter vertieft oder ist in wenigen Beispielen ohne weitere Erklärungen benutzt.

5.2.4.5. Andere Frameworks Während der Recherche wurden auch noch weitere Frameworks oder Ergänzungen untersucht, die zum Teil qualitativ nicht schlecht sind. Sie gelangten nicht in eine engere Auswahl, weil sie entweder sehr hohe Lizenzierungskosten verursachen, kein echtes Graphenframework sind oder bereits indirekt betrachtet wurden.

[176] s. [Fisher 05]
[177] s. [prefuse 09] (Homepage)
[178] s. [prefuse 07] (Galerie)
[179] Bis zu diesem Zeitpunkt existiert nur ein Beta-Release von 2007.

yFiles [180] ist ein unter anderem in Java erhältliches Framework[181], welches durchaus optisch ansprechende Graphen durch Renderer und automatische Layouter erstellt. Das Framework ist sowohl für den kommerziellen als auch für den nicht-kommerziellen Gebrauch kostenpflichtig. Auf eine weitere Bewertung wie etwa die mögliche Erweiterung wurde deshalb verzichtet. Die Preisliste[182] zeichnet eine Einzelentwicklerlizenz (Basis) für 1800 EUR aus; mit den entsprechenden Layoutkomponenten 3600 EUR. Dazu kommt die Möglichkeit, Zugriff auf weitere Komponenten sowie den Quellcode zu erhalten. Es gibt die Möglichkeit eines Rabatts von 80% für den akademischen Bedarf – also ab 360 EUR, respektive 720 EUR je Entwickler.

NetBeans Visual Library2.0 [183] ist ein Bestandteil der Entwicklungsumgebung NetBeans. Auch dieses Framework enthält eine unzureichende API, die aber durch eine umfangreiche Beispielsammlung aufgefangen wird. Allerdings zeigten sich in der ersten Phase der Evaluation deutliche Defizite gegenüber den anderen Frameworks, dabei vor allem in der Handhabung zum Erstellen und Verändern der visuellen Eigenschaften der Komponenten. Obwohl das Framework damit nicht in die enge Auswahl gelangt, lassen sich prinzipiell anspruchsvolle Graphen erstellen und darstellen, wie man der Homepage entnehmen kann. Der Hauptverwendungszweck dieses Frameworks ist dabei mehr die Visualisierung und weniger auf Graphenmodell selber. Das Framework wird unter der GPL vertrieben.

mxGraph [184] ist das Framework, welches in der neuen Version von JGraph (JGraph X) verwendet wird. Ähnlich wie JGraph bietet der Hersteller einen komfortabeln Freimodus-Editor an. Das Framework wird vom gleichen Hersteller wie JGraph entwickelt und veröffentlicht.

Weitere Frameworks, die sich auch für Bäume oder Netzwerke eignen, finden sich in der Arbeit von Tobias Kloss[185]. Im Rahmen dieser Arbeit sind sie jedoch funktional nicht relevant.

5.2.4.6. JavaFX Neben Java ist JavaFX[186] ein neues Produkt und Framework für Rich Internet Applications (RIA) der Firma Sun Microsystems und gilt als direkter Konkurrent zu Adobes Flash und Microsofts Silverlight. Seit Dezember 2008 ist es offiziell verfügbar[187].

[180] s. [yWorks 09c] (Homepage)
[181] s. [yWorks 09a] (Java-Version)
[182] s. [yWorks 09b] (Preisliste)
[183] s. [NetBeans 09]
[184] s. [JGraph 09d] (Homepage)
[185] s. [Tobias Kloss 05], S. 26ff
[186] s. [Sun Microsystems 09b] (Homepage)
[187] Die aktuelle Version ist 1.2 (Juni 2009).

JavaFX ist auf nahezu jedem Computer und Betriebssystem lauffähig, denn es baut auf der Programmiersprache Java auf und nutzt intern die riesige Java-Framework[188]. Dabei wird eine aktuelle Version der Java Releases 5 oder 6 erwartet. Obwohl der JavaFX-Code – das so genannte JavaFX Script – und auch die eigentliche Handhabung nicht direkt mit Java verwandt sind, gibt es Ähnlichkeiten, beispielsweise bei Klassennamen oder Typen. Ein großer Vorteil ist schließlich die Möglichkeit, Java- und JavaFX-Programme miteinander zu verknüpfen.

Durch die Möglichkeiten von zahlreiche Plugins wie Flash, modernen Anwendungsprofilen wie „Drag-to-Install“ und „Out-of-the-Box“ (also keine Installation notwendig und unmittelbar nach dem Download nutzbar) und nicht zuletzt wegen den technischen Möglichkeiten für visuelle Effekte, ist ein frischer und junger Konkurrent zu Flash und Silverlight entstanden. Außerdem versteht sich JavaFX als ein, programmiertechnisch gesehen, universelles Werkzeug für Produkte auf allen möglichen Plattformen – vom Computer bis zum Smartphone oder Handy[189].

Da bisher kein Graphenframework in JavaFX existiert, lässt sich dessen optimalen Einsatz in diesem Kontext heute nur schwer abschätzen. Ausgehend von den technischen Möglichkeiten[190] und einer einfachen Handhabung[191] mittels JavaFX Script könnte es aber eine durchaus lohnenswerte Alternative sein. Selbstverständlich ist es abzuwarten, wie das Potenzial von JavaFX allgemein in der Zukunft ausgeschöpft wird.

5.2.5. Fazit

Während der Untersuchung der Frameworks fiel uns auf, dass hinsichtlich verfügbarer Dokumentationen oder Hilfestellungen nahezu alle große Defizite aufweisen. Dabei muss man zwischen der API-Dokumentation unterscheiden, die der Programmierer während der Programmierung benötigt und der allgemeinen Hilfe in Form von Architektur, Tutorials oder Beispielen. Wir führen diese Probleme darauf zurück, dass viele der untersuchten Frameworks unter einer Open Source Lizenz veröffentlicht werden und oft durch ein loses und kleines Entwicklungsteam gepflegt werden. Es gibt keine klaren Releasezeiten und die Dokumentation steht oft nicht vorrangig an erster Stelle. Dies finden wir insbesondere deswegen sehr bedauernswert, weil der Zugang und mögliche Zugaben zu dem Projekt durch Dritte unnötig erschwert wird.

[188] vgl. [Sun Microsystems 09c]
[189] vgl. [Heise Developer 09]
[190] s. [Wikia 09] (Beispiele)
[191] s. [Sun Microsystems 09d] (Tutorials)

Keineswegs überraschend ist die Tatsache, dass die Dokumentation kommerzieller Produkte oft wesentlich umfangreicher ist.

In die letzendlich engere Auswahl kommen nur zwei Frameworks: prefuse und JUNG. Da die Funktionen, die für uns wichtig sind, in beiden ähnlich gut sind, fiel die Wahl schließlich aus folgenden Gründen auf JUNG in der aktuellen Version 2.0:

Aktualität Das JUNG-Framework wurde erst im April diesen Jahres in einer neuen Version veröffentlicht. Im Gegensatz dazu befindet sich prefuse auch nach zwei Jahren immer noch in einer Beta-Version mit ungewisser Wahrscheinlichkeit einer zeitnahen, finalen Version[192].

Erweiterungspotenzial Obwohl die Dokumentation zu JUNG keineswegs akzeptabel und die Javadoc in Teilen sogar qualitativ schlechter als prefuse ist, bietet JUNG mehr Freiheiten für individuelle und konfigurierbare Graphen. Die Vorteile von prefuse wie Support für Flash sind für diese Anwendung nicht von Bedeutung.

Wir schließen nicht aus, dass man mit prefuse oder JGraph(X) ebenfalls ein gutes Ergebnis erzielen könnte. Es ist jedoch im Ergebnis wenig effizient, die Funktionsweisen aller einzelnen Frameworks zu ermitteln, nachzuvollziehen und anzuwenden. Teilweise ist die Dokumentation nur schwer zugänglich, veraltet oder nicht auffindbar. Selbst für das hier zu Grunde liegende Evaluieren, ob eine Verwendung sinnvoll ist, erschien die einzige sinnvolle Möglichkeit das systematische Inspizieren der Beispiele und das Nachvollziehen über den zur Verfügung stehenden Quellcode zu sein. Da alle Frameworks in großen oder allen Teilen frei verfügbar sind, ist dies in der Regel kein Problem. Da diese Möglichkeit jedoch die einzige zur Verfügung stehende Option ist, könnte das Ergebnis nicht repräsentativ sein.

Bei der Entwicklung kleinerer Prototypen konnte sich das Framework JUNG schließlich aus technischen und logischen[193] Gründen behaupten. Im Vergleichskriterium Layout schneidet JUNG gegenüber JGraph ebenfalls besser ab, weil die Layouter nur im kommerziellen Paket von JGraph enthalten sind. Außerdem zeigten Testdurchläufe, dass beide Layouter nachträglich geändert werden müssen, um die Anforderungen an die View-Hierarchie zu erfüllen[194]. Die Verwendung eines kostenpflichtigen Layouters besitzt keine erkennbaren Vorteile.

Im Anhang A[195] finden sich einige Screenshots, Prototypen und Mockups, die im Rahmen der Bewertung erstellt wurden.

[192] Stand Juli 2009
[193] Damit ist gemeint, dass die Architektur des Systems als besser bewertet wird.
[194] s. Seite 106, Abschnitt 6.4.6 (6.4.6)
[195] s. Seite 154, Abschnitt A (A)

6. Implementierung

Die im Rahmen dieser Arbeit entstehende Software ist eine komplette, eigenständige Java-Applikation, die als Desktopanwendung realisiert ist[196]. Die Wahl auf Java als Programmiersprache erfolgt aus Gründen der Flexibilität gewählt: Die Anwendung ist auf jedem Computersystem verwendbar, welches eine aktuelle Version der Java Virtual Machine (JVM) installiert hat. Als primäre Quelle für die Sprachfeatures von Java wird die offizielle Javadoc[197] verwendet.

Die Entwicklung einer Webanwendung oder eines speziellen Java-Applets[198] sind weder möglich noch sinnvoll. Eine Webanwendung würde direkten Zugriff auf die Ressourcen (also etwa den Datenbankhost) notwendig machen. Dies ist weder in internen Unternehmensnetzen noch hinter Firewalls möglich. Außerdem wäre es zwingend notwendig, dass der Anwender die Zugangsdaten einer externen Anwendung anvertraut. Auch dies ist aus Gründen der IT-Sicherheit nicht erstrebenswert. Bei der Entwicklung von Java-Applets sind zusätzlich einige Sicherheitsaspekte zu beachten, da eine derartige Software erweiterte Zugriffsrechte außerhalb der Browserumgebung benötigt. Java-Applets erlauben einen Großteil von sicherheitsrelevanten Zugriffen wie auf das lokale Dateisystem nicht[199].

Da Java-Applets auch nur im Browser ausführbar sind, sind so genannte *Java Web Start*-Anwendungen eine durchaus interessante Alternative. Zwar werden auch diese Anwendungen über den Browser gestartet, allerdings können sie sich danach wie „normale“, reale Desktop-Anwendungen verhalten. Dies ist jedoch mit mehr oder weniger aufwendigen Zertifikaten und Signierungsschritten verbunden. Da *Web Start*-Anwendungen prinzipiell normale Anwendungen sind, kann auch zu einem späteren Zeitpunkt ein solcher Distributions- und Zugangsweg erwogen werden. Einen guten Einstieg bieten die offiziellen Hilfestellungen[200] seitens Sun.

Als Mindestvoraussetzung wird die derzeit aktuelle Version Java 1.6 (JavaSE6) gewählt. Diese Version bietet einige Detailverbesserungen gegenüber zum Vorgänger und steuert dazu bei, dass die Java-Anwendungen besser und effizienter funktionieren. Dazu gehören auch Ergänzungen der mitgelieferten Bibliotheken. Dies betrifft unter anderem Aspekte in der Thread- und Synchronisationsverwaltung von Java, die im Vorgänger 1.5 noch nicht existierten.

[196] Im weiteren Verlauf wird wahlweise Anwendung oder Software gleichbedeutend benutzt.
[197] s. [Sun Microsystems 09a]
[198] Java-Applets sind kleine Java-Programme, die sich im Browser über ein Plugin ausführen lassen.
[199] Diese Technik ist auch unter dem Begriff „Sandboxing“ bekannt.
[200] s. [Sun Microsystems 08a]

Für die grafische Oberfläche wird – wie in einer grafischen Java-Anwendung in der Regel üblich – die mitgelieferte Standard-Swing-Bibliothek verwendet. Damit lassen sich sowohl Betriebssystem unabhängige Layouts und Designs erzeugen als auch letztere einfach durch eigene Module erweitern.

Swing-Applikationen in Java haben sich in der Vergangenheit bewährt und werden empfohlen (*best practice*). Ein alternatives GUI-Framework wäre das Standard Widget Toolkit[201] (SWT) von IBM, wie es beispielsweise von der Entwicklungssuite Eclipse genutzt wird. SWT gelangt jedoch nicht in die engere Auswahl, da dieses Framework im Vergleich zu Swing für die Anwendung keinerlei Vorteile bietet, dafür aber wesentlich mehr Einarbeitungszeit beansprucht hätte. Außerdem haben Anwendungen, die mittels SWT realisiert werden, auf Nicht-Windows-Plattformen häufig mit Performanzproblemen zu kämpfen. Der Grund sind so genannte schwergewichtige Komponenten. Dies ist in der englischsprachigen Dokumentation[202] und auch im Artikel „SWT“ der deutschen Wikipedia[203] nachzulesen.

In den folgenden Abschnitten wird erläutert, welche Architektur dieser Implementierung zugrunde liegt und welche Fremdkomponenten für die Implementierung genutzt werden. Soweit notwendig, werden Anpassungen an externe Komponenten erwähnt – ansonsten ist zu erwarten, dass es sich um die jeweils derzeitige aktuelle[204] Version in unveränderter Form handelt.

Ein Hinweis zu Screenshots und Abbildungen der Software: Einige davon wurden unter dem Betriebssystem Mac OS X erstellt und weicht daher in manchen GUI-Elementen von der Darstellung in Windows oder in den zahlreichen Layoutmanagern unter Linux/Unix-Systemen ab. Sowohl Java selbst als auch diese Software-Entwicklung hat den Anspruch, dass das eigentliche Layout bis zu den Formularen sowohl systemunabhängig als auch systemtypisch angezeigt wird. Sollten entscheidende Änderungen in der Darstellung der Software in den unterschiedlichen Betriebssystemen vorhanden sein, so wird dies kenntlich gemacht.

6.1. Funktionen

In der eingangs erwähnten Aufgabenbeschreibung[205] wurde bereits festgestellt, dass die funktionalen Anforderungen zunächst als Teil dieser Arbeit ermittelt werden müssen. Da-

[201] s. [Eclipse Foundation 09d]
[202] s. [Eclipse Foundation 09c]
[203] s. [Wikipedia 09f]
[204] Mai bis Juni 2009
[205] s. Seite 18, Abschnitt 3 (3)

her beschreibt dieser Abschnitt, welche zentralen Funktionen an diese Software nun gestellt werden und welche Funktionen eventuell zu einem späteren Zeitpunkt dazu kommen könnten.

Das Anwendungsfalldiagramm[206] zeigt die vier wesentlichen Funktionen der Anwendung: Neben der Liste aller eingetragenen Verbindungen auch die Anzeigen der Views, Trigger und das *Entity-Relationship-Diagramm*.

6.1.1. Technischer Hinweis

Der übliche initiale Verbindungsvorgang zu einer Datenbank im Internet kann bis zu mehreren Sekunden dauern[207] – je nach Eigenschaft der Verbindung und Zustand des Hostsystems. Zusätzlich benötigt die Software einige Sekunden bis Minuten, um das gesamte Dictionary zu laden, zu analysieren und bei Bedarf auch zu parsen. Die dabei benötigte Zeit hängt von der Größe und dem Umfang der Datenbank ab. Vorab durchgeführte Tests zeigten, dass mit mehreren Sekunden bis wenigen Minuten bereits bei Verbindungen von kleinen Datenbanken über das Internet zu rechnen ist.

Aus diesen Gründen ist es empfehlenswert, die Daten in der Anwendung persistent vorzuhalten und bei Bedarf lokal zu laden. Damit ergibt sich der Vorteil, dass die Anwendung auch einen Offlinebetrieb ermöglicht.

6.1.2. Anforderungen

Angelehnt an vorhandene Programme im Umfeld der Datenbankadministration wird ein Dialog zum Erstellen neuer und Bearbeiten vorhandener Datenbankverbindungen konstruiert. Die eingegebenen Daten müssen im Hintergrund gespeichert (*Verbindung anlegen*) und bei Bedarf wieder angezeigt und bearbeitet werden (*Verbindung bearbeiten* und *Verbindung löschen*). Außerdem ist es wünschenswert, wenn das Programm unmittelbar beim Speichern eine Plausibilitätsprüfung der Eingabedaten vornimmt – beispielsweise durch das Testen der Verbindung.

Für eine gespeicherte Datenbankverbindung wird das vorhandene *Data Dictionary* ausgelesen; dabei sind alle Tabellen, Trigger und Views von Bedeutung. Da das Dictionary auch über die Java-Schnittstelle JDBC nicht immer herstellerunabhängig arbeitet, muss festgestellt werden, ob eine herstellerspezifische Analyse implementiert werden muss.

[206] s. S. 71 Abb. 6.1 (6.1)

[207] Bei Verwendung des aktuellen JDBC6-Treibers und eines nicht unter Last stehenden Datenbankservers (*idle server*).

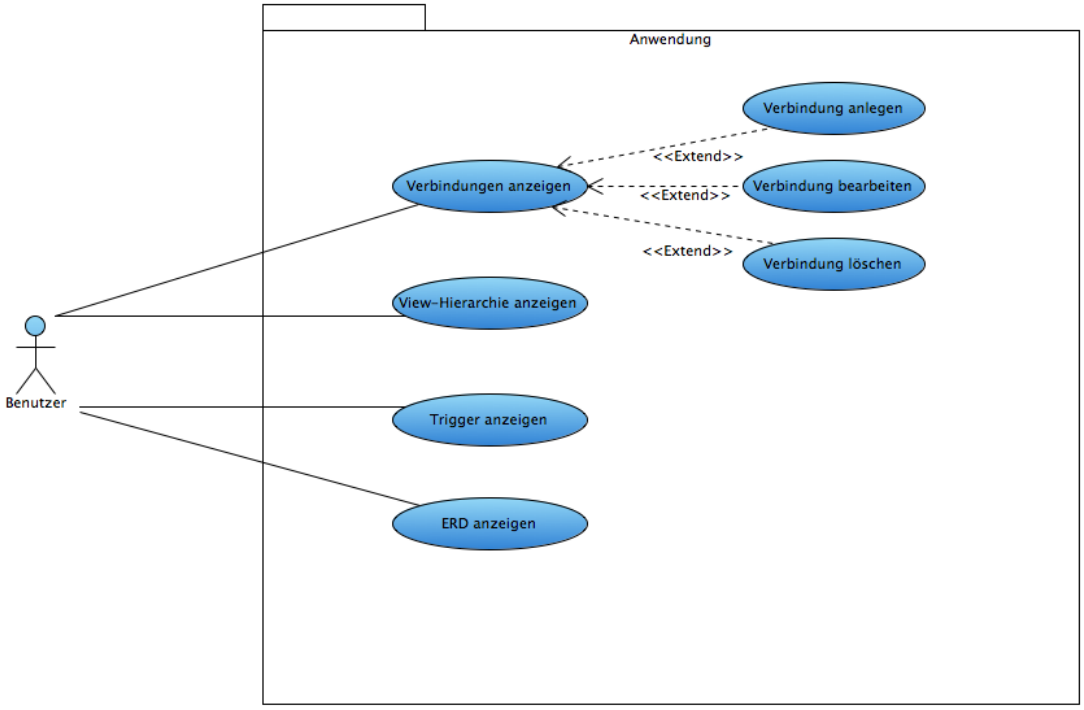


Abbildung 6.1: UML-Anwendungsfalldiagramm der gesamten Anwendung

Für Tabellen und Views sind allgemeine Daten über Spalten, Integritätsbedingungen und Indizes relevant. Außerdem muss die Definition der View (die entsprechende *SELECT*-Anfrage) ausgelesen und auf die Abhängigkeitsbeziehungen geprüft werden. Die Trigger der Datenbankverbindung besitzen einige Grundattribute, die relevant sind. Dazu gehören hauptsächlich: Tabellenname, Spalte und *Body*. Der Trigger-*Body* muss analysiert werden, um benutzte und verändernde Tabellen oder Views finden zu können. Aus den gesammelten Informationen können sich Rekursionen ableiten lassen.

Die analysierten Daten werden in einer Graphenansicht dargestellt. Dabei wird für die Darstellung der Views eine hierarchische und strukturelle Ansicht genutzt, in welcher jedes Objekt einen mathematisch festgelegten Rang besitzt[208]. Die Knotenobjekte sind Tabellen und Views, die Kanten sind die Abhängigkeiten der einzelnen Views. Jedes Objekt lässt sich interaktiv mit der Maus verschieben und die Gesamtansicht ist in der Größe veränderbar (Zoom-Funktion). Die Trigger werden in einem dynamischen Layout angezeigt, da die Gesamtstruktur nicht festgelegt ist.

Neben der grapfischen Darstellung werden auch die gesammelten Grunddaten (Spalten, Integritätsbedingungen, Indizes und Attribute) in geeigneter Weise dargestellt.

[208] s. Seite 106, Abschnitt 6.4.6 (6.4.6)

6.1.3. Spezialfunktionen

Um die Ausgabe des Graphens auch ausserhalb der Anwendung nutzen zu können, wird eine Exportfunktion eingebaut. Damit wird ermöglicht, dass die aktuelle Darstellung wahlweise als Bild oder als DOT-Datei (für eine Verwendung mit Graphviz) abgespeichert wird.

6.2. Architektur

6.2.1. Übersicht

Bei einer Softwareanwendung dieser Größe ist es elementar wichtig, dass sowohl die Spezifikation der Architektur als auch die Implementierung sauber, einheitlich und verständlich ist. Auf Basis allgemein anerkannter Muster und Programmierparadigmen lässt sich ein System entwickeln, welches nicht nur gewissenhaft und auf dem gegenwärtigen Stand der Technik implementiert ist, sondern sich auch später einfacher warten und ausbauen lässt. Dazu gehören natürlich eine gute Dokumentation, die Spezifizierung und Ausführung von Tests und eine modularisierte Entwicklung.

Darüberhinaus soll die Anwendung zu einem späteren Zeitpunkt in der Lage sein, Datenbanken anderer Hersteller zu analysieren. Ebenfalls wird in Aussicht gestellt, dass später weitere Funktionalitäten ergänzt werden könnten[209].

In diesem Abschnitt wird vor allem auf einen Teilaspekt der Architektur in der Informatik eingegangen: die so genannte *Softwarearchitektur*. Damit ist nach Balzert „eine strukturierte oder hierarchische Anordnung der Systemkomponenten sowie Beschreibung ihrer Beziehungen“[210] gemeint. Die Anwendung wird in wichtige Einzelaspekte und -komponenten zerlegt.

Wie bereits erwähnt[211], wird die Anwendung in Java entwickelt. Damit sind die Möglichkeiten einer komplett objektorientierten Entwicklung gegeben und zahlreiche fertige Entwurfsmuster können angewendet werden[212].

[209] s. Seite 143, Abschnitt 9.2 (9.2)
[210] [Balzert 01], S. 716
[211] vgl. Seite 68, Abschnitt 6 (6)
[212] Dies schließt selbstverständlich nicht andere höhere Programmiersprachen und Frameworks aus.

6.2.2. Muster

Die Verwendung anerkannter Mustern besagt die Aneignung der *best practice* für das zu behandelnde Problem. Ein Muster besteht dabei immer aus einem Problem und der dazugehörenden Problemlösung.

Die Muster in der Informatik sind äußerst zahlreich dokumentiert, mit Beispielen hinterlegt und finden sich in großer Verwendung in vielen Produkten. Nicht zuletzt mit den Veröffentlichungen der *Gang of Four*[213] haben sich zahlreiche neuartige Muster zu einer Vielzahl von Problemen entwickelt.

Dabei sind die Probleme und Lösungen vielfältiger Natur: Es gibt Muster sowohl für die Struktur einzelner Komponenten als auch für die Gesamtstruktur der Anwendung (Architekturmuster); es gibt sie sowohl für das Erstellen oder Verwalten von einzelnen Komponenten (Entwurfsmuster) als auch für spezielle Datenkomponenten. Selbst für Interaktionen gibt es fertige Lösungen: Wie funktioniert *Drag and Drop* oder wie realisiert man einen Kalender sind durchaus gefragte Muster auf der *Yahoo! Design Pattern Library*[214].

Gleichzeitig bestimmt die Wahl auch die gewünschte Intention für den Gebrauch der Komponenten. Beispielsweise ist es oft sinnvoll, dass Klassen zwar erweitert werden dürfen (offen), aber bestehende nicht verändert werden (geschlossen). Diese Vorgehensweise nennt man auch das *Open Closed Principle*. Konkret lässt sich dies sowohl mit der üblichen Generalisierung in Java als auch mit den Dekorations- oder Strukturmustern lösen[215].

Es gibt eine Reihe von üblichen Vorgehensweisen bei der Entwicklung von Softwareprojekten. Vor allem in der objektorientierten Programmierung empfiehlt sich das Verwenden des Musters *Model-View-Controller* (MVC), wenn grafische Komponenten eine Rolle spielen. Auch strukturelle und architektonische Muster wie *Singleton*, *Fabrik* oder *Builder* sind gängige Muster und finden sich in vielen Projekten wieder. Das Wiederaufnehmen solcher Problemlösungen verbessert das Verständnis der Software für Aussenstehende, da ein Muster erwartungsgemäß funktioniert.

Im folgenden werden nun einige wichtige Muster erwähnt und der Grund für ihre Benutzung erklärt.

6.2.2.1. Observer und Listener Das im deutschen als Beobachtermuster genannte Verfahren beschreibt, wie Informationen, Ereignisse und Veränderungen eines Objekts

[213] vgl. [Gamma 94]
[214] s. [Yahoo 09]
[215] vgl. [McLaughlin 07], [Freeman 06]

an andere Objekte mitgeteilt werden. Viele andere Entwurfsmuster wie das MVC basieren unter anderem darauf.

In der Regel besteht das Muster aus einem *Observable* und endlich vielen *Observern*. Das *Observable* ist demzufolge das zu beobachtende Objekt, welches Änderungen mitteilen möchte. Die *Beobachter* (*observers*) erhalten diese Mitteilungen und müssen die jeweilige Relevanz selbständig entscheiden.

Eine Variation des Musters sind die *Listener* oder auch *PropertyChangeListener*, die sich insbesondere im Kontext der Java- und Swing-Programmierung häufig wiederfinden. Diese Listener werden bei häufigen Attributänderungen von Objekten eingesetzt.

In Java wird das Observermuster mit der Klasse `java.util.Observable` und dem Interface `java.util.Observer` zur Verfügung gestellt. Abgesehen von diesem doch relativ konkreten `java.bean.PropertyChangedListeners` stellt der Java-Kern jedoch direkt keine fertigen Listener zur Verfügung. Allerdings haben viele Java-Pakete, auch Swing, interne Listener implementiert und in Verwendung.

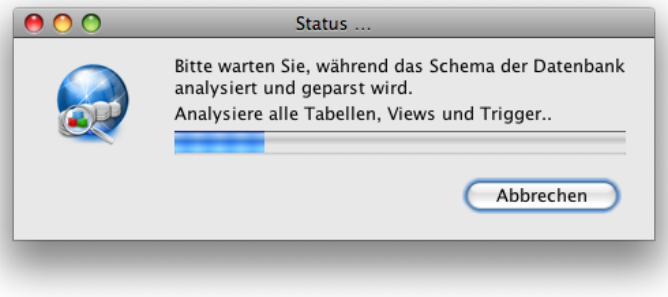


Abbildung 6.2: Screenshot des Fortschrittsbalkens mit Detailinformationen des Status

Aus Gründen der Thread/Prozess-Synchronisation, etwa für visuelle Feedbacks (siehe Abbildung 6.2), werden sowohl Observer als auch Listener in der Anwendung verwendet.

6.2.2.2. Model-View-Controller

Die Anwendung benutzt zunächst ganz klassisch aus dem Entwurfsmuster MVC[216] (siehe Abbildung[217]). Damit werden einerseits unnötige Komponentenkopplungen verhindert, andererseits aber auch die notwendigen Beziehungen wohldefiniert. Das MVC-Muster ist bei Anwendungen mit einem Graphical User Interface (GUI) das Standardvorgehen.

Konkret betreibt MVC die Aufteilung in die Teilsysteme Daten (*model*), Darstellung/Repräsentation (*view*) und Steuerung (*controller*). Nur zwischen diesen Komponenten be-

[216] s. [Robert Eckstein 07]
[217] s. S. 75 Abb. 6.3 (6.3)

stehen notwendige Verbindungen. Die Kopplung geschieht auf abstrakter Ebene und kann in Java über *Observer* und *Listener* realisiert werden. Obwohl das Konzept eine strikte Trennung vorschreibt, werden bei der Umsetzung öfters Kompromisse aus Gründen der Effizienz, Performance oder Sicherheit beschlossen. Dadurch entsteht sehr schnell eine Softwareanwendung, die zwar MVC-Komponenten verwendet, aber eine Vermischung der Aufgaben und deren Durchführung besitzt. Diese Variation von MVC ist auch unter dem Begriff *Document-View* bekannt[218]. Dennoch wird hier nur der Oberbegriff *Model-View-Controller* verwendet. Man bemüht sich darum, den Grundsatz von MVC während der Entwicklung stets beizubehalten.

Jedes dieser Teilsysteme ist je nach Bedarf wieder in verschiedene Subsysteme aufgeteilt. Die verwendeten Entwurfsmuster sind jeweils unterschiedlich und an die jeweilige Problemsituation angepasst.

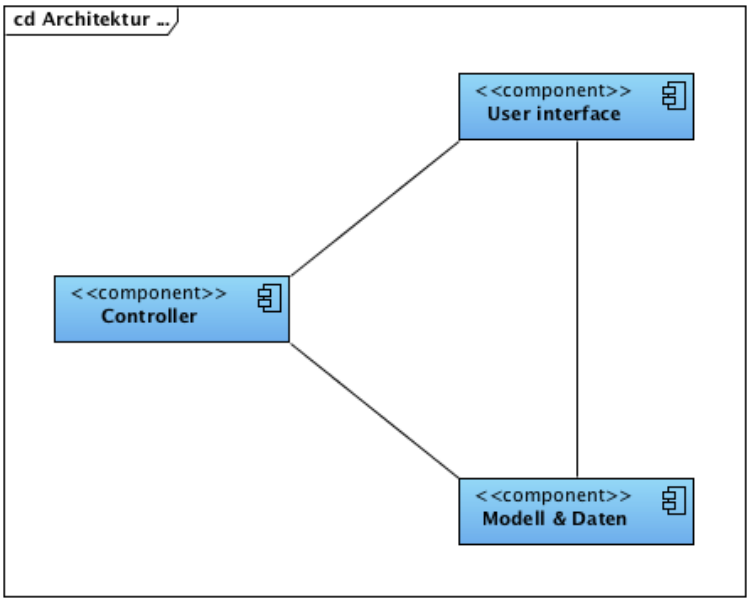


Abbildung 6.3: MVC Architektur als UML2-Diagramm

6.2.2.3. Allgemeine Erzeugermuster Die Entwurfsmuster für die Objektgenerierung werden in den Situationen angewendet, in welchen neue Objektinstanzen erstellt werden. Häufig wird mit einer Vielzahl von Klassen einer ganzen Hierarchie (beispielsweise durch Vererbung) gearbeitet. Um die konkrete Implementierung dieser konkreten Klassen vor der restlichen Anwendung zu verstecken, ist die Verwendung von Schnittstellen stets sinnvoll.

[218] vgl. [Microsoft Corp. 09a]

Neben dem *Factory*-Idiom und dem *Factory Methods*-Entwurfsmuster zum abstrakten Konstruieren von Objekten auf Basis von Schnittstellen existiert auch das *Data Access Object*-Muster (DAO) zum Verwalten persistenter Objekte. Alle drei haben gemeinsam, dass mit ihrer Hilfe Objekte erstellt oder gespeichert werden. Da DAO kein Erzeugermuster ist und hier eine besondere Rolle spielt, wird es später gesondert betrachtet[219].

Mit den verschiedenen Erzeugermustern wird die eigentliche, konkrete Objektkonstruktion nach aussen gekapselt. Dieses, auch *Verschleiern* genannte, Vorgehen sorgt für die eventuellen besonderen Initialisierungsschritte der Komponenten, die zur Verwendung notwendig sind. Außerdem wird mit dem Erzeugermuster die Arbeit gegen Schnittstellen forciert, da die konkreten Klassen nicht nach aussen gelangen. Als Anwender dieser Objekte agiert man auf Typebene auch nicht mit den konkreten Objekten, sondern nur mit einer Schnittstelle[220]. Dieses Vorgehen wird auch *Design by Contract* genannt.

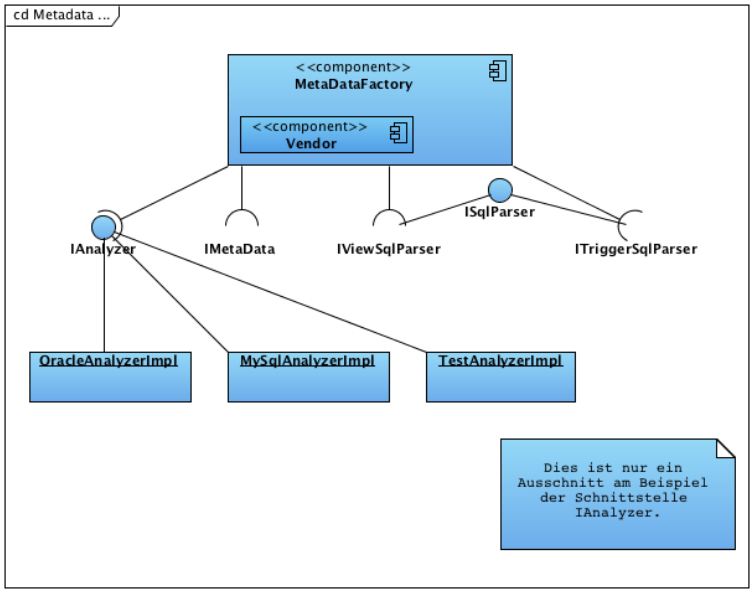


Abbildung 6.4: Die *einfache Fabrik* als UML2-Diagramm

```
1 IAnalyzer analyzer = MetaDataFactory.createAnalyzer(Vendor.ORACLE);
```

Listing 1: Fabrikmethode zum Erstellen eines Analyzerobjekts für Oracle-Datenbanken

Im Beispiel aus Abbildung 6.4 ist das Programmieridiom einer einfachen Fabrik[221] verdeutlicht. Es zeigt einen Ausschnitt des Paketes `Metadata`, welches für das Analysieren und Parsen der Objekte zuständig ist. Mit Hilfe eines Parameters (siehe Listing 1) konstruiert die Fabrik Objekte, die für bestimmte Datenbanksysteme konkret implementiert

[219] s. Seite 77, Abschnitt 6.2.2.4 (6.2.2.4)
[220] In manchen Programmiersprachen (nicht Java) sind abstrakte Klassen wie Schnittstellen zu behandeln.
[221] vgl. [Freeman 06], S. 117

wurden. Für den späteren Gebrauch von aussen ist jedoch nur wichtig, dass der *Analyzer* ein Objekt ist, welches die Schnittstelle *IAnalyzer* mit ihren vertraglichen Spezifikationen realisiert.

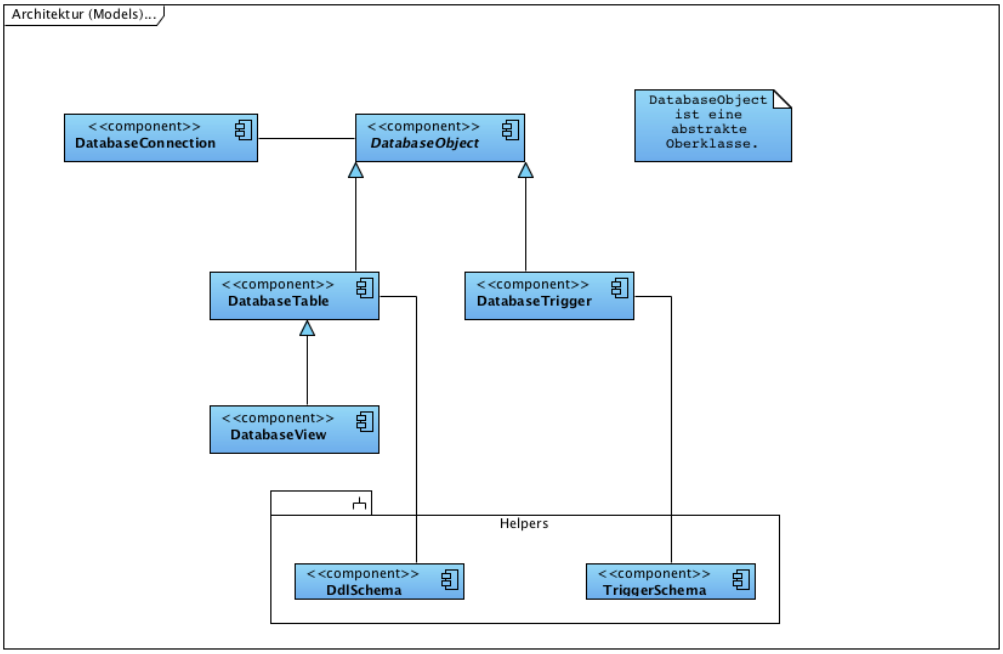


Abbildung 6.5: Die Entitäten als UML2-Diagramm

6.2.2.4. Data Access Object Das Muster *Data Access Object* (DAO) wird im Gegensatz zu den *Factory Methods* nicht unter die Erzeugermuster eingeordnet. Obwohl das Muster auch neue Objekte generieren kann, ist es vordergründig für eine universelle und nach aussen gekapselte Datenspeicherung vorgesehen. DAO wird vor allem in solchen Situationen verwendet, in denen die konkrete Art und Weise der Datenspeicherung (welche Datenbank oder ob es überhaupt eine klassische Datenbank ist) von der restlichen Anwendung verborgen werden soll. Ein DAO beinhaltet konkrete Anweisungen, wie eine persistente Entität gespeichert oder wieder geladen werden kann. Häufig werden diese Objekte auch um kreative und universelle Grundfunktionen ergänzt, beispielsweise um Methoden wie `findAll()` oder `findById()`. Andere Software-Frameworks wie das Rails-Framework[222] bieten solche Funktionalitäten direkt an. Hibernate[223] bietet solche DAO-Funktionalitäten zwar nicht direkt an, verfügt aber über alle notwendigen Voraussetzungen, um sie nachzubilden.

In Listing 2[224] wird ein solches generisches Interface für eine DAO-Klasse dargestellt.

[222] s. [o.V. 09b]
[223] s. Seite 80, Abschnitt 6.2.3.1 (6.2.3.1)
[224] s. S. 78 Listing 2 (2)

Durch die Verwendung der generischen Typen `T` für die Entitäten (Geschäftsobjekte) und `ID` für die Identifikationsobjekte (beispielsweise der `Primary key`) kann das Interface für jedes konkrete DAO-Objekt angewendet werden. Listing 3 zeigt eine beispielhafte Verwendung zum Erstellen einer Entität. In Abbildung 6.5 wird die Struktur der Entitäten der Anwendung dargestellt.

```
1 public interface IGenericDAO<T, ID extends Serializable> {
2   T findById(ID id, boolean lock);
3   T findByTitle(String title, boolean lock);
4   T findByAttributes(final boolean lock, final Criterion... criterion);
5   List<T> findAll();
6   T makePersistent(T entity);
7   void makeTransient(T entity);
8   T create();
9 }
```

Listing 2: Generisches Interface für DAO-Klassen (gekürzt)

```
1 IGenericDAO dao = HibernateDAOFactory.getConnectionDAO();
2 DatabaseConnection connection = dao.create();
```

Listing 3: Erzeugen einer neuen Entität „DatabaseConnection“ (gekürzt)

Bei der Verwendung von DAO werden häufig keine technischen Schnittstellen (Java-Interfaces) genutzt. Dennoch ist es wichtig, zwischen konkreten Klassennamen als solches und den logischen Typnamen zu unterscheiden.

6.2.2.5. Flyweight Eine weitere Problemsituation sind extrem viele Datenobjekte, häufig mit einer Fülle von Attributinformationen. Insbesondere bei Graphenframeworks werden oft eine große Menge an Knoten und Kanten dargestellt, dabei benötigt jedes Objekt einen entsprechenden Repräsentanten im Graphen. Dadurch ist die Skalierbarkeit der Anwendung für große Graphen nicht sicher gewährleistet.

Als Vorsichtsmaßnahme ist es bei Java-Anwendungen möglich, den zugewiesenen Speicher beim Start manuell zu erhöhen. Obwohl es so technisch lösbar ist, erweist es sich nicht als die eigentliche Problemlösung. Weder Computer mit weniger Speicher noch die vorhandene Ressourcenineffizienz werden berücksichtigt. An dieser Stelle setzt die Idee des *Flyweight* (deutsch: Fliegengewicht) auf.

Das Entwurfsmuster *Flyweight* löst die Probleme[225], bei denen theoretisch nicht genug Ressourcen zur Verfügung stehen und gehört zu der Gruppe der Strukturmuster. Die Verwendung von *Flyweight* ermöglicht es, dass man nur mit einer fixen Anzahl von Objekten

[225] s. [Wong 05], [Wikipedia 09d], [Wikipedia 09a]

arbeitet – idealerweise mit nur einem Objekt. Dies ist genau dann möglich, wenn sich die vielen Objekte von ihrer Datenstruktur nicht unterscheiden. Der Zustand ist in diesem Fall nur während der eigentlichen Benutzung relevant: Sowohl vorher als auch nachher ist das Objekt (also die Objektreferenz) ohne weitere Bedeutung.

Im konkreten Falle des Graphen lassen sich die gleichen Zeichenroutinen für alle Objekte des Graphen nutzen. Da als Zeichenbasis zusätzlich das entsprechende Swing-Objekt wiederverwendet werden kann, wird zusätzliche Effizienz erreicht. Dabei hat diese Struktur große Ähnlichkeiten mit der *Delegation* – dies ist nicht überraschend, weil *Flyweight* tatsächlich aus einer Komposition anderer Entwurfsmuster besteht[226].

Speziell in dem Kontext des Graphen bedeutet die Verwendung von *Flyweight*, dass nur ein Repräsentant für ein Objekt im Graphen verwendet wird (oft auch als Rendererkomponente bezeichnet). Das eigentliche Datenmodel ist unabhängig von der Graphenvisualisierung.

Das Muster Flyweight kann darüberhinaus auch mit Stellvertreterobjekten arbeiten. Dabei werden nach aussen hin nicht die eigentlichen Daten publiziert. Dadurch ist es möglich, einerseits nicht alle Objekte gleichzeitig im Speicher vorzuhalten und andererseits aber alle Objekte nach aussen verfügbar zu machen. Dieser Aspekt wird aufgrund der überschaubaren Graphenkomplexität und der infolgedessen fehlenden Notwendigkeit bei der Implementierung nicht weiter beachtet.

6.2.2.6. Strategiemuster Das *Strategiemuster* kommt dann zum Einsatz, wenn zwar die Benutzung einer Strategie erforderlich, aber die konkrete Realisierung unbekannt ist. Ein Beispiel für ein Strategiemuster sind Suchalgorithmen in einem Graphen oder Baum: Aus Sicht des Graphens ist es unerheblich, welcher Suchalgorithmus verwendet wird. Erst zur Laufzeit wird ein konkreter Suchalgorithmus wie die Tiefensuche implementiert, hinzugefügt und ausgeführt. Auch dies funktioniert über Schnittstellen und vertraglich zugesicherten Funktionalitäten. Das Muster wird bei der Darstellung der View- und Triggerobjekte im Graphen genutzt.

[226] Konkret besteht Flyweight häufig aus einer Fabrik, der Delegation und anderen Strukturmustern.

```
protected void initializeSatelliteViewRendererContext() {
  RenderContext renderContext = satelliteView.getRenderContext();

  renderContext.setVertexShapeTransformer(vertexShapeTf);
  renderContext.setEdgeDrawPaintTransformer(edgeDrawPaint);
  renderContext.setEdgeStrokeTransformer(new ConstantTransformer(new BasicStroke(2.5f)));
  renderContext.setEdgeShapeTransformer(new EdgeShape.Line());
}
```

Listing 4: Individualisierung des Renderers durch Transformatoren (gekürzt)

In Listing 4 wird ein beispielhafter Auszug für das individuelle Konfigurieren des Renderers dargestellt. Beispielsweise wird die Form (*shape*) des Knoten (*vertex*) durch den Transformator `vertexShapeTf` implementiert, damit die Knoten durch Rechtecke visualisiert werden. Hingegen wird die Form der Kanten (*edge*) nur durch einen simplen Linien zeichnenden Transformator realisiert.

6.2.2.7. Andere Muster

Auf folgende Muster sei der Vollständigkeits halber kurz eingegangen, da sie später noch erwähnt und benutzt werden.

Adapter Ein Adaptercode stellt sicher, dass eine neue Zuliefererkomponente in ein bestehendes System integriert wird. Sowohl Zulieferer als auch das System können und dürfen hierbei nicht verändert werden[227].

Iterator Der Iterator ermöglicht den universellen Zugriff auf Container von Objekten – also eine Liste oder Menge von Objekten. Die Verwendung von Iteratoren ist bei Java ein Bestandteil der Sprache und wird von nahezu allen Objektcontainern realisiert.

Singleton Ein Singleton erlaubt nur die Instanziierung eines Objektes dieser Klasse; gleichzeitig ist dieses Objekt aber auch im globalen Kontext angesiedelt. Das bedeutet einen bewussten Bruch der Kapselung. Singletons werden häufig bei zentralen Basiskomponenten verwendet – hier vor allem in den Controllern und Fabriken.

6.2.3. Externe Frameworks, APIs und Werkzeuge

6.2.3.1. Persistenz

Eine der funktionalen Anforderungen an die Anwendung betrifft die notwendige Persistenz der gesammelten und analysierten Daten. Zwangsläufig ist dafür also eine persistente Speicherung notwendig. Da die für diese Anwendung notwendigen

[227] vgl. [Freeman 06], S. 235ff

Objekte in einzelnen Geschäftsklassen abgelegt werden, ist es naheliegend, diese Objekte direkt persistent vorzuhalten. Die Wahl fällt aus folgenden Gründen auf den Object-Rational-Wrapper[228] (OR-Wrapper) *Hibernate*[229]:

Verbreitung In den letzten Jahren hat sich Hibernate als *De-Facto-Standard* für OR-Wrapper im Java-Umfeld entwickelt. Viele große Frameworks wie Springs oder Groovy & Grails unterstützen den Einsatz Hibernates durch spezielle Erweiterungen.

Datenbank Ein OR-Wrapper persistiert die Daten in einer Datenbank. Das Framework ist nahezu universell einsatzfähig und unterstützt zahlreiche Datenbanken. Da es unsinnig wäre, für diese Anwendung einen eigenen Datenbankserver einzurichten, wird als Datenbank eine lokale *HyperSQL*-Datenbank[230] verwendet. Diese Java-Datenbank ist sehr kompakt, benötigt keine weiteren Ressourcen und reicht deshalb für die genannten Zwecke aus. Da die HSQL-Datenbank mittels Hibernate unabhängig an die Software gekoppelt ist, lässt sie sich jederzeit flexibel austauschen.

Objektorientierung Hibernate unterstützt als OR-Wrapper die Verwendung von Objekten. Das bedeutet, es müssen keine Transformationen zwischen Objekten und Tupeln der Datenbank durchgeführt werden. Selbst komplexe Abfragen stellen kein Problem dar und vereinfachen die Geschäftslogik.

Unabhängigkeit Aus Sicht der Geschäftslogik vereinfacht ein OR-Wrapper die Nutzung der Datenbank, weil notwendige Beziehungen und Werte automatisch, aber unabhängig der Datenbank erstellt und aktualisiert werden. Bei der Verwendung des Entwurfsmusters *Data Access Object* lässt sich zudem auch Hibernate selbst unabhängig in die Anwendung integrieren.

Einfachheit Der OR-Wrapper ermöglicht die einfache Spezifizierung von Klassen mit der Konzeption der Plain Old Java Objects[231] (POJOs). Dabei wird in den Geschäftsklassen nur die eigentliche Datenlogik implementiert. Das führt dazu, das der umfangreiche JDBC-Code entfällt. Der Wrapper führt sowohl die Transformation der Daten in die JDBC-Anfrageparametern als auch das Extrahieren der Daten aus den JDBC-Ergebnisobjekten aus. Damit reduziert sich die Komplexität der Geschäftslogik und damit auch Fehleranfälligkeit bei JDBC-Operationen um ein Vielfaches.

Die eigentliche Funktionalität des Wrappers, vordergründig also die Persistenz, wird später mit diesen Objekten durchgeführt. Die Objekte sind damit immer passiv und

[228] In der Literatur und in anderen Quellen über Hibernate findet sich die ebenfalls gängige Abkürzung ORM (Object-Rational-Manager) wieder.
[229] s. [Red Hat Middlware 09] (Homepage)
[230] s. [The hsql Dev. Group 09] (Homepage)
[231] Ein POJO ist eine so genannte leichte Klasse, die in der Regel nur als Datencontainer dient. Daher besitzt ein POJO weder eigene Logik noch (komplexe) Abhängigkeiten zu anderen Klassen.

nie aktiv. Da das Muster DAO einen ähnlichen Ansatz verfolgt, lässt der Einsatz eines OR-Wrapper sich gut mit DAO-Erweiterungen verbinden.

```
1 AnnotationConfiguration configuration = new AnnotationConfiguration().configure();
2 SessionFactory factory = configuration.buildSessionFactory();

4 Session session = factory.openSession();
5 DatabaseConnection connection = new DatabaseConnection();
6 session.save(connection);
7 session.close();
```

Listing 5: Beispiel für Hibernate

In einer realen Anwendung werden die Arbeitsschritte der Zeilen 1 bis 4 aus Listing 5 in einer eigenen globalen Klasse ausgeführt. Dabei hat sich als *best practice* erwiesen, dass diese Klasse gebräuchlicher Weise den Namen `HibernateUtil` hat. Je nach Einsatzzweck beinhaltet diese Klasse auch das Öffnen und Schließen neuer Sessions. Eine Session ist dabei die Komponente zwischen Geschäftslogik und dem ORM, beispielsweise das Ausführen von Abfragen oder anderen Anweisungen.

Für das Ausführen von Testfällen ist es notwendig, die Konfiguration zur Laufzeit anzupassen. Da Testfälle ungern auf der produktiven Datenbank ausgeführt werden, empfiehlt sich der Einsatz der Memory-Datenbank von HyperSQL DataBase (HSQLDB). Dieser Datenbanktyp besteht nur zur Laufzeit, wird im Arbeitsspeicher angelegt und ist demzufolge sehr schnell. Hibernate erlaubt diese Konfigurationsänderungen zur Laufzeit (siehe Listing 6).

```
1 AnnotationConfiguration configuration = new AnnotationConfiguration().configure();

3 configuration.setProperty("hibernate.connection.url", "jdbc:hsqldb:mem:dbvisapptest");
4 configuration.setProperty("hbm2ddl.auto", "create-drop");
```

Listing 6: Änderung der Hibernatekonfiguration zur Laufzeit für Testfälle

Die Verwendung von Hibernate ist dabei sehr simpel und beinhaltet kein Wissen über die Datenbank. Damit ist der Code der Geschäftslogik datenbankunabhängig. Das Listing 5 zeigt den notwendigen Code, der hinter dem *Erzeugen einer neuen Entität* steckt. Das entsprechende Pendant in JDBC wäre um ein Vielfaches länger und komplexer. Das ist ein typisches Beispiel für das Paretoprinzip[232] in der Softwareentwicklung. Das Paretoprinzip besagt, dass 80% der Ergebnisse durch 20% der Gesamtarbeit erreicht werden. Umgekehrt bedeutet dies für die restlichen und schwierigeren 20% eine unverhältnismäßig hohe Belastung. Durch die Verwendung von Hibernate und im Zusammenspiel mit

[232] vgl. [Wikipedia 09e]

DAOs lässt sich der notwendige JDBC-Code sogar auf weniger als 20% reduzieren. Auch dadurch läst sich die Komplexität und die Fehleranfälligkeit der Software weiter reduzieren.

6.2.3.2. Layout In grafischen Anwendungen werden Dialogen und Formularen sehr oft zu wenig Bedeutung geschenkt. Dabei ist es aus Benutzersicht wichtig, dass Dialoge und Formulare klar und sauber strukturiert sind. In Java ist es, wie auch in anderen Bibliotheken, zuweilen nicht einfach, „gute“ Layouts zu erstellen. Erschwerend kommt hinzu, dass jeder Layoutmanager oder jedes Betriebssystem das Layouten unterschiedlich interpretiert und durchführt. Aus diesen Gründen verwendet die Anwendung das Paket *JavaBuilder*[233]. Sowohl die Homepage als auch eine separat erhältliche Online-Dokumentation[234] erleichtern die Anwendung.

Dieses Paket besteht aus aus dem Layoutmanager MiGLayout[235], einer in YAML[236] (rekursives Akronym für „YAML Ain't Markup Language“) geschrieben Konfigurationsdatei und einigen anderen Hilfsmodulen.

Der Layoutmanager MiGLayout ist der Nachfolger des JGoodies FormLayout und ist eine wohlbekannte Layoutalternative im Umfeld der Java-Entwicklung. Die offizielle Java-Bibliothek stellt im Paket `java.awt` bereits einige Implementierungen von Layoutern bereit. Dennoch stellt es sich als schwierige Aufgabe heraus, wenn es um die systemunabhängige Platzierung von Formularelementen oder die relative Anzeige in Bezug der umliegenden Komponenten geht. Diese Probleme werden mit MiGLayout gelöst.

Mit diesen Komponenten lassen sich kompakte und deklarative Layoutvorgaben für eine Vielzahl der Java Swingelemente erstellen. Zur Laufzeit werden diese Konfigurationsdateien eingelesen, der entsprechende Javacode erzeugt und in der JVM ausgeführt.

Das Formular[237] zum Anlegen einer neuen oder Bearbeiten einer vorhandenen Datenbankverbindung ist dabei ein Beispiel für die Verwendung von MigLayout und JavaBuilder. Die dazu gehörende YAML-Definition[238] ist im Anhang wieder zu finden.

[233] s. [Furmankiewicz 09a] (Homepage)
[234] s. [Furmankiewicz 09b] (Dokumentation)
[235] s. [MiG InfoCom AB 09]
[236] s. [Evans 09]
[237] s. S. 84 Abb. 6.6 (6.6)
[238] s. Anhang B, S. 180, Listing 25 (25)

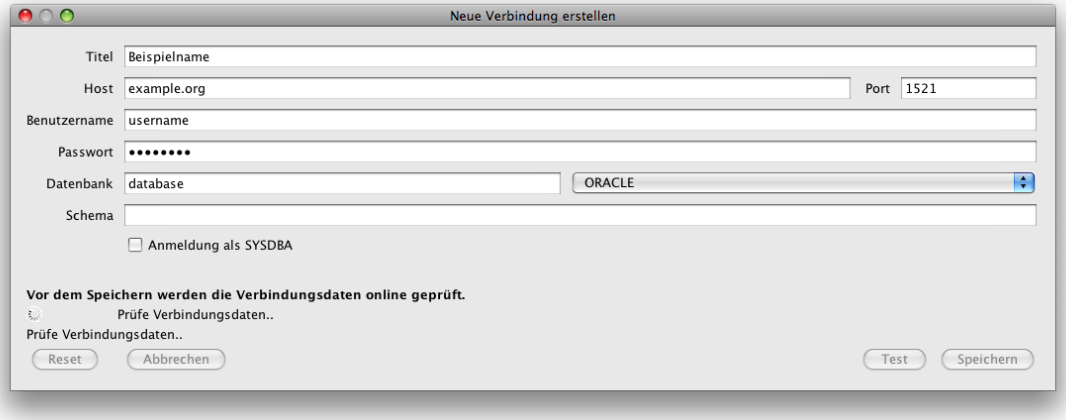


Abbildung 6.6: Formulardialog – Neue Verbindung anlegen

6.2.4. Allgemeiner Aufbau

Die Anwendung ist im globalen Namespace `de.unibonn.inf.dbdependenciesui` und in entsprechenden Unterpaketen angelegt. Im Weiteren wird dieser Präfix der Einfachheit halber weggelassen.

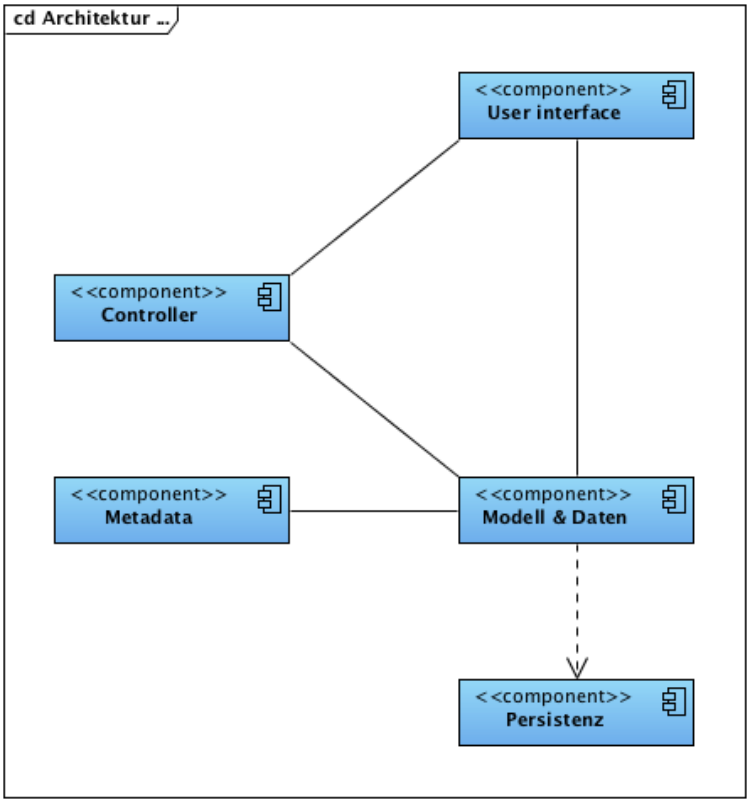


Abbildung 6.7: Die grobe Architektur als UML2-Diagramm

Im Wesentlichen existieren diese vier großen Pakete[239]:

<root> und controller An dieser Stelle sind alle initialen Startklassen, Konfigurationsklassen und der eigentliche Controller vorhanden. Letzterer ist dabei die entscheidende Komponente, die die einzelnen und zentralen Funktionalitäten mit den Geschäftsklassen durchführt oder an andere Objekte delegiert.

hibernate und hibernate.models In diesem Zusammenhang gehöhren dazu die speziellen Klassen von Hibeernate, wie die *HibernateUtil*, als auch die eigenen DAO-Klassen. Außerdem sind hier die konkreten Geschäftsklassen als POJOs[240] definiert. Auch alle notwendigen Hilfsklassen wie `DdlSchema` sind hier hinterlegt.

metadata Eine weitere Kernfunktionalität der Anwendung, das Analysieren und Parsen der Views und Trigger, liegt in diesem Paket. Erweiterungen für andere Datenbanktreiber gehören in das Unterpaket `metadata.impl`.

ui Alle Klassen, die im Rahmen der grafischen Darstellung eine Rolle spielen, sind in diesem Paket abgelegt. Damit stellt dieses Paket auch das umfangreichste aller genannten Pakete dar. Aus diesen Gründen ist das Paket `ui` in die funktionale Paketkomposition `views`, in die strukturellen Pakete `controller`, `factory` und `helpers` unterteilt.

Im Weiteren sind diese wichtigen und erwähnenswerten Pakete den oben genannten Paketen untergeordnet. Beide Pakete besitzen dabei selbstverständlich eine enge Bindung an das entsprechende Graphen-Framework.

graph Dieses Paket erhält alle Graphenklassen, die im Kontext ihrer Verwendung für die Views oder Trigger speziell angepasst wurden. Zusätzlich enthält dieses Paket die Umwandler, um aus den Entitäten einen konkreten Graphen zu erstellen.

ui.views In diesem Paket sind alle konkreten Visualisierungsklassen verfügbar, die zu einer Darstellung in Java notwendig sind.

6.2.5. Kapselung der konkreten Fremdkomponenten

Selbst mit der Bewertung und Auswahl eines geeigneten SQL-Parsers[241] und Graphenframeworks[242] sind die entsprechenden Komponenten *SQL-Parser* (als Bestandteil von *MetaData*) und *Graph* (als Bestandteil von *User Interface*) unabhängig von ihrer konkreten

[239] s. S. 84 Abb. 6.7 (6.7)
[240] s. S. 77 Abb. 6.5 (6.5)
[241] s. Seite 53, Abschnitt 5.1.5 (5.1.5)
[242] s. Seite 66, Abschnitt 5.2.5 (5.2.5)

Implementierung in die Anwendung einzubinden. Das bedeutet, dass jede Verwendung durch Schnittstellen oder Wrapper-Klassen in die Gesamtanwendung eingebunden wird und damit sicherstellt, dass es keine unnötigen Bindungen der Fremdkomponenten an die restliche Anwendung gibt.

Im konkreten Fall der SQL-Parser für Views bedeutet dies, dass nur die konkreten Klassen (beispielsweise zum Parsen der Oracle Views) den Parser ansprechen. Alle Klassen ausserhalb des entsprechenden Paketes `metadata.impl` kommunizieren ausschließlich mit der Schnittstelle `IViewSqlParser`.

Die Graphenklassen benötigen zweifelsohne jeweils eine entsprechende Geschäftslogik, weswegen auf eine Architektur über Schnittstellen und abstrakte Klassen verzichtet wird. Die Klassen sind selbstverständlich mit dem Graphen-Framework stark gekoppelt und in zwei Paketen abgelegt[243]. Damit ist sichergestellt, dass Klassen ausserhalb dieser Pakete keine Kopplung an das verwendete Graphenframework haben. Ein Wechsel des Graphenframeworks ist daher durchführbar, bedarf aber einer umfangreichen Anpassung und Umgliederung der internen Graphenimplementierung. Dieser Schwachpunkt wiegt jedoch geringer, da jedes Graphenframework von Grund auf anders aufgebaut ist und entsprechender Verbindungscode notwendig ist. Dieser Adaptercode ist die Realisierung des Adaptermusters[244]. Die Verwendung von Java-Interfaces ist nicht erforderlich und wurde daher nicht weiter ausgebaut.

6.2.6. Metadata

Bei der Implementierung der SQL-Parser[245] gibt es drei zentral wichtige Komponenten:

Analyzer ist die Komponente, die für eine vorgegebene JDBC-Datenbankverbindung eine Liste aller verfügbaren Tabellen, Trigger und Views erstellt.

Metadata bezeichnet die Komponente, die die Analyse und das Parsen weiter delegiert. Diese Komponente besitzt das Wissen, wie die Anmeldung zu der jeweils speziellen Datenbank konkret funktioniert und wie beispielsweise der Datenbank Uniform Resource Locator (Datenbank URL) aussieht.

SqlParser ist schließlich die Komponente, die dafür sorgt, dass die View-Definitionen (also *SELECT*-Queries) und Trigger-PL/SQL-Anweisungen geparst werden, um alle verwendeten Tabellen und Views zu erhalten. Deswegen wird diese Komponente in die zwei Subkomponenten `TriggerSqlParser` und `ViewSqlParser` unterteilt.

[243] s. Seite 84, Abschnitt 6.2.4 (6.2.4)
[244] s. Seite 80, Abschnitt 6.2.2.7 (6.2.2.7)
[245] s. Seite 89, Abschnitt 6.3 (6.3)

Diese Komponenten sind jeweils für jede zu unterstützende Datenbank unabdingbar und eigenständig zu implementieren. Der Grund sind die zum Teil komplett unterschiedlichen Daten und Interpretationen der jeweiligen Metadaten und Dictionary-Einträge.

Dieses Szenario lässt sich dabei mit der einfachen Fabrik lösen. Die dafür spezifizierten Schnittstellen `IAnalyzer`, `IMetaData` und `ISqlParser` (`ITriggerSqlParser` und `IViewSqlParser`) sind für die jeweiligen konkreten Komponenten die vertragliche[246] Grundlage. Diese Schnittstellen sind datenbankunabhängig.

Die eigentlichen konkreten Objekte – beispielsweise der `OracleAnalyzerImpl` für den Analyzer einer Oracle-Datenbank – werden durch die statische Fabrik `MetaDataFactory` erstellt. Der Aufzählungstyp `Vendor` ist dabei eine global verfügbare Liste aller gültigen Datenbanktreiber.

Da Teile der Komponenten allgemeine und wiederverwendbare Codebereiche beinhalten, werden die abstrakten Klassen – also entsprechend `AbstractAnalyzer`, `AbstractMetaData`, `AbstractTriggerSqlParser` und `AbstractViewSqlParser` – eingeführt. Durch diese Abstraktion und die Möglichkeiten der Erweiterbarkeit und Veränderbarkeit sind die konkreten Implementierungen der Analyse und des Parsens kompakter. Dennoch lässt diese Wahl das Vorgehen offen, falls die Notwendigkeit besteht, einzelne Schritte zu verändern. Die Hauptschritte Verbinden, Analysieren und Parsen sind unveränderbar vorgegeben.

Zu Test- und Demonstrationszwecken ist eine unvollständige MySQL-Variante implementiert. Dieser Treiber liest neben Tabellen, Views und Trigger auch View-Definitionen aus. Es werden jedoch keine Trigger-SQLs betrachtet oder berücksichtigt.

6.2.7. Serialisierte XML-Schemata

Im Rahmen der Analyse und des Parsens der unterschiedlichen Datenbankobjekte entstehen eine Vielzahl von Informationen je Objekt, die alle sinnvoll gespeichert werden müssen. Prinzipiell ist es natürlich möglich, jedes einzelne Datenpaket relational ganz klassisch in der Datenbank zu speichern. Es stellt sich aber heraus, dass in der Anwendung kein Vorteil aus dieser Datenspeicherung gezogen werden kann, zeitgleich diese aber sehr aufwendig wird. Es wird zugunsten einer partiellen, serialisierten Extensible Markup Language (XML)-Speicherung auf die relationale Abbildung der Schemata verzichtet. Die gleichen Informationen müssten in einem rein relationalen Modell in einer *Many-to-Many*-Assoziation abgelegt werden.

[246] s. Seite 75, Abschnitt 6.2.2.3 (6.2.2.3)

Am Beispiel der Schemadaten eines Viewobjektes wird deutlich, dass für jede View – egal ob Spalte, referentielle Integrität, Datenintegrität oder eine Abhängigkeit, welche über die View-Definition analysiert wurde – eine eigene Schemaabhängigkeit entsteht. Tatsächlich werden diese Informationen im Schema jedoch nicht benötigt, erhöhen aber die Komplexität des zugrunde liegenden Schemas sowie die entsprechenden Abfragen. Eine Speicherung als XML-Fragment reicht in diesem Falle aus. Das Beispiel eines View-Schemas in XML[247] zeigt, dass drei Spalten und zwei Abhängigkeiten vorhanden sind. Eine respektive, klassich relationale Speicherung bestünde in diesem Falle aus mindestens zwei weiteren Tabellen und Joins – ohne Berücksichtigung eventueller Zwischentabellen für m:n-Beziehungen.

In beiden Fällen wird das Schema mittels einem POJO implementiert. Für den Anwender des Modells ist es daher nachrangig interessant, ob das Objekt durch einen OR-Wrapper oder durch das Deserialisieren eines XMLs entstanden ist.

```
1 public String getDdlSchema() {
2   return ddlschema;
3 }

5 public DdlSchema getDdlSchemaObject() throws IllegalArgumentException {
6   return new DdlSchema(ddlschema);
7 }
```

Listing 7: Laden eines Tabellenschemas (gekürzt)

Die Speicherstruktur XML wurde deswegen gewählt, weil sie per Definition universell und damit flexibel anpassungsfähig ist. Das *Data Binding*, also das Verbinden des Objektes mit einer zugehörigen XML-Struktur, ist bei POJOs wie hier sehr einfach lösbar.

So sind Teile der Metadaten je Objekt in der Datenbank als XML-String[248] abgelegt und werden mittels Lese- und Schreibmethoden im entsprechenden Objekt gekapselt verfügbar gemacht. So ist etwa die Klasse `DdlSchema` ein POJO für das Schema einer Tabelle oder einer View – analog dazu ist die Klasse `TriggerSchema` das Schema für einen Trigger (siehe Listing 7). Zu einem Schema gehören in diesem Kontext neben den offensichtlichen Informationen wie Spaltennamen und -typen auch die vorhandenen Primär- und Fremdschlüssel sowie Unique-Bedingungen. Außerdem beinhaltet das Schema alle ermittelten Abhängigkeiten oder Verbindungen zu anderen Objekten.

Technisch gesehen beinhalten beide oben genannten *model*-Klassen die notwendigen

[247] s. Anhang B, S. 179, Listing 24 (24)

[248] Die Datenbank HSQL unterstützt keine XML-Typen wie Oracles *XMLTYPE*. Hibernate hat zwar ebenfalls keine direkte Unterstützung, aber es lassen sich entsprechende Erweiterungen selber entwickeln (s. [Batra 09]).

Instruktionen für das Marshalling und Unmarshalling. Dabei bedeutet Marshalling das Serialisieren der Daten, also das Überführen der Daten des (Java-) Objektes in einen XML-Baum und damit in einen XML-String. Das Unmarshalling ist dabei der dazugehörige Rückweg, also das Deserialisieren eines XML-Baumes in ein neues (Java-) Objekt. Des Weiteren beinhalten beide Klassen die bei POJOs üblichen *Getter*- und *Setter*-Methoden[249]. Um beispielsweise eine Liste der Primärschlüssel einer Tabelle zu erhalten, bedarf es ausschließlich der kompakten Quellcodezeile aus Abbildung 8.

```
1 List<PrimaryKey> primaryKeys = table.getSchemaObject().getPrimaryKeys();
```

Listing 8: Laden der Primärschlüssel einer Tabelle (Beispiel)

Da beide Transformationsvorgänge – das Marshalling und Unmarshalling – bei häufigem Gebrauch weder performant noch effizent sind, empfiehlt sich das Verwenden eines internen Schattenspeichers. Im Anhang finden sich zwei Beispiele für das Lesen[250] und Schreiben[251] des Caches wieder. Solange der Zustand des XML-Strings aus der Datenbank nicht verändert wird, ist das zugehörige *DdlSchema*-Objekt gültig[252]. Die Verwendung des Objekts wird davon nicht beeinflusst.

Seit Java 5 ist das Paket JAXB im Java-Framework enthalten. Es kann als Alternative zu der selbstentwickelten Serialisierung in Erwägung genommen werden, falls die XML-Daten in Zukunft wesentlich komplexer gestaltet werden. Im Vergleich zu vorhandenen JAXB oder Apache XML Beans ist die individuelle Lösung vergleichsweise kompakter und auf die Verwendung zugeschnitten. Eine gute Übersicht aktueller Frameworks und die möglichen Gründe zum Erwägen eines solchen finden sich im Onlinebereich von „Heise Developer“[253].

6.3. Analyse und Parsen

Dieser analytische Teil ist von zentraler Bedeutung für diese Arbeit. Er befasst sich mit der Analyse und dem Auslesen von vorhandenen Datenbankobjekten, dem Parsen der `SELECT`-Anweisungen der Views einschließlich dem Erkennen von Abhängigkeiten und dem Bestimmen der teilweise komplexen Triggeraktivitäten untereinander.

[249] Mit den Begriffen Getter und Setter bezeichnet man die aus der objektorientierten Programmierung stammenden Methodenzugriffe auf eine Variable, beispielsweise *getName()* und *setName()* auf die Variable *name*.
[250] s. Anhang B, S. 179, Listing 22 (22)
[251] s. Anhang B, S. 179, Listing 23 (23)
[252] s. S. 88 Listing 7 (7)
[253] s. [Frotscher 09]

Dabei werden die Aufgaben auf verschiedene Instanzen einzelner Komponenten verteilt. Entsprechende Interfaces, die die einzelnen Aufgaben beschreiben und somit die Methoden exportieren, werden hierfür angelegt. Durch diese universellen Schnittstellen sind die konkreten Klassen implementierungsunabhängig. Die abstrakten Klassen, die diese Interfaces importieren, erledigen die Aufgaben, die für die unterschiedlichen Datenbanksysteme gleich sind[254].

Die Analyse ist zuständig für das Auslesen der vorhandenen Datenbankobjekte und befragt die verschiedenen Data Dictionarys[255]. Durch diese Analyse werden die Integritätsbedingungen wie Primär- oder Fremdschlüssel entdeckt, aber auch Informationen über die Trigger wie Modus, Event, Aktionstyp und Bedingungen werden analysiert und gespeichert.

Mit dem Parsen beginnt der eigentliche Teil der Arbeit der Metadaten-Komponente. Anhand der ermittelten Tabellen, Views und Trigger werden die einzelnen Abhängigkeiten analysiert und aufgeschlüsselt[256]. Durch diese Auswertungen können die gespeicherten Daten im späteren Verlauf grafisch dargestellt werden. Dieser Teil muss also vor der eigentlichen Visualisierung stattfinden.

6.3.1. Aufbau

Das Analysieren und Parsen der benötigten Metadaten eines Datenbankschemas erfolgt in verschiedenen Schritten. Dabei werden die einzelnen Funktionen in verschiedene Klassen aufgeteilt, die die Anforderungen an diese Komponente implementieren.

Für eine gute Übersicht der Klassen werden verschiedene Pakete angelegt. In der nachstehenden Tabelle 2 erfolgt eine Beschreibung des Aufbaus dieses Paketes:

`metadata`	Interfaces zur Implementierung der benötigten Klassen und die Klasse `MetaDataFactory`
`metadata.impl`	Implementierung der abstrakten Oberklassen
`metadata.impl.*`	Konkrete Klassen für die Implementierung nach dem Datenbanksystem unterschieden (beispielsweise `metadata.impl.oracle11`)

Tabelle 2: Aufbau der Struktur des Metadaten-Paketes

Die Klasse `MetaDataFactory` verwaltet die einzelnen Datenbanksysteme und dient dabei als Fabrik[257]. Anhand der ausgewählten Verbindungseinstellung beim Anlegen einer

[254] s. Seite 90, Abschnitt 6.3.1 (6.3.1)
[255] s. Seite 93, Abschnitt 6.3.4 (6.3.4)
[256] s. Seite 97, Abschnitt 6.3.5 (6.3.5)
[257] s. Seite 91, Abschnitt 6.3.2 (6.3.2)

Verbindung wird über eine Enumeration[258] der Datenbankhersteller festgelegt. Die einzelnen Anforderungen erfüllen dabei die konkreten Instanzen, die anhand dieser Enumeration erstellt werden.

Durch die Implementierung abstrakter Klassen kann die Anwendung um weitere Datenbanksysteme erweitert werden. Dabei ist es ausreichend, die entsprechenden Implementierungen für das neue Datenbanksystem von der abstrakten Klasse abzuleiten. Die Funktionen, die sich auch bei verschiedenen Datenbanksystemen nicht unterscheiden, werden bereits hier vollständig implementiert wie beispielsweise das Verwalten der Verbindung.

Durch diese Art der Kapselung von Komponenten sind nur wenige Schritte notwendig, um weitere Datenbanksysteme einzubinden. Wird der volle Funktionsumfang von *Java Database Connectivity* (JDBC) unterstützt, ist die Integration leicht umzusetzen. Bei eventuell nötigen Anpassungen bedarf die Integration einen höheren Aufwand. Auch durch die unterschiedliche Art der Triggerimplementierung (Procedural Language/SQL für Trigger ist Oracle spezifisch) kann der Aufwand einer Integration steigen.

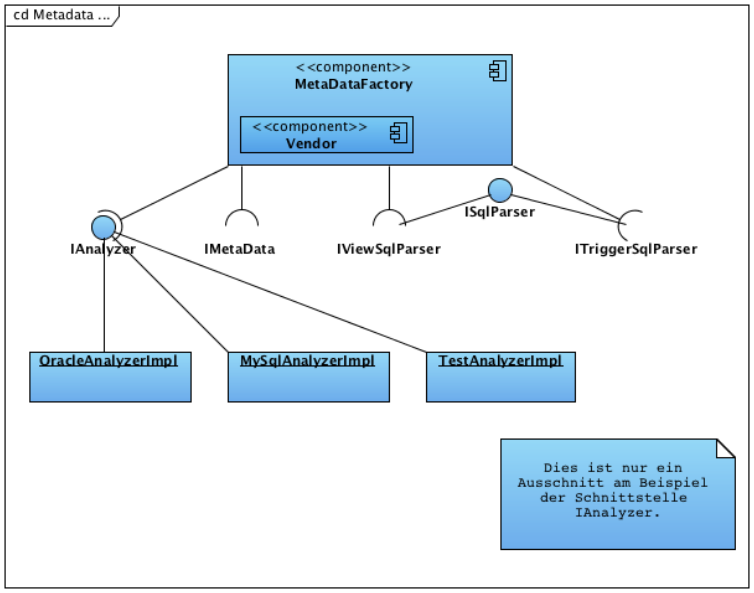


Abbildung 6.8: Die Klasse MetaDataFactory als UML-Diagramm.

6.3.2. Die Metadaten-Fabrik

Diese Arbeit befasst sich mit der Visualisierung der Abhängigkeiten von Datenbankobjekten. Dabei soll sie primär für eine Oracle-Datenbank implementiert werden[259]. Zusätzlich

[258] Aufzählungstyp, speziell in Java
[259] s. Seite 23, Abschnitt 3.6 (3.6)

soll die Anwendung für weitere Datenbanksysteme erweiterbar sein. Es bietet sich daher ein Vorgehen nach dem Designmuster einer Fabrik-Methode an[260].

Für dieses Vorgehen wird eine Klasse `MetaDataFactory` implementiert, die im Paket `metadata` liegt. Diese Klasse erstellt die benötigten Objekte für die Analyse und das Parsen eines Datenbankschema. Die Klasse beinhaltet die Enumeration `VENDOR`, anhand dessen das Datenbanksystem unterschieden werden kann. Um die Anwendung für weitere Datenbankhersteller erweitern zu können, müssen zunächst die erforderlichen Klassen erstellt und die einzelnen Methoden dieser Klasse einschließlich der Enumeration erweitert werden. Durch dieses Programmieridiom ist eine Erweiterung für andere Datenbanksysteme einfach umzusetzen.

Diese Klasse `MetaDataFactory` wird in diesem Projekt als eine statische Klasse angelegt[261] und besteht aus den einzelnen Methoden zum Erstellen der benötigten Objekte. Sie hat eine Methode für das Erstellen eines Metadaten-Objektes, welches eine Verbindung anlegt und diese Verbindung während der weiteren Analyse überprüft. Weiterhin besitzt diese Klasse eine Methode zum Erstellen eines Analyseobjektes mit der Aufgabe zum Auslesen der Metadaten des Datenbankschema. Diese Klasse hat noch zwei weitere Methoden, die jeweils ein Objekt für das Parsen der `SELECT`-Anweisung einer View und ein Objekt für das Parsen des Triggerrumpfes erstellen und zurückgeben.

Jede dieser Methoden bekommt die für das Datenbanksystem entsprechende Enumeration übergeben und entscheidet anhand dessen, welche datenbankspezifische Instanz erstellt werden muss. Diese Klasse dient ausschließlich als Fabrik und übernimmt nur das Erstellen dieser Objekte. Die einzelnen Aufgaben der Objekte werden in den nächsten Abschnitten erklärt und aufgeführt.

6.3.3. Die Metadaten

Die Klasse `AbstractMetaData` ist ein Hauptbestandteil des Paketes `metadata.impl`. Sie ist eine abtrakte Klasse und dient daher als Oberklasse für die konkreten Implementierungen. Die Funktionen, die für die diversen Datenbanksysteme gleich sind, wie das Erstellen der benötigten Objekte oder den Aufbau einer Verbindung, werden hier schon implementiert, während die konkreten Klassen die Methoden implementieren, die unterschiedlich sind.

[260] s. Seite 75, Abschnitt 6.2.2.3 (6.2.2.3)

[261] Es können keine Objekte erstellt werden und die Methoden können direkt über den Klassennamen aufgerufen werden.

Eine der Hauptaufgaben dieser Klasse ist der Aufbau einer Verbindung zur Datenbank. Hierfür wird ein entsprechender Datenbanktreiber aus dem *Java Database Connectivity* (JDBC)-Paket des Datenbankherstellers geladen. Durch diesen Treiber werden die Schnittstellen aus der Java-Bibliothek `java.sql` initialisiert. Der `DriverManager` aus dieser Bibliothek stellt dann die Verbindung anhand des Datenbanktreibers her[262]. Sobald eine Verbindung hergestellt wurde, wird während des Analysevorgangs das Bestehen dieser Verbindung überprüft. Bei einem Verbindungsabbruch kann an dieser Stelle eine Fehlermeldung ausgegeben werden.

```
1 ..
2 Class.forName("oracle.jdbc.OracleDriver");
3 ..
4 Connection connection = DriverManager.getConnection(url, properties);
```

Listing 9: Herstellen einer Verbindung zu einer Datenbank

Eine weitere Aufgabe, die diese Klasse übernimmt, ist das Abrufen der konkreten Instanzen für die Analyse und das Parsen. Dabei werden die entsprechenden Methoden der Klasse `MetaDataFactory`[263] aufgerufen. Die Objekte werden hier verwaltet und die weiteren Aufgaben wie das Auslesen der Tabellen, Views und Trigger werden an die entsprechenden Klassen verteilt. Eventuelle Fehlermeldungen, die durch die Analyse und das Parsen entstehen können, werden auch an diese Klasse weitergeleitet.

Die `AbstractMetaData` dient als eine Klasse zur Verwaltung der Aufgaben dieses Paketes. Sie erfragt die Listen der vorhandenen Tabellen, Views und Trigger nach einer erfolgreichen Analyse[264] und speichert sie. Diese Listen werden im weiteren Verlauf für das Parsen[265] benötigt. Weiterhin speichert diese Klasse die Anzahl der einzelnen Datenbankobjekte, um diese Informationen darstellen zu können.

6.3.4. Analyse

Die Analyse beschäftigt sich hauptsächlich mit dem Auslesen der vorhandenen Tabellen, Views und Trigger. Hier werden die Namen der einzelnen Objekte aus dem Data Dictionary sowie Spaltendefinitionen, Integritätsbedingungen (Primärschlüssel, Fremdschlüssel, Indizes oder Unique-Bedingungen), die *SELECT*-Anweisung einer View oder der Rumpf (*Body*) eines Triggers erfragt.

[262] s. S. 93 Listing 9 (9)
[263] s. Seite 91, Abschnitt 6.3.2 (6.3.2)
[264] s. Seite 93, Abschnitt 6.3.4 (6.3.4)
[265] s. Seite 97, Abschnitt 6.3.5 (6.3.5)

Das Interface `DatabaseMetaData` aus dem Paket `java.sql` bietet hierfür einige Methoden an. Eine konkrete Implementierung dieses Interfaces wird aus dem entsprechenden *JDBC*-Paket des Datenbankherstellers geladen. Über dieses Objekt können die meisten, wesentlichen Teile der Analyse erfragt werden. Das *JDBC*-Paket von Oracle kann für die verschiedenen Datenbankversionen eingesetzt werden.

Durch die Methode `getTables()` dieser Klasse können so die vorhandenen Tabellen und Views abgefragt werden[266]. Hier kann durch einen einfachen Methodenaufruf eine teils komplexe Abfrage an das Data Dictionary eingespart werden. Weiterhin lassen sich so auch die einzelnen Spalten einer Tabelle oder View erfragen. Für die Tabellen werden ebenfalls Methoden zur Abfrage der Primär- und Fremdschlüssel in dem Interface `DatabaseMetaData` zur Verfügung gestellt.

```
1  -- Gibt die vorhandenen Tabellen des erfragten Schemas zurück
2  -- (Katalog, Schema, regulärer Ausdruck als Suchbegriff, Objekttyp)
3  dbmd.getTables(null, null, "%", new String[] {"TABLE" });
4  -- Gibt die vorhandenen Views des erfragten Schemas zurück
5  -- wie bei Tabellen
6  dbmd.getTables(null, null, "%", new String[] {"VIEW" });
7  -- Abfrage der Spalten einer Tabelle oder View
8  -- (Katalog, Schema, Tabellenname, regulärer Ausdruck als Suchbegriff)
9  dbmd.getColumns(null, null, table.getTitle(), "%");
10 -- Abfrage der Primärschlüssel, nur für Tabellen definiert
11 -- (Katalog, Schema, Tabellenname)
12 dbmd.getPrimaryKeys(null, null, table.getTitle());
13 -- Abfrage der Fremdschlüssel nach einem Schema wie bei Primärschlüssel
14 -- Ebenfalls nur für Tabellen definiert
15 dbmd.getImportedKeys(null, null, table.getTitle());
```

Listing 10: Benötigte Methoden des Interfaces DatabaseMetaData

Bei der Implementierung der Analyse wird allerdings festgestellt, dass die Methode `getIndexInfo()` des Interfaces `DatabaseMetaData` zum Erfragen der Indizes und Unique-Bedingungen speziell unter Oracle keine erwarteten Ergebnisse liefert[267]. Diese Methode soll die vorhandenen Indizes und Unique-Bedingungen einer Tabelle zurückgeben. Während der Testphase dieser Methoden gegen eine reale Datenbank lies sich erkennen, dass diese Methode sehr schnell einen Fehler auftreten lässt oder keine akkuraten Ergebnisse liefert. Es wird für jeden Primärschlüssel zusätzlich in dieser Methode eine Unique-Bedingung angezeigt. Allerdings sollten hier ausnahmslos die Unique-Bedingungen und Indizes erscheinen, da die Informationen für die Primärschlüssel bereits verarbeitet und

[266] s. S. 94 Listing 10 (10)
[267] s. S. 95 Listing 11 (11)

gespeichert werden. Daher kann diese Methode leider nicht eingesetzt werden. Für die Informationen der Indizes und Uniques ist eine eigene *SELECT*-Anweisung nötig[268], die diese Anforderung erfüllt.

```
1 dbmd.getIndexInfo(null, null, table.getTitle(), false, false);
```

Listing 11: Methoden des Interfaces DatabaseMetaData (nicht funktionsbereit)

Es werden weitere *SELECT*-Abfragen benötigt, um die Anforderungen komplett zu realsieren. Nicht alle benötigten Informationen können über das Interface `DatabaseMetaData` erfragt werden.

Dazu gehören der Quelltext der *SELECT*-Abfrage einer View wie auch der einer materialisierten View. Das Interface bietet hier nur die Unterstützung zur Abfrage auf Tabellen. Es gibt dort zwar die Option, nach dem Typ *VIEW* zu suchen, allerdings haben Tabellen diese Spalte des Quelltextes nicht. Daher wird hier diese Art der Anfrage nicht unterstützt. Die materialisierten Views und die anderen Views können auch nicht zu einer Anweisung zusammengefasst werden, da sich die Spalten im Data Dictionary unterscheiden und sie für die Anwendung getrennt behandelt werden. Aus diesem Grund werden zwei unterschiedliche Anweisungen benötigt.

Weiterhin muss für die Informationen der Primär- und Fremdschlüssel sowie Indizes und Unique-Bedingungen eine weitere *SELECT*-Anweisung geschrieben werden. Die Integritätsbedingungen sind für Views in Oracle alle `disabled`. Das bedeutet, dass sie laut Oracle-Spezifikation[269] zwar möglich sind, aber sie sind nicht aktiv. Auch hier wird seitens der Schnittstelle keine Unterstützung zur Abfrage geboten.

Eine weitere Anweisung wird benötigt, um die Trigger auszulesen. Auch diese Art der Anfrage an das Data Dictionary wird nicht unterstützt. Die Triggerabhängigkeiten werden allerdings auch grafisch dargestellt, daher ist die Abfrage ebenfalls notwendig. Bei den verschiedenen Versionen von Oracle wurde die View `user_triggers` im Data Dictionary verändert. Daher werden an dieser Stelle zwei verschiedene Abfragen, jeweils für Oracle in der Version 11g und 10g/9i[270], benötigt.

Jede dieser Anweisungen wird über ein `PreparedStatement` an die Datenbank geschickt. Durch ein `PreparedStatement` kann die Anweisung dynamisch zur Laufzeit zusammengesetzt werden. Hier dient das Fragezeichen als Platzhalter und wird in einer Schleife durch den entsprechenden Tabellen- oder Viewnamen ersetzt. Diese Schleife

[268] s. S. 96 Listing 12 (12)
[269] vgl. [o.V. 09a]
[270] Zwischen den Versionen 9i und 10g von Oracle besteht in diesem Fall kein Unterschied.

wird bei den Triggern nicht benötigt, da eine Abfrage für alle benötigten Informationen implementiert wird.

```
1  -- SELECT-Abfrage für den Quelltext einer View für Oracle 11g
2  SELECT text FROM user_views WHERE UPPER(view_name) LIKE UPPER(?)
3  -- SELECT-Abfrage für den Quelltext einer materilisierten View für Oracle 11g
4  SELECT query FROM user_mviews WHERE UPPER(mview_name) LIKE UPPER(?)
5  -- SELECT-Abfrage für die Primär- und Fremdschlüssel sowie Uniques und weitere
       Indizes für Oracle 11g
6  SELECT ucc1.column_name, uc1.constraint_name, uc1.constraint_type, ucc2.column_name, ucc2
       .table_name FROM user_cons_columns ucc1, user_constraints uc1, user_cons_columns ucc2
        WHERE uc1.r_constraint_name = ucc2.constraint_name(+) AND uc1.constraint_name = ucc1
       .constraint_name AND uc1.table_name LIKE UPPER(?)
7  -- SELECT-Abfrage für die Indizes und Uniques einer Tabelle für Oracle 11g
8  SELECT ucc1.column_name, uc1.constraint_name, uc1.constraint_type, ucc2.column_name, ucc2
       .table_name FROM user_cons_columns ucc1, user_constraints uc1, user_cons_columns ucc2
        WHERE (uc1.constraint_type LIKE 'C' OR uc1.constraint_type LIKE 'U') AND uc1.
       r_constraint_name = ucc2.constraint_name(+) AND uc1.constraint_name = ucc1.
       constraint_name AND uc1.table_name LIKE UPPER(?)

10 -- SELECT-Abfrage für die Namen der materialiserten Views für Oracle 11g
11 SELECT mview_name FROM USER_MVIEWS
12 -- SELECT-Abfrage für die Trigger für Oracle 11g
13 SELECT trigger_name, trigger_type, triggering_event, table_owner, base_object_type,
       table_name, column_name, referencing_names, when_clause, status, description,
       action_type, trigger_body, crossedition, before_statement, before_row, after_row,
       after_statement, instead_of_row FROM user_triggers WHERE table_name IS NOT NULL
```

Listing 12: Eigene SELECT-Anweisungen

Die Tabellenabhängigkeiten durch Fremdschlüsselbeziehungen können durch diese Abfragen schon vollständig dargestellt werden. Für das Bestimmen der anderen Abhängigkeiten ist allerdings noch eine weitere Analyse notwendig, die von einer zusätzlichen Klasse, dem Parser, übernommen wird[271].

Jedes dieser ausgelesenen und analysierten Datenbankobjekte wird in einer Liste anhand des Names des Objektes gespeichert. Diese Listen werden von der Parserimplementierung benötigt, um später die beteiligten Tabellen und Views sowohl beim Parsen einer View wie auch beim Parsen eines Triggers ermitteln zu können. Diese Listen werden nach einer erfolgreichen Analyse des Datenbankschema von der Klasse `AbstractMetaData` beziehungsweise durch eine konkrete Instanz dieser Klasse[272] abgefragt, gespeichert und von dort aus verwaltet.

[271] s. Seite 97, Abschnitt 6.3.5 (6.3.5)
[272] s. Seite 92, Abschnitt 6.3.3 (6.3.3)

Zusätzlich wird für jede Tabelle, View oder Trigger eine Klasse implementiert, die ein *Hibernate*-Objekt[273] repräsentiert. Diese Objekte werden intern in einer HSQLDB gespeichert. Daher müssen die Datenbankobjekte nicht bei jedem Start der Anwendung neu analysiert werden. Dies geschieht aus Gründen der Performanz, da die einzelnen grafischen Ansichten somit schneller aufgebaut werden können.

Die Laufzeit der Analyse beträgt $O(n)$, wobei n für die Anzahl der Tabellen, Views und Trigger insgesamt steht. Für jedes dieser Objekte wird der Name ermittelt und gespeichert, wodurch eine Laufzeit von $O(n)$ entsteht. Im Anschluß werden die einzelnen Integritätsbedingungen sowie die Quelltexte der Views und Trigger gelesen und gespeichert. Durch diese zweite Schleife entsteht ebenfalls eine Laufzeit von $O(n)$. Das ergibt somit eine Gesamtkomplexität von $O(n + n) = O(n)$.

6.3.5. Parsen

Nach einer erfolgreichen Analyse und dem erschlossenen Wissen über die vorhandenen Tabellen, Views und Trigger und entsprechende Fremdschlüsselbeziehungen werden weitere Abhängigkeiten bestimmt. Dabei gilt es nun, die Abhängigkeiten der Views zu anderen Views und Tabellen sowie die Abhängigkeiten der Trigger durch ausgeführte Data Manipulation Language (DML)-Anweisungen im Triggerrumpf zu bestimmen.

Hierfür werden zwei unterschiedliche Parser benötitgt, da die Views in Structured Query Language (SQL) und die Trigger in Procedural Language/SQL (PL/SQL) geschrieben sind. Die beiden Parser werden in zwei verschiedene Klassen implementiert.

Für beide Arten von Anweisungen wird der SQLJEP[274] als Tokenizer[275] eingesetzt, um die einzelnen Wörter beziehungsweise Befehle einer Anweisung weiter verarbeiten zu können. Hierfür werden Grammatiken benötigt, um nach einem gelesenen Befehl entscheiden zu können, wie der weitere Verlauf des Parsens stattfinden soll. Für diese Fallunterscheidungen müssen in beiden Klassen Grammatiken implementiert werden, die auf den Backus-Naur-Formen (BNF)[276] der Datenbankobjekte beruhen.

Der Parser liest die Anweisungen der Views und Trigger jeweils gesondert ein und gibt die eingelesenen Befehle in Tokens zurück. Die benötigte Token-Klasse wird bei dem SQLJEP-Parser bereits mitgeliefert. Ein Token besteht aus einer vordefinierten Konstante. Wird ein Wort eingelesen, vergleicht der Parser diesen `STRING` mit den spezifizierten

[273] s. Seite 80, Abschnitt 6.2.3.1 (6.2.3.1)
[274] s. Seite 50, Abschnitt 5.1.4.3 (5.1.4.3)
[275] Scannt einen Text und gibt die Wörter in der eingelesenen Reihenfolge zurück.
[276] s. Anhang D, S. 195, Backus-Naur-Form 34 (34)

Tokens. Auf diese Wiese können SQL-Schlüsselwörter wie AND, OR oder auch EXISTS sowie Vergleichsoperatoren direkt erkannt werden. Weitere Befehle, aber auch Variablen-, Tabellen-, View- oder Triggernamen werden zunächst als Identifier zurückgegeben. An dieser Stelle wird dann die eigene Grammatik entscheidend, damit der weitere Verlauf des Parsens bestimmt werden kann.

Ein vom Parser vordefiniertes Token ist das EOF-Token, das auf End-Of-File beziehungsweise Ende des eingelesenen Textes[277] hindeutet. Wird dieses Token erkannt, kann der Vorgang des Parsens abgeschlossen werden, da keine weiteren Tokens mehr folgen.

6.3.5.1. Parsen einer View Die abstrakte Klasse AbstracViewSqlParser dient als Oberklasse für die einzelnen konkreten Implementationen der verschiedenen Datenbanksysteme. Durch eine vorhandene Enumeration (VENDOR) leitet diese Klasse die einzelnen Aufgaben an die konkreten Instanzen für das entsprechende Datenbanksystem weiter. Die Klasse bekommt beim Starten des Parsens die Listen der Tabellen- und Viewobjekte aus der Analyse übergeben, um die Tabellen und Views aus der SELECT-Anweisung bestimmen zu können[278].

Eine SELECT-Abfrage, die den eigentlichen Teil der View ausmacht, besteht in ihrer Grundform aus sechs Komponenten[279]. In der SELECT-Komponente werden die Spalten selektiert, die in der View angezeigt werden sollen. Hieraus können keine Abhängigkeiten erkannt werden. Allerdings wird eine Regel für das Entdecken dieser Komponente festgelegt, da der Parser die einzelnen Befehle bis zur nächsten Komponente lesen muss. Zwei weitere Komponenten sind ebenfalls nicht von Bedeutung, die GROUP BY- sowie die ORDER BY-Komponenten. Auch hier können ebenfalls keine Abhängigkeiten erkannt werden und brauchen dementsprechend nicht berücksichtigt werden.

Für die Implementierung bedeutet dies, dass eine Hauptschleife (Algorithmus 1) die Tokens abfragt, bis die Anweisung beendet ist (EOF-Token wird erkannt). Wird das Token als FROM-Komponente erkannt, wird eine Funktion aufgerufen, die die beteiligten Tabellen und Views erkennt und speichert. Im Falle der WHERE- beziehungsweise HAVING-Komponenten kann die Art der Abhängigkeiten erkannt werden. In beiden Komponenten können Bedingungen stehen, die erkennen lassen, ob die Abhängigkeit positiv oder negativ ist.

Bei der Untersuchung der FROM-Komponente bestehen zunächst drei Möglichkeiten, die erkannt werden können und als Regel implementiert werden.

[277] In diesem Fall eine *SELECT*-Anweisung.
[278] s. Seite 92, Abschnitt 6.3.3 (6.3.3)
[279] s. Seite 45, Abschnitt 5.1.2 (5.1.2)

Algorithmus 1 Hauptschleife $parse(listOfTable, listOfViews)$

Require: $listOfTable \neq NULL$ and $listOfViews \neq NULL$

```
1: Token t ⇐ getNextToken
2: while t ≠ Token(EOF do
3:    if kind(t) = Token(Identifier) then
4:       if t =“FROM“ then
5:          parseFrom
6:       else if t =“WHERE“ or t =“HAVING“ then
7:          parseCondition
8:       end if
9:    end if
10:   Tokent ⇐ getNextToken
11: end while
```

Tabellen-/Viewname Anhand der übergebenen Listen kann eine Tabelle oder View erkannt und in einer Liste der verwendeten Tabellen zwischengespeichert werden. Diese Liste wird benötigt, um im weiteren Verlauf anhand der `WHERE`- oder `HAVING`-Komponente entscheiden zu können, ob diese Tabelle oder View positiv beziehungsweise negativ abhängig ist.

Alias Ist das eingelese Wort kein Tabellen- oder Viewname und auch keine weitere Unterabfrage, dann wurde für diese Tabelle ein Alias vergeben. Diese werden in einer `HashMap` zu den entsprechenden Tabellen- oder Viewnamen gespeichert, um die Spalten später der ursprünglichen Tabelle oder View zuordnen zu können.

Unterabfrage In der `FROM`-Komponente sind weitere Unterabfragen möglich[280]. Anhand des Schlüsselwortes `SELECT` kann diese Unterabfrage erkannt und entsprechend weiterverarbeitet werden. Hierfür wird diese Unterabfrage komplett verarbeitet und dem definierten Alias am Ende der Unterabfrage zugeordnet.

Falls ein weiteres Schlüsselwort wie `WHERE`, `GROUP BY` oder `ORDER BY` erkannt wird, ist das Parsen dieser Komponenten abgeschlossen und die involvierten Tabellen und Views werden zwischengespeichert. Wird ein Mengenoperator wie `UNION` oder `MINUS` erkannt, folgt eine weitere *SELECT*-Anweisung, die als eigene Komponente, ebenso wie eine Unterabfrage, neu geparst wird.

Um die involvierten Tabellen einer positiven oder negativen Abhängigkeit zuordnen zu können, werden die beiden Komponenten `WHERE` und `HAVING` geparst. Diese Komponenten sind optional und müssen somit nicht auftauchen. Fehlen beide Komponenten und wird auch kein Mengenoperator wie `MINUS` angegeben, sind die involvierten Tabellen und Views komplett positiv abhängig.

[280] s. Anhang D, S. 195, Backus-Naur-Form 34 (34)

Der Parser liest die Anweisungen der beiden Komponenten aus und entscheidet anhand der Grammatik, ob eine negative oder positive Abhängigkeit erkannt wird[281]. Durch die beiden Schlüsselwörter `MINUS` sowie `NOT EXISTS` kann eine negative Abhängigkeit erkannt werden. In beiden Fällen erfolgt laut der Spezifikation dieser Komponente[282] eine weitere Unterabfrage. Die Komponenten dieser Unterabfrage sind somit negativ involviert und werden entsprechend in der Liste der negativen Abhängigkeiten gespeichert.

```
switch (t.kind) {
case ParserConstants.NE:
  t = parser.getNextToken();
  relationString = temp.toString() + " != " + t.toString();
  break;
```

Listing 13: Regeln der WHERE- oder HAVING-Komponente

Weiterhin besteht die Möglichkeit einer verschachtelten Abhängigkeit. Wird in dieser Unterabfrage ebenfalls eines der beiden Schlüsselwörter erkannt, ergibt sich hieraus eine doppelte Verneinung und die Abhängigkeit wird positiv. Für diesen Fall wird eine Zählervariable und ein `boolean` implementiert. Wird auf eine Spalte der entsprechenden Tabelle oder View zugegriffen, kann durch den Zähler und die Abfrage des `boolean` die Abhängigkeit bestimmt werden. Steht der Zähler auf ein Vielfaches von 2 oder auf 0, so besteht eine postive Abhängigkeit, ansonsten ist sie negativ. Ergibt die Abfrage des `boolean` ebenfalls `true`, so wird die ermittelte Abhängigkeit umgekehrt (positiv wird zu negativ und entsprechend negativ zu positiv).

```
private int notExists;
private boolean minus = false;
..
if (string.equals("MINUS")) {
  minus = true;
}
..
if (temp.toString().toUpperCase().equals("NOT")) {
  if (temp.toString().toUpperCase().equals("EXISTS")) {
    notExists++;
  }
}
```

Listing 14: Zähler für NOT EXISTS und MINUS

Die Laufzeit des Parsens beträgt $O(n)$, wobei n für die Anzahl der Views steht. Die implementierte Hauptschleife liest jeden Befehl (k) der `SELECT`-Anweisung für jede View

[281] s. S. 100 Listing 14 (14)
[282] s. Anhang D, S. 195, Backus-Naur-Form 34 (34)

einzeln ein. Hierbei entsteht die Laufzeit von $O(n * k)$. Bei der Laufzeitbetrachtung werden allerdings Konstanten (k) nicht beachtet, da sie die Laufzeit im wesentlichen nicht verändern. Daraus ergibt sich die Gesamtkomplexität von $O(n)$.

6.3.5.2. Parsen eines Triggers Zunächst muss der Quelltext des Triggers angepasst werden. Der Parser wurde für SQL entwickelt und lässt beispielsweise bei einem Semikolon eine Programmausnahme entstehen. Daher werden diese PL/SQL spezifischen Symbole durch vordefinierte Konstanten ersetzt.

Im Gegensatz zu der Anweisung einer View muss der Parser auf andere Schlüsselwörter reagieren. Jedoch erfolgt das Parsen dieser Anweisung ebenfalls in einer Hauptschleife wie bei einer View (vergleiche Listing 1) mit entsprechend gleicher Laufzeit. Der Unterschied besteht einzig aus der Abfrage der Schlüsselwörter. Diese werden ebenfalls anhand einer Fallunterscheidung verarbeitet[283].

Hier sind die vier Schlüsselwörter `SELECT`, `INSERT`, `UPDATE` und `DELETE` von Bedeutung. Diese Befehle sind optional und müssen somit nicht im Quelltext des Triggers auftauchen. Durch eine Data Query Language (DQL)-Anweisung wie `SELECT` werden keine weiteren Trigger angestoßen, aber es kann ein *Mutating-Table*-Problem auftauchen und sie dienen zur Übersicht der verwendeten Tabellen und Views. Auch hier werden die Listen der Tabellen und Views aus der Analyse benötigt[284], um die entsprechenden Abhängigkeiten zuordnen zu können.

```
1 if ((t.kind == ParserConstants.IDENTIFIER) && "SELECT".equalsIgnoreCase(token)) {
2   inSelect = true;
3 } else if ((t.kind == ParserConstants.IDENTIFIER) && "DELETE".equalsIgnoreCase(token)) {
4   inDelete = true;
5 } else if ((t.kind == ParserConstants.IDENTIFIER) && "INSERT".equalsIgnoreCase(token)) {
6   inInsert = true;
7 } else if ((t.kind == ParserConstants.IDENTIFIER) && "UPDATE".equalsIgnoreCase(token)) {
8   inUpdate = true;
9 }
```

Listing 15: Trigger - Abfrage der Schlüsselwörter in der Hauptschleife

Anders verhält es sich bei den drei Data Manipulation Language (DML)-Befehlen. Durch manipulierenden Zugriff auf weitere Tabellen können implizit andere Trigger angestoßen werden. Hierbei ist zu beachten, dass ein weiterer Trigger eventuell nur durch einen der drei DML-Befehle angestoßen wird. Um dies zu berücksichtigen, wird für jeden Befehl

[283] s. S. 101 Listing 15 (15)
[284] s. Seite 92, Abschnitt 6.3.3 (6.3.3)

eine Liste implementiert, da ein Trigger, der einen `DELETE`-Befehl ausführt, keinen Trigger anstößt, der nur für ein `INSERT` implementiert wurde.

Nach einem erfolgreichem Auslesen der Abhängigkeiten werden die Listen der verschiedenen Befehle von der Klasse `AbstractMetaData` verarbeitet. Diese Listen werden in einem internen Triggerschema gespeichert und können abgerufen werden. Dieses Triggerschema wird benötigt, um die erkannten Abhängigkeiten im weiteren Verlauf grafisch darstellen zu können.

6.4. Graphen

Natürlich ist der analytische Teil der Arbeit für das entstehende Produkt von zentraler Bedeutung – schließlich sind es die Schlüsselfunktionen: das Bestimmen relevanter Abhängigkeiten bei Views, das Parsen der SQL-Statements und nicht zuletzt das Erkennen von potenziell rekursiven Abfolgen bei Triggern. Jedoch ist es mindestens ebenso wichtig, diese gesammelten Daten auch in einer vernünftigen Form darzustellen. Nicht zuletzt behandelt diese Arbeit schließlich das Thema der visuellen Darstellung von Abhängigkeiten: Es ist entscheidend, dass die Visualisierung dem Benutzer auch erkannte Zyklen, Probleme oder Zusammenhänge in einer verständlichen Art und Weise präsentieren kann.

Bei der View-Hierarchie muss die Anwendung dabei eine Vielzahl von Wurzelknoten anzeigen, die weitere Verzweigungen mit Kindknoten nach unten besitzen. Jeder Pfad, ausgehend von einem Wurzelknoten, endet stets in einem Knoten, der immer eine Tabelle repräsentiert. Da Tabellen in der untersten Ebene liegen, sind sie auch als tiefste Knoten in einer Reihe im Graphen darzustellen[285]. Dabei werden nur die Tabellen dargestellt, welche auch in einer der ausgewählten Views verwendet werden.

Hingegen wird bei Triggern eine strukturfreie, organische oder konzentrische Darstellung bevorzugt: Es ist nicht vorherzusehen, welche und wie viele Verzweigungen der entstehende Graph enthalten kann. Deswegen ist von einer reinen strukturellen Darstellung abzusehen, denn es gibt weder eine festgelegte Hierarchie noch feste Abfolgen von Knoten.

Bei den Fremdschlüsselbeziehungen schließlich sind die gleichen Vorraussetzungen vorhanden wie bei den Triggerbeziehungen. Auch hier ist eine flexible und organische Darstellung in der Regel auch die beste und Platz effizienteste.

[285] s. S. 20 Abb. 3.1 (3.1)

6.4.1. Übersicht zur API

Für die Darstellung als Graphen wird das JUNG-Framework in der aktuellen Version 2.0 genutzt[286]. Dabei besteht das Framework JUNG aus mehreren großen Teilbereichen, die nun kurz vorgestellt werden.

Die objektorientierte Graphen-API, mit der sich sowohl einfache als auch komplexe Graphen in Java erstellen und verändern lassen, kann Graphen in Form von Bäumen, Wäldern, Netzwerken oder Hierarchien repräsentieren – wobei ein Graph immer aus einer Menge von Knoten und eine Menge von Kanten besteht. Die Bibliothek verwendet ausschließlich Javas Generics und stellt damit für große Anwendungen bedeutsame Werkzeuge für Graphen aller Art zur Verfügung. Alle notwendigen Klassen und Schnittstellen befinden sich in der Jar-Datei *jung-api-2.0.jar*.

Des Weiteren befinden sich im Framework eine Reihe fertiger Algorithmen. Das Spektrum der mitgelieferten Algorithmen erstreckt sich dabei von Layout- über Bewertungsbis hinzu Flussnetzalgorithmen. Neben speziellen Algorithmen, wie der Edmund-Karp-Algorithmus zum Ermitteln des maximalen Flusses in Netzwerken oder der Page Rank Algorithmus für die Knoten, sind dies hauptsächlich „Kürzeste Wege“-Algorithmen (*shortest path*) und Layoutalgorithmen: Erwähnenswert sind dabei in erster Linie die Algorithmen von Dijsktra (Wegfindung), Bellman-Ford (Wegfindung und Prüfung auf Zyklen) und der *Minimal aufspannende Baum* im Bereich der klassischen Graphenalgorithmen. Für das Darstellen des Graphen sind rund über ein Dutzend Layoutalgorithmen verfügbar: von klassischen Algorithmen von der Darstellung als Bäume, Kreise, Ballons („aufgeblasener"Baum oder Graph), radial orientierten Bäumen oder Fruchterman-Reingold bis hin zu speziellen Varianten für gerichtete, ungerichtete, baum- oder netzwerkartige Graphen. Für weitere Informationen wird auf die Dokumention und den, teilweise belegten, Quellcode verwiesen. Die Algorithmen befinden sich in der Jar-Datei *jung-algorithms-2.0.jar*.

Für die Visualisierung besonders wichtig ist der Inhalt der Jar-Datei *jung-visualization-2.0.jar* – nämlich die Visualisierungskomponenten für Graphen. In der Standardkonfiguration kann das Framework die Graphen in einer für algorithmische Anwendungen typischen Darstellung mit kleinen runden Knoten anzeigen. Außerdem befinden sich eine Reihe von fertigen Layoutern im Paket. Abgerundet wird die mitgelieferte Auswahl mit einer Reihe von Standardmodulen für individualisierte Graphen.

Alle genannten Pakete enthalten Klassen und Interfaces, die allesamt typisierbar mit Java Generics sind. Dies ist für den universellen Einsatz von erheblicher Bedeutung[287].

[286] s. Seite 66, Abschnitt 5.2.5 (5.2.5)
[287] s. Seite 105, Abschnitt 6.4.3 (6.4.3)

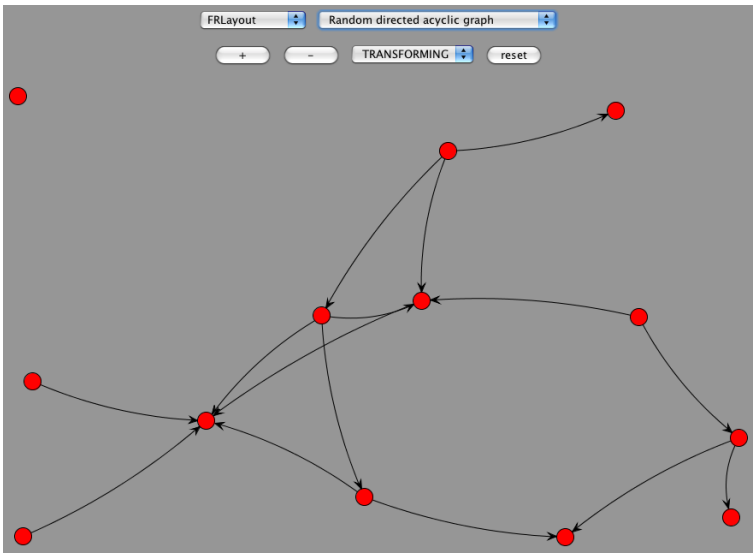


Abbildung 6.9: Beispiel einer Darstellung von Graphen des *JUNG2*-Frameworks

Außerdem gibt es noch Komponenten für die Darstellung dreidimensionaler Graphen und eine Schnittstelle für das XML-Austauschformat GraphML in zwei weiteren Jar-Dateien.

6.4.2. Das JUNG2-System

JUNG ist sehr modular aufgebaut und nutzt verschiedene Entwurfsmuster. Für die Kernfunktionalitäten gibt es eine Reihe von (typisierten) Schnittstellen – zum Beispiel die Schnittstelle `Graph<V, E>`[288] für den allgemeinen Graphen, die Schnittstelle `Forest<V, E>` für einen Wald von Bäumen oder auch die Schnittstelle `DirectedGraph<V, E>` für gerichtete Digraphen. Eine grobe Übersicht findet sich im Anhang als UML-Diagramm wieder[289].

Gesamtheitlich betrachtet ist das Framework sauber und umfangreich entwickelt. Verschiedene Komponenten haben jeweils ihre eigenen Kompetenzen, können aber durch eigene Module erweitert werden. Während ein Teil durch übliche Generalisierung erweiterbar ist, werden andere Teile des Frameworks durch bekannte Entwicklungsmuster modifiziert – beispielsweise das Strategiemuster in den Renderermodulen.

Dadurch, dass es möglich ist, auch nur kleine Änderungen vorzunehmen, lässt sich die Darstellung nahezu beliebig variieren. Beispiele folgen dazu in einem späteren Abschnitt[290].

[288] Sowohl das Framework JUNG als auch die Eigenentwicklungen nutzen die Generic-Typen E für Kante (englisch edge) und V für Knoten (englisch vertex).
[289] s. Anhang A, S. 162, Abb. A.11 (A.11)
[290] s. Seite 106, Abschnitt 6.4.5 (6.4.5)

6.4.3. Typisierter Graph

In JUNG besteht ein Objekt des Typs `Graph<V,E>` – oder einem davon abgeleiteten Typ – aus einer Menge von Knoten und Kanten. Die universellen Generic-Typen `V` und `E` entsprechen dabei den konkreten Entitäten der Anwendung sowie deren Beziehungen untereinander: die Tabellen, Trigger und Views der Datenbank als Knoten und die Relationen als Kanten.

Die konkreten Klassennamen in der Anwendung sind:

DatabaseTable Eine Tabelle besteht aus einem Namen (Titel) sowie einem definierten Schema der beinhalteten Spalten, Indizes, Schlüsseln und Relationen.

DatabaseTrigger Ein Trigger ist definiert durch Titel, seine unmittelbaren Attribute für Ausführungszeit (event), -bedingung (condition), und -modus (mode), den eigentlichen Code (action) sowie einer Reihe weiterer Randinformationen.

DatabaseView Eine View ist (im Kontext dieser Anwendung) wie eine Tabelle definiert – mit der Ergänzung, ob sie materialisiert ist.

Die Kantenobjekte in allen drei Graphen sind Objekte des Typs `Relation`. Dabei wird jedes Kantenobjekt entsprechend seiner semantischen Zugehörigkeit mit Daten aufgefüllt.

Alle weiteren Objekte sind entsprechend ihrer Verwendung und den gegebenen Umständen typisiert.

6.4.4. Unterschiedliche Ansichten von Graphen

Die Anwendung stellt neben der View-Hierarchie auch die Trigger und ein *ERD* der Tabellen als Graphen visuell dar. Für alle drei Ansichten werden die gleichen Visualisierungskomponenten genutzt. Daher können fast alle Visualisierungskomponenten wiederverwendet werden.

Das *vollständige ERD* eines Datenbankschemas zeigt nicht nur die Entitäten und bloßen Fremdschlüsselbeziehungen, sondern zusätzlich die Kardinalität und Optionalität der Beziehung an. Bei der Darstellung gibt es verschiedene Notationen, wobei die Martin-Notation[291] und die neuere UML-Notation sicherlich sehr verbreitet sind.

In der hier zu entwicklenden Anwendung wird als Darstellungstyp ein *einfaches Entity-Relationship-Diagramm* verwendet. Im Vergleich zu dem vollständigen *ERD* fehlen die

[291] Die Martin-Notation ist auch unter der Krähenfuß-Notation bekannt.

Kennzeichnungen der Kardinalität, Optionalität und Beschreibung der Assoziationen zwischen den Entitäten. Selbstverständlich ist es zu einem späteren Zeitpunkt möglich, ein *vollständiges ERD* zu implementieren[292].

6.4.5. Individuelle Visualisierung

Die Darstellung eines Graphen hängt von vier zentralen Komponenten ab. Der eigentliche `Graph` ist das Modell und beinhaltet, „was“ angezeigt wird. Der `VisualizationViewer`[293] bestimmt, „wo“ der Graph angezeigt wird und kann durch Interaktionsmöglichkeiten wie Maus- und Tastatureingaben erweitert werden. Als Verbindungskomponente dient der Layouter [294] dafür, die Darstellung in dem Kontext des „wie“ zu beschreiben. Schließlich dient die Komponente Renderer[295] dafür, die eigentlichen Objekte – also die Knoten und Kanten – grafisch darzustellen.

6.4.6. Layout

Der Layouter ist die Komponente, welche die einzelnen Knoten und Kanten in einem bestimmten Schema anordnet – etwa hierarchisch, binärisch oder konzentrisch. Eine Beispielanwendung ist als Screenshot in der Abbildung 6.9[296] dargestellt und stammt von der offiziellen Homepage[297].

Im Folgenden wird definiert, was Graphen und Bäume sind. Außerdem werden die Graphen der View-Hierarchien und Triggerbeziehungen untersucht. Zum Schluss wird erläutert, welche Anpassungen notwendig sind, um die Anforderungen zu erfüllen.

Die speziellen Anpassungen des so genannten *LeveledForestLayouts*[298] sind Bestandteil der folgenden Abschnitte.

6.4.6.1. Basis Als Basis wird der in JUNG mitgelieferte Layouter `TreeLayout` verwendet. Dieser Algorithmus dient zur Darstellung von freien Bäumen und ist überdies sogar in der Lage, einen Wald von Bäumen darzustellen.

[292] s. Seite 143, Abschnitt 9.2 (9.2)
[293] s. Seite 110, Abschnitt 6.4.7.1 (6.4.7.1)
[294] s. Seite 106, Abschnitt 6.4.6 (6.4.6)
[295] s. Seite 111, Abschnitt 6.4.7.2 (6.4.7.2)
[296] s. S. 104 Abb. 6.9 (6.9)
[297] s. [JUNG Dev. Team 09c]
[298] Die Namensgebung ergibt sich aus daraus, dass das Layout einen Wald von Bäumen darstellt. Jeder Baum wird mit speziellen Leveln gekennzeichnet.

In allen anderen Situationen wird als Grundlage ein fertiger Layouter verwendet. Der Layoutvorgang wird durch das Nachladen und Umpositionieren der Objekte abgeschlossen, falls der Benutzer zu einem früheren Zeitpunkt bereits das Layout verändert hat. Für diese Zwecke wird eine spezielle Fabrik[299] erstellt, welche verschiedene Layoutimplementierungen instanziiert und entsprechende Umpositionierungen durchführt.

6.4.6.2. Definition

Nach der Definition von Cormen[300] ist die darzustellende View-Hierarchie als Baum folgendermaßen einzuschätzen.

Freier Baum Mit der Prämisse, dass keine Zyklen auftauchen, ist der Graph ein allgemeiner freier Baum. Außerdem existiert eine Verbindung – also in diesem Kontext eine Abhängigkeit – nur einmal zwischen zwei Knoten.

Wald Ein Wald besteht aus einem freien Baum oder mehreren freien Bäume. Im Kontext betrachtet, ist es durchaus möglich, dass mehrere View-Hierarchien komplett unabhängig voneinander im System existieren.

Digraph Alle Verbindungen in einem freien Baum sind durch Wahl der Wurzel gerichtet. Das gleiche gilt auch für einen Wald aus gerichteten und freien Bäumen.

Das bedeutet: So lange der Graph, der die View-Hierarchien darstellt, nicht azyklisch ist, kann er nicht (einmal) mehr als freier Baum gelten[301].

Alle Graphen sind jederzeit vollständig gerichtet, weil es keine richtungslosen Abhängigkeiten gibt.

Alle anderen Graphen (Triggeraktivitäten und *ERD*) sind direkt gerichtet und können zyklisch sein und lassen sich daher nicht mit Hilfe eines Baumlayouts darstellen.

6.4.6.3. Rekursionsproblematik

Eine Rekursion von Views ist problematisch, weil der Baum damit nicht mehr azyklisch und demzufolge nach obiger Definitionsregel auch kein freier Baum mehr ist. Da das `TreeLayout` ebenfalls nur azyklische Bäume darstellen kann, muss diese Spezialfunktionalität nachträglich eingefügt werden. Eine Alternative ist die Verwendung anderer Layouttechniken.

Da im Rahmen dieser Arbeit zunächst keine rekursiven View-Hierarchien dargestellt werden, entfällt eine spezielle Implementierung. Sollte eine spätere Funktionsänderung

[299] Die Fabrik findet sich unter `GraphLayoutFactory` im Paket `ui.views.common.graph.layout`.
[300] vgl. [Cormen 07], S. 1087f
[301] s. Seite 107, Abschnitt 6.4.6.3 (6.4.6.3)

ergeben, dass Rekursionen und Zyklen erlaubt und darzustellen sind, muss eine entsprechende Anpassung in den Layoutern durchgeführt werden.

Für alle anderen Grundfunktionalitäten der Anwendung ist die Problematik irrelevant, weil diese andere Layouttechniken verwenden.

6.4.6.4. Erweiterungen

Das vorhandene `TreeLayout` muss um einige Funktionen ergänzt werden. Selbstverständlich müssen Teile der Funktionen auch in anderen Layoutern durchgeführt werden, falls jene für eine repräsentative Visualisierung verwendet werden sollen.

Levels Jeder Knoten soll auf seinem errechneten Level[302] sein. Dabei ist die Orientierung unüblich, weil der allgemeine Fall von einer Orientierung von oben nach unten spricht. Hier ist die Orientierung aber umgekehrt.

Überlappungen Ein weiteres Problem sind viele Knoten im Graphen. Der Algorithmus hat dann ein Problem und es können überlappende Knotenbereiche in der Darstellung auftreten. Dem Problem kann entgegengewirkt werden, indem wahlweise die Knoten verkleinert oder optimaler positioniert werden. Nichtsdestotrotz kann das Problem nicht vollständig gelöst werden. Der Renderer, der dieser Anwendung beiliegt, verzichtet auf eine Skalierung[303] zugunsten einer Verschiebung der Knotenobjekte.

Persistenz Schließlich soll der Benutzer in der Lage sein, die Knoten nach seiner beliebigen Wahl innerhalb des Graphen zu verschieben. Die Positionen müssen gespeichert und während des (initialen) Layout-Vorganges wieder hergestellt werden.

Für diese Implementierung nicht relevant sind die **zyklische Verbindungen**: Es kann vorkommen, dass Views (Knoten) in einem Zyklus miteinander verbunden sind. Der bisherige Algorithmus (`TreeLayout`) „vergisst“ deswegen einige Kanten, welche den Schluss für einen Zyklus bedeuten[304]. Allerdings sind sie für spätere Veränderungen festzuhalten und ihre Erkennung durchaus als *nice to have* zu klassifizieren.

6.4.6.5. Bestimmen der Knoten-Level

Die bereits angesprochene Bestimmung der unterschiedlichen Levels ist eine Variante der Höhenbestimmung von Knoten in einem Baum. Das Problem lässt sich mittels Rekursion relativ einfach lösen.

[302] vgl. Seite 107, Abschnitt 6.4.6.2 (6.4.6.2)

[303] Für eine Skalierung der Objekte muss beispielsweise auch der Layouter in der Lage sein, die unterschiedlichen Dimensionen effektiv zu nutzen. Außerdem müssen die Knotenattribute Name, Farbe und Form auch in der verkleinerter Form erkennbar sein.

[304] vgl. Seite 107, Abschnitt 6.4.6.2 (6.4.6.2)

Algorithmus 2 Initialisieren aller Level auf 0.

for all $table$ in $tables$ **do**
  $level(table) \leftarrow 0$
end for

Zunächst werden die Werte für alle Tabellen auf den Wert 0 initialisiert (Algorithmus 2). Anschließend wird für jeden Knoten im Baum, der eine Tabelle darstellt, der gesamte Pfad bis zur obersten Wurzel mittels Rekursion traversiert (Algorithmus 3).

Algorithmus 3 Definieren der Rekursion (Hauptschleife).

for all $table$ in $tables$ **do** {Berechnen der Level mit Rekursion}
  for all $relation$ in $relations(table)$ **do**
    if $target(relation) == table$ **then**
      $source_1 \leftarrow source(relation)$
      $computeLevel(source_1, target, 1)$
    end if
  end for
end for

Die rekursive Funktion $computeLevel$ (Algorithmus 5) stellt dabei sicher, das eine erreichte Höhe nie mehr verlassen werden kann und traversiert, falls notwendig, den Weg weiter aufwärts.

Algorithmus 4 Ermitteln des maximalen Levels.

for all $level$ in $levels$ **do** {Ermitteln des größten Levels (höchste Wurzel).}
  $maxLevel \leftarrow \max(maxLevel, level)$
end for

Die Berechnung des größten Levels ist nur für das Umpositionieren im speziellen Fall notwendig, wenn Objekte nicht an ihrem gewünschten Platz sind (Algorithmus 4).

Nachdem der eigentliche Prozess des Layoutvorganges abgeschlossen ist, werden alle Objekte hinsichtlich der Korrektheit des Levelwertes überprüft. Dies wird mit einer mathematischen Formulierung in Algorithmus 6 realisiert: Die Variablen $dist_y$ und $delta_y$ sind dabei Platzhalter für genaue Pixelangaben. Sollte ein Objekt nicht auf der erwarteten Höhe zu finden sein, wird es im Rahmen des initialen Layoutvorganges an die erwartete Stelle verschoben.

6.4.6.6. Laufzeit

Der gesamte Algorithmus zur Bestimmung aller Level hat eine Laufzeit von $O(n_t * n_v^3)$, wobei n_t die Anzahl der Tabellen und n_v die Anzahl der Views ist. Damit ist die Laufzeit direkt von der Anzahl der Objekte polynomisch abhängig.

Algorithmus 5 Die Funktion $computeLevel(source, target, level)$

Require: $source, table, level : source \neq table \wedge level \geq 1$
$level(source) \leftarrow \max(level(source), level)$ {Stellt sicher, dass eine Levelerhöhung nicht mehr zurückgenommen werden kann.}
for all $relation$ in $relations(source)$ **do**
if $target(relation) == source$ **then**
$source_2 \leftarrow source(relation)$
$computeLevel(source_2, source, level + 1)$
end if
end for

Algorithmus 6 Berechnung der erwarteten Höhe y.

Require: $maxLevel, level, dist_y, delta_y : maxLevel \geq level \geq 1 \wedge dist_y \in \mathcal{N} \wedge delta_y \in \mathcal{N}$
$expected_y \leftarrow (maxLevel - level) * dist_y + delta_y$

Im Detail ergibt sich diese Laufzeit aus der Hauptschleife $O(n_t * n_v * k')$ und der eigentlichen rekursiven Funktion $k^i = O(n_v * k^{i+1})$. Da n Views theoretisch maximal $n * n$ Abhängigkeiten untereinander haben können, ergibt das eine äußere theoretische Komplexität der Funktion von $k' = O(n_v^2)$. Das Initialisieren $O(n_t)$ und das Ermitteln des maximalen Levels $O((n_t + n_v)^2)$ sind dabei nebensächlich und ohne Bedeutung.

In der Realität wird diese Komplexität sicherlich nicht erreicht, denn diese ist nur eine Schätzung für den schlimmsten Fall (*worst case* Laufzeit). Darüber hinaus wird im aktuellen Stand keine Unterstützung für rekursive Views gegeben; damit verringert sich die Laufzeit der inneren Funktion auf $n * (n - 1)$.

Diese Berechnung ist selbstverständlich nur so lange gültig, wie keine doppelten Kanten im Graphen auftauchen.

6.4.7. Zentrale Komponenten

6.4.7.1. VisualizationViewer Der *VisualizationViewer* ist ein JPanel[305] und damit Swing-kompatibel. Der Viewer ist die Komponente in JUNG, die für die eigentliche grafische Ausgabe verantwortlich ist. Die Komponente besitzt überschreibbare Erweiterungen, welche für die Darstellung von Bedeutung sind: Größe der Darstellfläche, aktuelle Zoomstufe, verwendeter Renderer oder verwendete Rendererweiterungen. Außerdem werden mögliche Maus- und Tastaturaktivitäten mit dieser Komponente registriert und verbunden.

[305] Ein JPanel ist ein Standard-Container im Java-Swing-System für andere Container oder Elemente.

6.4.7.2. Renderer Ein *Renderer* ist verantwortlich für das eigentliche Zeichnen eines Datenobjektes. Im Kontext der Anwendung sorgt der Renderer etwa für die gewünschte Darstellung eines Objektes des Typs `DatabaseTable` als Tabellenknoten. Dabei kann der Renderer, je nach Framework, auf verschiedene Zusatzinformationen wie aktuell ausgewählter Knoten, Position des Mauszeigers oder Zoomstufe zurückgreifen.

Für die Darstellung aller im Rahmen der Anwendung vorkommenden Knotenobjekte – also Tabellen, Trigger und Views – ist die Komponente `DatabaseObjectComponent` zuständig. Diese Swing-kompatible Komponente ist für die grafischen Visualisierungsmerkmale wie Hintergrundfarbe, Titel, Beschreibung und Symbolik verantwortlich. Zusätzlich sorgt das Renderermodul `VertexRendererImpl` dafür, dass das JUNG-System für jeden Knoten die entsprechende Zeichenroutine nutzt. In der Kombination von JUNG entsteht dadurch eine ressourcenschonende und effiziente Anwendung mit den Entwurfsmustern *Delegation* und *Flyweight*.

Der *VisualizationViewer* verwendet einen so genannten „pluggable render context“. Diese Komponente realisiert das Entwurfsmuster „Strategiemuster“ und ermöglicht damit, die verschiedenen und einzelnen Transformationsschritte dynamisch in die Darstellung zu laden. Alleine für den Knoten gibt es neun verschiedene Transformationsmöglichkeiten: Von der Hintergrundfarbe, über die Form und Größe des Knoten bis hin zu dem Inhalt und der Schriftart des Knotennamens.

Der *RendererContext* enthält eine veränderbare Liste verschiedener Erweiterungen des oben genannten Renderers. Unterstützt der ausgewählte Renderer diese Erweiterungen, werden so objekt- und situationsspezifische Informationen nachgeladen. Da das JUNG-Framework an dieser Stelle auf die abstrakte `Transformer`-Schnittstellen zurückgreift, sind relativ geringe Grenzen für die eigene Kreativität gesetzt. So lassen sich beispielsweise nicht nur einfarbige Kanten(-hintergründe) erstellen – mittels einem geeigneten Gradiententransformators lässt sich auch ein entsprechender Farbverlauf produzieren. Weitere Beispiele für Transformationsschritte sind Hintergrundfarbe, Rahmenfarbe und die Objektform.

6.4.7.3. GraphMouse Die *GraphMouse* ist das entsprechende Pendant zu dem Userinterface „Maus“ des Benutzers und liegt im Paket `*.visualization.control` von JUNG. Die Klasse `DefaultModalGraphMouse` implementiert durch zahlreiche mitgelieferte Plugins eine Reihe von Features:

Scaling Dieses Plugin erlaubt das intuitive Vergrößern und Verkleinern (*Zoomen*) der Graphendarstellung über das Scrollrad der Maus. Zusätzlich gibt es ein Swing-

Spinner-Element für die genaue Steuerung der Zoomstufe.

Picking & Transforming Mit *Picking* ist das selektive Auswählen und Verschieben eines Objektes oder mehrerer Objekte mit der Maus gemeint. *Transforming* hingegen ist das komplette Verschieben der Perspektive – dies ist beispielsweise sinnvoll, wenn der eigentliche Graph größer als die Darstellfläche ist.

Maustasten Alle Maustasten lassen sich separat ansprechen – und darüber hinaus auch auf ihren individuellen Ereignistypen[306].

Editieren Prinzipiell lässt sich ein Knoten auch in den Bearbeitungsmodus versetzen; unter der Voraussetzung, dass der entsprechende Renderer dafür konfiguriert wurde.

In der Anwendung wird die oben genannte Mausimplementierung nicht verwendet, weil ein Popup (Kontextmenü) nach einem rechten Mausklick auftauchen soll. Die dafür notwendigen Änderungen werden in der Methode `handlePopup(MouseEvent e)` durchgeführt. Daher wird die entsprechende abstrakte Oberklasse `AbstractGraphMousePlugin` als Basis für eine eigene Implementierungsvariante genutzt.

6.4.8. Aufbau

Die Implementierung der Graphen wird gemäß der Arbeitsschritte *Erstellen* und *Darstellen* in zwei große Einheiten unterteilt.

Einerseits werden im Paket `graph.*` die eigentlichen Graphenklassen und Umwandlungsalgorithmen abgelegt. Ein Umwandlungsalgorithmus erstellt auf Basis der Datenentitäten die entsprechenden situationsspezifischen Graphenobjekte mit Knoten und Kanten. Andererseits beinhalten die Pakete `ui.views.*.graph` die jeweils angepassten Klassen zu einer Darstellung der Graphen.

`graph.common`	Abstrakte Oberklassen für die Klassen Graphen und Umwandlung, Klasse für den DOT-Export
`graph.entityrelations`	Konkrete Klassen für Graphen und Umwandlung für die Ansicht des ER-Modells
`graph.triggers`	Konkrete Klassen für Graphen und Umwandlung für die Ansicht der Triggerabhängigkeiten
`graph.viewhierarchy`	Konkrete Klasse für Graphen und Umwandlung für die Ansicht der Viewabhängigkeiten

Tabelle 3: Übersicht der Paketstruktur der Graphenimplementierung – Erstellen

[306] Bei einem Mausklick unterscheidet man in einer GUI in der Regel mindestens: wird-gedrückt, ist-gedrückt, wurde-losgelassen.

In beiden Paketen ist jeweils ein Unterpaket `common` für Klassen und Module angelegt, die für alle Ansichten der Anwendung geeignet sind. Außerdem sind in diesem Paket die entsprechenden Oberklassen oder Schnittstellen abgelegt. Eine detaillierte Übersicht der verwendeten Namensräume finden sich in den Tabellen 3 und 4.

`ui.views.*.graph`	Adapterklasse (JUNG in Swing)
`ui.views.common.layout`	Fabrikklasse für die Generierung der verschiedenen Layouts
`ui.views.*.graph.plugins`	Diverse GraphMousePlugins, wie zum Beispiel das Popupmenü auf Knoten
`ui.views.common.graph.renderer`	Konkrete Rendererkomponenten für die individuelle Visualisierung der Knotenobjekte
`ui.views.*.graph.transformers`	Zusätzliche Transformationsklassen für den *RendererContext* des Renderers

Tabelle 4: Übersicht der Paketstruktur der Graphenimplementierung – Darstellen

6.4.9. Fazit

Die Verwendung des Graphenframeworks JUNG ist keine schlechte Wahl für die Entwicklung und Visualisierung von Graphen in einer Java-Anwendung. Das Framework unterstützt Entwickler sowohl mit einer einfachen Graphen-API als auch mit voll Swing-kompatiblen Visualisierungsklassen. Durch die individuelle Renderer-Komponente lässt sich ein Graph erstellen, der komplett den eigenen Bedürfnissen entspricht.

Bei der Bewertung der verschiedenen Frameworks[307] wurde deutlich, dass auch andere Systeme in Erwägung gezogen wurden. Schlussendlich lässt sich sagen, dass wir die Wahl auf JUNG definitiv nicht bereut haben. Wir haben keine bemerkenswerte Probleme bei der Umsetzung feststellen können.

[307] s. Seite 66, Abschnitt 5.2.5 (5.2.5)

7. Vorstellung und Übersicht der entwickelten Software

Die im Zusammenhang mit dieser Diplomarbeit entwickelte Software *visualDependencies* verfügt insgesamt über vier Funktionalitäten, davon wurden im Vorfeld[308] zwei als Grundfunktionalitäten bestimmt.

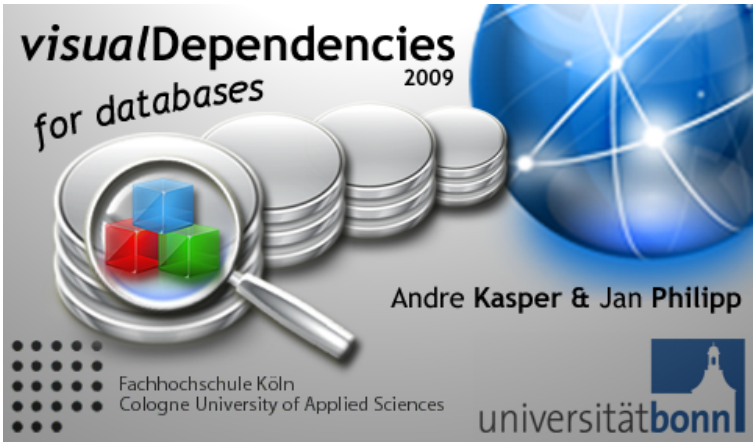


Abbildung 7.1: Logo und Ladebild der Anwendung *visualDependencies*

Auf eine Unterscheidung der Gewichtung der einzelnen Funktionalitäten wurde zugunsten der Darstellung verzichtet. Alle vier Funktionen sind gleichwertig über die Toolbar erreichbar, sobald eine Verbindung ausgewählt wurde.

Ein Willkommensbildschirm[309] begrüßt den Anwender und stellt die einzelnen Funktionalitäten kurz vor. Jede Funktionalität ist über ein eigenes Symbol identifizierbar.

In den folgenden vier Abschnitten werden kurz die einzelnen Funktionalitäten mit Beispielen erläutert. Als Basis wird ein Beispielschema „Flughafen“ verwendet, welches verschiedene Views, Trigger und Fremdschlüsselbeziehungen beinhaltet[310].

Zusätzlich beinhaltet die Anwendung noch zwei weitere, aber nicht-funktionale, Informationsquellen. Zum einen ist ein Protokollfenster verfügbar, welches Aufschluss darüber geben kann, wenn die Anwendung nicht wie gewünscht arbeitet oder Fehlermeldungen produziert. Darüberhinaus ist über *Menüleiste > Hilfe > Lizenzen* ein Fenster verfügbar, welches alle wichtigen und genutzten Drittherstellerkomponenten würdigt.

[308] s. Seite 18, Abschnitt 3 (3)
[309] s. Anhang A, S. 163, Abb. A.12 (A.12)
[310] s. Anhang B, S. 182, Listing 26 (26)

7.1. Anzeige der Verbindungsübersicht

Diese Anzeige wird beim Starten von *visualDependencies* direkt geöffnet und zeigt dem Benutzer einen Willkommensbildschirm[311], der eine Übersicht über die Funktionalitäten der Anwendung gewährt. In diesem Willkommensbildschirm werden die einzelnen Ansichten kurz vorgestellt und die Funktion dieser Ansicht wird kurz erklärt.

Die Verbindungsübersicht besteht aus zwei verschiedenen Bereichen. Im linken Bereich werden die bereits angelegten Datenbankverbindungen angezeigt. Neue Verbindungen können direkt über einen modalen Dialog erstellt werden. Die einzelnen Verbindungen können zu einer Baumansicht ausgeklappt werden. Die drei verschiedenen Datenbankobjekte, die im Rahmen dieser Diplomarbeit von Bedeutung sind (Tabellen, Views und Trigger), werden dabei alphabetisch zu der entsprechenden Verbindung angezeigt.

Die einzelnen Verbindungen verfügen über ein Kontextmenü, mit dem die Verbindung bearbeitet oder gelöscht werden kann. Weiterhin besteht die Möglichkeit, die vorhandene Verbindung zu aktualisieren, da beim Anlegen einer neuen Verbindung die einzelnen Daten intern gespeichert werden. Diese Art der Speicherung ermöglicht den Offline-Betrieb der Anwendung und die weiteren grafischen Ansichten können wesentlich schneller geladen werden. Es besteht außerdem noch die Möglichkeit, über dieses Kontextmenü direkt zu der Ansicht der Viewabhängigkeiten zu gelangen.

Um die einzelnen Objekte genauer zu betrachten, können die Tabellen- und Viewknoten nochmals weiter geöffnet werden. So erhält der Benutzer zu jeder Tabelle Informationen über die Spalten, die eingefügten Daten und die Integritätsbedingungen (Constraints), die auf dieser Tabelle definiert sind. Durch einen Doppelklick öffnet sich im rechten Bereich dieser Ansicht ein Reiter, der die gewünschte Information veranschaulicht. Entsprechendes gilt für die Anzeige der Views mit einer vierten Auswahl zur Darstellung der Definition (`SELECT`-Anweisung).

Anders als bei den Tabellen und Views kann der Baum bei den Triggern nicht weiter ausgeklappt werden. Die Informationen über die Definition des Triggers und die verwendeten oder veränderten Tabellen und Views werden hier in einer Ansicht dargestellt[312].

Die Verbindungsansicht dient zur Übersicht über die Tabellen, Views und Trigger eines Schemas und gewährt dem Benutzer einen ersten Überblick. Als Inspiration dieser Ansicht dienen die Softwareprogramme „Oracle SQL Developer[313]“ und „SQL Developer[314]“.

[311] s. Anhang A, S. 165, Abb. A.16 (A.16)
[312] s. Anhang A, S. 166, Abb. A.17 (A.17)
[313] s. Seite 33, Abschnitt 4.4.4 (4.4.4)
[314] s. Seite 35, Abschnitt 4.4.5 (4.4.5)

7.2. Anzeige der Viewabhängigkeiten

Die Anzeige der Abhängigkeiten der Views in der Datenbank ist eine Grundfunktionalität von *visualDependencies*. Sie zeigt jene Tabellen und Views an, die mindestens eine Abhängigkeit zu einem anderen Objekt haben. Nicht verwendete Tabellen werden im Graphen nicht angezeigt. Infolgedessen besteht die Darstellung aus einem Wald von einzelnen, freien Bäumen[315].

Zur Auswahl steht neben dem eigens entwickelten *Level-Layout* auch ein normales *TreeLayout* und ein *DAG-Layout*[316].

Die Knotenobjekte sind in den Farben Gelb und Blau verfügbar. Dabei symbolisieren die gelben Knoten Views und die blauen Knoten Tabellen. Sowohl Tabellen als auch Views beinhalten den Namen und die Anzahl der Spalten.

Jedes Knotenobjekt ist mit einem, über die rechte Maustaste erreichbaren, Kontextmenü ausgestattet. Damit kann dieser Knoten unmittelbar aus der Ansicht entfernt werden. Außerdem können weitere Informationen in der „Verbindungsübersicht“ angezeigt werden.

Die Farben der Kanten im Graphen sind farblich durch den Kontext ihrer Abhängigkeit bestimmt. Während grüne Kanten positive Abhängigkeiten repräsentieren, sind negative Abhängigkeiten durch rote Kanten hervorgehoben[317]. Die Kantenobjekte zeigen in einer kleinen Box die genauen Details der Abhängigkeit, wenn der Mauszeiger über sie gehalten wird[318].

Alle restlichen Funktionen wie die Auswahl der Layouts, das Speichern der Koordinaten oder die Begrenzung der dargestellten Objekte orientieren sich an den bereits beschrieben allgemeinen Funktionen[319].

[315] s. Anhang A, S. 164, Abb. A.13 (A.13)
[316] Ein DAG-Layout ist die gängige Abkürzung für das so genannte *Directed Acyclic Graph Layout*. Es ist ein alternatives Layout für azyklische Digraphen.
[317] s. Anhang A, S. 164, Abb. A.14 (A.14)
[318] Diese Technik nennt man auch *Hover-Effekt*.
[319] s. Seite 114, Abschnitt 7 (7)

7.3. Anzeige der Triggerabhängigkeiten

Die zweite Grundfunktionalität von *visualDependencies* ist die Anzeige der Abhängigkeiten der Trigger untereinander. In dieser Ansicht werden alle vorhandenen Trigger des ausgewählten Schemas angezeigt, auch die Trigger ohne Abhängigkeiten.

Für diese Ansicht werden verschiedene Layouts implementiert. So kann der Benutzer von *visualDependencies* entscheiden, wie die Triggerobjekte angezeigt werden sollen. Das Graphenframework JUNG bietet bereits einige fertige Layouts zur Auswahl an[320]. Insgesamt stehen in dieser Ansicht sechs verschiedene Layouts zur Verfügung.

Die Triggerobjekte und die Verbindungen (Kanten) werden in dieser Darstellung grün angezeigt, solange es zu diesem Objekt keine besondere Auffälligkeit gibt[321]. In einer Oracle-Datenbank kann ein Trigger ein *Mutating-Table*-Problem auslösen. Ein *Mutating-Table*-Problem ensteht, wenn der Trigger zeilenorientiert ist und auf die Tabelle, auf die er definiert ist, lesend oder manipulierend zugreift. Ein Trigger, der dieses Problem hervorrufen kann, wird in der Ansicht orange dargestellt und es gibt einen Hinweistext in diesem Objekt. Weiterhin wird eine rote Kante von diesem Trigger auf sich selbst angezeigt.

Die zweite Auffälligkeit, die gesondert dargestellt wird, ist die Rekursion unter den Triggern. Durch Data Manipulation Language (DML)-Befehle im Triggerrumpf können weitere Trigger angestoßen werden, was durchaus gewollt sein kann. Die Abhängigkeiten von Triggern können allerdings zu ungewollten rekursiven Aktivitäten führen[322]. Daher werden diese Trigger ebenfalls gesondert angezeigt. Die Farbe der Triggerobjekte ist rot und die rekursiven Kanten, die diese Trigger verbinden, ebenfalls.

Jedes Triggerobjekt kann durch ein Auswählmenü ein- oder ausgeblendet werden. Diese Funktion hilft bei größeren Schemata, einzelne Abhängigkeiten genauer zu untersuchen und dient dabei der Übersicht.

7.4. Anzeige des Entity-Relationship-Diagramms

Die Anzeige des *Entity-Relationship-Diagramms* ist eine weitere Funktionalität von *visualDependencies*. Im Gegensatz zu anderen Produkten beschränkt sich die Anzeige jedoch auf die lose Darstellung der Abhängigkeiten ohne weitere Details wie Kardinalität oder Art der Assoziation. Einzig und allein die Richtung der Abhängigkeit wird dargestellt, weshalb

[320] s. Seite 106, Abschnitt 6.4.6 (6.4.6)
[321] s. Anhang A, S. 166, Abb. A.18 (A.18)
[322] s. Seite 46, Abschnitt 5.1.2.4 (5.1.2.4)

die Visualisierung auch durch einen Digraph realisiert wird[323]. Aus diesem Grund ist die Bezeichnung des *einfachen Entity-Relationship-Diagramm* für eine Ausgabe[324] vorzuziehen.

Alle restlichen Funktionen, wie die Auswahl der Layouts, das Speichern der Koordinaten oder die Begrenzung der dargestellten Objekte, orientieren sich an den bereits beschrieben allgemeinen Funktionen[325].

[323] Die Leserichtung erfolgt auf Basis der exportierten Schlüssel.
[324] s. Anhang A, S. 165, Abb. A.15 (A.15)
[325] s. Seite 114, Abschnitt 7 (7)

8. Testen

Software hat in den letzten Jahren in vielen Bereichen eine weite Verbreitung gefunden. Dabei wird die Software immer mehr zur Realisierung oder Optimierung von Geschäftsprozessen eingesetzt. Die eingesetzte Software wird dabei aufgrund stetig steigender Anforderungen immer komplexer und funktionsreicher. Die Organisation oder der Ablauf des Betriebes ist von der Zuverlässigkeit der eingesetzten Software abhängig. Daraus ergeben sich hohe Qualitätsanforderungen an die eingesetzte Software. Um dieses Ziel zu erreichen, wird das systematische Testen und Prüfen der Software immer wichtiger.

> „. . . Das Testen von Software dient durch die Identifizierung von Defekten und deren anschließender Beseitigung durch das Debugging zur Steigerung der Softwarequalität. . . “[326]

Um eine gute Software zu entwickeln, werden auch im Rahmen dieser Diplomarbeit Testfälle entwickelt. Die Ergebnisse des Testens werden zur Fehlerbehebung verwendet, um die Qualität der Software zu steigern. Die Fehlerfreiheit kann das Testen nicht garantieren, weil das Testsubjekt für alle theoretischen Situationen von Ein- und Ausgaben überprüft werden müsste. Dies ist technisch nicht realisierbar, da die Anzahl der Eingaben, der möglichen Randbedingungen an die Umgebung des Testsubjektes und der Ausgaben in der Regel unendlich oder unverhältnismäßig hoch ist. Ausreichendes Testen kann die Wahrscheinlichkeit eines Fehlers im Programm verringern, aber es kann keine Fehlerfreiheit garantieren.

In diesem Projekt wird nach dem V-Modell getestet. Um eine testgetriebene Entwicklung durchführen zu können, sollte die Spezifikation der Anwendung bereits im Vorfeld bekannt sein. In diesem Projekt hat sich im Laufe der Zeit allerdings erst gezeigt, was die Anwendung genau leisten soll. Eine testgetriebene Entwicklung benötigt dadurch viele Ressourcen für ein Konfigurations- und Änderungsmanagement. Die Testfälle unter dem V-Modell können leicht modifiziert werden und daher kann schnell auf Änderungen der Anwendung reagiert werden. Durch diesen Vorteil fällt die Entscheidung gegen die testgetriebene Entwicklung und für das V-Modell.

Die einzelnen Teststufen nach dem V-Modell sind wie folgt aufeinander aufgebaut:

Komponententest Hier wird geprüft, ob die einzelnen Softwarebausteine den Vorgaben entsprechen.

[326] [Spillner 05], S. 11 K. 2.1.3 Softwarequalität

Integrationstest Dieser Test prüft, ob die einzelnen Softwarebausteine wie vorgesehen zusammenspielen oder ob es bei der Kommunikation zwischen den einzelnen Bausteinen Fehler gibt.

Systemtest Hierbei wird das System komplett geprüft beziehungsweise gegen die Programmspezifikation evaluiert und auf Fehlerwirkungen untersucht.

Abnahmetest Dieser Test geschieht bei der Auslieferung der Software an den Kunden[327].

Für einen *Komponententest* bietet sich das Java-Framework JUnit[328] als dynamiches Testverfahren an. Da die Anwendung in Java realisiert wird, können so die einzelnen Softwarekomponenten ohne weitere, spezielle Werkzeuge getestet werden. Die Umsetzung der einzelnen Testfälle in JUnit erweist sich als sehr einfach[329]. Der Integrationstest kann ebenfalls mit JUnit ohne große Probleme umgesetzt werden.

Der Systemtest erfolgt in diesem Projekt bereits während der Entwicklung. So können fertige Komponenten in die Anwendung integriert und auf ihre richtige Funktionsweise getestet werden. Es wird nach jedem neu hinzugefügten Softwarebaustein das komplette System getestet, um negative Auswirkungen auf bestehende Komponenten auszuschließen. Neben der offensichtlichen grafischen Ausgabe wird ein statisches Testverfahren (FindBugs/PMD[330]) eingesetzt, um Daten- und Kontrollflussanomalien aufzudecken. Ein weiteres dynamisches Testverfahren (Code Coverage[331]) wird ebenfalls verwendet, um nicht ausgeführten Quelltext oder die Komplexität (viele Verschachtelungen) einzelner Methoden aufzuzeigen.

Der Abnahmetest verläuft, genau wie der Systemtest, in diesem Projekt schon während der Entwicklung. Bereits fertiggestellte Softwarebausteine werden dem Auftraggeber zur Verfügung gestellt. So kann auch schon sehr frühzeitig auf Kundenwünsche reagiert werden. Ein kompletter Abnahmetest entfällt in diesem Fall am Ende der Entwicklung.

Das Programm wird zusätzlich mit weiteren Testverfahren überprüft. Diese Testverfahren sind zum Beispiel Crashtests, Interoperabilitätstest, Installationstest oder Oberflächentest[332]. Diese Testverfahren sollen zeigen, dass die Softwareanwendung auch in Ausnahmesituationen weiterhin funktioniert. Durch diese zusätzlichen Tests kann gezeigt werden, wie belastbar die Softwareanwendung ist.

[327] Ein Systemtest aus der Sicht des Kunden.
[328] vgl. [Beck 08]
[329] s. Seite 134, Abschnitt 8.4.2 (8.4.2)
[330] s. Seite 125, Abschnitt 8.3.1 (8.3.1)
[331] s. Seite 132, Abschnitt 8.4.1 (8.4.1)
[332] s. Seite 136, Abschnitt 8.5 (8.5)

Neben den hier vorgestellten Testverfahren werden im Vorfeld Testfälle für die Auswahl eines geeigneten SQL Parsers und eines Graphenframeworks ermittelt und spezifiziert. Diese Testfälle sind notwendig um entscheiden zu können, ob der ausgewählte Softwarebaustein für die Anwendung gute Ergebnisse erzielt. Somit können Fehlerwirkungen aus externen Softwarebausteinen weitestgehend ausgeschlossen werden.

Bei der Auswahl eines geeigneten Graphenframeworks bietet sich Prototyping an[333]. Durch eine kleine, einfache Implementierung des Softwarebausteins kann anhand einer grafischen Ausgabe entschieden werden, ob dieser Baustein den Anforderungen entspricht. Dieser entwickelte Prototyp kann auch dem Auftraggeber kurz vorgestellt werden. So kann sichergestellt werden, dass die Visualisierung den Vorstellungen des Auftraggebers entspricht.

Um sicherzustellen, dass der Parser exakte Ergebnisse liefert, werden Testfälle entwickelt. Durch die Auswertung der Testergebnisse kann entschieden werden, ob der Parser den funktionalen Anforderungen[334] entspricht und verwendet werden kann. Auch hier wird ein Prototyp erstellt und gegen die Spezifikation getestet[335].

8.1. Exkurs: Die Psychologie des Testens

In der Anwendung werden, bis auf den Test auf Robustheit, ausschließlich Entwicklertests durchgeführt. Es ist allgemein bekannt[336], dass die große Schwäche darin besteht, dass ein Entwickler blind gegenüber seinen eigenen Fehlern ist. So besteht die Gefahr, dass er sinnvolle Testfälle entfallen lässt, da er zu optimistisch seinem eigenen Programmcode gegenüber eingestellt ist.

Es bietet sich an, das Testen einer externen, unabhängigen Instanz zu überlassen. Dies ist für dieses Projektes nicht möglich, da der Aufwand und die Kosten für die Erstellung eines solchen Testteams den Rahmen des Projektes weit übersteigen würde. Um solch eine „Blindheit“ gegenüber den eigenen Fehlern zu vermeiden, werden die Testfälle bereits vor der Implementierung spezifiziert. So wird vor der Entwicklung festgelegt, wie die einzelnen Komponenten reagieren sollen.

[333] s. Seite 18, Abschnitt 3 (3)
[334] s. Seite 45, Abschnitt 5.1.2 (5.1.2)
[335] s. Seite 122, Abschnitt 8.2 (8.2)
[336] vgl. [Spillner 05], S. 33 K. 2.3 Psychologie des Testens

8.2. Testfälle für die Auswahl eines Parsers

Um die Abhängigkeiten der Datenbankobjekte bestimmen zu können, wird ein Structured Query Language (SQL)-Parser benötigt, der die unterschiedlichen Anweisungen einer View und die Procedural Language/SQL (PL/SQL) für die Trigger verstehen und verarbeiten kann. Die Syntax und Grammatik von PL/SQL ist stark unterschiedlich in Beziehung zu dem SQL einer View[337].

Für die Auswahl des SQL-Parsers werden zunächst Testfälle spezifiziert. Es wird versucht, mit einem Testfall möglichst viele verschiedene Anweisungen abzudecken. Dabei werden die Testfälle für die Views mit SQL und die Testfälle für die Trigger mit PL/SQL differenziert und einzeln betrachtet. Daher kann eine Auswertung der Testergebnisse für jeden Bereich übersichtlicher gestaltet werden.

Um die verschiedenen Parser testen zu können, werden Prototypen erstellt und durch die Testfälle verifiziert. Es wird hierfür ein Testrahmen entwickelt, der für alle getesteten Parser benutzt werden kann. Daher können die Tests relativ schnell durchgeführt werden und es entstehen keine großen Testressourcen.

8.2.1. Parsen der einzelnen SQL-Anweisungselemente

An dieser Stelle werden die Testfälle noch nicht für die Unterscheidung zwischen einer positiven oder negativen Abhängigkeit spezifiziert. Dies ist ein Schritt der eigenen Implementierung und dient nicht zur Überprüfung der Funktionsweise des Parsers. Um den Parser auf diese Funktionsweise zu überprüfen, sind weitere, dynamische Testverfahren nötig, die die eigene Implementierung ausgiebig testen[338].

Für jeden Testfall wird eine Liste mit vorhandenen Tabellen erstellt. Diese Liste soll im späteren Verlauf der Entwicklung durch eine reale Liste anhand der Rückgaben der Abfragen an das Data Dictionary ersetzt werden. In diesem Teststadium werden die Listen von Hand erstellt, wobei auch die *SELECT*-Abfragen zunächst nicht aus einer realen View stammen. Sie dienen nur zur Überprüfung der Funktionsweise des Parsers und stellen zu diesem Zeitpunkt noch kein reales Datenbankschema dar.

Die Liste mit den Tabellen wird für die Testfälle auch um nicht vorhandene Tabellen der *SELECT*-Anweisung ergänzt. Es wird eine Liste mit 20 Tabellen erstellt. Anhand der einzelnen Testfälle wird ein Ergebnis abgeleitet, wieviele und vor allem welche Tabellen gefunden werden dürfen.

[337] s. Seite 43, Abschnitt 5.1 (5.1)
[338] s. Seite 134, Abschnitt 8.4.2 (8.4.2)

Die Mengenoperationen werden zu einem Testfall zusammengeführt. Damit können alle möglichen Operationen dieser Art mit einem Testfall abgedeckt werden[339]. Weiterhin wird ein Testfall spezifiziert, um Unterabfragen in der *FROM*- beziehungweise *WHERE*-Klausel zu testen[340]. Ein dritter Testfall wird spezifiziert, um das Verhalten des Parsers bei einer Gruppenfunktion mit vorhandenen *HAVING*-Komponenten zu evaluieren[341]. Auch hier ist eine Unterabfrage möglich und sollte interpretiert und entsprechend geparst werden. Als vierten und letzten Testfall werden hier die *JOIN*-Operatoren untersucht und getestet[342]. Die mit einem *JOIN* verknüpften Tabellen sollten ebenfalls gefunden und in der Liste gespeichert werden.

8.2.2. Parsen von PL/SQL

Für das Testen, ob der Parser auch PL/SQL sowohl interpretieren als auch analysieren kann, ohne eine Programmausnahme zu werfen, reicht ein Testfall aus. Dabei ist es für die Auswahl des Testfalls nicht entscheidend, ob es sich um einen Data Manipulation Language (DML)-, um einen `INSTEAD OF`[343]- oder um einen Data Definition Language (DDL)-Trigger handelt. Die Art eines Triggers wird durch eine Anfrage an das Data Dictionary erschlossen und muss beim Parsen nicht berücksichtigt werden.

Dieser Testfall wird so spezifiziert, dass der Parser die vorhandenen Tabellen erkennt und in einer Liste speichert. Eine Fallunterscheidung, ob der Trigger nur beispielsweise bei einem `INSERT` oder `UPDATE` angestoßen wird, ist eine Aufgabe der späteren Analyse der geparsten Daten und wird daher in diesem Zusammenhang nicht betrachtet. Um dies auswerten zu können, werden andere, dynamische Testverfahren betrachtet, um diese Funktion zu gewährleisten[344].

In dem spezifizierten Testfall[345] wird als Ergebnis erwartet, dass der Parser zwei Tabellen findet (*tablename1* und *tablename2*). Weitere Tabellen dürfen nicht gefunden werden. Dafür wird ebenfalls wie bei den Testfällen für die Views eine Liste mit verschiedenen Tabellennamen spezifiziert und dem Parser beim Starten mitgegeben. Werden ausschließlich diese beiden Tabellen gefunden, kann eine richtige Funktionsweise des Parsers gewährleistet werden.

[339] s. Anhang B, S. 193, Listing 29 (29)
[340] s. Anhang B, S. 194, Listing 30 (30)
[341] s. Anhang B, S. 194, Listing 31 (31)
[342] s. Anhang B, S. 194, Listing 32 (32)
[343] Ein `INSTEAD OF`-Trigger führt den Quellcode des Rumpfes anstelle des definierten Ereignisses aus (Oracle-spezifisch).
[344] s. Seite 134, Abschnitt 8.4.2 (8.4.2)
[345] s. Anhang B, S. 194, Listing 33 (33)

Nach den ersten Testdurchläufen wurde festgestellt, dass alle getesteten Parser Probleme mit PL/SQL-spezifischen Symbolen wie einem Semikolon haben. Ein Semikolon ist in der Grammatik der Parser meist als Delimiter der einzelnen SQL-Anweisungen implementiert. Durch reguläre Ausdrücke werden die entsprechenden Anweisungen verändert und die problematischen Symbole werden durch Konstanten ersetzt. Nach einigen Anpassungen können die Parser somit auch für PL/SQL eingesetzt werden.

8.2.3. Fazit

Diese Testfälle werden alle spezifiziert, um die Vorgehensweise der vorhandenen Parser zu testen und zu evaluieren. Es ist obligatorisch, dass der Parser alle Testfälle besteht. Nur so können wir sicherstellen, dass die Anwendung nach Abschluß der Entwicklung funktioniert.

Weitere Testfälle, die das Verhalten des Parsers überprüfen, ob die Tabellen oder Views entsprechend mit einer positiven oder negativen Beziehung zu der eigentlichen View in Verbindung stehen, gehören nicht zur Evaluierung des vorliegenden Parsers. Die hierfür benötigten Regeln und Grammatiken werden später implementiert und in weiteren Tests auf ihre exakte Funktionsweise überprüft[346].

Wenn ein Parser nicht alle Testfälle besteht, kann er in diesem Projekt nicht eingesetzt werden. Es muss also ein Parser gefunden werden, der alle Testfälle ohne Probleme besteht, da ansonsten eine Eigenentwicklung vorzuziehen ist.

8.3. Statisches Testen

In einem statischen Test wird das Testobjekt nicht zur Ausführung gebracht, sondern nur als „solches“ direkt untersucht. Es gibt verschiedene statische Testverfahren. Zum Teil werden formelle und informelle Texte durch Mensch oder werkzeugunterstützt betrachtet und auf mögliche Fehlerwirkungen hin untersucht.

Eine Möglichkeit für statisches Testen ist das Review. Bei einem Review wird ein Dokument untersucht und auf Fehler überprüft. Ein solches Dokument kann beispielsweise die Anforderung sein. Da in diesem Projekt die Aufgabenstellung erst im Laufe der Zeit entstanden ist, wird auf die Untersuchung dieses Dokumentes verzichtet.

[346] s. Seite 134, Abschnitt 8.4.2 (8.4.2)

Weitere, eher formellere Dokumente wie Java-Klassen können werkzeuggestützt untersucht werden. Durch die Entwicklungsumgebung Eclipse[347] wird eine Verletzung der Syntax der Programmiersprache als Fehler angezeigt und kann direkt während der Entwicklung behoben werden. Daher sind weitere Testfälle in Form von Reviews nicht notwendig. Allerdings genügt dieses Testverfahren nicht als alleiniges Testkriterium. Drei weitere, statische Testverfahren bieten wesentlich mehr Möglichkeiten, Fehler aufzudecken. Das sind die Datenfluss-, Kontrollflussanalyse und die Prüfung auf Einhaltung von Konventionen und Standards. Diese drei Verfahren können werkzeuggestützt den vorliegenden Programmcode untersuchen und mögliche Fehlerwirkungen aufdecken.

So kann die Datenflussanalyse dazu eingesetzt werden, um Anomalien in der Verwendung der Daten aufzuzeigen. Es handelt sich um eine Anomalie, wenn die Verwendung der Daten zu einer Fehlerwirkung führen kann, aber nicht zwingend muss. Anomalien stellen ein mögliches Risiko dar, wobei allerdings im Einzelfall entschieden werden muss, ob es sich tatsächlich um ein Risiko oder um eine gewollte Ausnahme handelt.

Die Kontrollflussanalyse kann andererseits dazu benutzt werden, um Anomalien im Programmfluss zu erkennen und aufzudecken. Kontrollflussanomalien können unter anderem ungewollte Sprünge aus Schleifen sein, unerreichbarer Code einer Methode oder aber Programmstücke, die mehrere Ausgänge haben.

Dabei dient die Prüfung auf Einhaltung von Konventionen und Standards eher dem Zweck der Übersichtlichkeit. So kann sich ein Programmierer, der neu in das Projekt einsteigt, schneller einarbeiten.

Für die Entwicklungsumgebung Eclipse gibt es zwei kostenlose Plugins[348], die diese drei Testfahren vereinen, PMD[349] und FindBugs[350]. Diese Plugins wurden in diesem Projekt eingesetzt, um neben der statischen Analyse durch den Compiler die drei weiteren statischen Testverfahren auszuführen.

8.3.1. FindBugs/PMD

Diese beiden Eclipse-Plugins unterscheiden sich nur in ein paar Kleinigkeiten. PMD hat zum Beispiel eine feinere Abstufung der Fehlermeldungen (PMD besitzt insgesamt fünf Abstufungen, FindBugs hingegen drei). Der wesentliche Unterschied ist, dass PMD die letzte Stufe dafür benutzt, um ausschließlich Datenflussanomalien aufzuzeigen, während

[347] s. [Eclipse Foundation 09b]
[348] Diese Plugins sind auch für andere Entwicklungsumgebungen verfügbar oder als lauffähige Anwendung.
[349] s. [InfoEther 09]
[350] s. [Hovemeyer 09]

FindBugs diese Anomalien je nach Schweregrad in die drei vorhandenen Fehlerstufen einordnet. Eine explizite Filterung der Datenflussanomalien ist somit nicht möglich.

Ein weiterer Unterschied besteht in der Konsolenausgabe der beiden Plugins. Während FindBugs ausschließlich die Anzahl der gefunden Fehler und Warnungen im gesamten Projekt ausgibt[351], bietet PMD zusätzlich zwei weitere Metriken an. Zunächst teilt PMD die Anzahl der Fehler auf die einzelnen Pakete auf. Dadurch wird ersichtlich, wo sich eine größere Anzahl von Fehlern im Programm befindet. Meist lohnt es sich, diese Stellen genauer zu untersuchen, da dort die Wahrscheinlichkeit, weitere Fehler zu finden, höher ist. Weiterhin zeigt PMD die Anzahl der gefunden Fehler in einer Klasse pro Zeile Quellcode und auch pro Methode auf. Durch diese zusätzlichen Metriken bietet PMD eine bessere Möglichkeit die Fehlerhäufigkeit einzelner Programmcodeabschnitte genauer zu untersuchen.

Es werden beide Plugins ausgeführt, wobei an dieser Stelle wegen der besseren Übersicht nur auf PMD eingegangen wird. Das Plugin lässt sich ganz einfach in Eclipse direkt starten und ausführen. Anschließend erstellt dieses Plugin einen Fehlerbericht in Form einer Liste, wobei die Ausgabe in einem separaten Eclipse-Fenster erfolgt. Dieser Fehlerbericht kann zusätzlich exportiert werden. Der Export erfolgt in diversen Dateiformaten wie Hypertext Markup Language (HTML), Textdatei (TXT)[352], Character Separated Values (CSV) oder Extensible Markup Language (XML).

Die typischen Meldungen, die von PMD aufgezeigt werden, sind beispielsweise:

- Vermeidung von leeren (`Try-`)`Catch`-Blöcken
- Variablennamen-Konventionen (zu kurz; erster Buchstabe klein; Großschreibung nur bei Konstanten)
- Deklarierung lokaler Konstanten (finale Variablen)
- Vermeidung von mehreren Ausgängen einer Methode
- nicht initialisierte Verwendung einer Variablen

Dabei können die Fehlermeldungen in drei Kategorien eingeteilt werden, die Kontrollfluss-, die Datenflussanalyse und die Prüfung zur Einhaltung der Konventionen und Standards.

8.3.1.1. Datenflussanalyse

Die Datenflussanalyse dient zur Kontrolle bei der Verwendung der Daten. Während der Ausführung eines Programms werden die Daten in Va-

[351] s. Anhang A, S. 178, Abb. A.49 (A.49)
[352] s. Anhang C, S. 192, Listing 27 (27)

riablen geschrieben, gelesen und weiterverarbeitet. Dabei werden drei verschiedene Zustände unterschieden[353]:

definiert (d)	die Variable erhält einen Wert zugewiesen
referenziert (r)	die Variable wird gelesen
undefiniert (u)	die Variable hat einen undefinierten Wert (nicht initialisiert)

Tabelle 5: Datenflussanalyse – Verwendung einer Variablen

Bei der Verwendung der Variablen während der Ausführung eines Programms lassen sich drei entsprechende Datenflussanomalien unterscheiden[354]:

ur-Anomalie	die Variable wurde nicht initialisiert und wird gelesen (Anomalie, die auf jeden Fall behoben werden muss)
du-Anomalie	die Variable erhält einen Wert (d), der ungültig wird, bevor die Variable gelesen wird (beispielsweise Ende einer Methode)
dd-Anomalie	die Variable erhält einen Wert (d) und bekommt ein zweites Mal einen Wert zugewiesen (d), ohne zwischenzeitliches Lesen

Tabelle 6: Datenflussanalyse – Anomalien

Diese drei Anomalien lassen sich gut klassifizieren, da eine *ur*-Anomalie[355] durch Lesen eines nicht identifizierbaren Wertes unkontrollierbare Sprünge in der Anwendung hervorrufen kann, die eventuell zu einem Absturz der Anwendung führen. Währenddessen deuten die *du*-Anomalien darauf hin, dass der Programmcode an dieser Stelle nicht fertig implementiert wurde oder diese Variablen nach einer Überarbeitung nicht mehr benötigt werden, da keine weitere Verwendung stattfindet. Die *dd*-Anomalie gibt an, dass die erste Zuweisung eventuell eingespart werden kann und gehört dadurch in die Kategorie der Performanz-Warnungen, da jede Zuweisung Rechenleistung benötigt. Allerdings kann es auch darauf hinweisen, dass die Verwendung der Variablen nach der ersten Zuweisung nicht implementiert wurde und muss daher genauer untersucht werden.

Das Plugin untersucht den vorliegenden Programmcode und stellt die gefundenen Fehler in einer Liste zusammen. Allerdings sollte der Programmierer entscheiden, ob es sich tatsächlich um einen gefundenen Fehler handelt oder um eine gewollte Definition der Variablen. Eine werkzeuggestützte Untersuchung des Quellcodes ist hilfreich beim Fehler finden. So erhält der Entwickler eine Übersicht, die aber im Einzelnen genauer untersucht werden sollte.

[353] vgl. [Spillner 05], S. 97 K. 4.2.3
[354] vgl. [Spillner 05], S. 97 K. 4.2.3
[355] s. S. 127 Tabelle 6 (6)

PMD kann diese Fehler und Warnungen entdecken, allerdings handelt es sich hierbei nicht immer um einen Fehler oder eine Warnung. Es ist beispielsweise (je nach Programmierstil) durchaus üblich, Variablen im Voraus zu deklarieren und gegebenenfalls mit einem Wert zu initialisieren. Das Plugin zeigt dieses Vorinitialisieren einer Variablen als *dd*-Anomalie[356] auf, auch wenn dieses Vorgehen in der Praxis durchaus üblich ist[357].

```
1 Level level = null;
2 try {
3   level = Level.parse(Configuration.props.getProperty("loggerlevel").toUpperCase());
4 } catch (final Exception e) {} finally {
5   if (level == null) {
6     level = Level.SEVERE;
7   }
8 }
```

Listing 16: PMD-Warning DD-Anomalie (gekürzt)

Ein anderer Fehler, der auch relativ häufig angezeigt wird, ist die *du*-Anomalie. Dabei wird eine Variable in einer Methode angelegt und nicht mehr gelesen, bevor sie ungültig wird und das Ende der Methode erreicht ist. Bei einem verschachtelten Ausgang einer Methode zum Beispiel durch eine `IF-THEN-ELSE`- oder `SWITCH-CASE`-Anweisung wird diese Variable nicht in jeder Verzweigung benötigt. Das Plugin durchläuft die einzelnen Programmpfade und entdeckt dabei, dass diese Variable in einem der vielen Pfade nicht benötigt wird und zeigt diese Meldung auf[358]. Der Entwickler muss also bei diesen Meldungen von Fall zu Fall entscheiden, ob es sich tatsächlich um einen Fehler handelt.

```
1 try {
2   final LogRecord record = this.records.get(rowIndex);
3   switch (columnIndex) {
4   ..
5   case 3:
6     return record.getSourceMethodName();
7 } catch (final Exception e) {
8 }
9 return null;
```

Listing 17: PMD-Warning DU-Anomalie (gekürzt)

8.3.1.2. Kontrollflussanalyse

Die Kontrollflussanalyse dient zur Kontrolle der Programmabläufe. Somit wird hier das Programm nicht auf Datenebene untersucht, sondern auf die

[356] s. S. 127 Tabelle 6 (6)
[357] s. S. 128 Listing 16 (16)
[358] s. S. 128 Listing 17 (17)

Struktur und die internen Abläufe. Eine Anomalie entsteht durch Sprünge aus Schleifen heraus, durch ein Programmstück mit mehreren Ausgängen oder durch unerreichbaren Programmcode. Auch hierbei ist zu beachten, dass nicht jede Meldung, die dieses Plugin aufzeigt, tatsächlich ein Fehler ist.

Unter anderem ist es notwendig, in einer Methode zwei verschiedene Ausgänge zu definieren. So kann eine Überprüfung eines *NULL*-Wertes zu einer anderen Rückgabe führen als ein gelesener Wert einer Variablen[359]. Um einen Fehler bei einer Rückgabe eines `NULL`-Wertes zu vermeiden, ist es sogar sinnvoll, diese beiden Methodenausgänge zu definieren. Diese Meldung von PMD wurde untersucht, allerdings nicht verbessert.

```
1 if (result == null) {
2   return true;
3 }
4 return Boolean.valueOf(result.toString()).booleanValue();
```

Listing 18: PMD-Warning mehrere Ausgänge einer Methode (gekürzt)

Ein weiterer Fehler, der von PMD gefunden wird, ist das Fehlen eines privaten Konstruktors einer Klasse mit ausschließlich statischen Methoden (statische Klasse)[360]. Der private Konstruktor verhindert somit das Erzeugen einer Instanz dieser Klasse. Dies ist notwendig, da durch eine Instanz einer solchen Klasse auf Objekte zugegriffen werden kann, die eventuell nicht initialisiert wurden und somit das Programm zum Absturz bringen können. Diese Fehlermeldung gehört auch in den Bereich der Kontrollflussanomalien, da das Programm an dieser Stelle mehrere Eingänge einer Klasse aufweist, die nicht gewollt sind.

```
1 private Configuration() {}
```

Listing 19: PMD-Warning statische Klasse mit privatem Konstruktor (gekürzt)

Neben den bereits erwähnten Fehlermeldungen zeigt PMD auch Schleifen an, die zu ungewollten Sprüngen führen können. Daher meldet PMD auch das Erzeugen neuer Objektinstanzen in einer Schleife, da ein Fehler beim Erstellen des Objektes die Schleife nicht beenden würde[361]. Allerdings ist meist nur zur Laufzeit bekannt, wieviele Objekte erzeugt werden müssen, da die Listen dynamisch erstellt werden. Auch wenn die Anzahl der Objekte bekannt ist, ist es nicht sinnvoll, das Erzeugen für jedes einzelne Objekt von Hand fest in den Programmcode zu schreiben. Dies führt zu einem „aufgeblähten“ und unübersichtlichen Quellcode. Es liegt also in der Hand des Programmierers das Erzeu-

[359] s. S. 129 Listing 18 (18)
[360] s. S. 129 Listing 19 (19)
[361] s. S. 130 Listing 20 (20)

gen der Objekte durch Abfangen des Fehlers oder robuste Verwendung von fehlerhaften „Objektreferenzen“ sicher zu machen, so dass kein Schleifenabbruch stattfindet.

```
1 for (final DatabaseTrigger root : roots) {
2   result.add(new DatabaseTriggerGraphTree(this, root));
3 }
```

Listing 20: PMD-Warning Objekterzeugung in Schleifen (gekürzt)

8.3.1.3. Prüfung auf Einhaltung der Konventionen und Standards Diese Art der Überprüfung dient der Übersichtlichkeit des vorliegenden Programmcodes und deckt keine Fehler auf, die zum Absturz führen können. Es wird kaum Zeit benötigt und somit auch so gut wie keine Testkosten, diese Art der Überprüfung durchzuführen. Die Konventionen und Richtlinien sollten in einem Projekt nur dann zum Einsatz kommen, wenn sie auch werkzeuggestützt überprüft werden können[362]. So entsteht kein weiterer Testaufwand und diese Prüfung ist sehr schnell abgeschlossen. Das Plugin PMD überprüft den vorliegenden Quellcode auch auf die Konventionen und Standards, die in JAVA Verwendung finden.

So gibt es eine Richtlinie, dass Variablen ausschließlich groß geschrieben werden, wenn sie als Konstante definiert werden[363]. Eine Konstante wird daher mit dem Schlüsselwort `final` deklariert. Das bedeutet, dieser Konstante kann im weiteren Programmverlauf kein anderer Wert mehr zugewiesen werden. Diese Variablen können aber auch dazu dienen, die Einstellungen des Programms zu speichern. Werden die Einstellungen des Programms auf der Benutzerseite verändert, so wird auch diesen Variablen neue Werte zugewiesen. Somit sind diese Variable nur solange konstant, bis Einstellungsänderungen zur Laufzeit vorgenommen werden. Daher kann so eine Variable nicht als Konstante deklariert werden.

```
1 /**
2 * global logger identification name
3 */
4 public static String LOGGER = "dbvis";
5 ..
6 Configuration.LOGGER = Configuration.props.getProperty("loggername");
```

Listing 21: PMD-Warning Objektname (gekürzt)

[362] vgl. [Spillner 05], S. 96 K. 4.2.2
[363] s. S. 130 Listing 21 (21)

8.3.2. Fazit

Alle aufgeführten, statischen Testverfahren wurden in unserem Projekt eingesetzt. Durch die Unterstützung von Testwerkzeugen konnten diese Verfahren mit geringem Zeitaufwand durchgeführt werden und haben gute Ergebnisse geliefert. Die Werkzeuge zeigen dabei allerdings mehr Meldungen auf, als es tatsächlich Fehler gibt. Daher wurden die angezeigten Meldungen von uns genauer analysiert und bewertet.

Durch diese Verfahren kann ein wesentlicher Teil der Fehler behoben werden, die durch die auffällige, unregelmäßige Verwendung von Variablen oder ungewollten Programmabläufen hevorgerufen werden.

Die statischen Testverfahren reichen als alleiniges Testkriterium nicht aus. Es können keine Fehler, die erst zur Laufzeit der Anwendung entstehen, aufgezeigt werden. Diese Fehler können nur durch dynamische Testverfahren ermittelt und entdeckt werden. Daher haben wir in unserem Projekt neben den hier erwähnten statischen noch weitere dynamische Verfahren eingesetzt[364], um andere Fehlerwirkungen aufzudecken.

8.4. Dynamisches Testen

Im Gegensatz zum statischen Testen[365] wird bei einem dynamischen Testverfahren das Testobjekt während der Ausführung untersucht. Es muss daher eine lauffähige Anwendung vorliegen. Um auch Objekte auf den unteren Stufen[366] testen zu können, wird das Objekt in einen entsprechenden Testrahmen eingebettet. Die benötigten Eingabedaten werden durch Testtreiber der Anwendung zugeführt, wobei Platzhalter während der Ausführung das Ein- und Ausgabeverhalten an weitere Programmteile simulieren[367].

Ist das Testobjekt in so einen Testrahmen eingebettet, ist diese Komponente lauffähig und kann getestet werden. Die dynamischen Testverfahren können in zwei Kategorien unterteilt werden, die Blackbox- und Whitebox-Verfahren.

Bei einem Blackbox-Verfahren ist der innere Aufbau und der Quellcode des Testobjektes nicht bekannt. Hier wird das Verhalten des Testobjektes von aussen beobachtet. Eine Steuerung des Verhaltens des Objektes ist nur beschränkt möglich, da die interne Struktur nicht bekannt ist und die Wahl der Eingabedaten meist relativ wenige Auswirkungen hat. Die Testfälle werden ausschließlich anhand der Systemspezifikation erstellt[368]. Die

[364] s. Seite 131, Abschnitt 8.4 (8.4)
[365] s. Seite 124, Abschnitt 8.3 (8.3)
[366] s. Seite 119, Abschnitt 8 (8)
[367] vgl. [Spillner 05], S. 106 K. 5
[368] vgl. [Spillner 05], S. 108 K. 5

Blackbox-Verfahren sind eher dazu geeignet, die Tests auf den höheren Stufen durchzuführen.

Bei den Whitebox-Verfahren ist der Quellcode und daher auch der innere Aufbau des Testobjektes bekannt. Die Testfälle können anhand der Programmstruktur abgeleitet werden. Auch ist ein Eingriff in den Quellcode möglich, um eventuell eine Programmausnahme hervorzurufen und um dadurch die Reaktion der Anwendung im Fehlerfall testen zu können. Die Whiteboxverfahren eignen sich im Gegensatz zu den Blackbox-Verfahren für die unteren Teststufen wie Komponenten oder Integrationstest, da es nicht sinnvoll ist, ein komplettes System anhand des Quellcodes zu testen[369].

In diesem Projekt werden neben weiterführenden[370] auch zwei dynamische Testverfahren eingesetzt. Im ersten Verfahren werden sowohl die Komponenten im Einzelnen als auch die Integration beziehungsweise das Zusammenspiel mehrerer Komponenten untersucht und getestet[371]. Zusätzlich dient Code Coverage[372] als weiteres Verfahren zum Testen der Anweisungsüberdeckung. Hierbei wird das System als Ganzes untersucht.

8.4.1. Code-Coverage

Für den Test der Anweisungsüberdeckung[373] gibt es ein Plugin für die Laufzeitumgebung Eclipse, das „EclEmma 1.4.1 Java Code Coverage for Eclipse“[374]-Plugin. Das Plugin ist kostenlos und fällt unter die *Eclipse Public License*.

Ein Anweisungsüberdeckungstest ist ein einfaches, kontrollflussbasiertes Testverfahren, dass versucht, möglichst jede Anweisung im Quellcode der Anwendung einmal auszuführen. Dabei wird der Grad der Überdeckung ermittelt und dient als Bewertung des Testlaufs.

Um den Test auszuführen, wird die Anwendung über das Plugin in Eclipse gestartet und ausgeführt. Während der Ausführung wird versucht, alle verfügbaren Funktionen der Anwendung auszuführen, die über die Oberfläche erreichbar sind. Das Plugin läuft im Hintergrund mit und markiert jede Zeile des ausgeführten Quellcodes[375]. Es kann auch mehrmals ausgeführt und die einzelnen Sessions können miteinander vereint werden.

[369] vgl. [Spillner 05], S. 109 K. 5
[370] s. Seite 136, Abschnitt 8.5 (8.5)
[371] s. Seite 134, Abschnitt 8.4.2 (8.4.2)
[372] s. Seite 132, Abschnitt 8.4.1 (8.4.1)
[373] Dieser Test wird auch C0-Test genannt.
[374] s. [Hoffmann 08b]
[375] vgl. [Hoffmann 08a]

Nach dem Beenden der Anwendung berechnet das Plugin den Anweisungsüberdeckungsgrad[376]:

$$Anweisungsberdeckung\,(C0) = \frac{Anzahl\,durchlaufene\,Anweisungen}{Gesamtzahl\,Anweisungen} * 100\%$$

Es findet eine Ausgabe in Eclipse in einem extra eingefügten Fenster statt[377]. Hier wird die Gesamtzahl der Anweisungen den ausgeführten Anweisungen gegenübergestellt und die ermittelte Überdeckung prozentual dargestellt. Es erfolgt eine Ausgabe für das gesamte Projekt, allerdings kann die Anweisungsüberdeckung auch nach einzelnen Paketen unterschieden werden. Das Plugin markiert zusätzlich jede ausgeführte Zeile direkt im Quellcode grün, während nicht ausgeführte Zeilen rot markiert werden. Daher können nicht ausgeführte Quellcode-Zeilen schnell gefunden werden, und der Programmierer kann im einzelnen entscheiden, ob eine Anpassung notwendig ist.

Dieses Plugin verfügt weiterhin über eine Exportfunktion. Der Bericht, den dieses Plugin anfertigt, kann dabei in verschiedene Dateiversionen (beispielsweise als *Extensible Markup Language (XML)* oder *Hypertext Markup Language (HTML)*[378]) exportiert werden.

Insgesamt erreicht Code-Coverage eine Abdeckung von 73,4%. Das kommt daher, dass zum Beispiel für das Betriebssystem „Mac OS X“ einige Fallunterscheidungen für die grafische Darstellung implementiert werden. Weiterhin werden abstrakte Klassen implementiert, die als Weiche dienen, um neue Funktionen (wie zum Beispiel neue Datenbanktreiber) leichter einbauen zu können. Vor allem werden nicht alle Anweisungen in `Try-catch-finally`-Blöcken in einem Anweisungstest überdeckt. Dies ist dadurch begründet, das teilweise unwahrscheinliche und nicht provozierbare Fehlerzustände abgefangen werden müssen. Allerdings sind diese Anweisungen obligatorisch, da die Anwendung ansonsten in einem Fehlerfall abstürzen würde.

Diese Anweisungen werden in der Regel nicht ausgeführt, führen aber dazu, dass eine 100%-ige Anweisungsüberdeckung nicht möglich ist. Das bedeutet, dass der Programmierer diese Zahlen genauer untersuchen sollte, bevor er Änderungen im Quellcode vornimmt.

[376] vgl. [Spillner 05], S. 146
[377] s. Anhang A, S. 177, Abb. A.46 (A.46)
[378] s. Anhang A, S. 178, Abb. A.47 (A.47)

8.4.2. JUnit

JUnit[379] ist ein Testframework für Java und kann als Testwerkzeug eingesetzt werden, um die einzelnen Komponenten eines Java-Projektes zu testen.

Eine Testklasse unter JUnit besteht aus fünf verschiedenen Teilen[380]:

setUpBeforeClass ist eine Methode, die vor dem eigentlichen Start der Testfälle ausgeführt wird. Hier können alle benötigten Treiber geladen und die erforderlichen Einstellungen vollzogen werden, die sich während des gesamten Testvorgangs nicht verändern.

setUp ist eine Methode, die vor jedem einzelnen Testfall ausgeführt wird. Hier können die Einstellungen durchgeführt werden, die jeder Testfall neu benötigt. Dies können leere Listen sein oder vorhandene Iteratoren werden zurückgesetzt.

Ausführung der Testfälle Hier findet die eigentliche Ausführung der Testfälle statt. Die Testfälle werden nacheinander ausgeführt und es wird eine Meldung über den Erfolg beziehungsweise Misserfolg ausgegeben.

tearDown ist eine Methode, die nach jedem Testfall ausgeführt wird. So können hier beispielsweise geöffnete Verbindungen geschlossen werden.

tearDownAfterClass ist eine Methode, die nach dem Beenden aller Testfälle ausgeführt wird. So können alle geladenen Treiber geschlossen und die vorgenommenen Einstellungen können zurückgesetzt werden.

Die Testfälle können zu Testsuiten zusammengefasst werden. Dies hat den Vorteil, dass nicht viele Testklassen von Hand gestartet werden müssen. Es reicht aus, eine Testsuite zu starten, die dann die einzelen Testklassen aufruft und startet. In dieser Testsuite können außerdem die Treiber geladen werden, die für mehrere Tests benötigt werden. Die Tests können daher schneller durchgeführt werden und benötigen weniger Ressourcen.

JUnit zeigt die Auswertungen der Tests in einem weiteren Fenster in Eclipse an. Nach dem Ausführen der Tests wird bei Erfolg ein grüner Balken angezeigt. Bei einem fehlgeschlagenen Test ist dieser Balken komplett rot. JUnit zeigt in diesem Fenster den fehlerhaften Test an. Durch ein Klicken auf die Fehlermeldung wird die entsprechende Testklasse geöffnet und die Methode wird angezeigt. Die entsprechende Fehlerwirkung kann daher leicht ermittelt werden.

In diesem Projekt wird JUnit sowohl für die Komponententests als auch für den Integrationstest eingesetzt. Dabei können die Testfälle durch die Methoden von JUnit einfach

[379] s. [JUnit 09]
[380] vgl. [Beck 08]

umgesetzt werden. Dafür bietet die statische Klasse `Assert` aus dem JUnit Paket mehrere Methoden an, die alle entscheiden, ob ein Test fehlgeschlagen oder bestanden ist.

Einige Beispielmethoden dieser Klassen sind:

assertNotNull überprüft, ob eine Instanz einer Klasse initialisiert wurde und nicht `NULL` ist.

assertTrue überprüft, ob die Ausführung einer Methode einen positiven Rückgabewert ermittelt (`true`).

assertFalse erwartet, dass die Ausführung der Methode einen negativen Rückgabewert ermittelt (`false`).

assertEquals überprüft die Gleichheit zweier Objekte oder ob zwei ausgeführte Methoden den gleichen Rückgabewert ermitteln.

Mit diesen Methoden können die einzelnen Methoden einer Klasse auf ihr Verhalten und ihren Rückgabewert überprüft werden. Zusätzlich besteht die Möglichkeit zu testen, ob die einzelnen Objekte einer Klasse alle initialisiert worden sind und nicht mehr `NULL` zurückgeben. Auch das Zusammenspiel zweier Komponenten kann getestet werden, in dem der Rückgabewert einer Methode als Eingabewert der Methode einer zweiten Klasse benutzt wird. Nur wenn die Rückgabe der zweiten Klasse den Erwartungen entspricht, funktioniert die Kommunikation dieser beiden Komponenten.

Es werden insgesamt 68 Testfälle definiert, die die einzelnen Komponenten und die Rückgaben der entsprechenden Methoden überprüfen. Ein Auszug der spezifizierten Testfälle findet sich im Anhang C - Testfallspezifikationen und Auswertungen (Seite 193, Listing 28). Nach Abschluß der Entwicklung zeigen die Testfälle keine Fehlerwirkungen auf und liefern alle positive Ergebnisse.

8.4.3. Fazit

Die dynamischen Testverfahren zeigen im Gegensatz zu den statischen[381] das Verhalten der Anwendung und ihrer Komponenten bei der Ausführung. Diese Testverfahren sind notwendig, da durch die statischen Tests einige Fehlerwirkungen unentdeckt bleiben.

Die statischen Testverfahren untersuchen das Testobjekt nicht während der Ausführung. Dabei können nur einzelne Komponente alleine und ohne das Zusammenspiel mehrerer Komponente analysiert werden. Die Fehlerwirkungen, die hier entstehen wie ein falscher

[381] s. Seite 124, Abschnitt 8.3 (8.3)

Datentyp bei einem Übergabeparameter, können nur durch die dynamischen Testverfahren aufgedeckt werden. Aus diesem Grund werden die dynamischen Verfahren ebenfalls durchgeführt.

Alle bisher aufgeführten Testverfahren verliefen positiv, so dass wir an dieser Stelle sagen können, dass ein Testendekriterium der statischen sowie der dynamischen Tests erreicht ist. Wir führen noch einige weitere Testverfahren durch[382], die alle sehr schnell zu realisieren sind und sie runden die bisherigen Testverfahren ab. Somit erhalten wir eine gute Gesamtübersicht über die Qualität der Software.

8.5. Weitere Testverfahren

Es gibt noch eine Vielzahl an weiteren Testmethoden, um die Anwendung beispielsweise auf die Stabilität oder Performanz zu testen. Diese Testverfahren lassen sich schwer in eine Kategorie einordnen und werden daher gesondert aufgeführt.

Neben den bereits beschriebenen statischen[383] und dynamischen[384] Testverfahren wird die Anwendung weiteren Tests unterzogen. Diese Testverfahren dienen dazu, Aufschluss über die Reaktion der Anwendung unter besonderen Bedingungen zu erhalten.

8.5.1. Vorstellung der einzelnen Testverfahren

Die Testverfahren, die angewendet werden, sind im einzelnen:

Crashtest versucht die Anwendung unter Last zum Absturz zu bringen

Installationstest testet die Installationsroutinen auf verschiedenen Betriebssystemen oder Hardware

Interoperabilitätstest ist eine spezielle Testmethode für verteilte Systeme, bei der mindestens zwei Einzelkomponenten über ihre Schnittstellen interagieren

Internet-Abschalttest testet die Reaktion der Anwendung beim Verlust der Verbindung zum Internet, speziell zur entfernten Datenbank

Oberflächentest testet die Benutzerschnittstelle der Anwendung

Smoketest testet das System nach der Implementierung einer neuen Funktion oder Komponente

[382] s. Seite 136, Abschnitt 8.5 (8.5)
[383] s. Seite 124, Abschnitt 8.3 (8.3)
[384] s. Seite 131, Abschnitt 8.4 (8.4)

Wiederinbetriebnahmetest testet, ob die Anwendung nach dem Beenden ohne Probleme neu gestartet werden kann

8.5.1.1. Crashtest Ein *Crashtest* kann auf verschiedene Arten durchgeführt werden. Es wird jedoch immer versucht, die Anwendung durch außergewöhliche Belastungen zum Absturz zu bringen. In diesem Projekt wird der Crashtest durch Klicken der einzelnen Funktionen durchgeführt, ohne auf die Anwort beziehungsweise die Reaktion der Anwendung zu warten. Durch diese Art eines Crashtests wird das Verhalten der Anwendung untersucht, wenn eine Funktion nicht beendet wird, bevor eine andere Funktion aufgerufen wird. Auch nach unkontrolliertem Anklicken der einzelnen Funktionen läuft die Anwendung ohne Probleme weiter.

8.5.1.2. Installationstest Durch eine Implementierung in Java muss die Anwendung nicht installiert werden. Sie läuft auf jedem Betriebssystem, auf dem Java in der aktuellen Version 1.6[385] installiert ist. Die Anwendung ist als ausführbares Java Archiv (JAR) direkt lauffähig[386] und muss nicht installiert werden. Der *Installationstest* ist daher schnell durchgeführt, da die Anwendung keine Installation benötigt.

8.5.1.3. Interoperabilitätstest Der *Interoperabilitätstest* wird für diese Anwendung implizit benötigt, da zwei Einzelkomponenten (die Anwendung und die eigentliche Datenbank) involviert sind. Dieser Test zeigt auf, wie gut diese beiden Komponenten miteinander kommunizieren.

Die Schnittstelle, die diese beiden Komponenten miteinander verbindet, kommt aus dem Java Database Connectivity (JDBC)-Paket. Durch dieses Paket wird der Treiber einer Datenbank geladen und gestartet. Die weitere Kommunikation erfolgt über das Java-Interface `Connection` aus dem Paket `java.sql`.

Durch diesen Test konnte festgestellt werden, dass sich das Data Dictionary bei Oracle von der Version 10g zur Version 11g geändert hat[387]. Daher kann sichergestellt werden, dass die Anwendung ab Oracle-Version 9i lauffähig ist.

8.5.1.4. Internet-Abschalttest Der *Internet-Abschalttest*, auch *Rechnernetz-Test* genannt, wird auf zwei Arten durchgeführt, wobei entweder der Rechner vom Netz getrennt

[385] Stand Juli 2009

[386] Jedes Java-Programm wird in einer eigenen virtuellen Maschine gestartet.

[387] Die View für die vorhandenen Trigger (`user_triggers`) wurde in der Version 11g erweitert und kann nicht für eine frühere Version benutzt werden.

wird oder die Datenbank nicht erreichbar zu sein scheint. In beiden Fällen sollte die Anwendung gleichartig reagieren und eine entsprechende Fehlermeldung ausgeben. Die Anwendung benötigt in zwei Funktionen eine Verbindung zum Internet, beim Aktualisieren eines Schemas oder beim Anlegen einer neuen Verbindung, jeweils zu einer entfernten Datenbank.

Während dieser Verbindung werden die vorhandenen Datenbankobjekte abgefragt und ausgelesen. Für weitere Funktionen der Anwendung ist eine Verbindung nicht notwendig, da alle benötigten Ressourcen lokal in einer HyperSQL DataBase (HSQLDB) gespeichert werden und somit auch offline zur Verfügung stehen. Bei dem Anlegen einer neuen Verbindung zu einer entfernten Datenbank erfolgt ein Test, ob die angegebenen Einstellungen stimmen (Benutzername, Passwort, Host, Port, Datenbankname) und daher erfolgt auch implizit ein Test, ob sich der Rechner im Netz befindet und ob die Datenbank erreichbar ist.

Es kann passieren, dass die Verbindung während der Übertragung der Daten abbricht. Bei einer Aktualisierung des Schemas erfolgt eine Fehlermeldung und das Schema wird nicht aktualisiert. Eine Speicherung erfolgt erst nach dem kompletten Auslesen eines Schemas. Beim Anlegen einer neuen Verbindung erfolgt ebenfalls die Fehlermeldung und die Verbindung wird angelegt, allerdings werden keine Tabellen, Views oder Trigger gespeichert. Bereits gelesene Datenbankobjekte werden verworfen, da eine grafische Übersicht der Abhängigkeiten nur erfolgen kann, wenn alle beteiligten Datenbankobjekte verarbeitet wurden.

8.5.1.5. Oberflächentest

Um den *Oberflächentest* durchzuführen, ist es notwendig, ein eigenes Testszenario zu entwickelt. Bei diesem Testverfahren ist eine bereits implementierte, grafische Benutzeroberfläche Voraussetzung. Zwei unerfahrene Anwender, die nicht aus dem Informatikbereich kommen, erhalten nur grob eine kurze Einführung in die Thematik (Was ist ein „Datenbankhost“? Warum gebe ich einen Port an?) und werden bei der Ausführung der Anwendung beobachtet. Die Funktionalität der Anwendung wird im Vorfeld nicht erklärt.

Ein Grund dafür ist, dass die Benutzerfreundlichkeit der Anwendung getestet werden kann. Die Fragen wie beispielsweise „Finden die Anwender alle Funktionen? Kann eine Verbindung problemlos angelegt werden?“ können durch dieses Testverfahren geklärt werden.

Ein zweiter Grund für die Durchführung durch unerfahrene Anwender entsteht daraus, dass ein implizites, exploratives Testverfahren durchgeführt wird. Durch die Unerfahren-

heit lernt der Anwender nach und nach die Anwendung kennen und probiert die Funktionen aus. Daher können auch ungewollte, unkontrollierte Bedienungsabläufe stattfinden.

Dieses Testverfahren gibt also Aufschluss über die Benutzerfreundlichkeit der Anwendung und die Stabilität durch ungeplante Bedienungsabläufe. Anfangs werden die Funktionen durch die beiden Tester relativ schnell durchgeklickt und die Anwendung läuft stabil weiter. Nach einer kurzen Einarbeitungsphase werden alle Funktionen gefunden und es kann keine Fehlerwirkung festgestellt werden. Die Anwendung ist dementsprechend benutzerfreundlich und stabil gegenüber explorativem Ausprobieren.

8.5.1.6. Smoketest

Ein *Smoketest* ist ein iteratives Testen der Software nach jeder Implementierung einer neuen Funktion. Er wird auch als grundlegender Probelauf bezeichnet. Hierbei wird getestet, ob die neu hinzugefügte Funktion einer Anwendung nicht schon im Ansatz fehlschlägt. Bei einem Smoketest werden die neu implementierten Programmfunktionen oberflächlich getestet. Durch diesen Test kann sichergestellt werden, dass grundlegende Funktionen (beispielsweise ist ein Button anklickbar oder ein Eingabefeld beschreibbar) laut ihrer Spezifikation richtig umgesetzt werden. Dieses Testverfahren wird während der gesamten Entwicklung iterativ eingesetzt.

8.5.1.7. Wiederinbetriebnahmetest

Durch einen *Wiederinbetriebnahmetest* kann gezeigt werden, dass die Anwendung nach einem erfolgreichen Start und einer Ausführung wieder neu gestartet werden kann. Der Neustart der Anwendung sollte dabei ohne Probleme verlaufen und die benutzerdefinierten Programmeinstellungen sollten wiederhergestellt sein.

In diesem Projekt werden die erstellten Verbindungen lokal gespeichert. Außerdem werden die Objekte in den grafischen Ansichten der Abhängigkeiten gespeichert, so dass die Positionen der Objekte im Graphen wiederhergestellt werden können.

Die Anwendung wird mehrmals geöffnet, die Verbindungen werden bearbeitet, die Positionen der einzelnen Objekte im Graphen werden verändert und die Anwendung wird danach wieder geschlossen. Nach einem Neustart werden keine Veränderungen festgestellt und alle benutzerdefinierten Einstellungen werden wieder hergestellt.

8.5.2. Fazit

Diese Testverfahren wurden alle durchgeführt und die Ergebnisse wurden abschließend ausgewertet. Diese Tests sind als alleiniges Testkriterium nicht ausreichend, aber sie

runden die durchgeführten statischen und dynamischen Testverfahren ab.

Durch diese weiteren Testverfahren kann sichergestellt werden, dass die Anwendung auch unter besonderen Bedingungen stabil läuft oder die Benutzbarkeit der grafischen Oberfläche gewährleistet ist. In der Regel können diese Testverfahren innerhalb kürzester Zeit durchgeführt werden und benötigen keine großartigen Ressourcen.

Nach dem Abschluss dieser Verfahren kann das Testen der Anwendung beendet werden, da alle Tests keine weiteren Fehler angezeigt haben und somit ein Testendekriterium erreicht wird.

9. Fazit

In den bisherigen Kapiteln wurde die Vorgehensweise zur Visualisierung der Abhängigkeiten von Datenbankobjekten erläutert. Die aus dieser Arbeit entstandende Software wurde ebenfalls in Teilen dokumentiert. Dabei wurde sowohl auf den analytischen als auch auf den grafischen Teil detailliert und in Beispielen eingegangen.

Im Folgenden werden die Erkenntnisse der Recherche und der Entwicklung der Software kurz zusammengefasst. Außerdem werden die erzielten Ergebnisse in einer Selbstreflexion bewertet. Dazu gehören auch weitere Ideen im Zusammenhang des Themas.

9.1. Zusammenfassung

Die vorliegende Arbeit beschäftigt sich thematisch mit der Untersuchung und Darstellung unterschiedlicher Abhängigkeiten der einzelnen Datenbankobjekte. Die Motivation war die Visualisierung dieser ermittelten Abhängigkeiten aus einem Datenbankschema in einer geeigneten Art und Weise. Dies sollte als Softwareanwendung mit der Sprache Java und zunächst primär für das Datenbanksystem Oracle in der Version 11g entwickelt werden.

Eine Analyse der Aufgaben und Funktionen an die Anwendung ergab, dass drei verschiedene Ansichten implementiert werden müssen: die View-Hierarchie, die Triggeraktivitäten und das Entity-Relationship-Diagramm. Die View-Hierarchie beinhaltet alle Views und deren Beziehungen untereinander. Dabei wird zwischen positiven und negativen Abhängigkeiten unterschieden. Potenzielle Triggerrekursionen und -anomalien werden in der Triggeransicht dargestellt. Schließlich werden die einzelnen Fremdschlüsselbeziehungen in dem ER-Diagramm visualisiert.

Eine Evaluation bestehender Werkzeuge im Bereich des Datenbankmanagements ergab, dass nahezu keine Anwendung derartige Funktionalitäten aufweist. Zwar können die meisten Anwendungen die Informationen über die Abhängigkeiten der Views ermitteln und (tabellarisch) anzeigen, aber keine Anwendung bietet eine visuelle und anspruchsvolle Darstellung an. Der SQL Developer bietet eine „Diagramm“-Funktion mit integrierter Anzeige der Viewabhängigkeiten an, jedoch nicht im Kontext der hierarchischen Abhängigkeitsstruktur. Des Weiteren schweigt sich der SQL Developer sowohl über die Details einer View-Abhängigkeit als auch Trigger im Allgemeinen aus.

Zur Ermittlung eines geeigneten SQL Parsers wurden verschiedene Parser miteinander verglichen und auf die Tauglichkeit in Bezug zur Aufgabenstellung hin bewertet. Dabei hat sich herausgestellt, dass ein Großteil der untersuchten Parser Probleme mit verschach-

telten Abfragen (Unterabfragen) hat. Da der Parser Sqljep rein tokenbasiert und für eine Integration in Java verfügbar ist, haben wir uns für dessen Verwendung entschieden. Zwar reicht Sqljep einerseits für das reine syntaktische Analysieren (Lesen) von SQL-Anweisungen aus. Spezielle Herstellererweiterungen wie PL/SQL machen jedoch individuelle Anpassungen nötig. Leider gibt es dafür bisher[388] keine vernünftige und universell einsatzfähige Lösung, die uns bekannt ist. Daher war es in Teilen notwendig, individuelle und meist datenbankherstellerabhängige Workarounds in die Analyse und das Parsen zu integrieren.

Bei der Evaluation eines brauchbaren Graphenframeworks war es unter Anderem von Bedeutung, dass es sich gut in Komponenten der grafischen Swing-Bibliothek von Java integrieren ließ. Die Betrachtung der unterschiedlichen Frameworks, Dokumentationen und Beispiele zeigte, dass einige Frameworks nur eine mäßig gute Dokumentation vorweisen können. Wir stellten fest, dass die Frameworks JUNG und JGraph die für unsere Zwecke nützlichsten Voraussetzungen und Funktionalitäten anbieten. Die Wahl auf JUNG wurde schließlich auf Basis der internen API-Struktur, dem Informationsgehalt vorhandener Dokumentationsquellen und dem Vorhandensein fertiger Layouts getroffen. Alle Frameworks besitzen eine teils breit gefächerte Auswahl an Layoutvarianten. Die kommerziellen Produkte bieten als Mehrwerte in Teilen eine bessere Optik des Graphen verbunden mit leistungsfähigeren Layoutalgorithmen.

Die Analyse der Metadaten konnte in Teilen zügig mit Hilfe des JDBC-Pakets von Java umgesetzt werden. Mittels einem herstellerabhängigen Datenbanktreiber, der die Spezifikationen und Konzepte von JDBC realisiert, werden die Abfragen an das Data Dictionary in einzelnen Methodenaufrufe und Ergebnismengen gekapselt. Allerdings wurde bei der Implementierung des universellen Analysecodes festgestellt, dass der aktuelle Datenbanktreiber[389] von Oracle falsche Ergebnisse oder sogar Fehlermeldungen wirft: im konkreten Fall bei der Abfrage der Indexe und Unique-Bedingungen für Tabellen. Aber auch die fehlermeldungsfreien Ergebnisse sind ungenügend, da neben den einfachen Unique-Bedingungen ebenfalls die Primärschlüssel angezeigt werden. Daher wird anstelle der dafür im JDBC-Paket vorgesehen Methoden zum Laden der Indexe und Primärschlüssel eine eigene Anfrage an das Data Dictionary geschrieben.

Nach kleineren Anpassungen für das Auslesen eines Triggerrumpfes in PL/SQL konnte der SQL-Parser SQLJEP ohne weitere Probleme in die Anwendung integriert werden. Durch eine tokenbasierte Arbeitsweise kann dieser Parser für jedes Datenbanksystem eingesetzt werden. Es wird für jedes Datenbanksystem durch die verschiedenen Daten-

[388] Stand Juli 2009
[389] Oracle Database 11g Release 1 (11.1.0.7.0) JDBC Drivers, Stand Juli 2009

bankdialekte eine entsprechende Grammatik benötigt. Die Datenbankobjekte in Oracle werden zuerst formal in einer Backus-Naur-Form beschrieben und die Grammatik wird hiervon abgeleitet. Durch die Implementierung dieser Grammatik kann der Parser, und daher auch die Anwendung, jeder Zeit für andere Datenbanksysteme erweitert werden.

Die Implementierung der Graphenvisualisierung mit JUNG war überschaubar und durch Beispiele und eigene Prototypen gut lösbar. Mangels technischer Unterstützung von zusätzlichen Events[390] konnten leider bereits früher erstellte Prototypen[391] nicht verwendet werden. Dennoch bietet die nun implementierte Darstellung neben Zoom- & Verschiebemöglichkeiten auch filterbare Darstellungen sowie kontextsensitive Popupmenüs.

9.2. Ausblick

Bereits bei der Formulierung der konkreten Aufgabenstellung ergaben sich stets neue mögliche Ideen und Vorschläge, die nicht nur maßgeblich den Umfang, sondern auch die Richtung der Software bestimmt haben. Natürlich konnten nicht alle Ideen oder Fragen aufgegriffen werden. Es ist aber wichtig darauf hinzuweisen, dass die erreichte Visualisierung der strukturellen Zusammenhänge der unterschiedlichen Datenbankobjekte noch weiter ausbaufähig ist. Je nach Kontext ist eine anderer Grad der Komplexität und der darzustellenden Dynamik der Daten von Bedeutung.

Die erstellte Anwendung beschränkt sich in der vorliegenden Version auf die Schemaanalyse der Datenbanksysteme Oracle und MySQL. Dabei werden die Oracle-Versionen 9i, 10g und 11g für die Zwecke der Anwendung vollständig unterstützt. Das Plugin für eine MySQL-Datenbank ist in einem experimentellen Zustand und daher nicht vollständig. Durch ein geeignetes Abstraktionslevel lassen sich jederzeit weitere Datenbanksysteme unterstützen. Auch eine individuelle Implementierung aufgrund verschiedener Datenbankdialekte oder spezieller Features[392] ist möglich. Prinzipiell ist jedoch ein JDBC-Treiberplugin für ein entsprechendes Datenbanksystem notwendig.

Für eine geeignete Darstellung der Knoten und Kanten eines Graphen sind weitere Alternativlayouts möglich. Insbesondere die Problematik der sich überschneidenden Kanten kann durch effizientere Algorithmen gelöst werden. Ein Beispiel für solche Algorithmen ist der Algorithmus von Sugiyama zur Minimierung von Kantenschnitten. Für eine optimalere Darstellung ist auch die Bildung von Clustern, also zusammenhängenden Untermengen,

[390] An dieser Stelle sind damit die abfragbaren Mausklicks für zusätzliche Popups oder Buttons eines Graphenknotens gemeint.
[391] s. Anhang A, S. 159, Abb. A.8 (A.8)
[392] Ein besonderes Oracle-Feature ist PL/SQL.

eine mögliche Option.

Das dargestellte Entity-Relationship-Diagramm beinhaltet keine zusätzlichen Informationen wie die Kardinalität, die Optionalität und identifizierende Beziehungen. Die dafür notwendigen Informationen sind bereits durch die Analyse[393] ermittelt worden. Dabei werden die Zwischentabellen von m:n-Beziehungen zu zwei 1:n-Beziehungen[394] aufgeschlüsselt. Nur durch Interpretationen lassen sich echte m:n-Beziehungen nachweisen und anzeigen.

Im Zusammenhang mit der Analyse und Visualisierung der View-Hierarchien sind auch die rekursiven Views interessant. Diese sind zwar als *RECURSIVE VIEW* bereits im SQL-Standard *SQL:1999* vorhanden, allerdings werden sie noch nicht von allen Datenbankherstellern unterstützt. Im Gegensatz zu DB2 von IBM bietet Oracle bisher[395] nur die *CONNECT BY*-Klausel[396] an, die mit komplett anderer Vorangehensweise ähnliche Ergebnisse liefert. Als Grundlage für die Darstellung rekursiver Viewabhängigkeiten kann die visuelle Veranschaulichung der Triggeraktivitäten dienen.

Bei der Visualisierung der Abhängigkeiten in einer Datenbank gibt es weitere, für diese Anwendung interessante Objekte[397]: Integritätsbedingungen, Assertions, Funktionen, Prozeduren und Typen. Ähnlich wie Trigger enthalten Funktionen und Prozeduren SQL-Code[398], der sowohl vordefinierte als auch dynamisch zusammengesetzte SQL-Anweisungen enthalten kann. Damit ergeben sich weitere direkte oder indirekte Abhängigkeiten auf andere Objekte in der Datenbank. Auch die Aufschlüsselung einer objektrelationalen Typhierarchie kann in Erwägung gezogen werden.

Im Rahmen dieser Arbeit wird nur eine statische Analyse der Objekte und deren Abhängigkeiten verfolgt. Eine tiefergehende Untersuchtung der Abhängigkeiten kann durch die Betrachtung der spaltenbezogenen *ON*-Klausel oder der Bedingungsklausel *WHEN* in PL/SQL-Triggern ermöglicht werden.

Eine andere visuelle Darstellungsmöglichkeit, eventuell auch dreidimensional, wäre, die Tabellen mehr in das Zentrum der Betrachtung zu stellen. Dabei würden alle drei Ansichten der Anwendung – Views, Trigger und das ERD mit Fremdschlüsseln – vereinigt und bei Bedarf im Graphen jeweils zu den Tabellenobjekten auf unterschiedliche Weise angezeigt.

[393] s. Seite 89, Abschnitt 6.3 (6.3)
[394] Selbstverständlich sind hiermit stellvertretend alle möglichen Beziehungstypen gemeint.
[395] Stand Juli 2009
[396] in Oracle nur in Verbindung mit einer rekursiven Select-Abfrage möglich
[397] s. Seite 18, Abschnitt 3.1 (3.1)
[398] Je nach Hersteller in verschiedenen, nicht standardisierten SQL-Standard-Erweiterungen, beispielsweise Oracle PL/SQL und DB2 SQL/PL.

Als einen zusätzlichen Ansatz zur Visualisierung könnte die Darstellung eines dynamischen Kontrollflusses dienen. Ähnlich wie in einem UML-Aktivitätsdiagramm zeigt das Kontrollflussdiagramm einzelne Aktivitäten und Kontrollflüsse an. Dabei sind die Aktivitäten die einzelnen Objekte der Datenbank und die Kontrollflüsse resultieren aus den Abhängigkeiten. Damit ergibt sich ein weiteres und interessantes Anwendungsszenario. Nach der Spezifizierung einer DML-Anweisung durch den Benutzer zeigt die Visualisierung die einzelnen Kontrollflüsse durch alle benutzten und verwendeten Datenbankobjekte mit den entsprechend konkreten Daten und Entscheidungen an. Diese teils indirekten Implikationen, beispielsweise auch kaskadierendes Löschen bei Integritätsbedingungen, können somit ebenfalls in einem Graphen dargestellt werden. Dafür ist es allerdings notwendig, dass der Parser den Inhalt des Triggerkörpers komplett einliest und verarbeitet. Für dynamische Kontrollflüsse sind alle Blöcke (*Compoundtrigger*, Bedingungen, Schleifen) notwendig zu betrachten und bei Ausführung auszuwerten.

Mit Hilfe dieser Software – und unter Berücksichtigung der möglichen Erweiterungen – gibt es nun ein hilfreiches Werkzeug, um die direkten und indirekten Abhängigkeiten von Objekten in einem Datenbanksystem sowohl zu erkennen als auch grafisch darzustellen. Durch eine flexible und erweiterbare Architektur ist das Softwareprodukt auch für neue Anforderungen gut einzusetzen.

Literaturverzeichnis

[Alder 03] Gaudenz Alder. *The JGraph Tutorial*. http://dev.cs.uni-magdeburg.de/java/jgraph/tutorial/t1.html, 2003. Internet, abgerufen im Mai 2009.

[Alder 08] Gaudenz Alder. *JGraph v5.12.4.0 API Specification*. http://www.jgraph.com/doc/jgraph, 2008. Internet, abgerufen im Juni 2009.

[Asal 09] Martin Asal. *Microsoft Office Access Tutorial*. http://www.access-tutorial.de/, 2009. Internet, abgerufen im Juli 2009.

[Balzert 01] Helmut Balzert. Lehrbuch der Software-Technik. Spektrum Akademischer Verlag, 2001.

[Batra 09] Gautam Batra. *Mapping XML to Oracle XMLTYPE in IBM Webspshere platform*. http://www.hibernate.org/405.html, 2009. Internet, abgerufen im Juli 2009.

[Beck 08] Kent Beck & Erich Gamma. *JUnit Cookbook*. http://junit.sourceforge.net/doc/cookbook/cookbook.htm, Dezember 2008. Internet, abgerufen im Juni 2009.

[Black Duck Software 08] Black Duck Software. *Open Source Code Search Engine – Koders*. http://www.koders.com, 2008. Internet, abgerufen im Juli 2009.

[Borchers 07] Jan Borchers. *SQL Developer*. http://sqldeveloper.solyp.com, 2007. Internet, abgerufen im Juni 2009.

[Cormen 07] Thomas H. Cormen *et al.* Algorithmen – Eine Einführung, Ausgabe 2. Oldenbourg, Maerz 2007.

[Eclipse Foundation 06] Eclipse Foundation. *SQL Query Parser User documentation*. http://www.eclipse.org/datatools/project_sqldevtools/sqltools_doc/SQL%20Query%20Parser%20User%20documentation.htm, Maerz 2006. Internet, abgerufen im Juni 2009.

[Eclipse Foundation 09a] Eclipse Foundation. *DTP SQL Development Tools Project*. http://www.eclipse.org/datatools/project_sqldevtools/, 2009. Internet, abgerufen im Juli 2009.

[Eclipse Foundation 09b] Eclipse Foundation. *Eclipse Foundation*. http://www.eclipse.org, 2009. Internet, abgerufen im Juli 2009.

[Eclipse Foundation 09c] Eclipse Foundation. *SWT Documentation*. http://www.eclipse.org/swt/docs.php, 2009. Internet, abgerufen im Mai 2009.

[Eclipse Foundation 09d] Eclipse Foundation. *SWT: The Standard Widget Toolkit*. http://www.eclipse.org/swt, 2009. Internet, abgerufen im Juni 2009.

[Estier 98] Thibault Estier & Jacques Guyot. *The BNF Web Club Language SQL, ADA, JAVA, MODULA2, PL/SQL, ...* http://cui.unige.ch/db-research/Enseignement/analyseinfo/BNFweb.html, 1998. Internet, abgerufen im Juli 2009.

[Evans 09] Clark C. Evans. *The Official YAML Web Site*. http://www.yaml.org, 2009. Internet, abgerufen im Juli 2009.

[Faeskorn-Woyke 07] Heide Faeskorn-Woyke *et al.* Datenbanksysteme – Theorie und Praxis mit SQL2003, Oracle und MySQL. pearson Verlag, Mai 2007.

[Fisher 05] Danyel Fisher & Joshua O'Madadhain. *Understanding the JUNG Visualization System*. http://jung.sourceforge.net/doc/JUNGVisualizationGuide.html, September 2005. Internet, abgerufen im Mai 2009.

[Freeman 06] Eric Freeman *et al.* Entwurfsmuster. O'Reilly Verlag, 2006.

[Frotscher 09] Thilo Frotscher. http://www.heise.de/developer/Kriterien-fuer-die-Auswahl-eines-XML-Data-Binding-Frameworks--/artikel/140242/0, Juni 2009. Internet, abgerufen im Juni 2009.

[Furmankiewicz 09a] Jacek Furmankiewicz. *javabuilders*. http://code.google.com/p/javabuilders/, 2009. Internet, abgerufen im Juni 2009.

[Furmankiewicz 09b] Jacek Furmankiewicz. *Swing JavaBuilders – Achieving maximum productivity with minimum code via declarative UIs* . http://github.com/jacek99/javabuilders/raw/39a2f1854c1bd53162e53e845f5ce663d7214d22/org.javabuilders.docs/javabuilders/out/pdf/swing.

javabuilder.pdf, Maerz 2009. Internet, abgerufen im Juni 2009.

[Gaidukov 07] Alexey Gaidukov. *SQLJEP Usage*. http://sqljep.sourceforge.net/, Oktober 2007. Internet, abgerufen im Juli 2009.

[Gamma 94] Erich Gamma *et al.* Design patterns – elements of reusable object-oriented software. Addison Wesley, 1994.

[Gansner 09] Emden R. Gansner. *Drawing graphs with Graphviz*. http://www.graphviz.org/pdf/libguide.pdf, April 2009. Internet, abgerufen im Mai 2009.

[Gibello 08a] Pierre-Yves Gibello. *ZQL API Specification*. http://www.gibello.com/code/zql/api/index.html, Mai 2008. Internet, abgerufen im Juni 2009.

[Gibello 08b] Pierre-Yves Gibello. *ZQL Parser Homepage*. http://www.gibello.com/code/zql/, Mai 2008. Internet, abgerufen im Juni 2009.

[Graphviz-Team 04] AT&T Graphviz-Team. *Command-line Invocation*, 2004. Internet, abgerufen im Juni 2009.

[Graphviz 09] Graphviz. *Graph Visualization Software*. http://www.graphviz.org, 2009. Internet, abgerufen im Mai 2009.

[Gudu Software 09a] Gudu Software. *General SQL Parser*. http://www.sqlparser.com/index.php, 2001-2009. Internet, abgerufen im Juni 2009.

[Gudu Software 09b] Gudu Software. *GSP Documentations*. http://www.sqlparser.com/docs.php, 2001-2009. Internet, abgerufen im Juni 2009.

[Gudu Software 09c] Gudu Software. *GSP How To*. http://www.sqlparser.com/howto.php, 2001-2009. Internet, abgerufen im Juni 2009.

[Heise Developer 09] Heise Developer. *JavaFX – eine Einschätzung*. http://www.heise.de/developer/JavaOne-Top-Thema-JavaFX-eine-Einschaetzung--/news/meldung/139853, Juni 2009. Internet, abgerufen im Juni 2009.

[Hoffmann 08a] Marc R. Hoffmann. *EclEmma 1.4.1 – User Guide*. http://www.eclemma.org/userdoc/index.html, Dezember 2008. Internet, abgerufen im Juni 2009.

[Hoffmann 08b] Marc R. Hoffmann. *EclEmma 1.4.1 Java Code Coverage for Eclipse*. http://www.eclemma.org/, Dezember 2008. Internet, abgerufen im Juni 2009.

[Hovemeyer 09] David H. Hovemeyer & William W. Pugh. *FindBugs Manual*. http://findbugs.sourceforge.net/manual/index.html, Juni 2009. Internet, abgerufen im Juli 2009.

[InfoEther 09] InfoEther. *PMD HowTo*. http://pmd.sourceforge.net/, Februar 2009. Internet, abgerufen im Juli 2009.

[JGraph 05] JGraph. *JGraphpad Pro v6.0.7.3 API Specification*. http://www.jgraph.com/doc/jgraphpadpro/, 2005. Internet, abgerufen im Juni 2009.

[JGraph 09a] JGraph. *JGraph – Customers*. http://www.jgraph.com/customers.html, 2009. Internet, abgerufen im Juni 2009.

[JGraph 09b] JGraph. *JGraph – Purchase*. http://www.jgraph.com/purchase.html, 2009. Internet, abgerufen im Juni 2009.

[JGraph 09c] JGraph. *JGraph – Screenshots*. http://www.jgraph.com/screenshots.html, 2009. Internet, abgerufen im Juni 2009.

[JGraph 09d] Ltd. JGraph. *mxGraph – Demo page*. http://www.mxgraph.com/demo.html, 2009. Internet, abgerufen im Juni 2009.

[JUNG Dev. Team 09a] JUNG Dev. Team. *Java Universal Network/Graph Framework*. http://jung.sourceforge.net, 2009. Internet, abgerufen im Mai 2009.

[JUNG Dev. Team 09b] JUNG Dev. Team. *JUNG 2.0 Tutorial*. http://www.grotto-networking.com/JUNG/JUNG2-Tutorial.pdf, April 2009. Internet, abgerufen im Mai 2009.

[JUNG Dev. Team 09c] JUNG Dev. Team. *JUNG Demo: Show Layouts*. http://jung.sourceforge.net/applet/showlayouts2.html, 2009. Internet, abgerufen im Mai 2009.

[JUNG Dev. Team 09d] JUNG Dev. Team. *JUNG2 2.0 API (Javadoc)*. http://jung.sourceforge.net/site/apidocs/index.html, 2009. Internet, abgerufen im Mai 2009.

[JUnit 09] JUnit. *JUnit Webpage*. http://www.junit.org/, 2009. Internet, abgerufen im Juni 2009.

[Lee 09] Trustin Lee. *APIviz – JavaDoc doclet for API visualization*. http://code.google.com/p/apiviz, 2009. Internet, abgerufen im Juni 2009.

[McLaughlin 07] Brett D. McLaughlin *et al.* Objektorientierte Analyse & Design. O'Reilly Verlag, 2007.

[Microsoft Corp. 09a] Microsoft Corp. *Document/View Architecture* . http://msdn.microsoft.com/en-us/library/4x1xy43a(VS.80).aspx, 2009. Internet, abgerufen im Mai 2009.

[Microsoft Corp. 09b] Microsoft Corp. *Microsoft Office Access*. http://office.microsoft.com/de-de/access/default.aspx, 2009. Internet, abgerufen im Juli 2009.

[MiG InfoCom AB 09] MiG InfoCom AB. *MiG Layout – Java Layout Manager for Swing and SWT*. http://www.miglayout.com, 2009. Internet, abgerufen im Juni 2009.

[Milakovic 03] Zoran Milakovic. *SQL Parser by Zoran Milakovic*. http://www.koders.com/java/fid875376E771EE4511E298CB10723A5447A9D6D587.aspx?s=cdef%3Aparser#L30, 2002-2003. Internet, abgerufen im Juni 2009.

[Minq Software AB 09a] Minq Software AB. *DbVisualizer Datasheet*. http://www.dbvis.com/products/dbvis/doc/dbvis-datasheet.pdf, 2009. Internet, abgerufen im Juli 2009.

[Minq Software AB 09b] Minq Software AB. *DbVisualizer Webpage*. http://www.dbvis.com/products/dbvis/, 2009. Internet, abgerufen im Juli 2009.

[NetBeans 09] NetBeans. *Netbeans Visual Library in NetBeans Platform 6.0*. http://graph.netbeans.org, 2009. Internet, abgerufen im Juni 2009.

[O'Madadhain 06] Joshua O'Madadhain *et al. Technical Report UCI-ICS 03-17* . Rapport technique, School of Information and Computer Scienc, University of California, Irvine, August 2006. http://www.datalab.uci.edu/papers/JUNG_tech_report.html. Internet, abgerufen im Mai 2009.

[Oracle 07] Oracle. *Oracle SQL Developer*. http://www.oracle.com/

technology/products/database/sql_developer/index.html, 2007. Internet, abgerufen im Juni 2009.

[o.V. 09a] o.V. *Oracle PL/SQL Code Library and Resources – Oracle Views*. http://www.psoug.org/reference/views.html, 2009. Internet, abgerufen im Juli 2009.

[o.V. 09b] o.V. *Rails Framework Documentation*. http://api.rubyonrails.org, Juli 2009. Internet, abgerufen im Juli 2009.

[prefuse 07] Regents of the University of California prefuse. *prefuse – gallery*. http://prefuse.org/gallery, August 2007. Internet, abgerufen im Juni 2009.

[prefuse 09] Regents of the University of California prefuse. *prefuse – interactive information visualization toolkit*. http://prefuse.org, Mai 2009. Internet, abgerufen im Juni 2009.

[Quest Software 09] Inc. Quest Software. *Toad Webpage*. http://www.toadsoft.com/, 2009. Internet, abgerufen im Juli 2009.

[Red Hat Middlware 09] LLC Red Hat Middlware. *Hibernate*. http://www.hibernate.org, 2009. Internet, abgerufen im Juni 2009.

[Robert Eckstein 07] Robert Eckstein. *Java SE Application Design With MVC*. http://java.sun.com/developer/technicalArticles/javase/mvc/, Maerz 2007. Internet, abgerufen im Mai 2009.

[SourceForge 09] Inc. SourceForge. *Lexer Parser Generator Download*. http://sourceforge.net/projects/lpg/files/, 2009. Internet, abgerufen im Juli 2009.

[Spillner 05] Andreas Spillner & Tilo Linz. Basiswissen Softwaretest – Aus- und Weiterbildung zum Certified Tester, Ausgabe 3. dpunkt.verlag, 2005.

[Sun Microsystems 06] Inc. Sun Microsystems. *JDK 6 Java Native Interface*. http://java.sun.com/javase/6/docs/technotes/guides/jni/index.html, 2006. Internet, abgerufen im Juni 2009.

[Sun Microsystems 08a] Inc. Sun Microsystems. *The Java Tutorials: Java Web Start*. http://java.sun.com/docs/books/tutorial/deployment/webstart, 2008. Internet, abgerufen im Juni 2009.

[Sun Microsystems 08b] Inc. Sun Microsystems. *ResultSet Java Documentati-*

on. http://java.sun.com/javase/6/docs/api/java/sql/ResultSet.html, 2008. Internet, abgerufen im Juni 2009.

[Sun Microsystems 09a] Inc. Sun Microsystems. *Java Platform, Standard Edition 6 – API Specification*. http://java.sun.com/javase/6/docs/api, 2009. Internet, abgerufen im Juni 2009.

[Sun Microsystems 09b] Inc. Sun Microsystems. *JavaFX – Rich Internet Applications Development*. http://javafx.com, 2009. Internet, abgerufen im Juni 2009.

[Sun Microsystems 09c] Inc. Sun Microsystems. *JavaFX Developer Technologies*. http://java.sun.com/javafx/technologies, 2009. Internet, abgerufen im Juni 2009.

[Sun Microsystems 09d] Inc. Sun Microsystems. *JavaFX Tutorial – Learn JavaFX Script for Web Application Design*. http://java.sun.com/javafx/1/tutorials/core, 2009. Internet, abgerufen im Juni 2009.

[Sun Microsystems 09e] Inc. Sun Microsystems. *MySQL Workbench*. http://www.mysql.de/products/workbench/, 2009. Internet, abgerufen im Juli 2009.

[Sun Microsystems 09f] Inc. Sun Microsystems. *MySQL Workbench Documentation*. http://dev.mysql.com/doc/workbench/en/index.html, 2009. Internet, abgerufen im Juli 2009.

[The hsql Dev. Group 09] The hsql Dev. Group. *Hibernate*. http://www.hsqldb.org, Juni 2009. Internet, abgerufen im Juni 2009.

[Tobias Kloss 05] Tobias Kloss. *Automatisches Layout von Statecharts unter Verwendung von GraphViz (Diplomarbeit)*. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/tkl-dt.pdf, Mai 2005. Internet, abgerufen im Mai 2009.

[Vinni 09] Mikko Vinni & Moti Ben-Ari. *iDot – Incremental Dot Viewer*. http://stwww.weizmann.ac.il/g-cs/benari/idot/index.html, Mai 2009. Internet, abgerufen im Juni 2009.

[Wikia 09] Inc. Wikia. *Planet JFX – Demos*. http://jfx.wikia.com/wiki/Demos, 2009. Internet, abgerufen im Juni 2009.

[Wikipedia 09a] Wikipedia. http://de.wikipedia.org/w/index.php?title=Flyweight&oldid=59119929, 2009. Internet, abgerufen im Juni 2009.

[Wikipedia 09b] Wikipedia. *ADA Programmiersprache*. http://de.wikipedia.org/wiki/Ada_(Programmiersprache), Juni 2009. Internet, abgerufen im Juni 2009.

[Wikipedia 09c] Wikipedia. *Dot language — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=DOT_language&oldid=284291406, 2009. Internet, abgerufen im Mai 2009.

[Wikipedia 09d] Wikipedia. *Flyweight pattern — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Flyweight_pattern&oldid=289985207, 2009. Internet, abgerufen im Juni 2009.

[Wikipedia 09e] Wikipedia. *Paretoprinzip*. http://de.wikipedia.org/w/index.php?title=Paretoprinzip&oldid=61133089, Juni 2009. Internet, abgerufen im Juli 2009.

[Wikipedia 09f] Wikipedia. *Standard Widet Toolkit*. http://de.wikipedia.org/w/index.php?title=Standard_Widget_Toolkit&oldid=58505048, 2009. Internet, abgerufen im Mai 2009.

[Wong 05] Prof. Stephen Wong. *The Flyweight Design Pattern*. http://www.exciton.cs.rice.edu/javaresources/DesignPatterns/FlyweightPattern.htm, 2005. Internet, abgerufen im Juni 2009.

[Yahoo 09] Inc. Yahoo. *Yahoo! Design Pattern Library*. http://developer.yahoo.com/ypatterns, 2009. Internet, abgerufen im Juli 2009.

[yWorks 09a] GmbH yWorks. *yFiles – Java Graph Layout and Visualization Library*. http://www.yfiles.com/en/products_yfiles_about.html, 2009. Internet, abgerufen im Juni 2009.

[yWorks 09b] GmbH yWorks. *yFiles – Prices (Euro)*. http://www.yfiles.com/en/products_yfiles_commercialinfo_priceseur.html, 2009. Internet, abgerufen im Juni 2009.

[yWorks 09c] GmbH yWorks. *yWorks*. http://www.yfiles.com, 2009. Internet, abgerufen im Juni 2009.

A. Anhang – Bilder

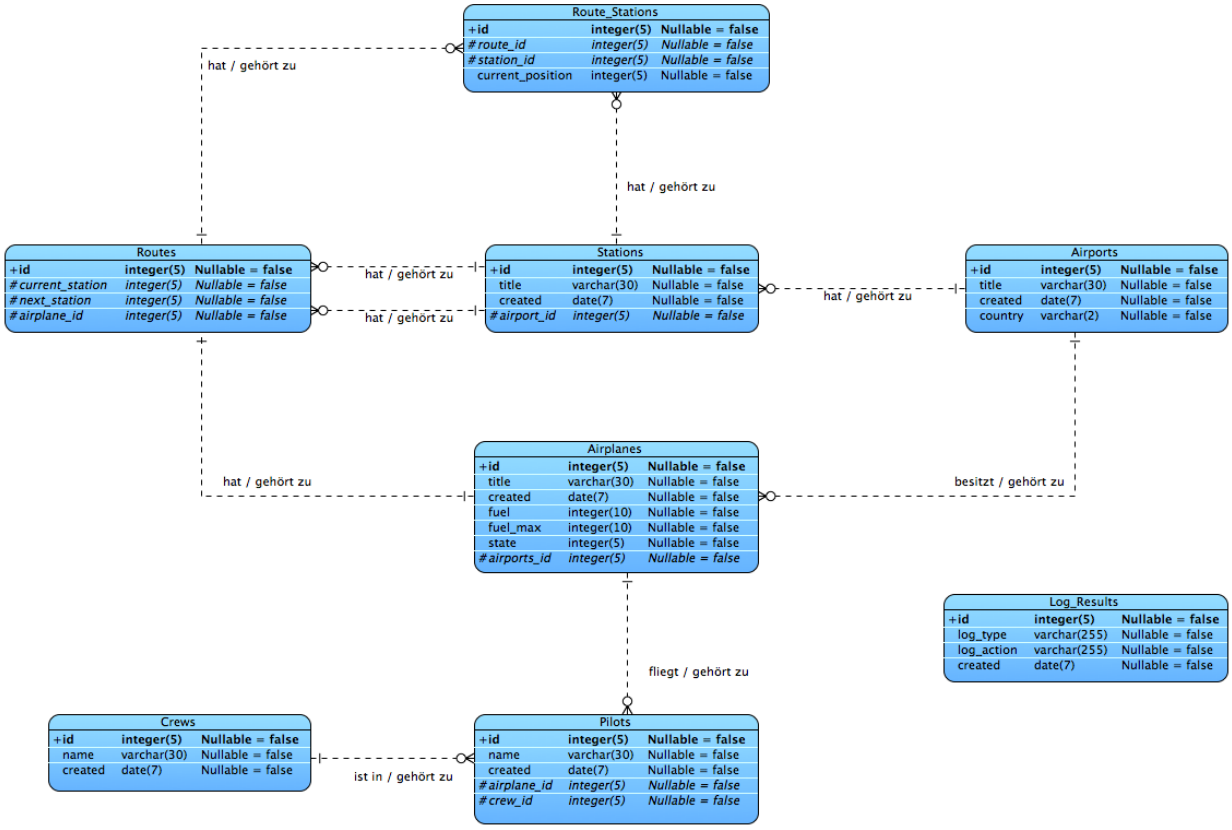


Abbildung A.1: ER-Diagramm der Tabellen aus dem Beispielschema „Flughafen & Flugzeuge“

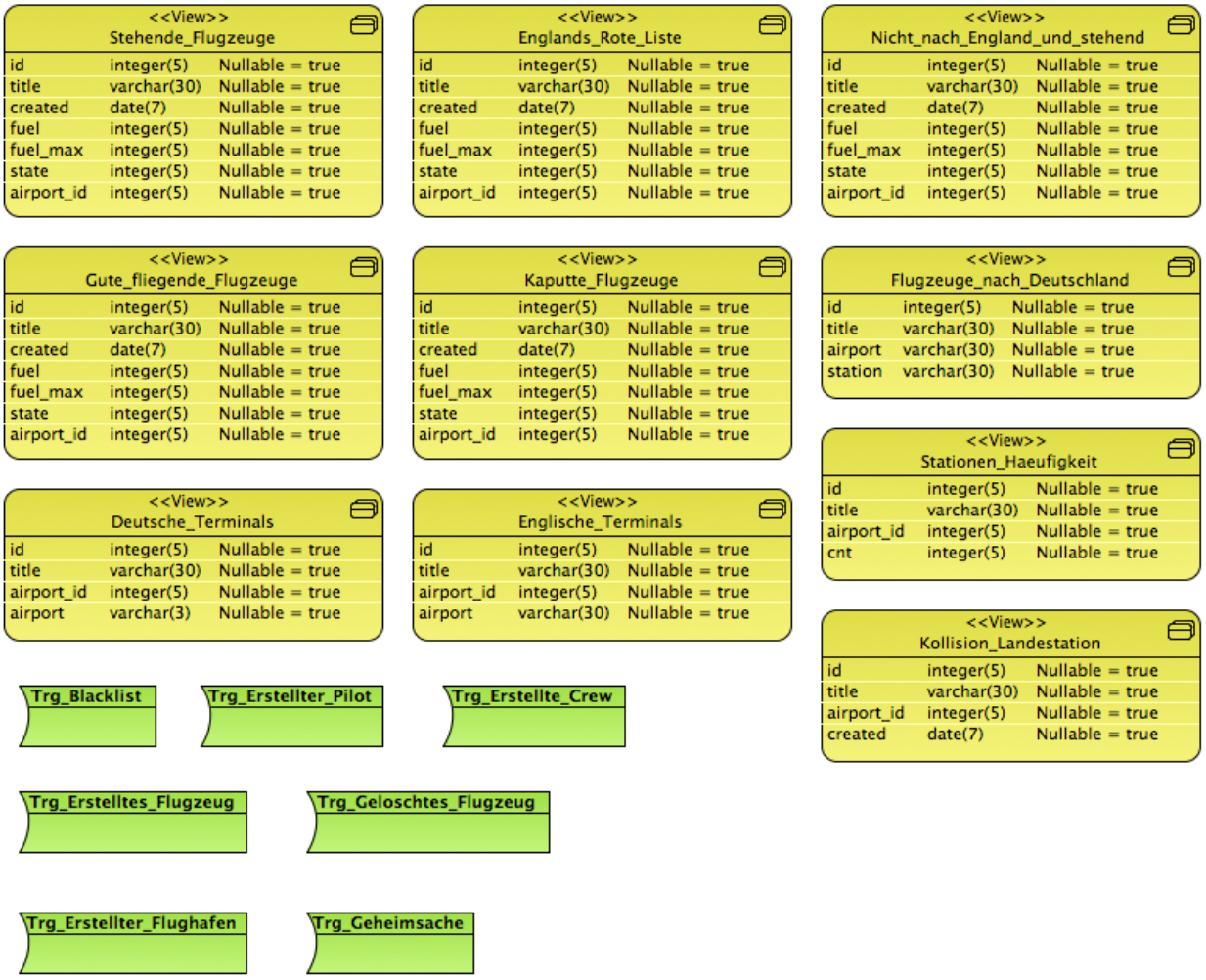


Abbildung A.2: ER-Diagramm der Views und Trigger aus dem Beispielschema „Flughafen & Flugzeuge“

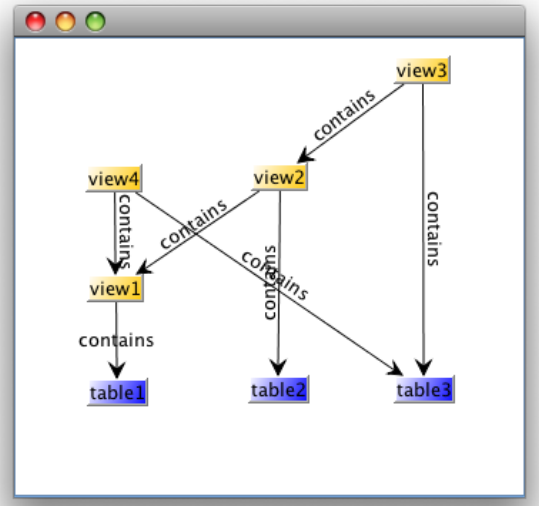


Abbildung A.3: Früher Prototyp einer View-Hierarchie mit *JGraph*. Die Objekte wurden nachträglich repositioniert.

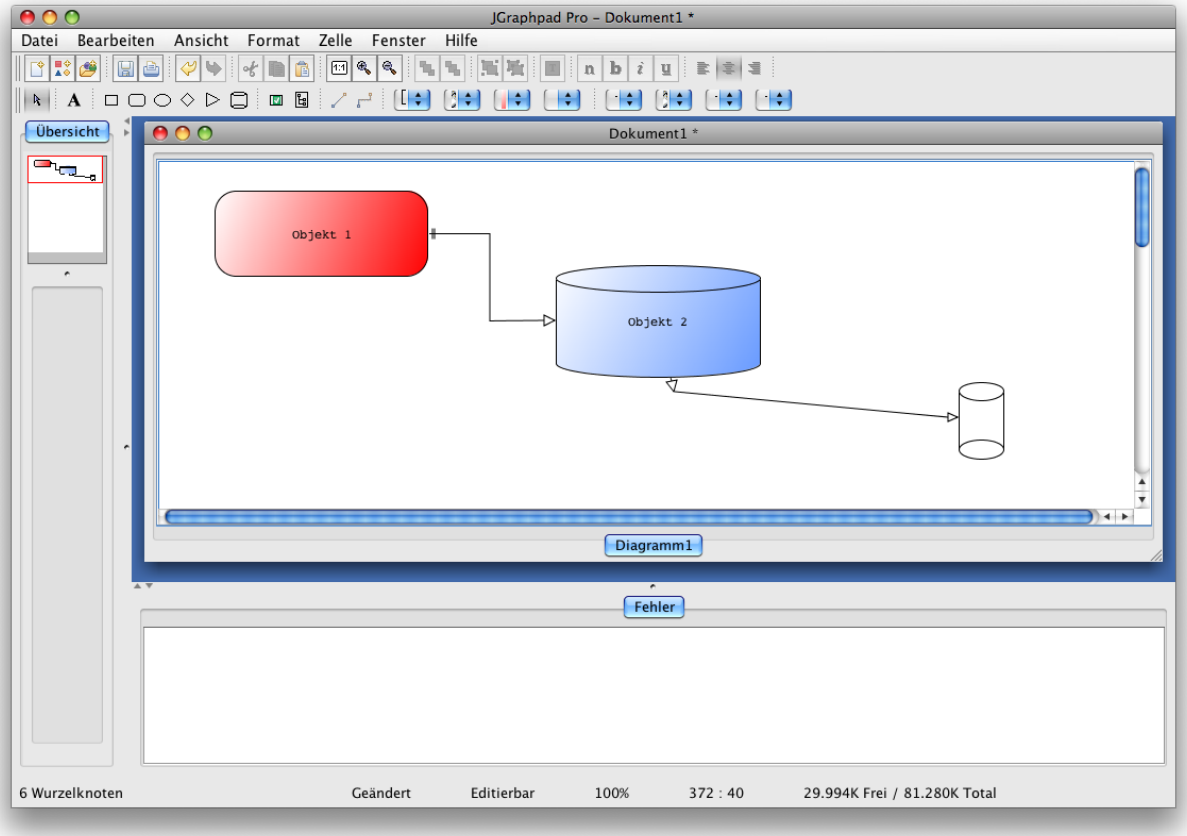


Abbildung A.4: Beispielhafter Screenshot des Programmes *JGraphpad Pro Diagram Editor*.

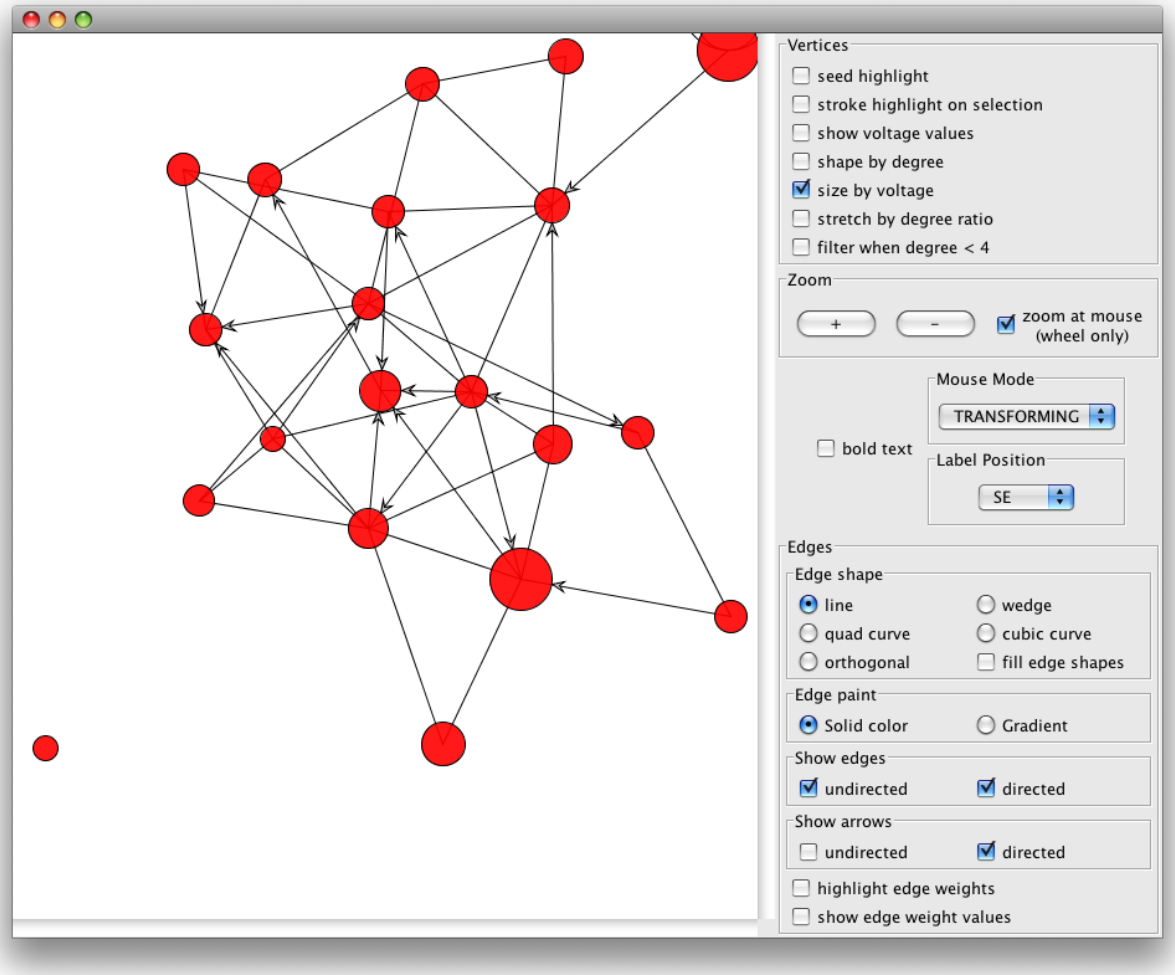


Abbildung A.5: Screenshot 1/2 einer *JUNG*-Anwendung zur Demonstrationszwecken (in JUNG enthalten).

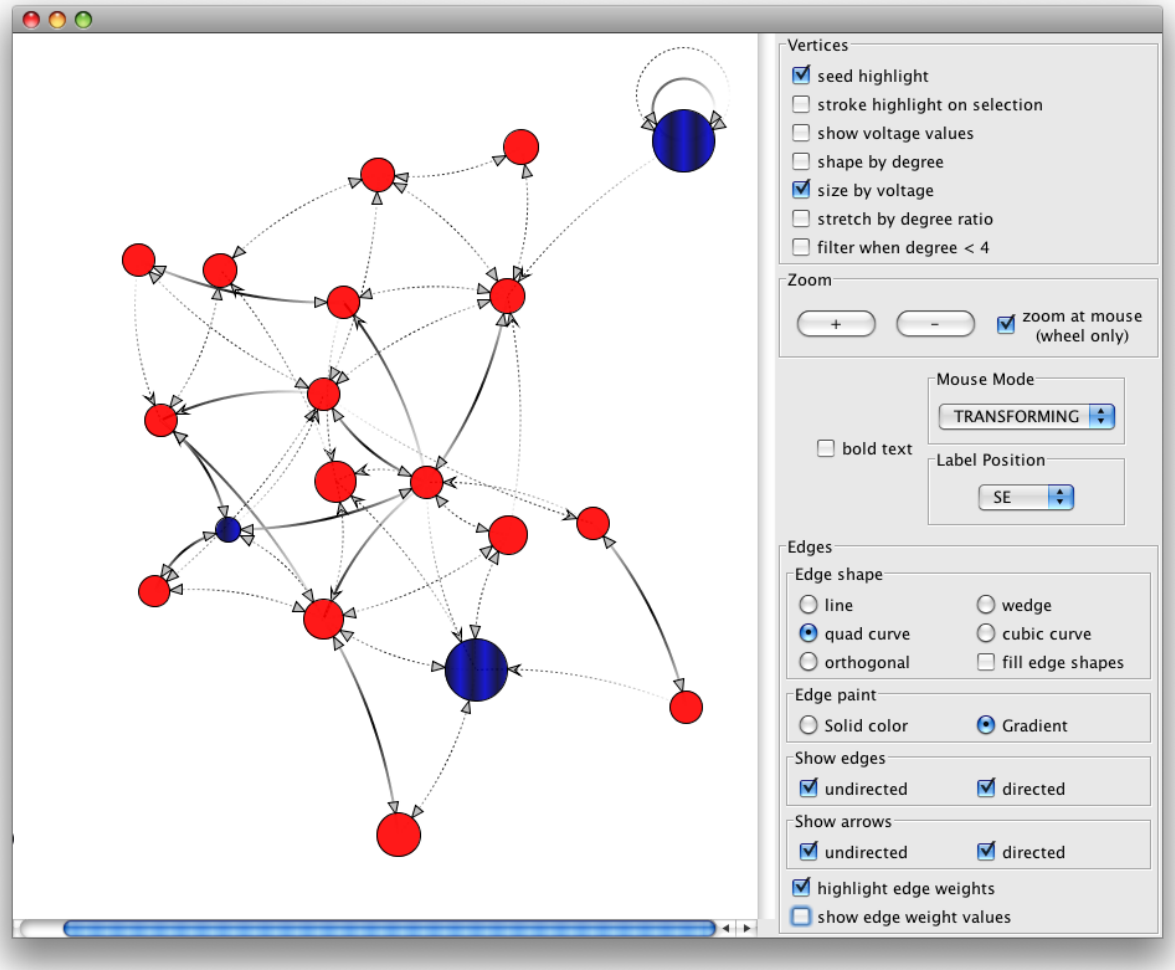


Abbildung A.6: Screenshot 2/2 einer *JUNG*-Anwendung zur Demonstrationszwecken (in JUNG enthalten).

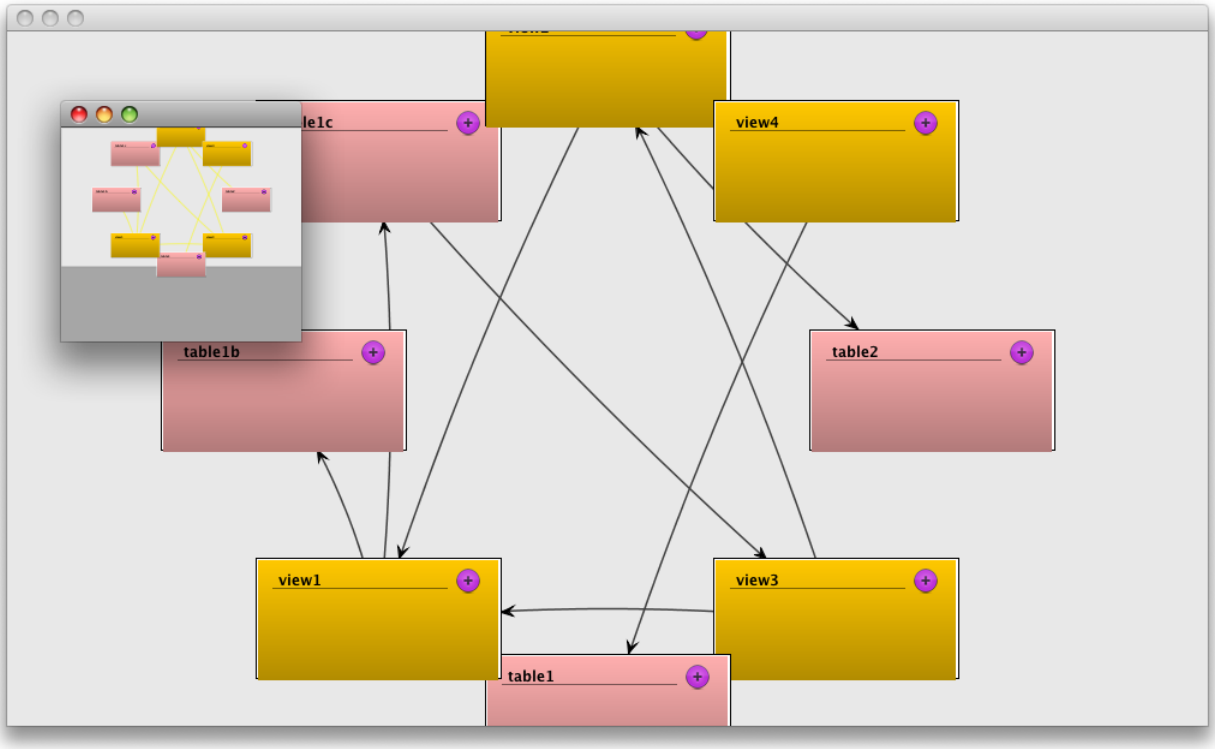


Abbildung A.7: Fortgeschrittener Prototyp einer View-Hierarchie inklusive eines Satelliten (*JUNG*). Die Darstellung nutzt ein alternatives Layout zu Testzwecken.

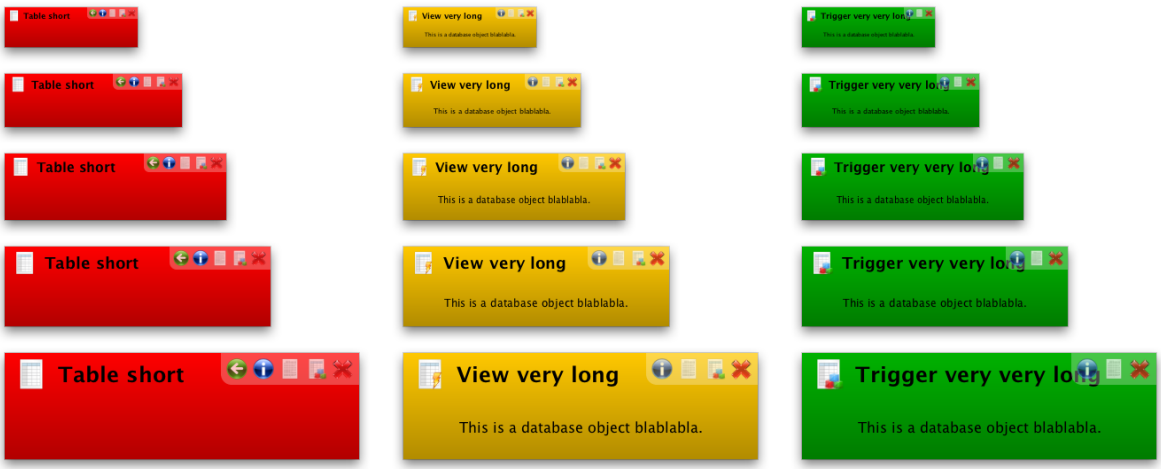


Abbildung A.8: Fortgeschrittene Swing-Mockups für das Visualisieren der Datenbankobjekte

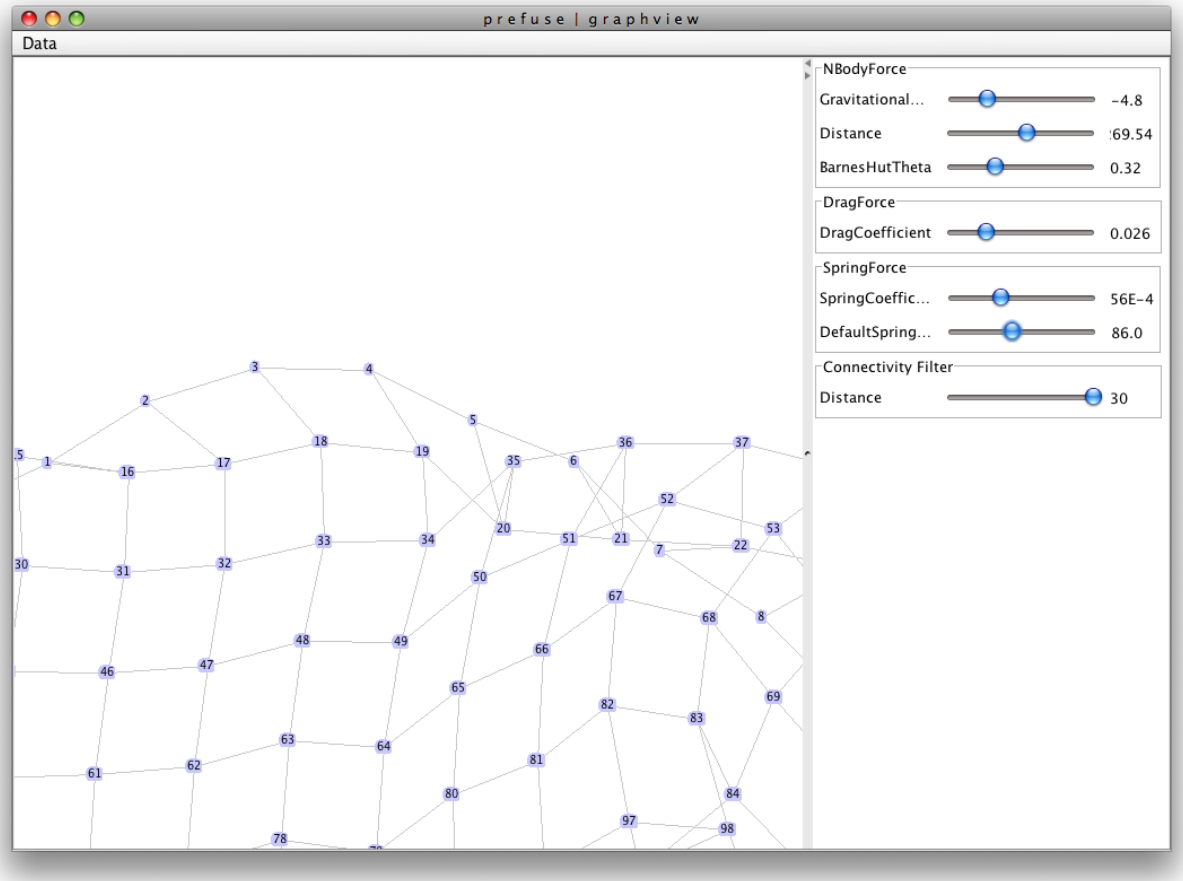


Abbildung A.9: Screenshot einer *prefuse*-Anwendung zur Demonstrationszwecken (Anwendung auf [prefuse 09] verfügbar).

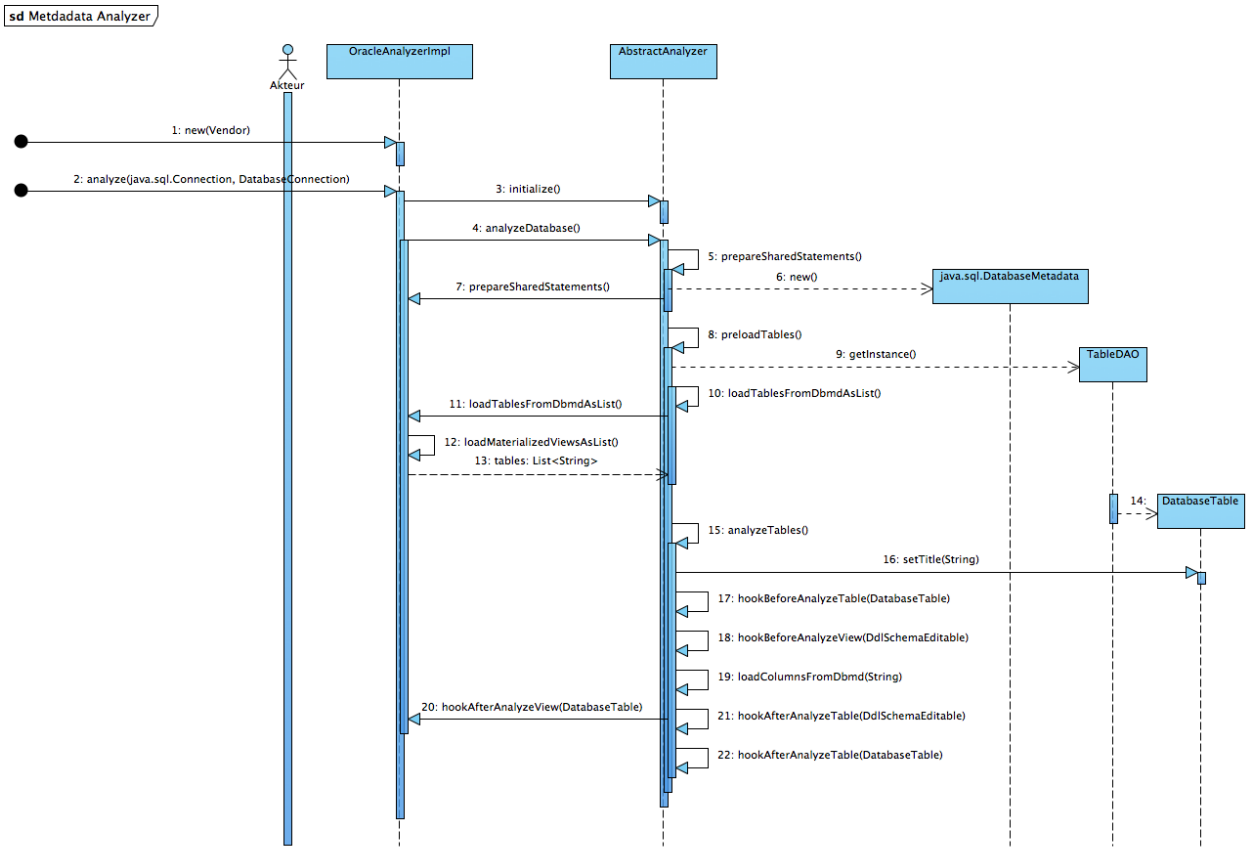


Abbildung A.10: Interner Ablauf des Analyseprozesses im Metadata-Paket

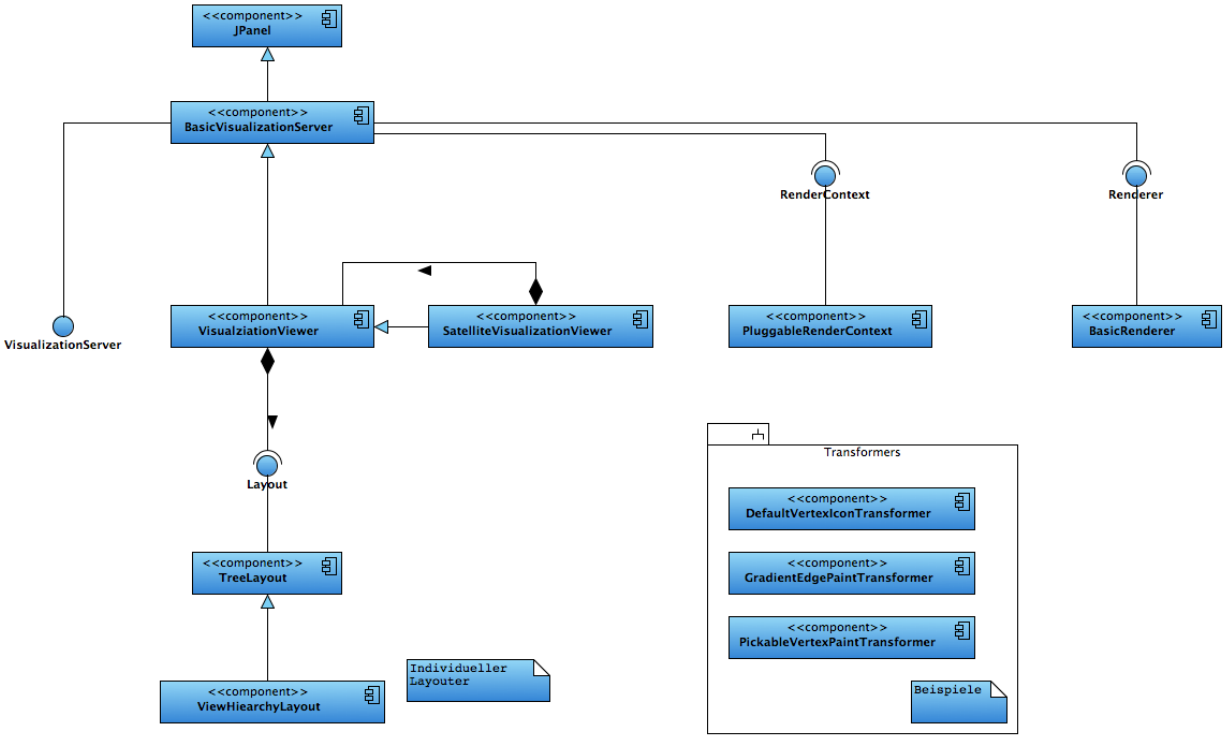


Abbildung A.11: Die JUNG-Architektur im Überblick

Abbildung A.12: Anwendung mit Willkommensbildschirm

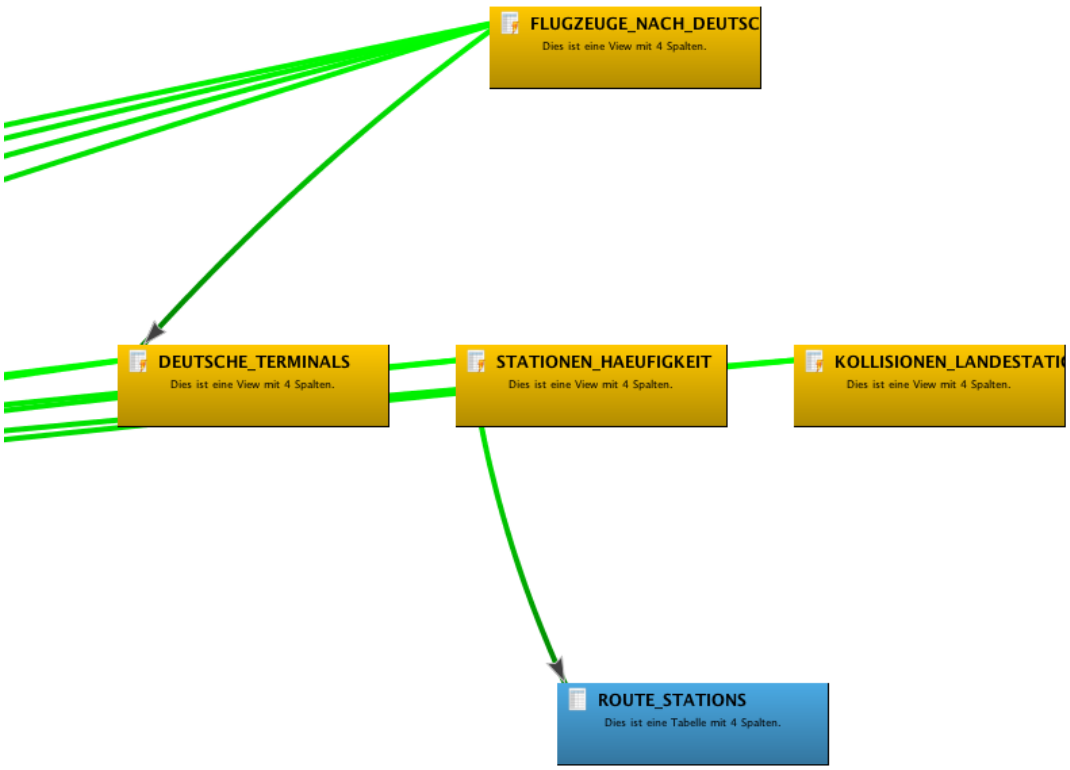


Abbildung A.13: Anwendung mit Übersicht der View-Hierarchie

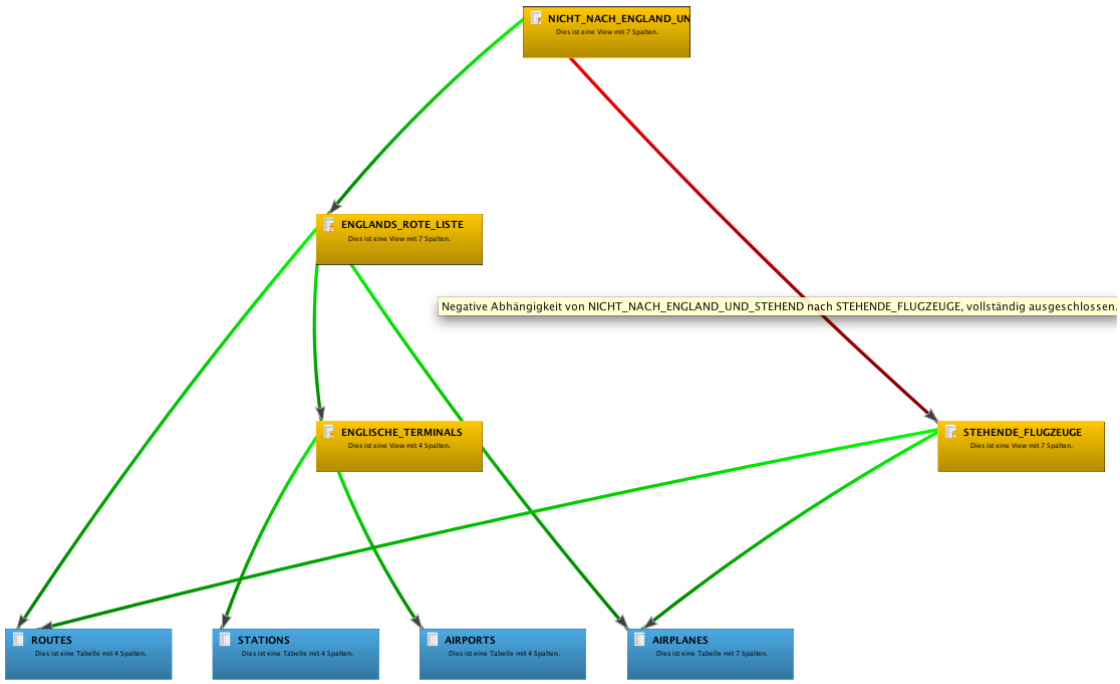


Abbildung A.14: Anwendung mit Anzeige einer negativen View-Abhängigkeit

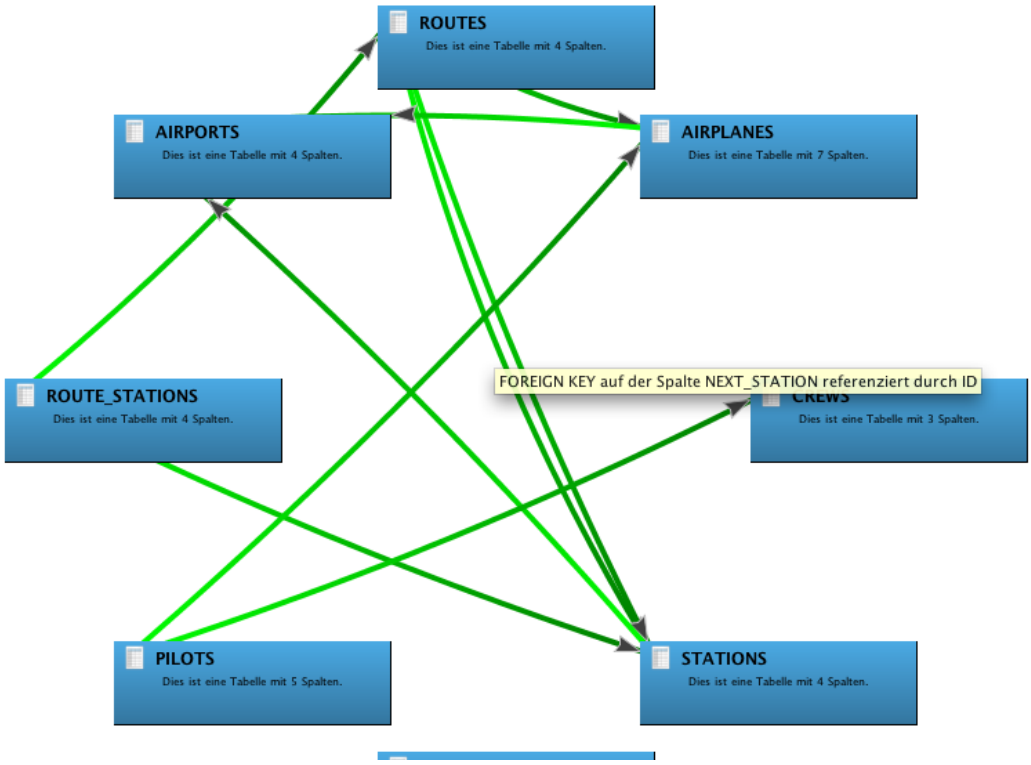


Abbildung A.15: Anwendung mit Anzeige eines einfachen ERDs

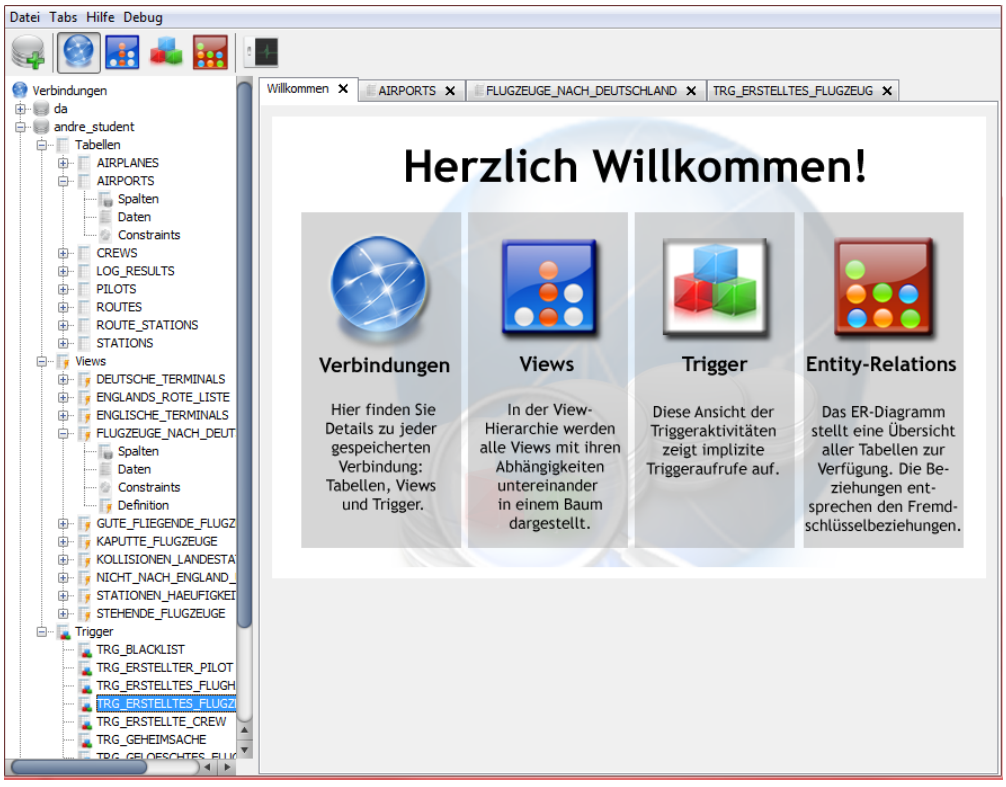


Abbildung A.16: Anwendung mit Anzeige der Verbindungsübersicht

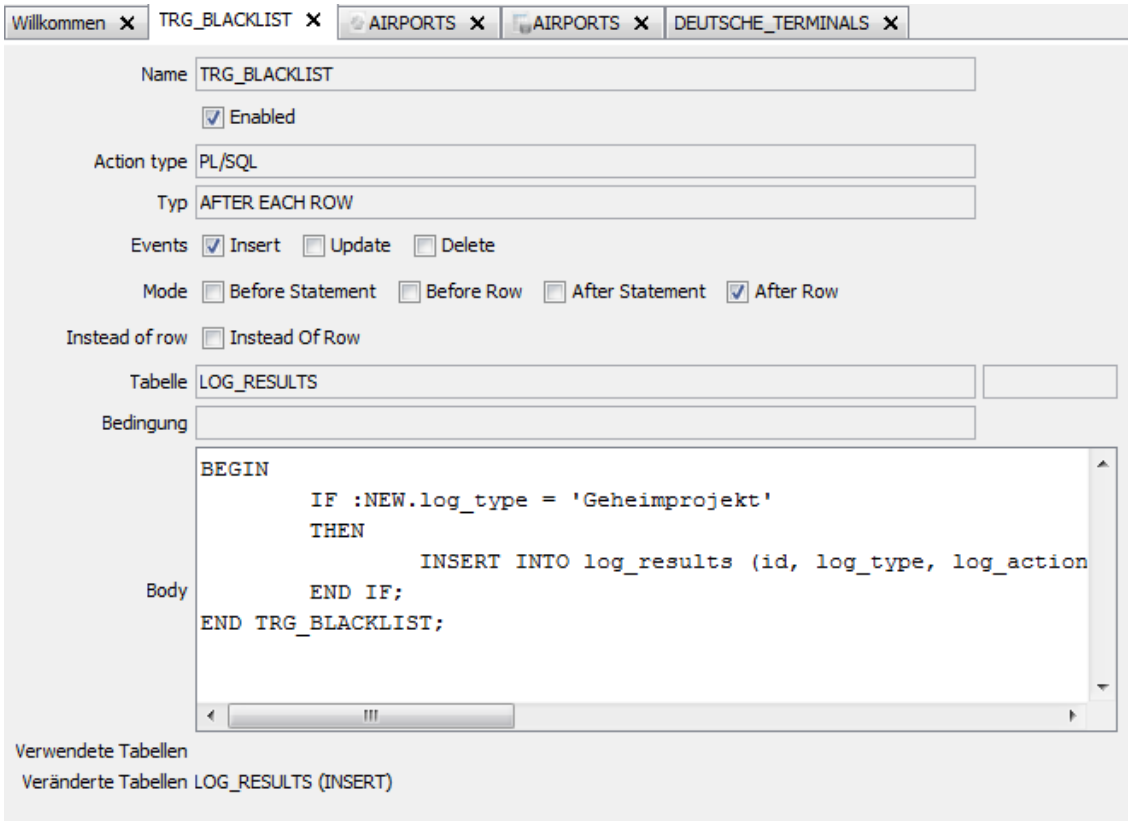


Abbildung A.17: Anwendung mit Anzeige der Verbindungsübersicht – Triggerdefinition

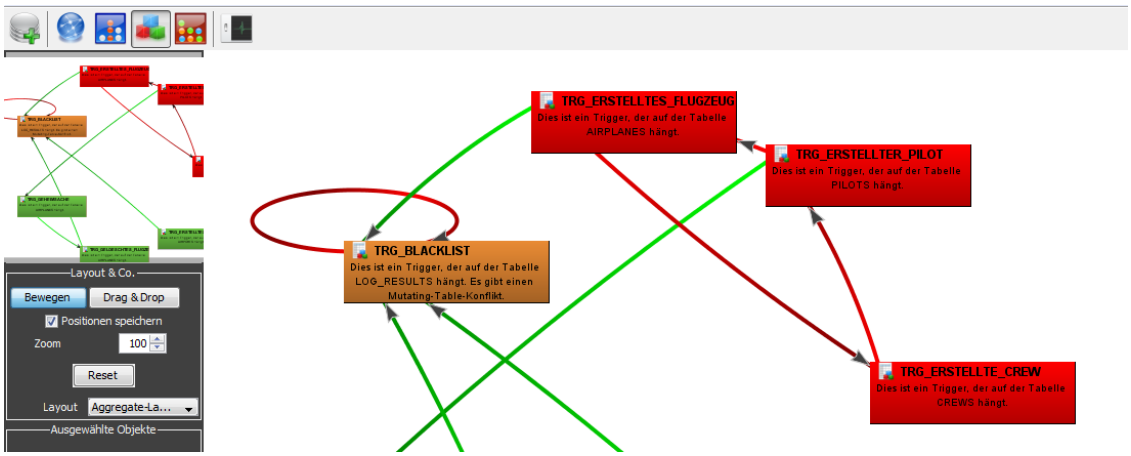


Abbildung A.18: Anwendung mit Anzeige der Triggerabhängigkeiten

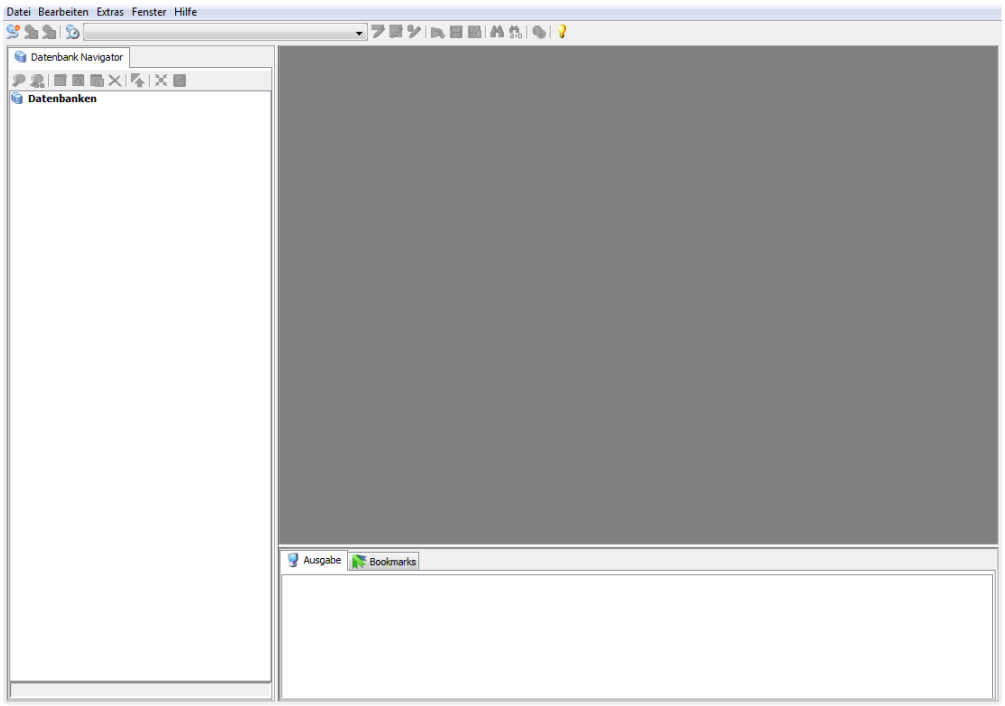


Abbildung A.19: SQL Developer – Übersicht

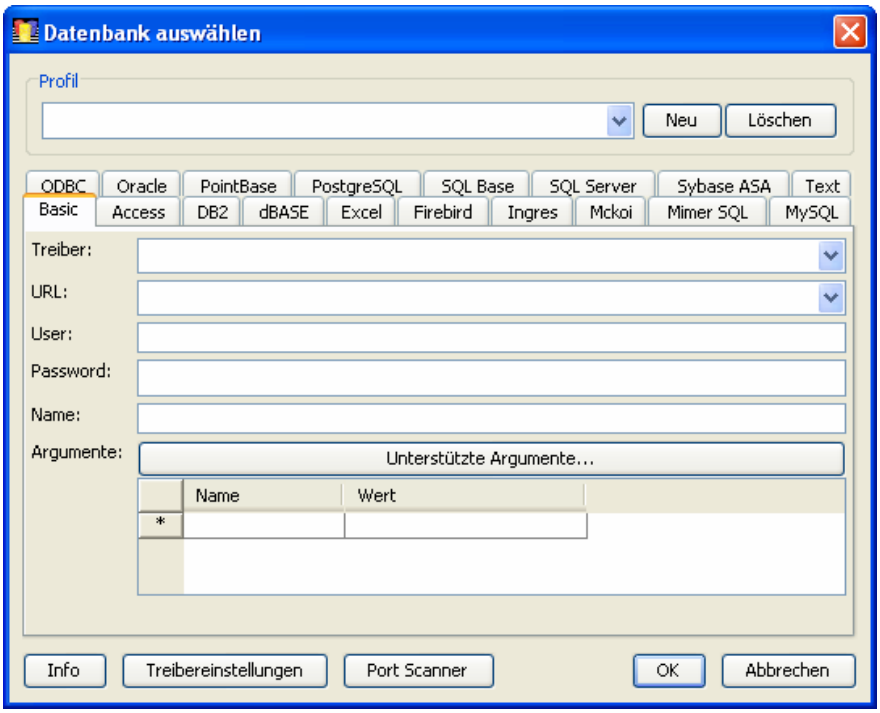


Abbildung A.20: SQL Developer – Verbindungen

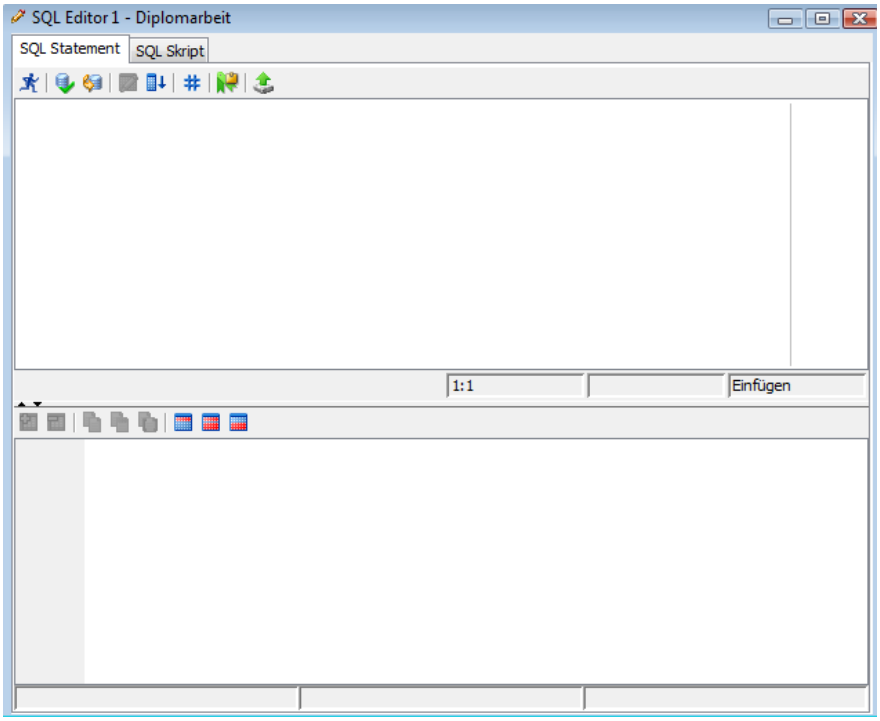


Abbildung A.21: SQL Developer – Editor

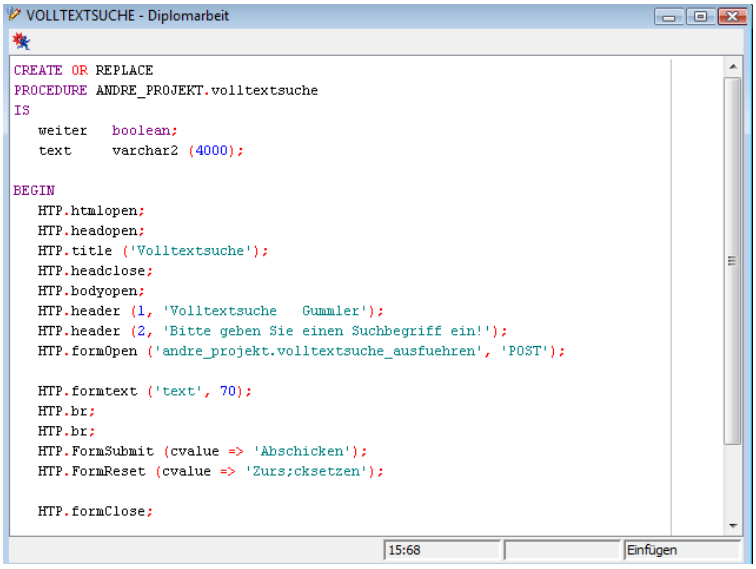


Abbildung A.22: SQL Developer – Procedure-Viewer

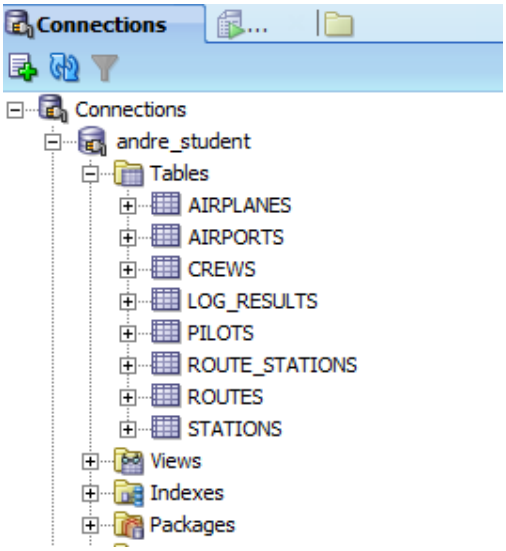


Abbildung A.23: Oracle SQL Developer – Übersicht

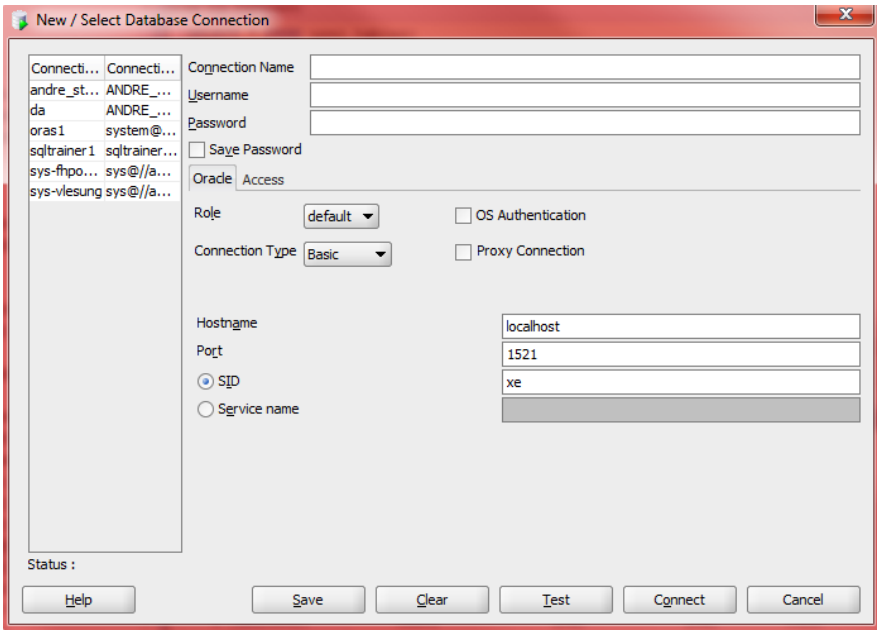
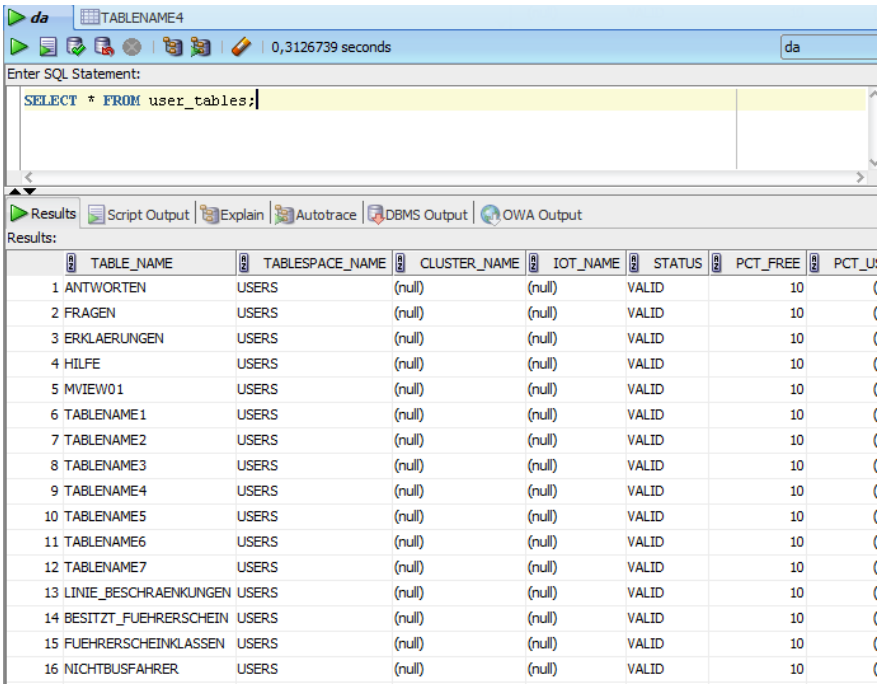


Abbildung A.24: Oracle SQL Developer – Verbindung



	TABLE_NAME	TABLESPACE_NAME	CLUSTER_NAME	IOT_NAME	STATUS	PCT_FREE	PCT_U
1	ANTWORTEN	USERS	(null)	(null)	VALID	10	(
2	FRAGEN	USERS	(null)	(null)	VALID	10	(
3	ERKLAERUNGEN	USERS	(null)	(null)	VALID	10	(
4	HILFE	USERS	(null)	(null)	VALID	10	(
5	MVIEW01	USERS	(null)	(null)	VALID	10	(
6	TABLENAME1	USERS	(null)	(null)	VALID	10	(
7	TABLENAME2	USERS	(null)	(null)	VALID	10	(
8	TABLENAME3	USERS	(null)	(null)	VALID	10	(
9	TABLENAME4	USERS	(null)	(null)	VALID	10	(
10	TABLENAME5	USERS	(null)	(null)	VALID	10	(
11	TABLENAME6	USERS	(null)	(null)	VALID	10	(
12	TABLENAME7	USERS	(null)	(null)	VALID	10	(
13	LINIE_BESCHRAENKUNGEN	USERS	(null)	(null)	VALID	10	(
14	BESITZT_FUEHRERSCHEIN	USERS	(null)	(null)	VALID	10	(
15	FUEHRERSCHEINKLASSEN	USERS	(null)	(null)	VALID	10	(
16	NICHTBUSFAHRER	USERS	(null)	(null)	VALID	10	(

Abbildung A.25: Oracle SQL Developer – Editor

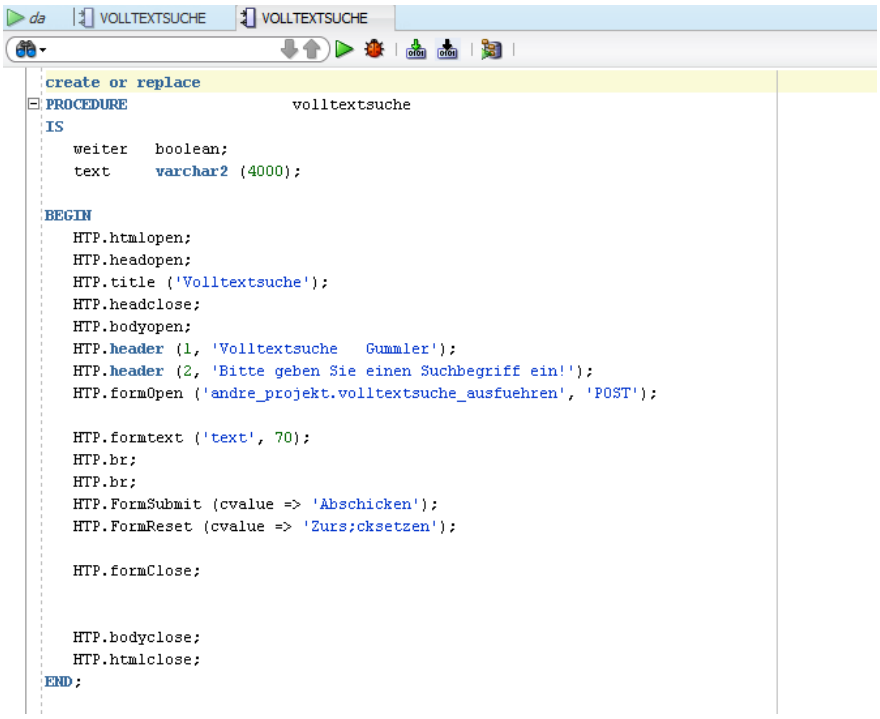


Abbildung A.26: Oracle SQL Developer – Procedure Editor

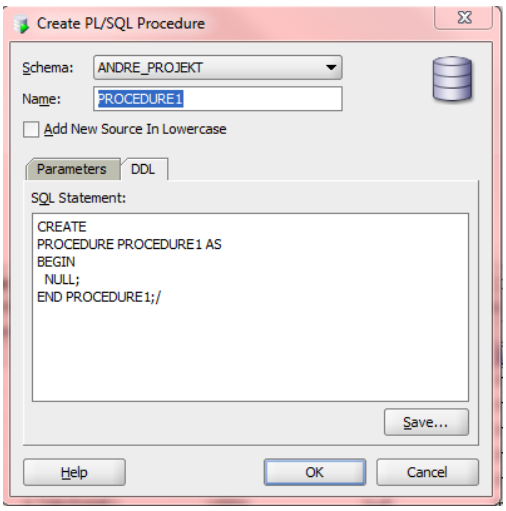


Abbildung A.27: Oracle SQL Developer – Neue Prozedur

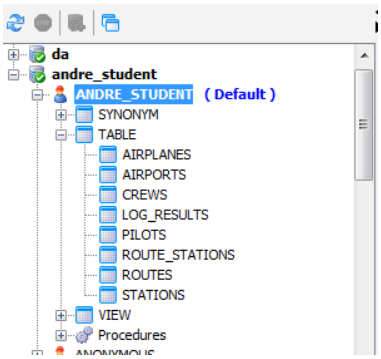


Abbildung A.28: DbVisualizer – Verbindungsübersicht

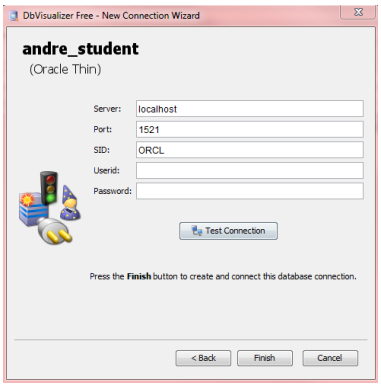


Abbildung A.29: DbVisualizer – Neue Verbindung

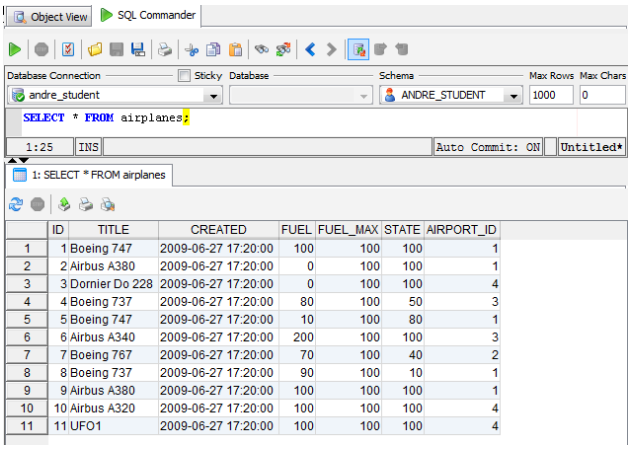


Abbildung A.30: DbVisualizer – Editor

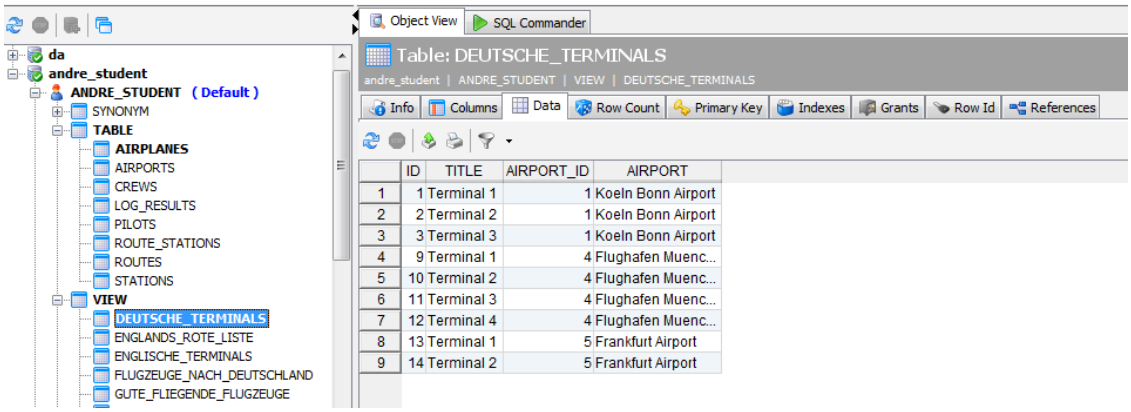


Abbildung A.31: DbVisualizer – Objektdaten-Anzeige

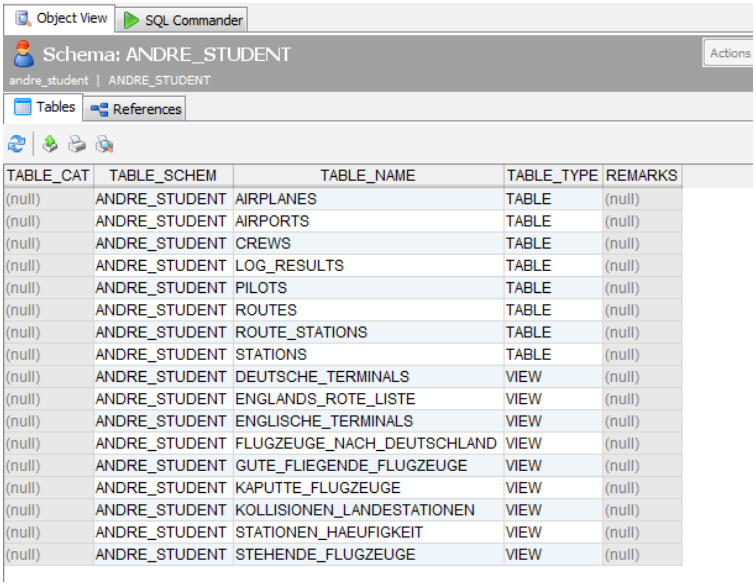


Abbildung A.32: DbVisualizer – Objekt-Anzeige

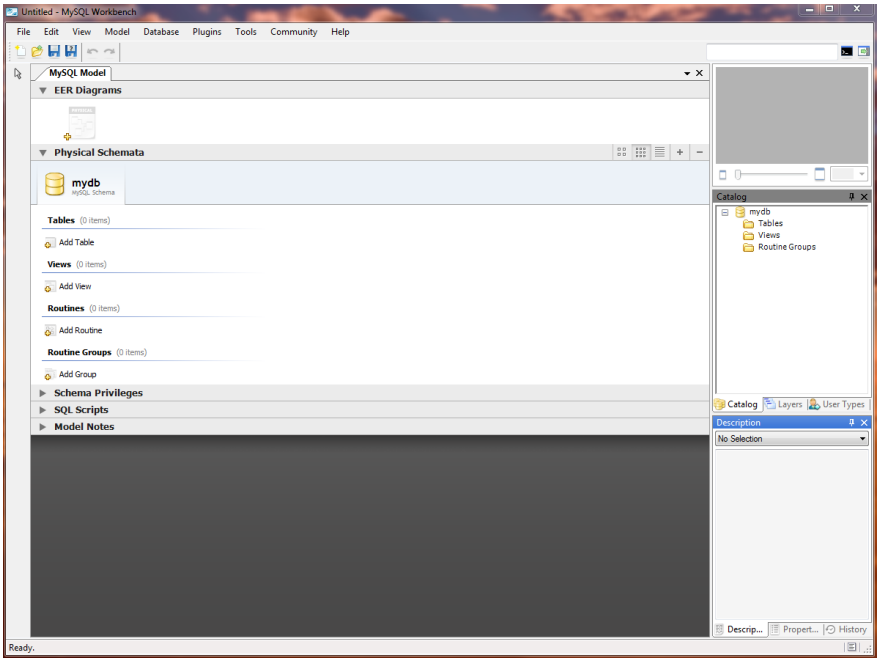


Abbildung A.33: MySQL Workbench – Übersicht

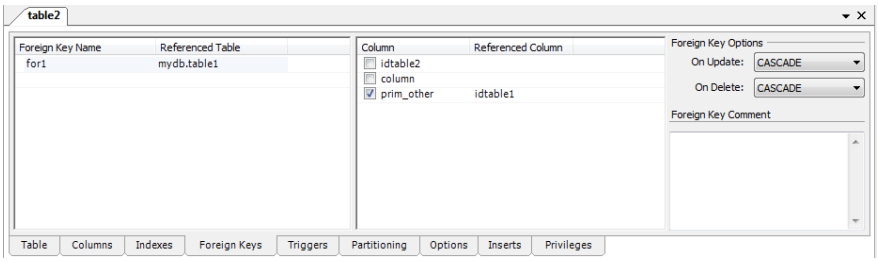


Abbildung A.34: MySQL Workbench – Tabellen anlegen

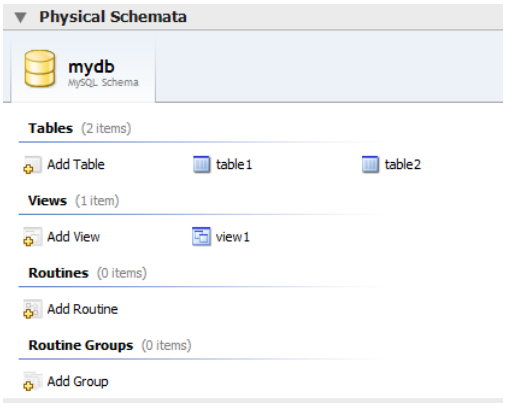


Abbildung A.35: MySQL Workbench – Schemaübersicht

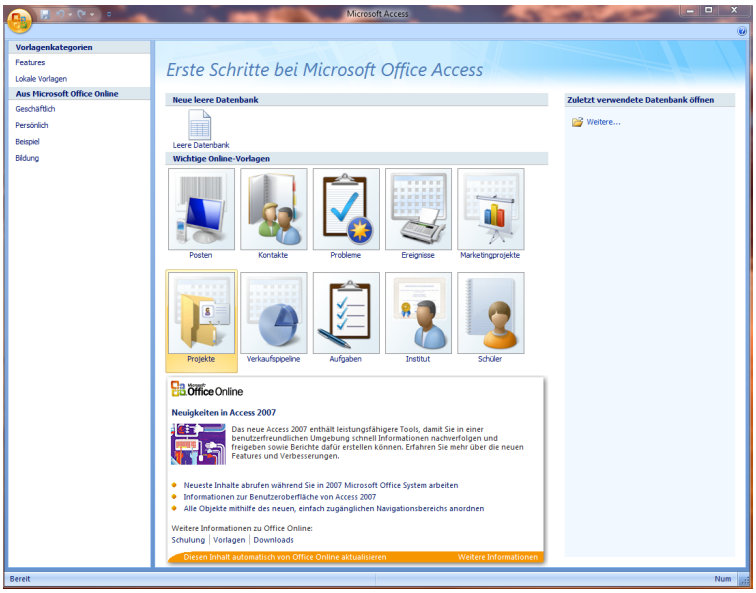


Abbildung A.36: Microsoft Office Access – Übersicht

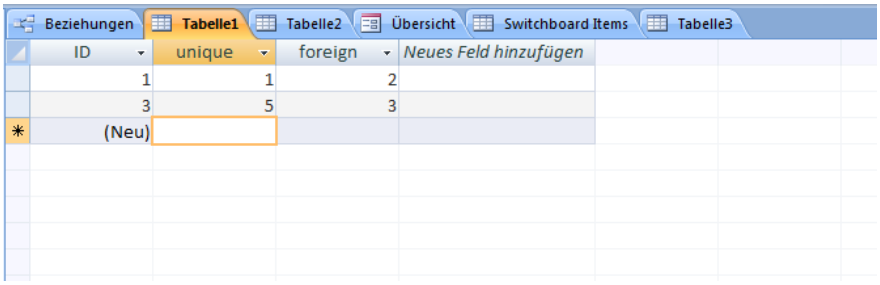


Abbildung A.37: Microsoft Office Access – Tabellen anlegen

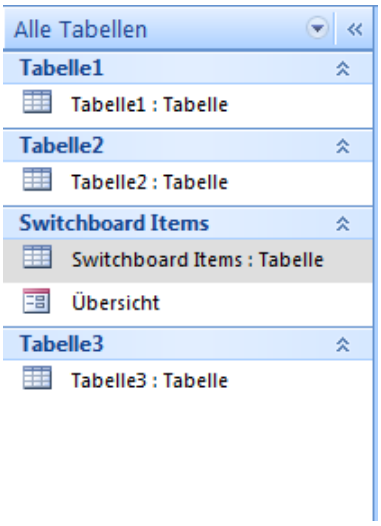


Abbildung A.38: Microsoft Office Access – Tabellenübersicht

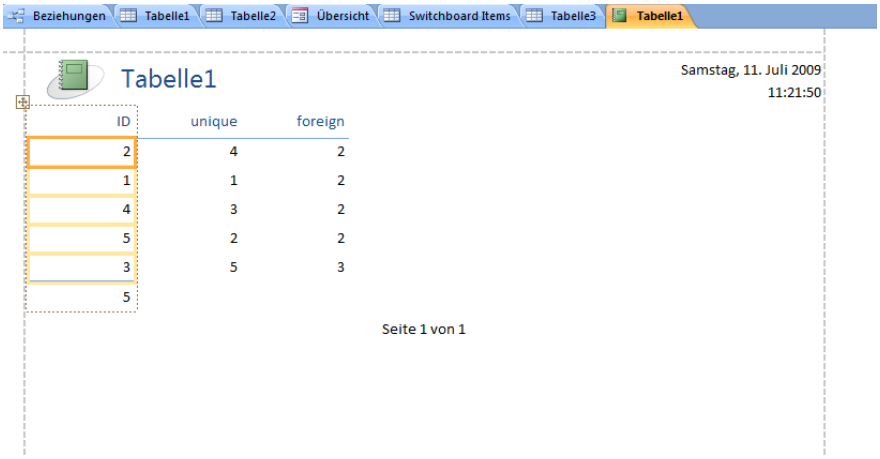


Abbildung A.39: Microsoft Office Access – Bericht

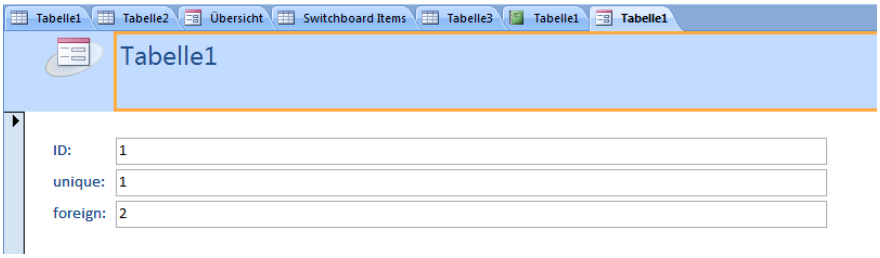


Abbildung A.40: Microsoft Office Access – Eingabeformular

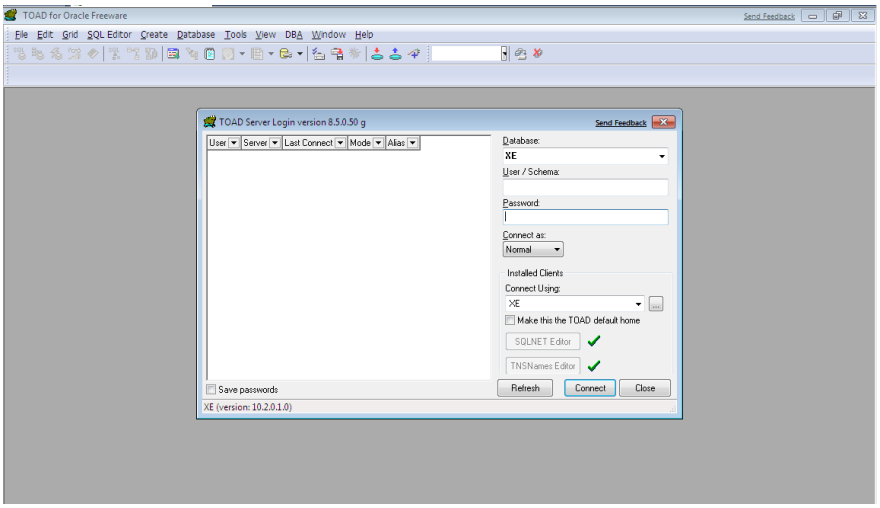


Abbildung A.41: TOAD – Startansicht

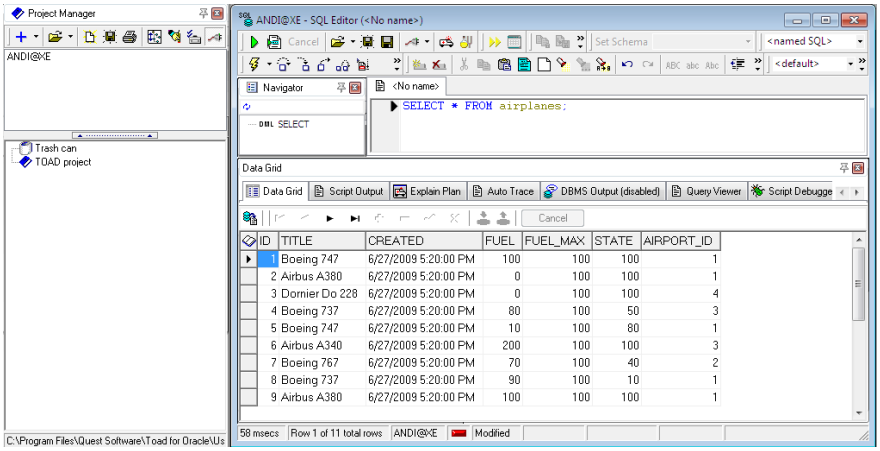


Abbildung A.42: TOAD – SQL Editor

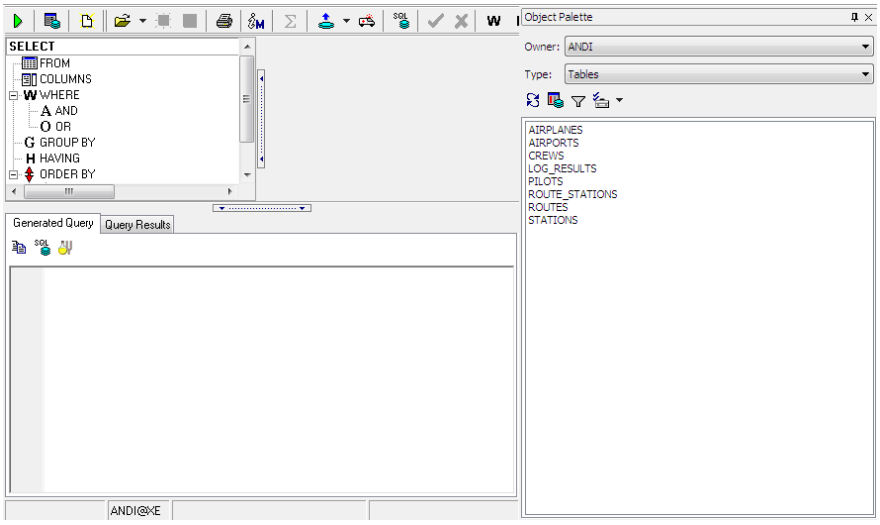


Abbildung A.43: TOAD – SQL Modeler

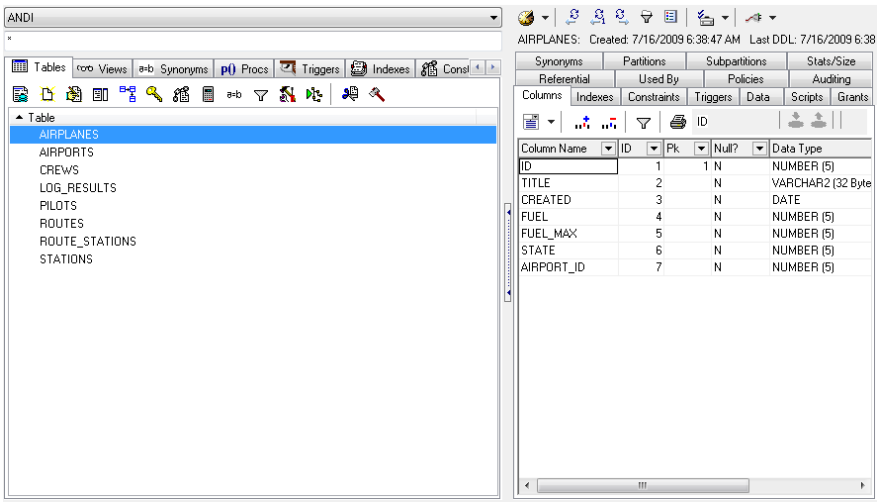


Abbildung A.44: TOAD – Schema Browser

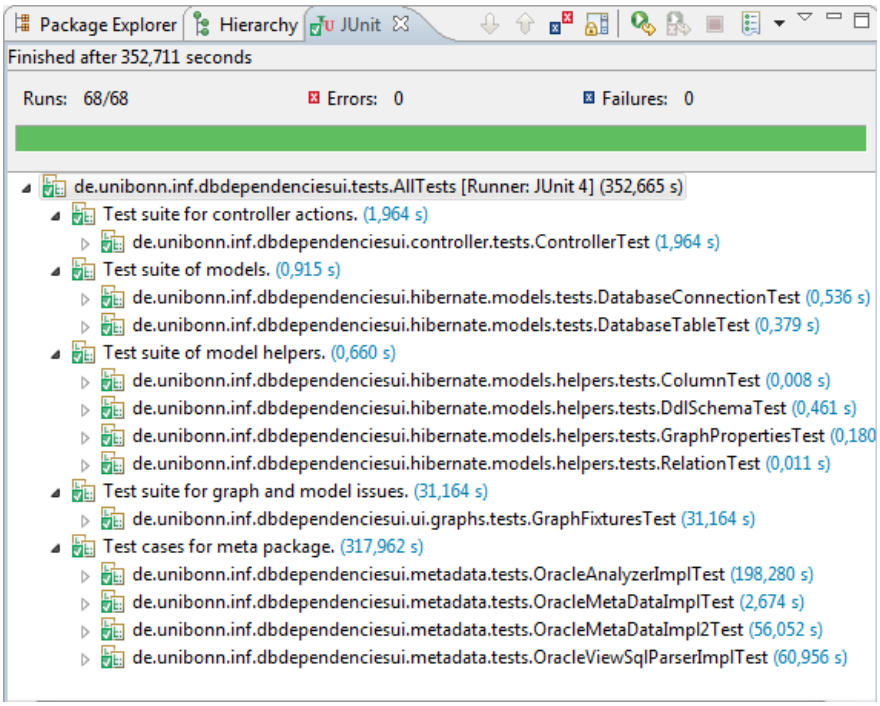


Abbildung A.45: Testergebnis – JUnit-Testfälle

Element	Coverage	Covered Instructions	Total Instructions
DA-App	73,4 %	28724	39122
src	73,4 %	28724	39122
de.unibonn.inf.dbdependenciesui	66,9 %	174	260
de.unibonn.inf.dbdependenciesui.apple	0,0 %	0	314
de.unibonn.inf.dbdependenciesui.controller	68,1 %	323	474
de.unibonn.inf.dbdependenciesui.controller.tests	92,7 %	38	41
de.unibonn.inf.dbdependenciesui.graph.common	9,6 %	35	364
de.unibonn.inf.dbdependenciesui.graph.entityrelations	62,0 %	150	242
de.unibonn.inf.dbdependenciesui.graph.triggers	75,7 %	287	379
de.unibonn.inf.dbdependenciesui.graph.viewhierarchy	85,8 %	494	576
de.unibonn.inf.dbdependenciesui.hibernate	92,4 %	97	105
de.unibonn.inf.dbdependenciesui.hibernate.impl	90,9 %	170	187
de.unibonn.inf.dbdependenciesui.hibernate.models	76,9 %	627	815
de.unibonn.inf.dbdependenciesui.hibernate.models.helpers	92,4 %	3355	3630
de.unibonn.inf.dbdependenciesui.hibernate.models.helpers.tests	47,0 %	641	1363
de.unibonn.inf.dbdependenciesui.hibernate.models.tests	93,6 %	265	283
de.unibonn.inf.dbdependenciesui.metadata	62,9 %	122	194
de.unibonn.inf.dbdependenciesui.metadata.impl	46,3 %	3610	7804
de.unibonn.inf.dbdependenciesui.metadata.progresstask	91,7 %	516	563
de.unibonn.inf.dbdependenciesui.metadata.tests	98,4 %	1999	2031
de.unibonn.inf.dbdependenciesui.misc	87,8 %	338	385
de.unibonn.inf.dbdependenciesui.tests	76,6 %	36	47
de.unibonn.inf.dbdependenciesui.ui	84,1 %	1891	2248
de.unibonn.inf.dbdependenciesui.ui.controller	80,8 %	463	573
de.unibonn.inf.dbdependenciesui.ui.factory	37,0 %	10	27
de.unibonn.inf.dbdependenciesui.ui.graphs.tests	92,1 %	35	38
de.unibonn.inf.dbdependenciesui.ui.helpers	48,8 %	390	799
de.unibonn.inf.dbdependenciesui.ui.misc	66,2 %	479	724
de.unibonn.inf.dbdependenciesui.ui.views.common	90,0 %	1929	2143
de.unibonn.inf.dbdependenciesui.ui.views.common.graph	96,1 %	637	663
de.unibonn.inf.dbdependenciesui.ui.views.common.graph.layout	76,9 %	170	221
de.unibonn.inf.dbdependenciesui.ui.views.common.graph.plugins	89,5 %	238	266

Abbildung A.46: Testergebnis – Code-Coverage

EMMA Coverage Report (generated Mon Jun 22 19:40:46 CEST 2009)

[all classes]

OVERALL COVERAGE SUMMARY

name	class, %	method, %	block, %	line, %
all classes	89% (172/194)	73% (1104/1506)	73% (29086/40106)	72% (6318.4/8785)

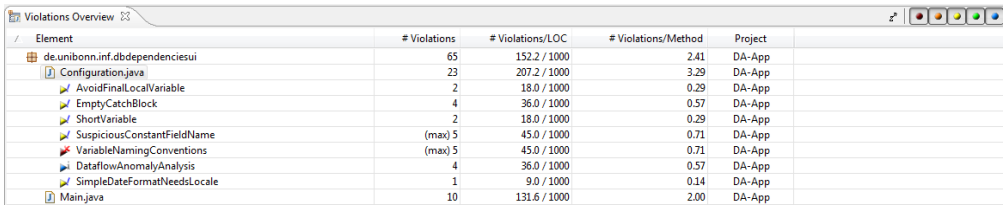
OVERALL STATS SUMMARY

total packages: 30
total executable files: 125
total classes: 194
total methods: 1506
total executable lines: 8785

COVERAGE BREAKDOWN BY PACKAGE

name	class, %	method, %	block, %	
de.unibonn.inf.dbdependenciesui.apple	0% (0/2)	0% (0/12)	0% (0/314)	0%
de.unibonn.inf.dbdependenciesui.ui.hierarchicalview.graph.plugins	25% (1/4)	15% (2/13)	6% (15/251)	12%
de.unibonn.inf.dbdependenciesui.ui.hierarchicalview.graph.renderer	38% (3/8)	40% (21/52)	37% (808/2185)	41%
de.unibonn.inf.dbdependenciesui.ui.hierarchicalview.graph.transformers	100% (7/7)	77% (17/22)	48% (195/410)	59%
de.unibonn.inf.dbdependenciesui.metadata.impl	81% (13/16)	60% (127/212)	49% (4589/9415)	49%
de.unibonn.inf.dbdependenciesui.ui.helpers	88% (7/8)	40% (19/47)	49% (390/799)	38%
de.unibonn.inf.dbdependenciesui.hibernate.models.tests	67% (2/3)	35% (8/23)	51% (126/245)	45%
de.unibonn.inf.dbdependenciesui.ui.misc	86% (6/7)	58% (22/38)	66% (479/724)	64%
de.unibonn.inf.dbdependenciesui.controller	100% (2/2)	73% (22/30)	69% (262/377)	70%
de.unibonn.inf.dbdependenciesui.metadata	100% (2/2)	85% (11/13)	72% (139/194)	61%
de.unibonn.inf.dbdependenciesui.hibernate.models	100% (6/6)	76% (65/85)	75% (608/809)	74%
de.unibonn.inf.dbdependenciesui.ui.factory	100% (2/2)	74% (14/19)	76% (257/339)	83%
de.unibonn.inf.dbdependenciesui.tests	50% (1/2)	40% (2/5)	77% (36/47)	78%
de.unibonn.inf.dbdependenciesui.hibernate.models.helpers.tests	90% (9/10)	73% (55/75)	77% (1172/1528)	79%
de.unibonn.inf.dbdependenciesui	100% (4/4)	75% (21/28)	79% (792/997)	80%
de.unibonn.inf.dbdependenciesui.ui.connectionview.helpers	100% (14/14)	77% (54/70)	80% (1924/2395)	81%
de.unibonn.inf.dbdependenciesui.ui.hierarchicalview.graph	100% (8/8)	69% (52/75)	82% (1269/1545)	81%
de.unibonn.inf.dbdependenciesui.metadata.progresstask	100% (4/4)	68% (26/38)	82% (879/1067)	85%
de.unibonn.inf.dbdependenciesui.misc	100% (4/4)	90% (26/29)	86% (333/385)	80%
de.unibonn.inf.dbdependenciesui.ui	100% (14/14)	93% (63/68)	87% (1303/1492)	87%
de.unibonn.inf.dbdependenciesui.ui.hierarchicalview	88% (15/17)	85% (93/110)	89% (2411/2723)	87%
de.unibonn.inf.dbdependenciesui.ui.hierarchicalview.graph.layout	100% (2/2)	81% (17/21)	89% (621/695)	86%
de.unibonn.inf.dbdependenciesui.ui.controller	100% (4/4)	95% (42/44)	90% (438/487)	89%
de.unibonn.inf.dbdependenciesui.hibernate.impl	100% (5/5)	87% (20/23)	91% (170/187)	88%
de.unibonn.inf.dbdependenciesui.hibernate.models.helpers	100% (14/14)	84% (156/185)	91% (2953/3237)	89%
de.unibonn.inf.dbdependenciesui.ui.graphs.tests	100% (2/2)	83% (5/6)	92% (35/38)	94%
de.unibonn.inf.dbdependenciesui.hibernate	100% (2/2)	80% (12/15)	92% (97/105)	94%
de.unibonn.inf.dbdependenciesui.controller.tests	100% (2/2)	83% (5/6)	93% (38/41)	94%
de.unibonn.inf.dbdependenciesui.ui.connectionview	92% (12/13)	95% (70/74)	94% (1858/1986)	95%
de.unibonn.inf.dbdependenciesui.metadata.tests	83% (5/6)	84% (57/68)	96% (4889/5089)	97%

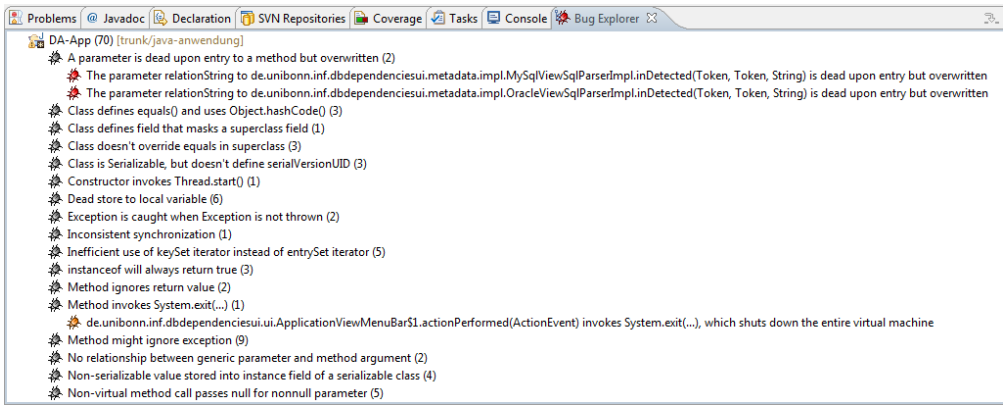
Abbildung A.47: Testergebnis – Code-Coverage als HTML-Ausgabe



Violations Overview

Element	# Violations	# Violations/LOC	# Violations/Method	Project
de.unibonn.inf.dbdependenciesui	65	152.2 / 1000	2.41	DA-App
Configuration.java	23	207.2 / 1000	3.29	DA-App
AvoidFinalLocalVariable	2	18.0 / 1000	0.29	DA-App
EmptyCatchBlock	4	36.0 / 1000	0.57	DA-App
ShortVariable	2	18.0 / 1000	0.29	DA-App
SuspiciousConstantFieldName	(max) 5	45.0 / 1000	0.71	DA-App
VariableNamingConventions	(max) 5	45.0 / 1000	0.71	DA-App
DataflowAnomalyAnalysis	4	36.0 / 1000	0.57	DA-App
SimpleDateFormatNeedsLocale	1	9.0 / 1000	0.14	DA-App
Main.java	10	131.6 / 1000	2.00	DA-App

Abbildung A.48: Testergebnis – PMD Konsolen-Ausgabe



Problems | Javadoc | Declaration | SVN Repositories | Coverage | Tasks | Console | Bug Explorer

DA-App (70) [trunk/java-anwendung]

- A parameter is dead upon entry to a method but overwritten (2)
  - The parameter relationString to de.unibonn.inf.dbdependenciesui.metadata.impl.MySqlViewSqlParserImpl.inDetected(Token, Token, String) is dead upon entry but overwritten
  - The parameter relationString to de.unibonn.inf.dbdependenciesui.metadata.impl.OracleViewSqlParserImpl.inDetected(Token, Token, String) is dead upon entry but overwritten
- Class defines equals() and uses Object.hashCode() (3)
- Class defines field that masks a superclass field (1)
- Class doesn't override equals in superclass (3)
- Class is Serializable, but doesn't define serialVersionUID (3)
- Constructor invokes Thread.start() (1)
- Dead store to local variable (6)
- Exception is caught when Exception is not thrown (2)
- Inconsistent synchronization (1)
- Inefficient use of keySet iterator instead of entrySet iterator (5)
- instanceof will always return true (3)
- Method ignores return value (2)
- Method invokes System.exit(...) (1)
  - de.unibonn.inf.dbdependenciesui.ui.ApplicationViewMenuBar$1.actionPerformed(ActionEvent) invokes System.exit(...), which shuts down the entire virtual machine
- Method might ignore exception (9)
- No relationship between generic parameter and method argument (2)
- Non-serializable value stored into instance field of a serializable class (4)
- Non-virtual method call passes null for nonnull parameter (5)

Abbildung A.49: Testergebnis – FingBugs Konsolen-Ausgabe

B. Anhang – Programmcodes

Listing 22: Laden des serialisierten Schemas

```
public DdlSchema getDdlSchemaObject() throws IllegalArgumentException {
  if (cachedDdlschema == null) {
    DdlSchema result = null;
    try {
      result = new DdlSchema(ddlschema);
    } catch (final Exception e) {
      result = new DdlSchema();
    }
    result.setConnection(getConnection());
    cachedDdlschema = result;
    return result;
  } else {
    return cachedDdlschema;
  }
}
```

Listing 23: Serialisieren des Schemas

```
public void setDdlSchemaObject(final DdlSchema ddlschema) {
  try {
    setDdlSchema(ddlschema.serialize());
    cachedDdlschema = ddlschema;
  } catch (final Exception e) {
    Logger.getLogger(Configuration.LOGGER).warning("Could not serialize ddlschema.");
  }
}
```

Listing 24: Beispiel eines View-Schemas in XML

```
<schema>
  <columns>
    <column>
      <name>MITA_ID</name>
      <type>NUMBER</type>
      <length>
        <size>9</size>
        <fractionaldigits>0</fractionaldigits>
      </length>
      <is-nullable>true</is-nullable>
    </column>
    <column>
      <name>VORNAME</name>
```

```
      <type>VARCHAR2</type>
      <length>
        <size>20</size>
        <fractionaldigits>0</fractionaldigits>
      </length>
      <is-nullable>true</is-nullable>
    </column>
    <column>
      <name>NACHNAME</name>
      <type>VARCHAR2</type>
      <length>
        <size>25</size>
        <fractionaldigits>0</fractionaldigits>
      </length>
      <is-nullable>true</is-nullable>
    </column>
  </columns>
  <keys/>
  <relations>
    <view>
      <name>MITARBEITER_VIEW_O</name>
      <source>MITARBEITER_VIEW_O</source>
      <target>MITARBEITER</target>
      <column>MITA_ID</column>
      <condition>e.mita_id = m.mita_id</condition>
      <is-positive>true</is-positive>
      <is-and>false</is-and>
    </view>
    <view>
      <name>MITARBEITER_VIEW_O</name>
      <source>MITARBEITER_VIEW_O</source>
      <target>EINSATZPLAN</target>
      <column>MITA_ID</column>
      <condition>e.mita_id = m.mita_id</condition>
      <is-positive>true</is-positive>
      <is-and>false</is-and>
    </view>
  </relations>
  <triggers/>
</schema>
```

Listing 25: YAML-Definition des Verbindungsdialogs

```
JDialog(name=frame, title=application.connection.newconnection, size=packed,
    defaultCloseOperation=disposeOnClose):
```

```
content:
- JButton(name=btnReset,text=application.connection.action.reset,onAction=resetDialog)
- JButton(name=btnCancel,text=application.connection.action.cancel,onAction=
    cancelDialog)
- JButton(name=btnTest,text=application.connection.action.test,onAction=($validate,
    testConnection))
- JButton(name=btnSave,text=application.connection.action.save,onAction=($validate,
    saveConnection))
- JLabel(name=lblTitle,text=application.connection.title)
- JLabel(name=lblHost,text=application.connection.host)
- JLabel(name=lblPort,text=application.connection.port)
- JLabel(name=lblUsername,text=application.connection.username)
- JLabel(name=lblPassword,text=application.connection.password)
- JLabel(name=lblDatabase,text=application.connection.database)
- JLabel(name=lblSchema,text=application.connection.schema)
- JLabel(name=help, text=application.connection.help, font=bold)
- JLabel(name=statusIcon)
- JLabel(name=statusText, text=application.connection.status)
- JLabel(name=feedbackText, text=application.connection.status)
- JLabel(name=lblSysdba, text=application.connection.sysdba)
- JTextField(name=title)
- JComboBox(name=vendors)
- JTextField(name=host)
- JTextField(name=port)
- JTextField(name=username)
- JPasswordField(name=password)
- JTextField(name=database)
- JTextField(name=schema)
- JCheckBox(name=updateSchema, text=application.connection.updateSchema)
- JCheckBox(name=checkSysdba, text=application.connection.sysdba)
- MigLayout: |
  [pref]              [grow]        [pref]   [80]
  >lblTitle           title+*
  >lblHost            host          >lblPort port
  >lblUsername        username+*
  >lblPassword        password+*
  >lblDatabase        database+* vendors
  >lblSchema          schema+*
           <checkSysdba+*
           <updateSchema+*
  <help+*
  <statusIcon statusText+*
  <feedbackText+*
  <btnReset           btnCancel >btnTest+* btnSave
```

```
validate:
  - title.text: {label: application.connection.title, mandatory: true}
  - host.text: {label: application.connection.host, mandatory: true}
  - port.text: {label: application.connection.port, mandatory: true, regex: "
      (6553[0-5]|655[0-2]\\d|65[0-4]\\d\\d|6[0-4]\\d{3}|[1-5]\\d{4}|[1-9]\\d{0,3}|0)",
      regexMessage: "\"{0}\" must be a port number."}
  - username.text: {label: application.connection.username, mandatory: true}
  - database.text: {label: application.connection.database, mandatory: true}
```

Listing 26: Beispielschema Flughafen

```
DROP VIEW deutsche_terminals;
DROP VIEW englische_terminals;
DROP VIEW flugzeuge_nach_deutschland;
DROP VIEW englands_rote_liste;
DROP VIEW stationen_haeufigkeit;
DROP VIEW kaputte_flugzeuge;
DROP VIEW gute_fliegende_flugzeuge;
DROP VIEW stehende_flugzeuge;
DROP VIEW kollisionen_landestationen;
DROP VIEW NICHT_NACH_ENGLAND_UND_STEHEND;

DROP TRIGGER trg_erstelltes_flugzeug;
DROP TRIGGER trg_geloeschtes_flugzeug;
DROP TRIGGER trg_erstelltes_flughafen;
DROP TRIGGER trg_erstellte_crew;
DROP TRIGGER trg_erstellter_pilot;
DROP TRIGGER trg_blacklist;
DROP TRIGGER trg_geheimsache;

DROP TABLE pilots;
DROP TABLE crews;
DROP TABLE route_stations;
DROP TABLE routes;
DROP TABLE stations;
DROP TABLE airplanes;
DROP TABLE airports;
DROP TABLE log_results;

DROP SEQUENCE airports_seq;
DROP SEQUENCE pilots_seq;
DROP SEQUENCE crews_seq;
DROP SEQUENCE airplanes_seq;
DROP SEQUENCE routes_seq;
DROP SEQUENCE stations_seq;
```

```
35 DROP SEQUENCE route_stations_seq;
36 DROP SEQUENCE log_results_seq;

38 CREATE SEQUENCE airports_seq;
39 CREATE SEQUENCE pilots_seq;
40 CREATE SEQUENCE crews_seq;
41 CREATE SEQUENCE airplanes_seq;
42 CREATE SEQUENCE routes_seq;
43 CREATE SEQUENCE stations_seq;
44 CREATE SEQUENCE route_stations_seq;
45 CREATE SEQUENCE log_results_seq;

47 CREATE TABLE airports (
48 id number(5) CONSTRAINT airports_pk PRIMARY KEY,
49 title varchar2(32) NOT NULL,
50 created date NOT NULL,
51 country varchar2(4) NOT NULL
52 );

54 CREATE TABLE airplanes (
55 id number(5) CONSTRAINT airplanes_pk PRIMARY KEY,
56 title varchar2(32) NOT NULL,
57 created date NOT NULL,
58 fuel number(5) NOT NULL,
59 fuel_max number(5) NOT NULL,
60 state number(5) NOT NULL,
61 airport_id number(5) NOT NULL,
62 constraint fk_airplane_airport foreign key (airport_id) references airports(id)
63 );

65 CREATE TABLE crews (
66 id number(5) CONSTRAINT crews_pk PRIMARY KEY,
67 name varchar2(32) NOT NULL,
68 created date NOT NULL
69 );

71 CREATE TABLE pilots (
72 id number(5) CONSTRAINT pilots_pk PRIMARY KEY,
73 name varchar2(32) NOT NULL,
74 created date NOT NULL,
75 airplane_id number(5) NULL,
76 crew_id number(5) NULL,
77 CONSTRAINT fk_pilot_airplane FOREIGN KEY (airplane_id) REFERENCES airplanes(id),
78 CONSTRAINT fk_pilot_crew FOREIGN KEY (crew_id) REFERENCES crews(id)
```

```
79 );

81 CREATE TABLE stations (
82 id number(5) CONSTRAINT stations_pk PRIMARY KEY,
83 title varchar2(32) NOT NULL,
84 airport_id number(5) NOT NULL,
85 created date NOT NULL,
86 constraint fk_station_airport foreign key (airport_id) references airports(id)
87 );

89 CREATE TABLE routes (
90 id number(5) CONSTRAINT routes_pk PRIMARY KEY,
91 current_station number(5) NOT NULL,
92 next_station number(5) NOT NULL,
93 airplane_id number(5) NOT NULL,
94 constraint fk_route_airplane foreign key (airplane_id) references airplanes(id),
95 constraint fk_route_station1 foreign key (current_station) references stations(id),
96 constraint fk_route_station2 foreign key (next_station) references stations(id)
97 );

99 CREATE TABLE route_stations (
100 id number(5) CONSTRAINT route_stations_pk PRIMARY KEY,
101 route_id number(5) NOT NULL,
102 station_id number(5) NOT NULL,
103 current_position number(5) NULL,
104 constraint fk_routestation_route foreign key (route_id) references routes(id),
105 constraint fk_routestation_station foreign key (station_id) references stations(id)
106 );

108 CREATE TABLE log_results (
109 id number(5) CONSTRAINT log_results_pk PRIMARY KEY,
110 log_type VARCHAR2(50) NOT NULL,
111 log_action VARCHAR2(255) NOT NULL,
112 created date NOT NULL
113 );



117 CREATE OR REPLACE TRIGGER trg_erstellte_crew
118 AFTER INSERT ON CREWS
119 FOR EACH ROW
120 BEGIN
121   INSERT INTO pilots (id, name, created, crew_id) VALUES (pilots_seq.nextval, 'John Doe',
        SYSDATE, :new.id);
```

```
122 END trg_erstellte_crew;
123 /

125 CREATE OR REPLACE TRIGGER trg_erstellter_pilot
126 AFTER INSERT ON PILOTS
127 FOR EACH ROW
128 BEGIN
129   INSERT INTO airplanes (id,title,created,fuel,fuel_max,state,airport_id) VALUES(
        airplanes_seq.nextval,'Privatmaschine',TO_DATE('2009-06-27 17:20:00', 'YYYY-MM-DD
        HH24:MI:SS'),100,100,100,4);
130 END trg_erstellter_pilot;
131 /

133 CREATE OR REPLACE TRIGGER TRG_GELOESCHTES_FLUGZEUG
134 AFTER DELETE ON AIRPLANES
135 FOR EACH ROW
136 BEGIN
137   INSERT INTO log_results (id, log_type, log_action, created) VALUES (log_results_seq.
        nextval, 'Gelöschtes Flugzeug', 'Es wurde ein Flugzeug "' || :OLD.title || '"
        gelöscht.', SYSDATE);
138 END TRG_GELOESCHTES_FLUGZEUG;
139 /

141 CREATE OR REPLACE TRIGGER trg_erstelltes_flughafen
142 AFTER INSERT ON AIRPORTS
143 FOR EACH ROW
144 BEGIN
145   INSERT INTO log_results (id, log_type, log_action, created) VALUES (log_results_seq.
        nextval, 'Neuer Flughafen', 'Es wurde ein neuer Flughafen "' || :new.title || '"
        erstellt.', SYSDATE);
146 END trg_erstelltes_flughafen;
147 /




152 CREATE OR REPLACE TRIGGER TRG_GEHEIMSACHE
153 AFTER INSERT OR UPDATE ON AIRPLANES
154 DECLARE
155   CURSOR cur_airplanes IS
156     SELECT id, title
157       FROM airplanes
158     WHERE title = 'UFO';
159 BEGIN
```

```
160   FOR rec_airplane IN cur_airplanes LOOP
161     DELETE FROM airplanes WHERE id = rec_airplane.id;
162   END LOOP;
163 END;
164 /

166 CREATE VIEW DEUTSCHE_TERMINALS AS
167   SELECT s.id, s.title, s.airport_id, a.title as airport
168   FROM stations s, airports a
169   WHERE s.airport_id = a.id
170   AND a.country = 'DE';

172 CREATE VIEW ENGLISCHE_TERMINALS AS
173   SELECT s.id, s.title, s.airport_id, a.title as airport
174   FROM stations s, airports a
175   WHERE s.airport_id = a.id
176   AND a.country = 'GB';

178 CREATE VIEW flugzeuge_nach_deutschland AS
179   SELECT p.id, p.title, a.title airport, s.title station
180   FROM airplanes p, routes r, stations s, airports a
181   WHERE p.id = r.airplane_id
182     AND r.next_station = s.id
183     AND a.id = s.airport_id
184     AND p.id IN (
185     SELECT r.airplane_id
186       FROM routes r, stations s
187     WHERE r.next_station = s.id
188       AND r.next_station IN (SELECT id FROM DEUTSCHE_TERMINALS)
189   );

191 CREATE VIEW englands_rote_liste AS
192   SELECT p.*
193   FROM airplanes p
194   WHERE p.id NOT IN (
195     SELECT r.airplane_id
196       FROM routes r, route_stations rs, stations s
197     WHERE r.id = rs.route_id
198       AND s.id = rs.station_id
199       AND s.id IN (SELECT id FROM ENGLISCHE_TERMINALS)
200   );

202 CREATE VIEW STATIONEN_HAEUFIGKEIT AS
203   SELECT s.id, s.title, s.airport_id, COUNT(s.id) cnt
```

```
204     FROM routes r, route_stations rs, stations s
205   WHERE rs.route_id = r.id
206     AND rs.station_id = s.id
207   GROUP BY s.id, s.title, s.airport_id
208   ORDER BY cnt DESC;

210 CREATE VIEW KAPUTTE_FLUGZEUGE AS
211   SELECT * FROM airplanes WHERE state < 50;

213 CREATE VIEW STEHENDE_FLUGZEUGE AS
214 SELECT p.* FROM airplanes p LEFT JOIN routes r ON (p.id = r.airplane_id) WHERE r.id IS
      NULL;

216 CREATE VIEW GUTE_FLIEGENDE_FLUGZEUGE AS
217   SELECT *
218     FROM airplanes
219   WHERE id NOT IN (SELECT id FROM KAPUTTE_FLUGZEUGE)
220     AND id NOT IN (SELECT id FROM STEHENDE_FLUGZEUGE);

222 CREATE VIEW KOLLISIONEN_LANDESTATIONEN AS
223   SELECT * FROM stations
224   WHERE id IN (
225       SELECT r.next_station
226         FROM routes r
227       GROUP BY (r.next_station)
228       HAVING COUNT(r.next_station) > 1
229     );

231 CREATE VIEW NICHT_NACH_ENGLAND_UND_STEHEND AS
232   SELECT * FROM englands_rote_liste
233   MINUS
234   SELECT * FROM stehende_flugzeuge;



237 INSERT INTO airports (id,title,created,country) VALUES (airports_seq.nextval,'Koeln Bonn
      Airport',TO_DATE('2009-06-27 17:20:00', 'YYYY-MM-DD HH24:MI:SS'),'DE');
238 INSERT INTO airports (id,title,created,country) VALUES (airports_seq.nextval,'Heathrow
      Airport',TO_DATE('2009-06-27 17:20:00', 'YYYY-MM-DD HH24:MI:SS'),'GB');
239 INSERT INTO airports (id,title,created,country) VALUES (airports_seq.nextval,'JFK Airport
      ',TO_DATE('2009-06-27 17:20:00', 'YYYY-MM-DD HH24:MI:SS'),'US');
240 INSERT INTO airports (id,title,created,country) VALUES (airports_seq.nextval,'Flughafen
      Muenchen',TO_DATE('2009-06-27 17:20:00', 'YYYY-MM-DD HH24:MI:SS'),'DE');
241 INSERT INTO airports (id,title,created,country) VALUES (airports_seq.nextval,'Frankfurt
      Airport',TO_DATE('2009-06-27 17:20:00', 'YYYY-MM-DD HH24:MI:SS'),'DE');
```

```
242 INSERT INTO airports (id,title,created,country) VALUES (airports_seq.nextval,'Flughafen
    Mailand-Malpensa',TO_DATE('2009-06-27 17:20:00', 'YYYY-MM-DD HH24:MI:SS'),'IT');
243 INSERT INTO airports (id,title,created,country) VALUES (airports_seq.nextval,'Canberra
    International Airport',TO_DATE('2009-06-27 17:20:00', 'YYYY-MM-DD HH24:MI:SS'),'AU');

245 INSERT INTO airplanes (id,title,created,fuel,fuel_max,state,airport_id) VALUES (
    airplanes_seq.nextval,'Boeing 747',TO_DATE('2009-06-27 17:20:00', 'YYYY-MM-DD HH24:MI
    :SS'),100,100,100,1);
246 INSERT INTO airplanes (id,title,created,fuel,fuel_max,state,airport_id) VALUES (
    airplanes_seq.nextval,'Airbus A380',TO_DATE('2009-06-27 17:20:00', 'YYYY-MM-DD HH24:
    MI:SS'),0,100,100,1);
247 INSERT INTO airplanes (id,title,created,fuel,fuel_max,state,airport_id) VALUES(
    airplanes_seq.nextval,'Dornier Do 228',TO_DATE('2009-06-27 17:20:00', 'YYYY-MM-DD
    HH24:MI:SS'),0,100,100,4);
248 INSERT INTO airplanes (id,title,created,fuel,fuel_max,state,airport_id) VALUES(
    airplanes_seq.nextval,'Boeing 737',TO_DATE('2009-06-27 17:20:00', 'YYYY-MM-DD HH24:MI
    :SS'),80,100,50,3);
249 INSERT INTO airplanes (id,title,created,fuel,fuel_max,state,airport_id) VALUES(
    airplanes_seq.nextval,'Boeing 747',TO_DATE('2009-06-27 17:20:00', 'YYYY-MM-DD HH24:MI
    :SS'),10,100,80,1);
250 INSERT INTO airplanes (id,title,created,fuel,fuel_max,state,airport_id) VALUES(
    airplanes_seq.nextval,'Airbus A340',TO_DATE('2009-06-27 17:20:00', 'YYYY-MM-DD HH24:
    MI:SS'),200,100,100,3);
251 INSERT INTO airplanes (id,title,created,fuel,fuel_max,state,airport_id) VALUES(
    airplanes_seq.nextval,'Boeing 767',TO_DATE('2009-06-27 17:20:00', 'YYYY-MM-DD HH24:MI
    :SS'),70,100,40,2);
252 INSERT INTO airplanes (id,title,created,fuel,fuel_max,state,airport_id) VALUES(
    airplanes_seq.nextval,'Boeing 737',TO_DATE('2009-06-27 17:20:00', 'YYYY-MM-DD HH24:MI
    :SS'),90,100,10,1);
253 INSERT INTO airplanes (id,title,created,fuel,fuel_max,state,airport_id) VALUES(
    airplanes_seq.nextval,'Airbus A380',TO_DATE('2009-06-27 17:20:00', 'YYYY-MM-DD HH24:
    MI:SS'),100,100,100,1);
254 INSERT INTO airplanes (id,title,created,fuel,fuel_max,state,airport_id) VALUES(
    airplanes_seq.nextval,'Airbus A320',TO_DATE('2009-06-27 17:20:00', 'YYYY-MM-DD HH24:
    MI:SS'),100,100,100,4);

256 INSERT INTO stations (id,title,airport_id,created) VALUES (stations_seq.nextval,'Terminal
     1',1,TO_DATE('2009-06-27 17:20:00', 'YYYY-MM-DD HH24:MI:SS')); /* Koeln Bonn
    Airport */
257 INSERT INTO stations (id,title,airport_id,created) VALUES (stations_seq.nextval,'Terminal
     2',1,TO_DATE('2009-06-27 17:20:00', 'YYYY-MM-DD HH24:MI:SS')); /* Koeln Bonn
    Airport */
258 INSERT INTO stations (id,title,airport_id,created) VALUES (stations_seq.nextval,'Terminal
     3',1,TO_DATE('2009-06-27 17:20:00', 'YYYY-MM-DD HH24:MI:SS')); /* Koeln Bonn
```

```
    Airport */
259 INSERT INTO stations (id,title,airport_id,created) VALUES (stations_seq.nextval,'Terminal
     1',2,TO_DATE('2009-06-27 17:20:00', 'YYYY-MM-DD HH24:MI:SS')); /* Heathrow Airport
    */
260 INSERT INTO stations (id,title,airport_id,created) VALUES (stations_seq.nextval,'Terminal
     2',2,TO_DATE('2009-06-27 17:20:00', 'YYYY-MM-DD HH24:MI:SS')); /* Heathrow Airport
    */
261 INSERT INTO stations (id,title,airport_id,created) VALUES (stations_seq.nextval,'Terminal
     3',2,TO_DATE('2009-06-27 17:20:00', 'YYYY-MM-DD HH24:MI:SS')); /* Heathrow Airport
    */
262 INSERT INTO stations (id,title,airport_id,created) VALUES (stations_seq.nextval,'Terminal
     A',3,TO_DATE('2009-06-27 17:20:00', 'YYYY-MM-DD HH24:MI:SS')); /* Heathrow Airport
    */
263 INSERT INTO stations (id,title,airport_id,created) VALUES (stations_seq.nextval,'Terminal
     B',3,TO_DATE('2009-06-27 17:20:00', 'YYYY-MM-DD HH24:MI:SS')); /* Heathrow Airport
    */
264 INSERT INTO stations (id,title,airport_id,created) VALUES (stations_seq.nextval,'Terminal
     1',4,TO_DATE('2009-06-27 17:20:00', 'YYYY-MM-DD HH24:MI:SS')); /* JFK Airport */
265 INSERT INTO stations (id,title,airport_id,created) VALUES (stations_seq.nextval,'Terminal
     2',4,TO_DATE('2009-06-27 17:20:00', 'YYYY-MM-DD HH24:MI:SS')); /* JFK Airport */
266 INSERT INTO stations (id,title,airport_id,created) VALUES (stations_seq.nextval,'Terminal
     3',4,TO_DATE('2009-06-27 17:20:00', 'YYYY-MM-DD HH24:MI:SS')); /* JFK Airport */
267 INSERT INTO stations (id,title,airport_id,created) VALUES (stations_seq.nextval,'Terminal
     4',4,TO_DATE('2009-06-27 17:20:00', 'YYYY-MM-DD HH24:MI:SS')); /* JFK Airport */
268 INSERT INTO stations (id,title,airport_id,created) VALUES (stations_seq.nextval,'Terminal
     1',5,TO_DATE('2009-06-27 17:20:00', 'YYYY-MM-DD HH24:MI:SS')); /* FF Airport */
269 INSERT INTO stations (id,title,airport_id,created) VALUES (stations_seq.nextval,'Terminal
     2',5,TO_DATE('2009-06-27 17:20:00', 'YYYY-MM-DD HH24:MI:SS')); /* FF Airport */
270 INSERT INTO stations (id,title,airport_id,created) VALUES (stations_seq.nextval,'Terminal
     1',6,TO_DATE('2009-06-27 17:20:00', 'YYYY-MM-DD HH24:MI:SS')); /* Mailand Airport
    */

272 INSERT INTO routes (id,current_station,next_station,airplane_id) VALUES (routes_seq.
    nextval,4,3,1); /* FZ 1 */
273 INSERT INTO routes (id,current_station,next_station,airplane_id) VALUES (routes_seq.
    nextval,9,14,2); /* FZ 2 */
274 INSERT INTO routes (id,current_station,next_station,airplane_id) VALUES (routes_seq.
    nextval,1,10,3); /* FZ 3 */
275 INSERT INTO routes (id,current_station,next_station,airplane_id) VALUES (routes_seq.
    nextval,11,2,4); /* FZ 4 */
276 INSERT INTO routes (id,current_station,next_station,airplane_id) VALUES (routes_seq.
    nextval,2,15,5); /* FZ 5 */
277 INSERT INTO routes (id,current_station,next_station,airplane_id) VALUES (routes_seq.
    nextval,9,7,6); /* FZ 6 */
```

```
278 INSERT INTO routes (id,current_station,next_station,airplane_id) VALUES (routes_seq.
    nextval,2,15,7); /* FZ 7 */
279 INSERT INTO routes (id,current_station,next_station,airplane_id) VALUES (routes_seq.
    nextval,1,8,8); /* FZ 8 */
280 INSERT INTO routes (id,current_station,next_station,airplane_id) VALUES (routes_seq.
    nextval,8,1,9); /* FZ 9 */


283 INSERT INTO route_stations (id,route_id,station_id,current_position) VALUES (
    route_stations_seq.nextval,1,9,1); /* Route FZ 1 */
284 INSERT INTO route_stations (id,route_id,station_id,current_position) VALUES (
    route_stations_seq.nextval,1,4,2); /* Route FZ 1 */
285 INSERT INTO route_stations (id,route_id,station_id,current_position) VALUES (
    route_stations_seq.nextval,1,3,3); /* Route FZ 1 */
286 INSERT INTO route_stations (id,route_id,station_id,current_position) VALUES (
    route_stations_seq.nextval,2,9,1); /* Route FZ 2 */
287 INSERT INTO route_stations (id,route_id,station_id,current_position) VALUES (
    route_stations_seq.nextval,2,14,2); /* Route FZ 2 */
288 INSERT INTO route_stations (id,route_id,station_id,current_position) VALUES (
    route_stations_seq.nextval,2,1,3); /* Route FZ 2 */
289 INSERT INTO route_stations (id,route_id,station_id,current_position) VALUES (
    route_stations_seq.nextval,2,4,4); /* Route FZ 2 */
290 INSERT INTO route_stations (id,route_id,station_id,current_position) VALUES (
    route_stations_seq.nextval,3,10,1); /* Route FZ 3 */
291 INSERT INTO route_stations (id,route_id,station_id,current_position) VALUES (
    route_stations_seq.nextval,3,5,2); /* Route FZ 3 */
292 INSERT INTO route_stations (id,route_id,station_id,current_position) VALUES (
    route_stations_seq.nextval,3,1,3); /* Route FZ 3 */
293 INSERT INTO route_stations (id,route_id,station_id,current_position) VALUES (
    route_stations_seq.nextval,4,2,1); /* Route FZ 4 */
294 INSERT INTO route_stations (id,route_id,station_id,current_position) VALUES (
    route_stations_seq.nextval,4,11,2); /* Route FZ 4 */
295 INSERT INTO route_stations (id,route_id,station_id,current_position) VALUES (
    route_stations_seq.nextval,5,2,1); /* Route FZ 5 */
296 INSERT INTO route_stations (id,route_id,station_id,current_position) VALUES (
    route_stations_seq.nextval,5,15,2); /* Route FZ 5 */
297 INSERT INTO route_stations (id,route_id,station_id,current_position) VALUES (
    route_stations_seq.nextval,6,7,1); /* Route FZ 6 */
298 INSERT INTO route_stations (id,route_id,station_id,current_position) VALUES (
    route_stations_seq.nextval,6,9,2); /* Route FZ 6 */
299 INSERT INTO route_stations (id,route_id,station_id,current_position) VALUES (
    route_stations_seq.nextval,7,2,1); /* Route FZ 7 */
300 INSERT INTO route_stations (id,route_id,station_id,current_position) VALUES (
    route_stations_seq.nextval,7,15,2); /* Route FZ 7 */
```

```
301 INSERT INTO route_stations (id,route_id,station_id,current_position) VALUES (
        route_stations_seq.nextval,8,15,1); /* Route FZ 8 */
302 INSERT INTO route_stations (id,route_id,station_id,current_position) VALUES (
        route_stations_seq.nextval,8,1,2); /* Route FZ 8 */
303 INSERT INTO route_stations (id,route_id,station_id,current_position) VALUES (
        route_stations_seq.nextval,9,15,1); /* Route FZ 9 */
304 INSERT INTO route_stations (id,route_id,station_id,current_position) VALUES (
        route_stations_seq.nextval,9,1,2); /* Route FZ 9 */

306 INSERT INTO airplanes (id,title,created,fuel,fuel_max,state,airport_id) VALUES(
        airplanes_seq.nextval,'UFO1',TO_DATE('2009-06-27 17:20:00', 'YYYY-MM-DD HH24:MI:SS')
        ,100,100,100,4);
307 INSERT INTO airplanes (id,title,created,fuel,fuel_max,state,airport_id) VALUES(
        airplanes_seq.nextval,'UFO',TO_DATE('2009-06-27 17:20:00', 'YYYY-MM-DD HH24:MI:SS')
        ,100,100,100,4);
308 COMMIT;

310 CREATE OR REPLACE TRIGGER TRG_BLACKLIST
311 AFTER INSERT ON log_results
312 FOR EACH ROW
313 BEGIN
314   IF :NEW.log_type = 'Geheimprojekt'
315   THEN
316     INSERT INTO log_results (id, log_type, log_action, created) VALUES (log_results_seq.
          nextval, 'ACHTUNG', 'Stationen für "' || :new.log_action || '" überprüfen.',
          SYSDATE);
317   END IF;
318 END TRG_BLACKLIST;
319 /

321 CREATE OR REPLACE TRIGGER trg_erstelltes_flugzeug
322 AFTER INSERT ON AIRPLANES
323 FOR EACH ROW
324 BEGIN
325   IF (:new.title != 'Privatmaschine') THEN
326     INSERT INTO log_results (id, log_type, log_action, created) VALUES (log_results_seq.
          nextval, 'Neues Flugzeug', 'Es wurde ein neues Flugzeug "' || :new.title || '"
          erstellt.', SYSDATE);
327     INSERT INTO crews (id, name, created) VALUES (crews_seq.nextval, 'John Doe Crew',
          SYSDATE);
328   END IF;
329 END trg_erstelltes_flugzeug;
330 /
```

C. Anhang – Testfallspezifikationen und Auswertungen

Listing 27: Auszug - PMD Fehlerbericht

```
1  src/de/unibonn/inf/dbdependenciesui/Configuration.java:24 All methods are static.
       Consider using Singleton instead. Alternatively, you could add a private constructor
       or make the class abstract to silence this warning.
2  src/de/unibonn/inf/dbdependenciesui/Configuration.java:28 The field name indicates a
       constant but its modifiers do not
3  src/de/unibonn/inf/dbdependenciesui/Configuration.java:28 Variables should start with a
       lowercase character
4  src/de/unibonn/inf/dbdependenciesui/Configuration.java:91 Avoid empty catch blocks
5  src/de/unibonn/inf/dbdependenciesui/Configuration.java:136 Found 'DD'-anomaly for
       variable 'level' (lines '136'-'138').
6  src/de/unibonn/inf/dbdependenciesui/Configuration.java:150 Avoid variables with short
       names like df
7  src/de/unibonn/inf/dbdependenciesui/Configuration.java:150 When instantiating a
       SimpleDateFormat object, specify a Locale
8  src/de/unibonn/inf/dbdependenciesui/Configuration.java:182 Found 'DD'-anomaly for
       variable 'value' (lines '182'-'188').
9  src/de/unibonn/inf/dbdependenciesui/Configuration.java:203 Avoid using final local
       variables, turn them into fields
10 src/de/unibonn/inf/dbdependenciesui/Configuration.java:203 Avoid variables with short
       names like ds
11 src/de/unibonn/inf/dbdependenciesui/Configuration.java:205 Found 'DD'-anomaly for
       variable 'path' (lines '205'-'209').
12 src/de/unibonn/inf/dbdependenciesui/Main.java:35 The Logger variable declaration does not
        contain the static and final modifiers
13 src/de/unibonn/inf/dbdependenciesui/Main.java:42 The field name indicates a constant but
       its modifiers do not
14 src/de/unibonn/inf/dbdependenciesui/Main.java:42 Variables should start with a lowercase
       character
15 src/de/unibonn/inf/dbdependenciesui/Main.java:42 Variables that are not final should not
       contain underscores (except for underscores in standard prefix/suffix).
16 src/de/unibonn/inf/dbdependenciesui/Main.java:66 Avoid excessively long variable names
       like irrelevantException
17 src/de/unibonn/inf/dbdependenciesui/Main.java:85 Avoid excessively long variable names
       like irrelevantException
18 src/de/unibonn/inf/dbdependenciesui/Main.java:107 To be compliant to J2EE, a webapp
       should not use any thread.
19 src/de/unibonn/inf/dbdependenciesui/Main.java:107 To be compliant to J2EE, a webapp
       should not use any thread.
20 src/de/unibonn/inf/dbdependenciesui/Main.java:145 Avoid empty catch blocks
21 src/de/unibonn/inf/dbdependenciesui/Main.java:158 System.exit() should not be used in
```

```
    J2EE/JEE apps
22 src/de/unibonn/inf/dbdependenciesui/TestFixtures.java:33 Unnecessary final modifier in
    final class
23 src/de/unibonn/inf/dbdependenciesui/apple/OSXAdapter.java:103 System.out.print is used
```

Listing 28: Auszug - Testfälle JUnit

```
1  // Testet, ob die Objekte initialisiert sind und nicht null zurückgeben
2  Assert.assertNotNull(analyser);
3  ..
4  // Testet die Anzahl der Trigger des Testschemas
5  Assert.assertEquals("Number of triggers", amountTriggers, databaseTriggers.size());
6  ..
7  // Testet, ob eine View nicht in der Liste der Tabellen vorhanden ist
8  Assert.assertFalse(databaseTables.contains(view));
9  ..
10 // Testet, ob der Trigger mit dem spezifierten Namen im Testschema vorhanden ist
11 trigger.setTitle("TEST_TRIGGER");
12 Assert.assertTrue("List contains " + trigger, databaseTriggers.contains(trigger));
13 ..
14 // Testet vor dem Einfügen einer Verbindung, ob die Anzahl 0 ist und nach dem
       Einfügen sollte die Anzahl 1 sein
15 Assert.assertEquals("There should be no connection", 0, Controller.getConnections().
       size());
16 Controller.updateConnection(TestFixtures.createDatabaseModels01());
17 Assert.assertEquals("There should be only one connection", 1, Controller.getConnections
       ().size());
18 ..
19 // Testet die Rückgabe der Funktion getName der Klasse Column
20 @Test
21 public final void testGetName() {
22   final Column column = new Column("Abcd");
23   Assert.assertEquals("Name has to be equal", "Abcd", column.getName());
24 }
25 ..
26 // Test case for constructor with nulls expected NullPointerException
27 @Test(expected = NullPointerException.class)
28 public final void testRelationConstructor1() {
29   new Relation(new DatabaseTable(), null, true);
30 }
```

Listing 29: Testfälle Parser - Mengenoperationen

```
1 SELECT prim FROM tablename1
2 UNION
3 SELECT prim FROM tablename2
```

```
4 INTERSECT
5 SELECT prim FROM tablename3
6 MINUS
7 SELECT prim FROM tablename4

9 Erwartetes Ergebnis: tablename1, tablename2, tablename3 und tablename4
```

Listing 30: Testfälle Parser - Unterabfragen

```
1 SELECT t1.prim FROM tablename1 t1, (SELECT t2.* FROM tablename2 t2, tablename3) t2
2 WHERE t1.prim != ALL (SELECT prim FROM tablename4)
3 OR NOT EXISTS(SELECT prim FROM tablename5
4 WHERE NOT EXISTS(SELECT prim FROM tablename6))

6 Erwartetes Ergebnis: tablename1, tablename2, tablename3, tablename4, tablename5 und
      tablename6
```

Listing 31: Testfälle Parser - HAVING-Klausel

```
1 SELECT COUNT(*) FROM tablename1
2 GROUP BY prim
3 HAVING prim < ANY (SELECT COUNT(*) FROM tablename2);

5 Erwartetes Ergebnis: tablename1 und tablename2
```

Listing 32: Testfälle Parser - Joins

```
1 SELECT prim FROM
2 tablename1 t1 LEFT JOIN tablename2 t2 ON (t1.prim = t2.prim)
3 RIGHT JOIN tablename3 t3 ON (t1.prim = t3.prim)
4 FULL OUTER JOIN tablename5 t5 ON (t1.prim = t5.prim)
5 CROSS JOIN tablename4 t4
6 NATURAL JOIN tablename5 t6
7 JOIN tablename5 t7 USING (prim)

9 Erwartetes Ergebnis: tablename1, tablename2, tablename3, tablename4 und tablename5
```

Listing 33: Testfälle Parser - PL/SQL-Trigger

```
1 BEGIN
2   :NEW.prim := :NEW.prim * 2;
3   SELECT COUNT(*) INTO my_counter FROM tablename1;
4   INSERT INTO tablename2 VALUES (:NEW.prim, my_counter, 'TEST','TEST');
5 END;

7 Erwartetes Ergebnis: tablename1 und tablename2
```

D. Anhang – Backus-Naur-Formen

Diese „Backus-Naur-Formen“ sind verfügbar auf den Webseiten der „Centre Universitaire d’Informatique (CUI)“[399] und entstanden durch Zuhilfenahme der Inhalte. Dabei werden die Backus-Naur-Formen für eine Oracle-Datenbank in der Version 7 spezifiert. Eventuelle Erweiterungen und Änderungen bezüglich der Syntax werden geändert.

Listing 34: BN-Formen einer View und eines Triggers in Oracle

```
// Startsymbol - View
create_view
  ::= "create" [ "or" "replace" ] [ "force" | "noforce" ] "view"
  [ schema_name "." ] view_name
  [ "(" alias { "," alias } ")" ]
  "as" query
  [ "with" "check" "option" [ "constraint" constraint_name ] ]

// Startsymbol - Trigger
create_trigger
  ::= "create" [ "or" "replace" ] "trigger"
  [ schema_name "." ] trigger_name
  trigger_sync
  trigger_ref
  trigger_when
  plsql_block



view_name
  ::= identifier
constraint_name
  ::= identifier
schema_name
  ::= identifier
alias
  ::= identifier
query
  ::= select_command
identifier
  ::= ( "letter" { "letter" | "digit" | "underline" | "dollar" | "sharp" } )
  | "quote" { "any character" } "quote"
select_command
  ::= "select" [ "all" | "distinct" ]
  ( "*" | ( displayed_column { "," displayed_column } ) )
```

[399] vgl. [Estier 98]

```
  "from" ( selected_table { "," selected_table } )
  [ "where" condition ]
  { connect_clause }
  { group_clause }
  { set_clause }
  { order_clause }
  { update_clause }
displayed_column
  ::= ( [ schema_name "." ] table_name "." "*" ) | ( expression [ alias ] )
expression
  ::= [ "+" | "-" ] term { ( "+" | "-" ) term }
term
  ::= factor { ( "*" | "/" ) factor }
factor
  ::= constante_nonsigne | variable
  | ( function ( [ "(" expression { "," expression } ")" ] ) )
  | ( group_function "(" [ "*" | "all" | "distinct" ] expression ")"
constante_nonsigne
  ::= identifier
variable
  ::= [ table_name "." ] column_name [ "(+)" ]
column_name
  ::= identifier
function
  ::= number_function | char_function | group_function | conversion_function |
      other_function
group_function
  ::= "avg" | "count" | "max" | "min" | "stddev" | "sum" | "variance"
number_function
  ::= "abs" | "ceil" | "floor" | "mod" | "power" | "round" | "sign" | "sqrt" | "trunc"
char_function
  ::= "chr" | "initcap" | "lower" | "lpad" | "ltrim" | "replace"
  | "rpad" | "rtrim" | "soundex" | "substr" | "translate" | "upper"
  | "ascii" | "instr" | "length"
conversion_function
  ::= "chartorowid" | "convert" | "hextoraw" | "rawtohex" | "rowidtochar"
  | "to_char" | "to_date" | "to_number"
other_function
  ::= "decode" | "dump" | "greatest" | "least" | "nvl"
  | "uid" | "user" | "userenv" | "vsize"
selected_table
  ::= [ schema_name "." ] table_name [ "@" link_name ] [ alias ]
table_name
  ::= identifier
```

```
78 link_name
79   ::= identifier
80 condition
81   ::= [ "not" ] logical_term { "or" logical_term }
82 logical_term
83   ::= logical_factor { "and" logical_factor }
84 logical_factor
85   ::= ( expression comparaison_op expression )
86   | ( [ "not" ] "in" exp_set )
87   | ( [ "not" ] "like" match_string )
88   | ( [ "not" ] "between" expression "and" expression )
89   | ( "is" [ "not" ] "null" )
90   | quantified_factor
91   | ( "(" condition ")" )
92 comparaison_op
93   ::= "=" | "<" | ">" | "<>" | "!=" | "^=" | "<=" | ">="
94 exp_set
95   ::= ( ( number | quoted_string ) { "," ( number | quoted_string ) } ) | subquery
96 number
97   ::= n [ "." [ n ] ] [ "E" [ "+" | "-" ] n ]
98 n
99   ::= "digit" { "digit" }
100 quoted_string
101   ::= "'" { "any_character" } "'"
102 subquery
103   ::= "(" select_command ")"
104 match_string
105   ::= "'" { "any_character" | "_" | "\%" } "'"
106 quantified_factor
107   ::= ( expression comparaison_op [ "all" | "any" ] expression )
108   | ( [ "not" ] "exists" subquery )
109 connect_clause
110   ::= "connect" "by"
111   ( ( "prior" expression comparaison_op expression )
112   | ( expression comparaison_op expression "prior" ) )
113   [ "start" "with" condition ]
114 group_clause
115   ::= "group" "by" expression { "," expression } [ "having" condition ]
116 set_clause
117   ::= ( ( "union" "all" ) | "intersect" | "minus" ) select_command
118 order_clause
119   ::= "order" "by" sorted_def { "," sorted_def }
120 sorted_def
121   ::= ( expression | n ) ( "asc" | "desc" )
```

```
122 update_clause
123   ::= "for" "update" "of" column_name { "," column_name } [ "nowait" ]
124 trigger_name
125   ::= identifier
126 trigger_sync
127   ::= ( "before" | "after" )
128   ( "delete" | "insert" | "update" [ "of" column_list ] )
129   { "or" ( "delete" | "insert" | "update" [ "of" column_list ] ) }
130   "on" [ schema_name "." ] table_name
131 column_list
132   ::= column_name { "," column_name }
133 trigger_ref
134   ::= "referencing"
135   [ "old" [ "as" ] identifier ]
136   [ "new" [ "as" ] identifier ]
137 trigger_when
138   ::= "for" "each" "row" [ "when" "(" condition ")" ]

140 plsql_block
141   ::= [ "<<" label_name ">>" ]
142   [ "declare" declare_spec { declare_spec } ]
143   "begin"
144   seq_of_statements
145   [ "exception" exception_handler { exception_handler } ]
146   "end" [ label_name ] ";"

148 exception_handler
149   ::= "when" exception_name "then" exception_dec ";"
150 exception_name
151   ::= identifier
152 exception_dec
153   ::= raise_statement
154 label_name
155   ::= identifier
156 declare_spec
157   ::= variable_declaration | subtype_declaration | cursor_declaration |
        exception_declaration
158   | record_declaration | plsql_table_declaration | procedure_declaration |
        function_declaration
159 variable_declaration
160   ::= variable_name [ "constant" ] type_name [ "not" "null" ]
161   [ ( ":=" | "default" ) plsql_expression ] ";"
162 variable_name
163   ::= identifier
```

```
type_name
  ::= identifier
plsql_expression
  ::= num_expression | char_expression | date_expression | boolean_expression
num_expression
  ::= [ "+" | "-" ] num_term { ( "+" | "-" ) num_term }
num_term
  ::= num_factor { ( "*" | "/" ) num_factor }
num_factor
  ::= ( number | variable_name | host_variable
  | ( function [ "(" plsql_exp_list ")" ] )
  | ( "(" num_expression ")" )
plsql_exp_list
  ::= plsql_expression { "," plsql_expression }
cursor_name
  ::= identifier | "null"
  | ( ( cursor_name | subquery ) "%rowcount" )
  ) [ "**" num_expression ]
host_variable
  ::= identifier
char_expression
  ::= char_term { "||" char_term }
char_term
  ::= ( char_literal | variable_name | host_variable
  | ( function [ "(" plsql_exp_list ")" ] )
  | ( "(" char_expression ")" )
  | "null" )
char_literal
  ::= quoted_string
date_expression
  ::= ( date_literal | variable_name | host_variable
  | ( function [ "(" plsql_exp_list ")" ] )
  | ( "(" date_expression ")" )
  | "null" )
date_literal
  ::= quoted_string
boolean_expression
  ::= [ "not" ] boolean_term { "or" boolean_term }
boolean_term
  ::= boolean_factor { "and" boolean_factor }
boolean_factor
  ::= boolean_literal | variable_name
  | ( function [ "(" plsql_exp_list ")" ] )
  | ( "(" boolean_exp ")" )
```

```
208   | ( plsql_expression
209   ( ( comparaison_op plsql_expression )
210   | ( "is" [ "not" ] "null" )
211   | ( [ "not" ] "like" match_string )
212   | ( [ "not" ] "between" plsql_expression
213   "and" plsql_expression )
214   | ( [ "not" ] "in" "(" plsql_exp_list ")" )
215   | ( ( cursor_name | subquery )
216   ( "%notfound" | "%found" | "%isopen" ) ) ) )
217 boolean_literal
218   ::= "true" | "false" | "null"
219 subtype_declaration
220   ::= "subtype" type_name "is" type_spec ";"
221 type_spec
222   ::= datatype | ( variable_name "%type" )
223   | ( table_name "." column_name "%type" )
224   | ( table_name "%rowtype" ) | type_name
225 datatype
226   ::= "binary_integer" | "natural" | "positive" | ( "number" [ "(" n [ "," n ] ")" ] )
227   | ( "char" [ "(" n ")" ] ) | ( "long" ) | ( "raw" ) | ( "long" "raw" )
228   | ( "boolean" ) | ( "date" )
229 cursor_declaration
230       ::=
231       "cursor" cursor_name
232       [ "(" parameter_spec { "," parameter_spec } ")" ]
233       "is" select_statement ";"
234 cursor_name
235   ::= identifier
236 parameter_spec
237   ::= parameter_name datatype
238 parameter_name
239   ::= identifier
240 select_statement
241   ::= "select" sql_item { "," sql_item }
242   "into" ( ( variable_name { "," variable_name } )
243   | record_name )
244   "from" ( table_reference { "," table_reference } )
245   sql_rest_of_select
246 sql_item
247   ::= "as in sql select_command"
248 record_name
249   ::= identifier
250 table_reference
251   ::= [ schema_name "." ] ( table_name | view_name ) [ "@" link_name ]
```

```
252 sql_rest_of_select
253   ::= "as in sql select_command"
254 exception_declaration
255   ::= exception_name "exception" ";"
256 exception_name
257   ::= identifier
258 record_declaration
259   ::= record_type_dec | record_var_dec
260 record_type_dec
261   ::= "type" type_name "is" "record" "(" field_spec { "," field_spec } ")" ";"
262 field_spec
263   ::= identifier
264 plsql_table_declaration
265   ::= table_type_dec | table_var_dec
266 table_type_dec
267   ::= "type" type_name "is" "table" "of" field_spec "indexed" "by" "binary_integer" ";"
268 table_var_dec
269   ::= plsql_table_name type_name ";"
270 plsql_table_name
271   ::= identifier
272 procedure_declaration
273   ::= procedure_body
274 procedure_body
275   ::= "procedure" procedure_name
276   [ "(" argument { "," argument } ")" ]
277   "return" return_type
278   "is"
279   [ "declare" declare_spec ";" { declare_spec ";" } ]
280   "begin"
281   seq_of_statements
282   [ "exception" exception_handler { exception_handler } ]
283   "end" [ procedure_name ] ";"
284 argument
285   ::= argument_name [ "in" | "out" | ( "in" "out" ) ] argument_type
286   [ ":=" "default" value
287 argument_name
288   ::= identifier
289 argument_type
290   ::= type_spec
291 value
292   ::= ( [ "+" | "_" ] number ) | quoted_string
293 return_type
294   ::= type_spec
295 function_declaration
```

```
296   ::= function_body
297 function_body
298   ::= "function" function_name
299   "(" argument { "," argument } ")"
300   "return" return_type ";"
301 function_name
302   ::= [ schema_name "." ] identifier
303 seq_of_statements
304   ::= statement ";" { statement ";" }
305 statement
306   ::= comment | assignment_statement | exit_statement | if_statement | loop_statement
307   | null_statement | raise_statement | return_statement | sql_statement | plsql_block
308 comment
309   ::= ( "--" "text_until_end_of_line" ) | ( "/*" "multi_line_text" "*/"
310 assignment_statement
311   ::= object ":=" plsql_expression ";"
312 object
313   ::= ( variable_name ) | ( record_name "." field_name ) | ( plsql_table_name "("
        subscript ")" ) | ( ":" host_variable )
314 field_name
315   ::= identifier
316 subscript
317   ::= plsql_expression
318 exit_statement
319   ::= "exit" [ label_name ] [ "when" plsql_condition ]
320 plsql_condition
321   ::= boolean_exp
322 if_statement
323   ::= "if" [ "(" ] boolean_exp [ ")" ] "then" seq_of_statements
324   [ ("else" seq_of_statements) | ("elsif" [ "(" ] boolean_exp [ ")" "then"
        seq_of_statements )]
325   "end" "if" ";"
326 loop_statement
327   ::= [ label_name ] [ ( "while" plsql_condition )
328   | ( "for" ( numeric_loop_param | cursor_loop_param ) ) ]
329   "loop" seq_of_statements "end" "loop" [ label_name ]
330 numeric_loop_param
331   ::= index_name "in" [ "reverse" ] num_expression ".." num_expression
332 index_name
333   ::= identifier
334 cursor_loop_param
335   ::= record_name "in" ( ( cursor_name
336   [ "(" plsql_exp_list ")" ] )
337   | ( "(" select_statement ")" ) )
```

```
338 null_statement
339   ::= "null"
340 raise_statement
341   ::= "raise" [ exception_name ]
342 return_statement
343   ::= "return" [ plsql_expression ]
344 sql_statement
345   ::= close_statement | commit_statement | delete_statement | fetch_statement
346   | insert_statement | lock_table_statement | open_statement | rollback_statement
347   | savepoint_statement | select_statement | set_transaction_statement | update_statement
348 close_statement
349   ::= "close" cursor_name
350 commit_statement
351   ::= "commit" [ "work" ] [ "comment" quoted_string ]
352 fetch_statement
353   ::= "fetch" cursor_name "into" ( ( variable_name { "," variable_name } ) | record_name
        )
354 lock_table_statement
355   ::= "lock" "table" table_reference "in" lock_mode "mode" [ "nowait" ]
356 lock_mode
357   ::= identifier
358 open_statement
359   ::= "open" cursor_name [ "(" plsql_exp_list ")" ]
360 rollback_statement
361   ::= "rollback" [ "work" ] [ "to" [ "savepoint" ] savepoint_name ] [ "comment"
        quoted_string ]
362 savepoint_statement
363   ::= "savepoint" savepoint_name
364 savepoint_name
365   ::= identifier
366 set_transaction_statement
367   ::= "set" "transaction" "read" "only"
368 delete_statement
369   ::= "delete" "from" table_name [ "cascade" ( "constraint" | "constraints") ]
370 insert_statement
371   ::= "insert" "into" table_name [ "(" column_name { "," column_name } ")" ]
372   ( "values" "(" column_def { "," column_def } ")"
373   | "as" select_command )
374 column_def:
375   ::= number | quoted_string | variable | boolean_literal | date_literal
376 update_statement
377   ::= "update" table_name "set" column_name "=" [ "where" condition ]
```

Erklärung

Wir versichern, die von uns vorgelegte Arbeit selbständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, haben wir als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die wir für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

Köln, 10. Juni 2010

Andre Kasper und Jan Philipp