Fachhochschule Köln
Cologne University of Applied Sciences

Weiterentwicklung der Software „Visual Dependencies“ zur Visualisierung von Datenbankabhängigkeiten

Analyse von Graphenalgorithmen und Erweiterung um Prozeduren und Funktionen und eine MySQL-Anbindung

Bachelorarbeit

zur Erlangung
des akademischen Grades
BACHELOR OF SCIENCE (B. SC.)
im Studiengang Allgemeine Informatik

vorgelegt an der Fachhochschule Köln
Campus Gummersbach
Fakultät für Informatik und Ingenieurwissenschaften

ausgearbeitet von:

Marc Kastleiner

Erster Prüfer: Prof. Dr. Heide Faeskorn-Woyke (Fachhochschule Köln)
Zweiter Prüfer: Prof. Dr. Birgit Bertelsmeier (Fachhochschule Köln)

Gummersbach, im September 2010

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

BNF	Backus-Naur-Form
ERD	Entity-Relationshop-Diagram
GUI	Graphical User Interface
ISO	Internationale Organisation für Normung
JDBC	Java Database Connectivity
JUNG	Java Universal Network/Graph Framework
MVC	Model-View-Controller
PL/SQL	Procedural Language/Structured Query Language
PNG	Portable Network Graphics
SQL	Structured Query Language

1. Einleitung

Durch die rasant fortschreitende Weiterentwicklung der Technik, insbesondere der computerbasierten Systeme, wird die Speicherung von relevanten Daten immer wichtiger. Fast kein Softwaresystem kommt heutzutage ohne Datenspeicherung aus. Hierbei kommen vorwiegend relationale Datenbanksysteme zum Einsatz, aber auch objektorientierte Lösungen finden Verwendung bei der Datenspeicherung für Softwarelösungen[1].

Durch die wachsenden Anforderungen an Datenbanksysteme, wie beispielsweise die zunehmende Menge an Daten und die dadurch immer komplexer werdenden Beziehungen zwischen selbigen, erhöht sich auch der Aufwand, den Datenbankadministratoren (oder andere mit den Datenbanken arbeitende Personen) betreiben müssen, um die Daten zu verwalten und selbst den Überblick über das Gesamtsystem zu behalten. Aufgrund der gesteigerten Anforderungen wächst auch die Nachfrage nach geeigneten Softwarelösungen, die die Datenbankadministratoren bei ihrer Arbeit unterstützen, indem sie die verschiedenartigen und komplexen Beziehungen zwischen den Datenbankobjekten visualisieren[2]. Dadurch soll es dem Verwaltungspersonal erleichtert werden, mögliche Probleme im Datenbanksystem aufzudecken und anschließend zu beseitigen.

Im Rahmen der Diplomarbeit von Andre Kasper und Jan Philipp „Visualisierung der Abhängigkeiten von Datenbankobjekten“, die im Jahr 2009 in einer Zusammenarbeit der Fachhochschule Köln Campus Gummersbach und der Universität Bonn entstanden ist, wurden die verfügbaren Softwarelösungen auf ihre Funktionalitäten hin analysiert. Interessanterweise war bis dato nahezu kein Programm in der Lage, die komplexen Beziehungen zwischen den Datenbankobjekten innerhalb eines Datenbanksystems darzustellen bzw. grafisch zu veranschaulichen.[3] Lediglich Fremdschlüsselbeziehungen, also die Beziehungen zwischen den Tabellenobjekten, werden oftmals in den Softwarelösungen mit Hilfe so genannten Entity-Relationship-Diagrammen (kurz ERD) grafisch visualisiert. Weiterführende Beziehungen zwischen den Datenbankobjekten, wie beispielsweise das Zusammenspiel zwischen Tabellen und virtuellen Sichten (Views)[2] oder auch die Abhängigkeiten zwischen Trigger-Objekten[2], werden meist vernachlässigt. Dabei sind gerade solche Beziehungen interessant, um Probleme innerhalb der Datenbank aufdecken und beseitigen zu können.

[1] Siehe Kapitel 3.1 Datenbanktypen
[2] Siehe Kapitel 3.3 Beziehungsarten in Datenbanken
[3] Vgl. [KP 09]

Aufgrund dieser Feststellung wurde im Zuge der eben genannten Diplomarbeit von Andre Kasper und Jan Philipp eine Softwarelösung entwickelt, die sich dieser fehlenden Visualisierung der Datenbankabhängigkeiten annimmt. Das entwickelte Programm wurde in der Programmiersprache Java geschrieben und verfügt bislang über drei verschiedene Ansichten, in denen der Benutzer sich die Beziehungen zwischen den Tabellen (ERD) untereinander, zwischen den Tabellen und Views und zwischen den Trigger-Objekten untereinander darstellen lassen kann. Desweiteren verfügt das Programm über die Möglichkeit, sich die Definitionen der einzelnen Datenbankobjekte und die Daten an sich anzeigen zu lassen. Bisher beschränkt sich die Software jedoch lediglich auf einen einzigen Datenbanktypen: Die Datenbank „Oracle“ (jedoch in verschiedenen Versionen 9i, 10g und 11g). Eine Anbindung an weitere Datenbanksysteme, vor allem das weit verbreitete Open Source Datenbanksystem „MySQL“, wäre demnach wünschenswert. Weiterführend ist zu sagen, dass die Software noch nicht alle möglichen Datenbankobjekte in die Ansichten aufnimmt. Beispielsweise werden die in Oracle durchaus wichtigen Prozeduren und Funktionen nicht in der Software dargestellt.

Aus diesen fehlenden bzw. noch zu ergänzenden Funktionen, entwickelte sich die grundlegende Anforderung, die Software weiterzuentwickeln. Dies soll im Rahmen dieser Bachelorarbeit geschehen. Neben der Umsetzung der Anbindung an weitere Datenbanksysteme und der Ergänzung der Ansichten, soll weiterführend noch eine kurze Analyse durchgeführt werden, ob und inwieweit die grafische Darstellung der Abhängigkeiten der Datenbankobjekte verbessert werden kann. Hierbei soll das Augenmerk auf die Verbesserung der graphischen Algorithmen und die potenzielle Verwendung von dreidimensionalen Räumen zur Optimierung der Ansichten gelegt werden. Eine Implementierung selbiger ist vorerst nicht angedacht.

2. Aufgabenbeschreibung

2.1 Ausgangszustand der Software

Bevor die genannte Weiterentwicklung stattfinden kann, ist jedoch erst einmal der genaue Ausgangszustand der Software zu ermitteln.

Die ursprüngliche Idee, die hinter der Entwicklung der zugrunde liegenden Software steckt, ist die Tatsache, dass zuvor nahezu keine Softwarelösung existierte, die die verschiedenartigen Beziehungen innerhalb einer Datenbank darstellen kann. Diese Visualisierung dient dem Zweck, Probleme in den Datenbankstrukturen und Abhängigkeiten zwischen den Datenbankobjekten leichter erkenntlich zu machen. Somit besteht die Möglichkeit, die unterstützten Datenbanken auf etwaige Fehlerquellen zu analysieren, um diese später beseitigen zu können.

Die zu Grunde liegende Software wurde in der Programmiersprache Java entwickelt und unterstützt bisher die Auflistung der Datenstrukturen einer Datenbank, wie beispielsweise die Spalten der Tabellen oder die Definitionen von Triggern und die Anzeige der in den Tabellen und Views enthaltenen Daten. Nachdem der Benutzer eine neue Verbindung zur Datenbank hergestellt hat und die Daten geladen und analysiert worden sind, steht dem Benutzer eine Baumdarstellung zur Verfügung, die die einzelnen Datenbankobjekte der gewählten Verbindung auflistet. Der Verbindungsname bildet hierbei die Wurzel des Baumes. Bei der Auswahl eines Objektes öffnet sich ein neuer Reiter/Tab, der die Definitionen bzw. Daten des ausgewählten Objektes anzeigt. Der Benutzer kann mehrere Verbindungen anlegen, die Datenstrukturen werden hierbei beim Laden der Verbindung lokal gespeichert. Dies geschieht mit Hilfe des Open-Source-Persistenz-Frameworks Hibernate[4], welches eine objektrelationale lokale Abbildung der Datenstrukturen ermöglicht. Durch diese Speicherung wird es dem Benutzer ermöglicht, die Datenbank zu untersuchen, ohne dass eine Verbindung zu selbiger bestehen muss. Eine Verbindung zur Datenbank wird lediglich für das erste Laden der Datenstrukturen und die Anzeige der in den Tabellen und Views enthaltenen Daten benötigt. Die Speicherung der Daten an sich erfolgt nicht, da die Datenmenge deutlich größer und weniger abschätzbar wäre und die Daten für die Analyse der Datenbank, für die die Software ursprünglich vorgesehen war, nicht relevant sind.

[4] http://www.hibernate.org/
http://de.wikipedia.org/wiki/Hibernate_(Framework)

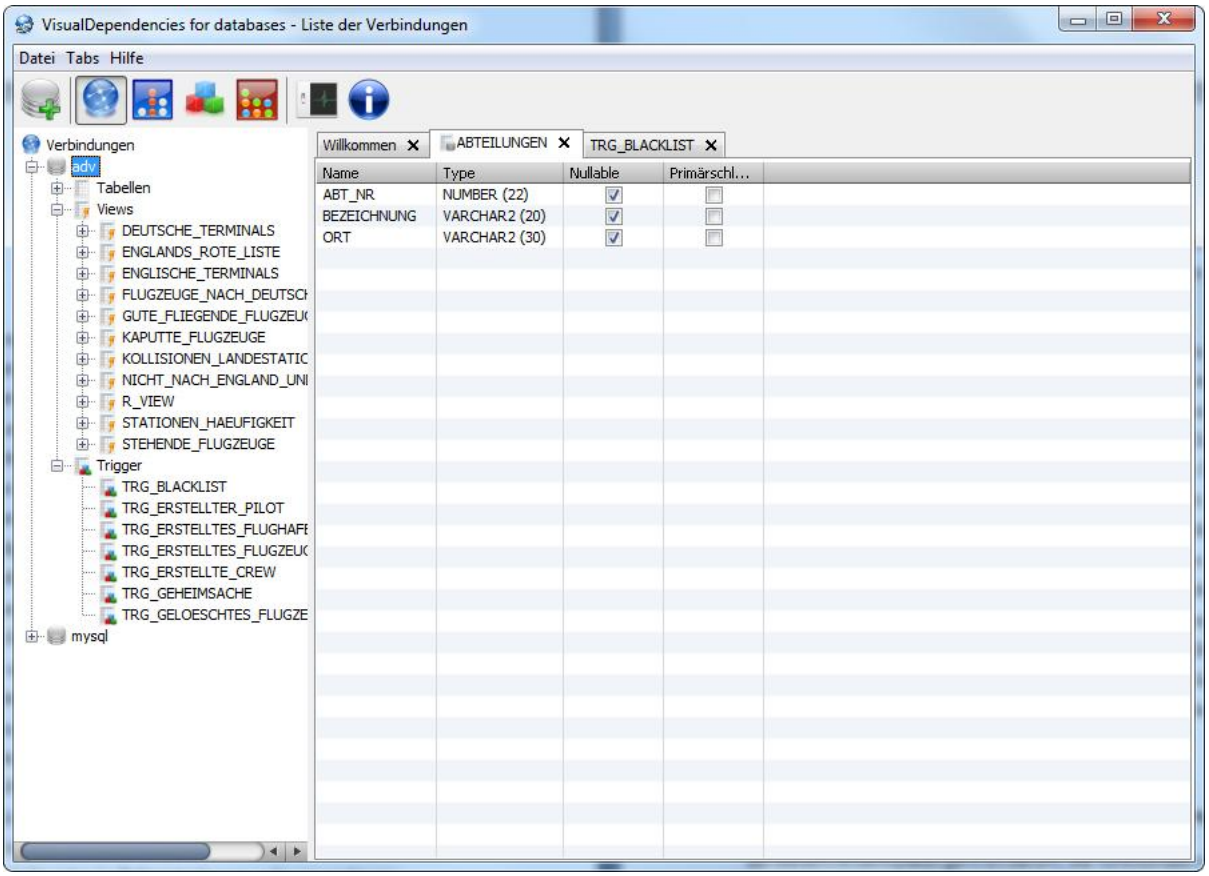

Abbildung 1: Ansicht der Datenstrukturen und Detailansicht der Spalten einer Tabelle

Zusätzlich zur Ansicht der Datenstrukturen und Daten können in drei verschiedenen Ansichten die komplexen Abhängigkeiten zwischen den Datenbankobjekten mit Hilfe von Graphen visualisiert werden. Hierbei werden die Beziehungen zwischen Tabellen (z.B. Fremdschlüsselbeziehungen), Triggern und Views berücksichtigt. Um die Darstellung realisieren zu können, wird bei der Darstellung der Graphen das Open Source Framework „Java Universal Network/Graph Framework" (JUNG)[5] eingesetzt. In den verschiedenen Ansichten werden dem Benutzer die Datenbankobjekte als Rechteecke dargestellt, die untereinander durch Pfeile verbunden sind. Die Objekte sind farblich voneinander abgegrenzt, die Pfeile stellen die Beziehungen zwischen den Objekten dar. Die Software liefert ebenfalls eine farbliche Rückmeldung darüber, ob Probleme bei den bestehenden Beziehungen auftreten können. Beispielsweise wird der Benutzer auf Mutating-Table-Probleme bei Triggern aufmerksam gemacht, die entstehen können, wenn ein Trigger-Objekt die aufrufende Tabelle nutzt bzw. etwas an ihr ändert. Außerdem werden Zyklen zwischen Trigger-Objekten erkannt und dargestellt. Diese Zyklen oder auch Rekursionen sind ohne grafische Veranschaulichung meist nicht oder nur schwer auffindbar.

[5] http://jung.sourceforge.net/

Dem Benutzer steht in den Ansichten außerdem die Möglichkeit zur Verfügung, die Graphen zu manipulieren. Neben der Auswahl verschiedener Layouts kann der Nutzer mittels Drag & Drop- und Zoom-Funktionalitäten den Graphen verändern. Außerdem kann der Nutzer in einer Liste die anzuzeigenden Objekte auswählen und somit die Ansicht auf den für ihn relevanten Bereich ausrichten. Um die Ansichten auch beispielsweise in Präsentationen oder in anderen Programmen (z.B. Graphviz [6]) nutzen zu können, wurde die Funktionalität eingefügt, die dargestellten Graphen als Grafik im „Protable Network Graphics“-Format (PNG) oder als DOT-Datei, eine formale Beschreibung von Graphen, zu exportieren.

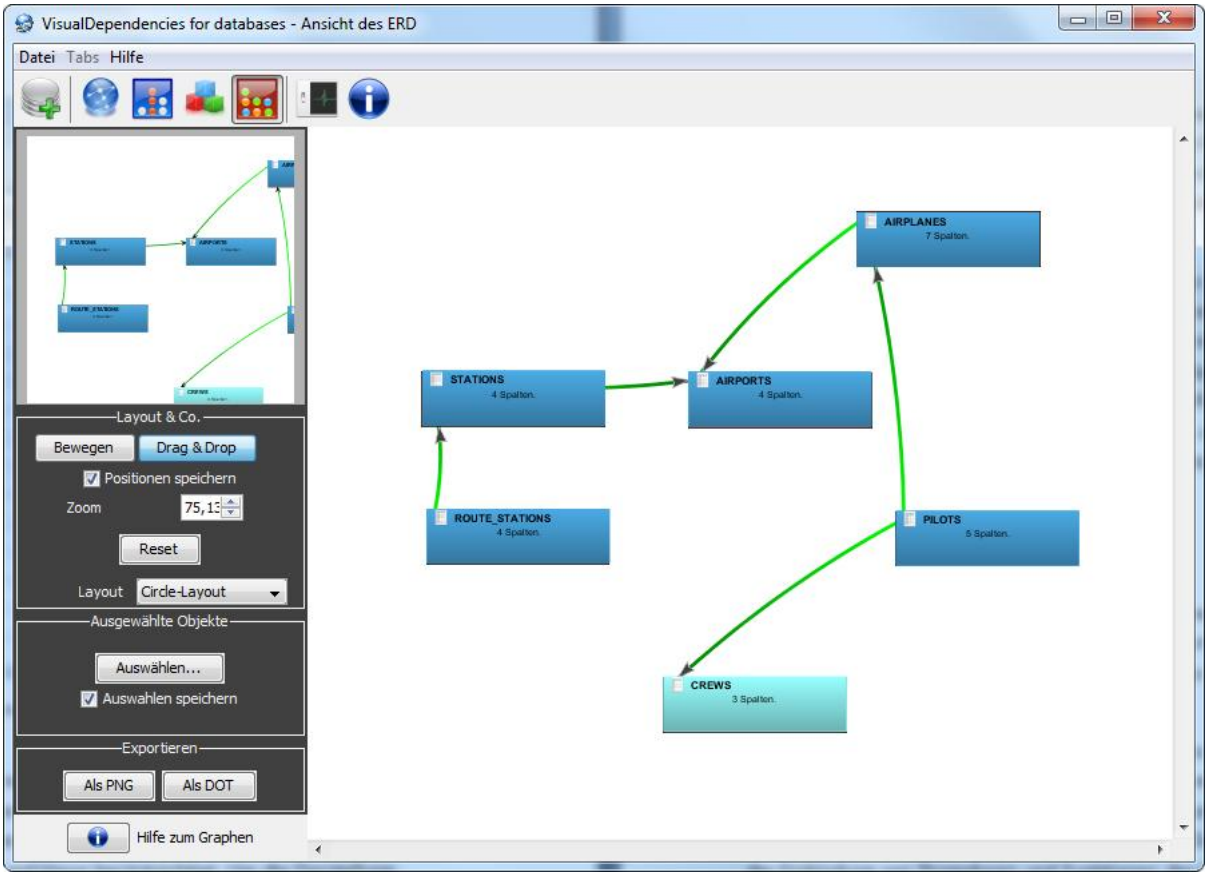

Abbildung 2: Beispielhafte Graphenansicht anhand eines ER-Diagrammes

Alle diese Funktionen sind jedoch bisher lediglich für die Datenbank „Oracle“ in den Versionen 9i, 10g und 11g implementiert. Eine Anbindung an die weit verbreitete Open Source Datenbank „MySQL“ wird derzeit nicht unterstützt. Desweiteren finden Prozeduren und Funktionen, zwei weitere wichtige Datenbankobjekte, in der Software keine Berücksichtigung. Diese können jedoch genauso Einfluss auf die anderen Datenbankobjekte

[6] http://www.graphviz.org/

nehmen, weshalb sie in der Software ergänzt werden sollten, um dem Nutzer einen Überblick zu bieten, welche Objekte in den Prozeduren und Funktionen genutzt bzw. verändert werden.

2.2 Funktionen und Arbeitspakete

An dieser Stelle soll die Weiterentwicklung im Rahmen dieser Arbeit ansetzen. Hierzu ist zunächst zu klären, welche Funktionen implementiert werden sollen, was zusätzlich noch analysiert werden muss und in welche Arbeitspakete die Aufgaben unterteilt werden können.

Aufgrund der anstehenden Themengebiete ist es sinnvoll, das Projekt in drei Teilpakete zu unterteilen. Im ersten Schritt ist die Anbindung an das Datenbanksystem „MySQL“ vorgesehen, wobei die schon zur Verfügung stehenden Schnittstellen der Software genutzt werden sollen. Das zweite Teilpaket beschäftigt sich mit der Einbindung von Prozeduren und Funktionen des Datenbanksystems Oracle und der Implementierung einer neuen Ansicht. Diese Ansicht soll die Tabelle mehr ins Zentrum der Betrachtung rücken, indem es dem Nutzer ermöglicht wird, eine Tabelle mit ihren zugehörigen Datenbankobjekten und Beziehungen auszuwählen. Nach Auswahl einer Tabelle werden dann alle von dieser Tabelle abhängigen Datenbankobjekte angezeigt, wozu dann auch die Prozeduren und Funktionen zählen. Vorerst ist für diese neue Ansicht nur eine Implementierung für Oracle vorgesehen, daher soll die neue Ansicht nur bei Oracle-Verbindungen verfügbar sein.

Im letzten Schritt ist angedacht, das System weiterführend auf mögliche Erweiterungen und Verbesserungen zu analysieren. In diesem Vorgang liegt das Augenmerk vorwiegend auf der Verbesserung der grafischen Algorithmen. Eine Implementierung ist bislang nicht vorgesehen.

Arbeitspaket 1 – Implementierung der MySQL Anbindung

Bei diesem Arbeitspaket geht es darum, die funktionalen Anforderungen an die Anbindung des MySQL-Datenbanksystems zu analysieren und umzusetzen. Bei der Umsetzung ist darauf zu achten, die vorhandenen Schnittstellen weitestgehend zu nutzen. Folgende Funktionen müssen dabei in der gegebenen Reihenfolge umgesetzt werden.

- Grundsätzliche Analyse der Machbarkeit der Funktionen
- Einbindung der Datenbankverbindung
- Auslesen der Datenstrukturen (Tabellendefinitionen etc.)

- Auslesen der Daten
- Analyse (Parse-Vorgang)[7] der Trigger-Inhalte zur Darstellung der Abhängigkeitsgraphen
- Analyse (Parse-Vorgang) der View-Definitionen zur Darstellung der Abhängigkeitsgraphen
- Einbindung der Funktionen in die bestehende Oberfläche

Arbeitspaket 2 – Einbindung der Funktionalitäten für Prozeduren und Funktionen

Dieses Arbeitspaket soll die Implementierung der Prozeduren und Funktionen und die Umsetzung einer neuen Ansicht beinhalten. Dabei sind folgende Schritte angedacht:

- Analyse des Aufbaus von Oracle-spezifischen Prozeduren und Funktionen und Untersuchung des Oracle Data Dictionary[8] im Hinblick auf die genannten Datenbankobjekte
- Auslesen der Datenstrukturen
- Analyse (Parse-Vorgang) der Datenstrukturen zur späteren Darstellung
- Aufbau eines neuen Abschnitts in der Verbindungsansicht, der die neuen Objekte (Prozeduren und Funktionen) darstellt
- Entwurf eines neuen Reiters/Tabs für die Detailansicht der neuen Objekte
- Aufbau der neuen Ansicht zur Veranschaulichung, welche Datenbankobjekte von der ausgewählten Tabelle abhängen. Hierzu zählt auch die Gestaltung einer neuen Auswahlmaske im linken Menü, das eine Tabellenauswahl ermöglicht
- Implementierung des eigentlichen Graphen. Hierbei muss auch eine geeignete Darstellung für die neuen Datenbankobjekte (Prozeduren und Funktionen) gefunden werden
- Entwurf und Einbau neuer Icons für die neu entstandenen Funktionen

[7] Beim Parse-Vorgang werden die gegebenen Informationen analysiert, gefiltert und in ein weiterverarbeitbares Format gebracht.
[8] Metadaten-Verzeichnis, welches die Definitionen und Regeln der Datenbankobjekte enthält.

Arbeitspaket 3 – Graphenanalysen

Das letzte Arbeitspaket beschäftigt sich vorwiegend mit der Analyse der Graphen. Es soll festgestellt werden inwieweit die Graphenansichten verbessert werden können.

- Analyse der vorhandenen Graphenalgorithmen, ob und wie Überschneidungen zwischen den Objekten verhindert bzw. verringert werden können.
- Überprüfung der Möglichkeit den dreidimensionalen Raum zur Darstellung nutzen zu können.
- Sammeln von Verbesserungsideen für die Graphenansichten

Weitere Aufgaben

Bei der gründlichen Analyse der Software sind zusätzlich zu den Arbeitspaketen noch ein paar Fehler aufgefallen. Diese sollen ebenfalls im Zuge dieser Bachelorarbeit beseitigt werden.
Ein größerer Fehler wurde in der Analyse der Trigger festgestellt. Der vorhandene Parser für die Trigger-Objekte aus Oracle-Datenbanken sollte ursprünglich alle Tabellenobjekte, mit denen der Trigger arbeitet, aus der Trigger-Definition auslesen. Diese Funktionalität ist auch implementiert worden, jedoch findet der Trigger keinerlei Tabellenobjekte mehr, wenn die Definition des Triggers von den „einfachen Standardausdrücken“ abweicht. Ein kurzes Beispiel soll diese Problematik verdeutlichen (siehe Abbildung 3).

```
DECLARE
     CURSOR cur_airplanes IS
       SELECT * FROM airplanes;
BEGIN
  […]
END;
```

Abbildung 3: Beispielhafte Trigger-Definition mit einem einfachen SELECT-Statement

In dieser kurzen Trigger-Definition ist zu sehen, dass Datensätze von der Tabelle „airplanes“ in einen Cursor[9] geladen werden. Hierbei soll der Parser bei seiner Analyse die genannte Tabelle finden und abspeichern. Dies funktioniert auch reibungslos. Diesbezüglich soll nun jedoch ein weiteres Beispiel zeigen, dass dies nicht immer funktioniert.

[9] Vereinfacht gesagt sind Cursor mit Variablen in Programmiersprachen zu vergleichen.

```
DECLARE
     CURSOR cur_airplanes IS
       SELECT * FROM (airplanes), airports;
BEGIN
  […]
END;
```

Abbildung 4: Veränderte Trigger-Definition mit einem erweiterten SELECT-Statement

Diese leicht veränderte Definition ist weiterhin ein korrektes SQL-Statement, hierbei wird jedoch ein kartesisches Produkt[10] der beiden Tabellen „airplanes“ und „airports“ gebildet. Es ist auch darauf zu achten, dass die erste Tabelle in Klammern gefasst ist. Der bisher implementierte Parser findet bei dieser Definition leider keine der beiden Tabellen mehr, da er darauf ausgerichtet ist, das erste Wort nach dem Schlüsselwort „FROM“ als Tabelle anzusehen, mit dem vorher geladenen Tabellenverzeichnis zu vergleichen und anschließend abzubrechen, um mit dem nächsten Statement in der Definition fortzufahren. Da jedoch die öffnende Klammer natürlicherweise keine Tabelle darstellt (und demnach auch nicht im Tabellenverzeichnis enthalten ist), verwirft der Parser dieses Statement. Bei der Analyse des Parsers wurde somit festgestellt, dass er bei komplexeren Definitionen nicht mehr zum gewünschten Ergebnis führt, da beispielsweise JOIN-Operationen, Subselects oder einfache Klammersetzungen nicht berücksichtigt werden. Eine Neuimplementierung ist demnach unumgänglich.

Ein weiterer nicht so schwerwiegender Fehler, der bei der Analyse der Software aufgefallen ist, ist die Tatsache, dass das Programm einen Fehler wirft, wenn man die Funktionen „Alle Tabs schließen“ oder „Alle anderen Tabs schließen“ nutzt, um alle offenen bzw. alle anderen Reiter zu schließen. Es wurde festgestellt, dass man zwar neue Tabs öffnen kann, jedoch nicht mehr diejenigen, die zuvor schon geöffnet waren, bevor die Funktion genutzt wurde. Im Rahmen der Weiterentwicklungen soll auch dieses Problem gelöst werden.

Es ist also festzuhalten, dass zusätzlich zu den genannten Arbeitspaketen noch folgende Funktionen umgesetzt werden müssen:

- Neuentwicklung des Trigger-Parser-Algorithmus
- Fehlerbehebung der „Tabs schließen“-Funktionen

Zusätzlich ist noch zu erwähnen, dass alle Funktionen zweisprachig implementiert werden sollen, da die Ausgangssoftware ebenfalls in englischer und deutscher Sprache vorliegt.

[10] Es wird ein Ergebnis erzeugt, dass jede Zeile der ersten Tabelle mit jeder Zeile der zweiten Tabelle kombiniert.

3. Definitionen und Beschreibungen

Da in dieser Bachelorarbeit einige Begriffe verwendet werden, die nicht unbedingt geläufig sind, sollen in diesem Kapitel die Ausdrücke zunächst erklärt werden. Desweiteren sollen schon hier Rahmenbedingungen definiert werden, die für die Entwicklung der Software von Bedeutung sind. Dazu zählt zum Beispiel die Festlegung, welche Datenbanktypen relevant für das Projekt sind.

3.1 Datenbanktypen

Wenn mit Datenbanksystemen gearbeitet wird, ist die zweckgebundene Auswahl des Datenbanktypen von essentieller Bedeutung. Jeder Typ hat spezielle Vor- und Nachteile, die in Bezug auf den Einsatzfall analysiert werden müssen. Der folgende Abschnitt soll lediglich anschneiden, welche Datenbanksysteme derzeit die relevantesten sind, und kurz erklären, was die Systeme auszeichnet.

Zunächst ist an dieser Stelle der wohl am weitesten verbreitete Datenbanktyp, die **relationale Datenbank**, zu nennen. Die relationale Datenbank wird hauptsächlich durch Tabellen und die Beziehungen zwischen selbigen definiert. Jede Tabelle kann über mehrere Spalten verfügen, wobei jede Spalte für ein Attribut steht. Demnach stellt eine Zeile in der Tabelle einen Datensatz dar. Daten unterschiedlicher Tabellen können über Schlüsselbeziehungen miteinander in Verbindung stehen[11]. Durch den relativ simplen Aufbau der Datenbank ist gewährleistet, dass Datenabfragen auf die Tabellen der relationalen Datenbank sehr performant erfolgen können. Diese Abfragen werden mit Hilfe der relationalen Algebra ausgeführt, welche stark an den Begriff der mathematischen Algebra angelehnt ist. Beispielsweise existiert auch in der Datenbanksprache SQL, die unter anderem für die Abfragen genutzt wird, ebenso wie in der Mathematik, das kartesische Produkt (oder auch Kreuzprodukt). In der Mathematik bedeutet dies, dass eine neue Menge aus zwei Mengen gebildet wird, die alle möglichen geordneten Kombinationen der Inhalte der zwei Ausgangsmengen enthält (sprich: Jedes Objekt der einen Menge wird mit jedem Objekt der anderen Menge kombiniert). Analog dazu wird bei einer SQL-Abfrage jeder Datensatz der einen Tabelle mit jedem Datensatz der anderen Tabelle kombiniert.

Die relationalen Datenbanken eignen sich aufgrund ihrer Struktur am besten für einfach strukturierte Daten mit komplexen Abfragen.[12]

[11] Siehe Kapitel 3.3 Beziehungsarten in Datenbanken
[12] Vgl. [DBS 07], S.32f

Als weiterer Datenbanktyp ist das Konzept der **objektorientierten Datenbank** zu nennen. Mit der vermehrten Verwendung von objektorientierten Konzepten in der Programmierung wurden auch die Objektdatenbanken eingeführt. Durch die direkte Speicherung der Objekte mit Attributen und Methoden innerhalb der Datenbank, entfällt die Schwierigkeit, die Objekte aus der Programmierung in die Strukturen einer relationalen Datenbank zu übertragen. Dieser Vorteil der durch die Nähe zu den objektorientierten Programmiersprachen entsteht, weist leider bei der Abfrage komplexerer Daten auch deutliche Schwächen auf. Durch die Verwendung von objektorientierten Konzepten wie beispielsweise der Vererbung und der Assoziation, können bei Abfrage- oder Speichervorgängen komplexe Pfade zu den gesuchten Objekten entstehen. Durch diese Komplexität der Pfade wird eine hohe Laufzeit erreicht, die letztendlich bei vielen (vor allem komplexen) Abfragen zu starken Performanceproblemen führen kann.
Demnach sind diese objektorientierten Datenbanksysteme zwar für die Speicherung komplexer Daten geeignet, jedoch muss darauf geachtet werden, dass keine komplexen Abfragen benötigt werden.[13]

Aus diesem Grund wurden **objektrelationale Datenbanksysteme** entwickelt. Sie sollen die Lücke zwischen den objektorientierten Programmiersprachen und den relationalen Datenbanken schließen, ohne Performanceeinbußen aufzuweisen. Hierzu werden zusätzlichen zum relationalen Datenbanksystem einige Konzepte der Objektorientierung verwendet. So ist es beispielsweise möglich Methoden zu definieren oder eigene Datentypen festzulegen. Jedoch werden nicht alle Konzepte der Objektorientierung verwendet, die objektrelationalen Datenbanksysteme nähern sich lediglich der Objektorientierung an, wodurch es möglich wird, auch komplexere Daten zu speichern und die Abfragen trotzdem performant zu halten.[13]

Es existieren noch weitere Datenbanktypen, wie beispielsweise die auf einem hierarchischen Modell basierenden Datenbanken oder die Netzwerkdatenbanken, die an dieser Stelle jedoch nicht genau erklärt werden sollen. Diese Typen existieren schon sehr lange, wurden aber immer mehr vor allem von den relationalen Datenbanksystemen verdrängt.

Im Zusammenhang dieser Bachelorarbeit sind nur relationale Datenbanksysteme von Bedeutung. Diese Einschränkung ist vor allem darauf zurückzuführen, dass es sich bei diesem Projekt um eine Weiterentwicklung einer bestehenden Software handelt. Die weiterzuentwickelnde Software beschäftigt sich ausschließlich mit relationalen Datenbanksystemen, was wohl darauf zurückzuführen ist, dass Beziehungen zwischen den

[13] Vgl. [DBS 07], S.32f

Daten und Datenbankobjekten vor allem in relationalen Datenbanksystemen eine große Bedeutung haben.

3.2 Datenbankobjekte

In diesem Abschnitt soll geklärt werden, was die folgenden Begrifflichkeiten im Zusammenhang mit relationalen Datenbanken bedeuten. Hierzu sollen die wichtigsten Datenbankobjekte, die im Rahmen dieser Bachelorarbeit von Bedeutung sind, kurz vorgestellt werden.

Tabellen

Tabellen sind Datenbankobjekte, die die eigentlichen Daten enthalten. Sie bestehen aus mindestens einer Spalte, wobei jede Spalte ein Attribut eines Datensatzes enthält. Die Tabellen haben eindeutige Namen, jedes Attribut hat einen Datentypen. Die enthaltenen Datensätze sind über sogenannte Schlüsselattribute eindeutig identifizierbar, beispielsweise eine fortlaufende Nummer. Die Schlüsselattribute können auch über mehrere Spalten der Tabelle hinweg definiert werden.

Views

Views (oder auch Sichten genannt) können als Abbildungen von Tabellen angesehen werden. Genauso wie Tabellen haben auch Views Spalten mit Attributen und Zeilen mit Datensätzen. Views stellen jedoch nur eine logische Relation dar, die durch eine in der Datenbank gespeicherte Abfrage definiert wird. Ein Einsatzzweck hierfür wäre zum Beispiel die Tatsache, dass nicht jeder Benutzer alle Daten sehen soll. Nehmen wir an, es gibt in der Datenbank eine Tabelle, die Personen mit ihrer Adresse enthält. Ein Mitarbeiter soll jedoch nur die Namen, nicht jedoch die Adressen zu Gesicht bekommen. Hierzu kann eine View angelegt werden, die nur die relevanten Daten abfragt. Der Benutzer erhält dann nur Zugriff auf die View, nicht jedoch die Tabelle selbst.

Trigger

Trigger sind Datenbankobjekte, die jeweils an eine Tabelle gebunden sind. Sie enthalten einen Programmcode, der zu einem bestimmten Ereignis ausgeführt wird. Dieses Ereignis ist eine Änderung (Einfügen, Ändern und Löschen) an der Tabelle sein, an der der Trigger hängt. Außerdem ist in der Definition eines Triggers festgelegt, ob der Trigger vor oder nach dem Ereignis ausgelöst wird. Abhängig vom Datenbanksystem sind verschiedene weitere Parameter möglich.

Prozeduren und Funktionen

Die bisher genannten Objekte sind schon im derzeitigen Stand der Software integriert, jedoch gibt es noch zwei weitere Datenbankobjekte, die in den gängigen Datenbanksystemen relevant sind: die Prozeduren und Funktionen.
Prozeduren und Funktionen werden durch ihren Programmcode definiert, was sie den Triggern ähneln lässt. Sie werden jedoch nicht wie die Trigger über ein Ereignis auf einer Tabelle ausgelöst, sondern explizit von einem Benutzer oder einem Programm aufgerufen. Prozeduren können Ein- und Ausgabeparameter besitzen. Funktionen sind Prozeduren sehr ähnlich, jedoch müssen sie über einen Rückgabewert verfügen.

3.3 Beziehungsarten in Datenbanken[14]

Bevor mit der Entwicklung der Software begonnen werden kann, muss zunächst einmal geklärt werden, welche Arten von Beziehungen überhaupt innerhalb von Datenbanken existieren können und welchen Nutzen es hat, diese grafisch darzustellen. In den vorhergehenden Kapiteln wurde das Thema zwar angeschnitten, es ist jedoch zu komplex, weshalb an dieser Stelle noch einmal eine genauere Übersicht erstellt werden soll.

Fremdschlüsselbeziehungen

Die simpelste Beziehungsart stellen sicherlich die Fremdschlüsselbeziehungen zwischen den einzelnen Tabellen dar. Hierbei werden in den Tabellen einzelne Spalten als so genannte Fremdschlüssel ausgewiesen, die wiederrum auf andere Tabellen zeigen. Beispielsweise könnte man sich eine Tabelle „Personen“ vorstellen, in der verschiedene Personen verzeichnet sind. Diese Personen üben jeweils einen Beruf aus, alle Berufe sind in einer separaten Tabelle gespeichert. Nun kann über die Fremdschlüsselbeziehung eine Verbindung zwischen den beiden Tabellen hergestellt werden, die aussagt, welche Person welchen Beruf ausübt. Diese Beziehungen werden in der Software in einem Entity-Relationship-Diagramm (kurz ERD) dargestellt. Daraus wird ersichtlich, welche Tabellen miteinander in Verbindung stehen. Somit ist es für den Betrachter einfacher ersichtlich, welche Beziehungen zwischen den Daten bestehen und ob das gewollte Modell der Daten korrekt in der Datenbank abgebildet wurde. Neben dem Diagramm kann der Nutzer auch in der Verbindungsansicht nachlesen, welche Fremdschlüssel auf einer Tabelle definiert sind.

[14] Vgl. auch [KP 09]

Triggerbeziehungen

Als nächster relevanter Beziehungstyp lässt sich die Beziehung von Triggern untereinander nennen. Zunächst fragt man sich, was die Trigger überhaupt miteinander in Verbindung stehen lässt, da die Trigger normalerweise nur mit ihren aufrufenden Tabellen verbunden sind. Diese Beziehungsdarstellung würde jedoch dem Betrachter wenig Nutzen bringen, da jeder Trigger nur an einer Tabelle hängt und der Betrachter in Textform diese Beziehungen wahrscheinlich ebenfalls erkennen würde. Viel interessanter ist jedoch die Beziehung der einzelnen Trigger zu anderen Tabellen an denen ebenfalls Trigger hängen. Dies sieht folgendermaßen aus:

Durch eine Aktion (beispielsweise ein Löschvorgang oder das Bearbeiten eines Datensatzes) auf der Tabelle, an der der Trigger hängt, wird der Trigger aufgerufen. Nun führt der Trigger Operationen auf anderen Tabellen aus. Durch diese Aktionen könnten weitere Trigger angestoßen werden. Somit kann ein Beziehungsnetzwerk der Trigger untereinander entstehen. Die Darstellung dieser Beziehungen hat den Vorteil, dass auf einen Blick Probleme erkannt werden können. Solche Probleme können beispielsweise Zyklen sein, die zwischen den Triggern entstehen.

Hierzu soll anhand eines Beispiels ein solcher Zyklus erklärt werden: Trigger A ändert etwas an der Tabelle TB und löst somit Trigger B aus. Dieser löst wiederrum Trigger C aus, der wieder Trigger A auslöst. Somit ist ein Zyklus entstanden der womöglich zu einer Endlosschleife führt. Diesen kleinen Kreis hätte man vielleicht noch anhand der Triggerdefinitionen erkennen können, wird dieser Zyklus jedoch größer, so ist der Fehler nur noch schwer ersichtlich. Mit einer grafischen Darstellung wird dem Benutzer diese Sucharbeit abgenommen. In der Software wird ein solcher Zyklus derzeit rot markiert, wodurch der Betrachter sofort einen Fehler erkennen kann.
Aber nicht nur Trigger, die auf anderen Tabellen etwas ändern, sind von möglichen Problemen betroffen. Unter Oracle existiert das so genannte Mutating-Table-Problem, was auftritt wenn ein Trigger auf seiner aufrufenden Tabelle arbeitet und somit Inkonsistenzen entstehen können. Auch diese Art von Problemen in Triggerbeziehungen kann mit Hilfe der Software einfach aufgedeckt werden.

Viewbeziehungen

Als letzte Beziehungsart sind in der Software die Viewbeziehungen implementiert. Views werden immer durch Abfragen auf Tabellen oder andere Views definiert. Diese Beziehungen können ebenfalls im Programm dargestellt werden, wodurch ersichtlich wird, von welchen Stellen die Views ihre Daten beziehen. Desweiteren lassen sich die Beziehungen in positive und negative Beziehungen unterteilen. Eine positive Beziehung stellt beispielsweise der Zusammenschluss zweier Tabellen mit Hilfe einer „JOIN-Operation“ dar. Eine negative Beziehung hingegen stellt immer die Einschränkung von Daten durch eine oder mehrere andere Tabellen oder Views dar. Die Darstellung dieser Beziehungen erfolgt in unterschiedlichen Farben und ist vor allem für den Fall der Rekursionen (Klausel unter Oracle: „start with … connect by …“) wichtig, da Rekursionen eine Abbruchbedingung benötigen, welche meist durch eine negative Beziehung realisiert wird. Somit wird dem Betrachter in der Software aufgezeigt, ob diese Abbruchbedingung tatsächlich vorhanden ist.

All diese Beziehungsdarstellungen sollen dem Benutzer helfen, Probleme innerhalb einer Datenbank schneller und mit geringerem Aufwand feststellen zu können.

Beziehungen der Datenbankobjekte zu einer bestimmten Tabelle

Zusätzlich zu den bereits implementierten Beziehungstypen, könnte es für den Nutzer möglicherweise interessant sein, eine Ansicht zur Verfügung zu haben, in der er sehen kann, welche Datenbankobjekte mit einer bestimmten Tabelle in Verbindung stehen. Diese Ansicht hätte den Vorteil, dass der Nutzer auf einen Blick erkennen kann, welche Objekte von Änderungen an der gewählten Tabelle betroffen sein könnten. Im Rahmen dieser Bachelorarbeit soll die genannte neue Ansicht implementiert werden.

4. Arbeitspaket 1 - Implementierung der MySQL - Anbindung

4.1 Zielsetzung

Im Rahmen dieses Kapitels soll die MySQL-Funktionalität, die in Arbeitspaket 1 beschrieben wurde, analysiert und implementiert werden.

Bevor mit der eigentlichen Implementierung begonnen werden kann, muss zunächst festgestellt werden, inwieweit die geforderten Funktionalitäten überhaupt umgesetzt werden können. Hierzu sollen die Unterschiede und Probleme zwischen der bisher verwendeten Datenbank „Oracle“ und der neu hinzuzufügenden Datenbank „MySQL“ bestehen.

4.2 Unterschiede / Probleme zwischen MySQL und Oracle

Im Hinblick auf die Implementierung der genannten Funktionen, soll zunächst eine Abgrenzung der Unterschiede zwischen MySQL und bekannten kommerziellen Datenbanken, insbesondere der in Industrie häufig zum Einsatz kommenden Datenbank Oracle, vorgenommen werden. Durch diese Analyse soll festgestellt werden, wo sich diese Unterschiede eventuell auf die Umsetzung der Funktionen in der Software auswirken und wo dadurch Probleme entstehen könnten.

Der wohl gravierendste Unterschied der Datenbank MySQL zu kommerziellen Datenbanken wie Oracle, ist wohl die Tatsache, dass MySQL die Möglichkeit besitzt, verschiedene Storage Engines einzusetzen. Storage Engines definieren hierbei die Art der physikalischen Speicherung der Daten innerhalb der Datenbank. Aber nicht nur in der Speicherung unterscheiden sich diese Storage Engines, auch die Funktionalitäten innerhalb der verschiedenen Engines können variieren. Beispielsweise unterstützt nicht jede Engine die Möglichkeit Transaktionen durchzuführen oder referentielle Integrität durch Fremdschlüsselbeziehungen zu gewährleisten, welche Standardfunktionalitäten der Datenbank Oracle darstellen. Durch diese Vielzahl von verschiedenen Speicherungsmöglichkeiten und Funktionalitäten ist es möglich, die Datenbank auf den Zweck der Datenspeicherung genau anzupassen. Leider kann sich diese Variierbarkeit negativ auf die Umsetzung der Softwarefunktionen auswirken, da nicht zwingend alle Daten zur korrekten Ausführung der Funktionen ausgelesen werden können. Beispielsweise ist die Storage Engine „InnoDB“ in MySQL derzeit die einzige Engine, die Fremdschlüsselbeziehungen (sprich referentielle Integrität) unterstützt. Wenn jedoch eine andere Engine benutzt wird, können Fremdschlüsselbeziehungen zwischen den einzelnen

Tabellen nicht erkannt und somit auch nicht im Entity-Relationship-Diagramm dargestellt werden. Diese fehlende Unterstützung würde jedoch nicht zu einem Fehler innerhalb der Software führen. Es würden dann lediglich die Tabellen als Objekte im Graphen dargestellt, jedoch ohne ihre Beziehungen, was dem Nutzer keinerlei relevante Informationen liefert.

Ein weiterer großer Unterschied war bislang die Tatsache, dass MySQL schlichtweg keine Trigger, Views oder Stored Procedures unterstützte und ebenfalls kein Data Dictionary implementiert war. Eine Einbindung von MySQL wäre demnach fast sinnfrei, wenn nicht gar unmöglich, da die Hauptfunktionalitäten der Software, sprich die graphische Veranschaulichung der Tabellen-, Trigger- und View-Beziehungen nicht umsetzbar wäre. Mit der Version 5 der nicht kommerziellen Datenbank MySQL hat sich am Funktionsumfang jedoch einiges geändert. Die Version 5 von MySQL wurde Ende 2005 eingeführt und unterstützt alle im SQL3-Standard definierten Objekttypen. Dies bedeutet, dass nun auch Trigger, Views und Stored Procedures möglich sind. Des Weiteren wurde in MySQL 5 ein Data Dictionary eingeführt, wie es unter Oracle schon lange implementiert ist. Unter MySQL ist das Data Dictionary unter dem Namen „INFORMATION_SCHEMA“ bekannt. Mit diesen Neuerungen setzt MySQL in der Version 5 also die grundlegenden Voraussetzungen, um eine Implementierung der Schnittstelle zu MySQL in der Software sinnvoll umsetzen zu können. Das Data Dictionary von MySQL ist zwar bei weitem noch nicht so ausgereift wie jenes von Oracle, jedoch enthält es alle wichtigen Informationen, die in der Software benötigt werden.

Weitere relevante Unterschiede zwischen MySQL und Oracle lassen sich erkennen, wenn man die Datenbankobjekte genauer betrachtet. Nimmt man beispielsweise die Trigger von MySQL genauer unter die Lupe, so lässt sich erkennen, dass MySQL lediglich so genannte Row-Trigger, jedoch keine Statement-Trigger unterstützt. Row-Trigger werden für jeden betroffenen Datensatz des feuernden Statements ausgeführt. Statement-Trigger hingegen werden nur einmal für das gesamte Statement ausgeführt. Weiterführend ist es in MySQL nicht möglich die aufrufende Tabelle zu bearbeiten, dies verhindert demnach das aus Oracle bekannte Mutating-Table-Problem. Die Software ist derzeit so implementiert, dass sie Mutating-Table-Probleme graphisch veranschaulicht. Diese Funktion kann demnach in einer MySQL-Implementierung nicht genutzt werden, da dieses Problem nicht existiert.

4.3 Betrachtung des INFORMATION_SCHEMA

Um die Datenbankobjekte aus MySQL-Datenbanken analysieren zu können, müssen zunächst die Informationen über die Datenstrukturen zur Verfügung stehen. Unter Oracle sind diese Informationen im Data Dictionary gespeichert. Unter MySQL ab der Version 5 sind die relevanten Daten im so genannten INFORMATION_SCHEMA hinterlegt. Die Metadaten (also die Daten über die eigentlichen Daten), wie beispielsweise die Namen von Tabellen oder die Datentypen von Spalten, sind in verschiedenen schreibgeschützten Tabellen aufbereitet, die alle wichtigen Informationen zu den Datenstrukturen des zu untersuchenden Datenbankschemas enthalten [15]. Aus diesen Informationen kann die Darstellung der Tabellenstrukturen erfolgen und die enthaltenen Views und Trigger können angezeigt werden. Die Metadaten über die Trigger und Views enthalten ebenfalls ihre Definitionen. Diese werden benötigt, um die betroffenen Tabellen und Views in den Parsern auf ihre Abhängigkeiten hin zu filtern und später die Beziehungen grafisch darstellen zu können.

Bei einer Analyse des Information_Schema von MySQL hat sich ergeben, dass die im folgenden Abschnitt genannten Tabellen relevant für die Darstellung der gewünschten Informationen sind:

Die Tabellen INFORMATION_SCHEMA - COLUMNS und - KEY_COLUMN_USAGE

Die Tabelle „COLUMNS“ enthält alle Informationen, die den Aufbau der Tabellen und Views des MySQL-Schemas bezeichnen. Hierzu zählen neben den Namen der Spalten auch ihre Datentypen und die Größe der Datenfelder. Da die genannte Tabelle ebenfalls den Tabellen- bzw. Viewnamen enthält und jede Tabelle zwangsläufig in der Tabelle „COLUMNS“ verzeichnet sein muss, ist es nicht zwingend notwendig diese Informationen von der Tabelle „TABLES“ im Information_Schema abzufragen. Die Tabelle „TABLES“ enthält noch zusätzlich Informationen zu den für die Tabellen verwendeten Speicher-Engines, was jedoch vorerst noch nicht gespeichert werden soll.

Zusätzlich zu den Spalten der Tabellen müssen noch die Constraints[16] der Tabellen geladen werden. Diese Informationen können aus der Tabelle „KEY_COLUMN_USAGE“ entnommen werden. Mit Hilfe eines JOINS können die Informationen über die Tabellen und Views und ihre Constraints mit einer einzigen Abfrage eingelesen werden.

[15] Prinzipiell handelt es sich bei den Tabellen nicht um Basistabellen, sondern um Views. Wenn über das INFORMATION_SCHEMA gesprochen wird, ist jedoch meist von Tabellen die Rede.

[16] Constraints sind Bedingungen an eine oder mehrere Spalten. Hierzu zählen die Primär- und Fremdschlüssel, aber auch Prüfbedingungen für Werte.

Die Tabelle INFORMATION_SCHEMA - VIEWS

Da Views Tabellen zwar ähneln, jedoch zusätzlich noch eine Definition in Form einer Abfrage auf andere Tabellen und Views besitzt, muss diese noch abgefragt werden. Dies geschieht mittels einer Abfrage der Tabelle „VIEWS“ im Information_Schema. Die View-Definition ist die einzige Spalte, die auf dieser Tabelle abgefragt werden muss.

Die Tabelle INFORMATION_SCHEMA - TRIGGERS

In dieser Tabelle sind alle relevanten Informationen zu den Triggerobjekten enthalten. Neben den Namen der Trigger können hier auch die Ausführungszeitpunkte[17], die Informationen an welchen Tabellen die Trigger hängen und die Trigger-Definitionen abgerufen werden.

Derzeit steckt das Information_Schema von MySQL noch ein wenig in den Kinderschuhen. Die für die Software relevanten Daten können zwar abgerufen werden, aber im Vergleich zum Data Dictionary von Oracle ist die Informationsfülle noch sehr spärlich. Es ist jedoch scheinbar zu erwarten, dass diese Informationsquelle noch weiter ausgebaut wird, was daran zu erkennen ist, dass einige zusätzliche Spalten schon vorhanden, aber noch nicht mit Daten gefüllt sind.

4.4 Softwarearchitektur und relevante Entwurfsmuster

Aufgrund der Tatsache, dass die Weiterentwicklungen an vorhandenen Schnittstellen der Software anknüpfen sollen, ist zunächst zu klären, welche Schnittstellen überhaupt angeboten und welche Entwurfsmuster verwendet werden.

Model-View-Controller

Zuerst ist die grundsätzliche Architektur der Software zu nennen. Die Architektur verwendet in den Grundzügen das Model-View-Controller-Prinzip (MVC). Bei diesem Architekturmuster wird eine Trennung zwischen dem Datenmodell (Model) mit der damit verbundenen Logik und der Präsentation für den Benutzer (View) vorgenommen. Der Controller übernimmt hierbei die Steuerung und reagiert auf Benutzeraktionen. Ändert beispielsweise der Benutzer etwas, so wird der Controller von der View über diese Änderung informiert. Dieser sorgt für die notwendigen Änderungen im Model, welches seinerseits wiederrum die View über diese Änderung informiert, damit die Daten neu geladen werden können.

[17] Dies meint bei welchem Event die Trigger aufgerufen werden (INSERT, UPDATE, DELETE) und ob sie vor (BEFORE) oder nach (AFTER) dem triggernden Statement ausgeführt werden.

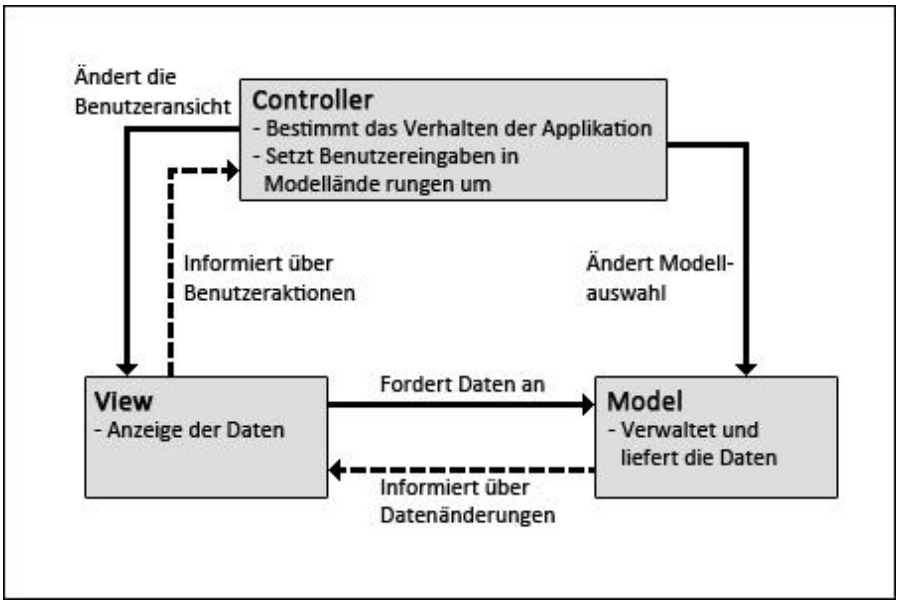

Abbildung 5: Vereinfachte Ansicht des MVC-Architekturmusters

Die gezeigte Architektur ist nur ein Beispiel für das MVC-Muster, die Implementierung hängt vom konkreten Anwendungsfall ab. So findet man beispielsweise auch oft den Ansatz, dass die View keine direkte Verbindung zum Model besitzt, der Controller die gesamte Steuerung übernimmt und die View lediglich für die Darstellung der vom Controller gelieferten Daten zuständig ist.

Durch die Kapselung der einzelnen Komponenten wird eine bessere Wart- und Erweiterbarkeit erzielt, außerdem wird es einfacher die Komponenten auszutauschen bzw. wiederzuverwenden. Es wäre durch den flexiblen Entwurf beispielsweise einfacher möglich, aus der lokalen Anwendung eine Webanwendung zu entwerfen. Hierzu muss nicht die gesamte Software neu entwickelt, sondern lediglich die View-Komponenten müssen ausgetauscht und ein paar Änderungen am Controller vorgenommen werden. Die Model-Implementierungen bleiben, eine korrekte Umsetzung des MVC-Musters vorausgesetzt, bestehen.

Aufgrund der Verwendung des MVC-Musters für den Architekturentwurf der Software, ist es nun nur noch nötig, für die Anbindung eines neuen Datenbanktyps, die Model-Komponente auszutauschen bzw. um einen Datenbanktypen zu erweitern. Da die Darstellung der Daten die gleiche bleibt, ist ein Austausch von View und Controller nicht notwendig, hier sind nur minimale Änderungen bzw. Erweiterungen vorzunehmen.

Fabrikmethode

Bei der Fabrikmethode (engl. factory method) handelt es sich um ein Entwurfsmuster zur Erzeugung von Objekten. Es wird hierfür eine Klasse definiert, die mindestens eine Methode enthält, die bei ihrem Aufruf ein Objekt eines bestimmten Typs produziert. Die Methode gibt immer ein Objekt als Objekt einer gemeinsamen Oberklasse oder einer Schnittstellendefinition zurück. Dies hat den Vorteil, dass die Klassen, die die Objekte nutzen, die konkreten Objekte nicht kennen müssen, sondern nur mit der Oberklasse bzw. der Schnittstelle arbeiten. Die eigentliche Funktionalität ist in den konkreten Unterklassen gekapselt.
Im konkreten Fall kann durch die Fabrikmethode eine Kapselung der Verbindungen und Analysen der verschiedenen Datenbanktypen erfolgen. Nach außen sind nur die Schnittstellen sichtbar. Die Methoden der Fabrik-Klasse werden mit einem Eintrag aus einer Aufzählungs-Klasse (enum) namens „Vendor“ als Parameter aufgerufen. Diese Aufzählung enthält alle implementierten Datenbanktypen. Anhand dieses Eintrags entscheidet die Fabrikmethode, welche konkrete Objektinstanz erzeugt werden muss.
Demnach muss für die Implementierung eines neuen Datenbanktyps (in diesem Fall MySQL) zunächst ein Eintrag in der Aufzählung erfolgen. Im Anschluss daran können die konreten Klassen für MySQL implementiert und die Methoden der Fabrik erweitert werden.

4.5 Implementierung

Wie im vorhergehenden Kapitel schon erwähnt, werden einige Schnittstellen angeboten, die eine Erweiterung um einen neuen Datenbanktypen erleichtern. Die Implementierung findet lediglich auf Ebene des Modells statt, die Benutzeransichten (Views) sind nicht betroffen, weshalb keine Analyse selbiger nötig ist.

Verbindungsaufbau

Zunächst muss die Oberfläche um die Möglichkeit erweitert werden, eine Verbindung zu einer MySQL-Datenbank aufzubauen. Hierzu wurde zunächst eine Klasse „MySQLMetaDataImpl“ als Erweiterung der abstrakten Klasse „AbstractMetaData“ erzeugt, die neben der Verwaltung der verbindungsspezifischen Metadaten auch für das Laden des JDBC-Treibers[18] zuständig ist. Innerhalb der Klasse ist eine Subklasse implementiert, die die

[18] Java Database Connectivity: eine Datenbankschnittelle für Java, die unter anderem für den korrekten Verbindungsaufbau zuständig ist.

abstrakte Verbindungsklasse „AbstractConnectionBuilder“ erweitert und für das Erzeugen einer neuen Verbindung zu einer MySQL-Datenbank zuständig ist.

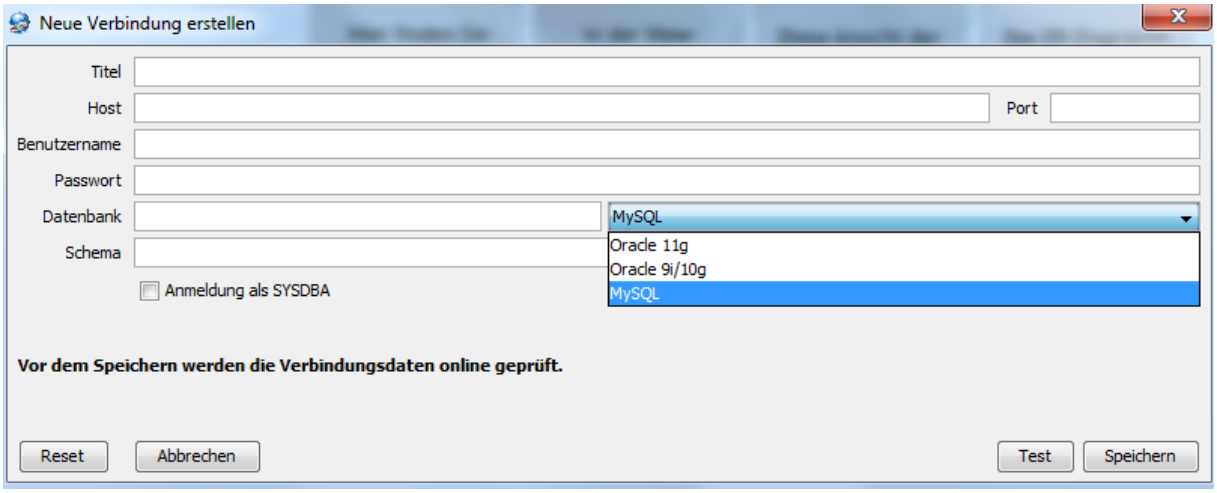

Abbildung 6: Eingabemaske für die Verbindungsdaten

Durch das Einfügen eines neuen Wertes in die Aufzählungsklasse, erscheint automatisch ein neuer Eintrag in der Datenbankauswahl beim Erzeugen einer neuen Verbindung.

Auslesen und Analysieren der Datenstrukturen

Das Auslesen der Datenstrukturen erfolgt in der Klasse „MySqlAnalyzerImpl“. Sobald die Verbindungsdaten eingegeben und gespeichert wurden, werden alle Datenstrukturen und Daten aus der Datenbank gelesen. Dies geschieht, indem sich das Programm zur angegeben Datenbank verbindet und mit Hilfe von vordefinierten Abfragen, die Informationen aus dem Data Dictionary von MySQL (INFORMATION_SCHEMA) liest. Dabei werden sowohl die Tabellendefinitionen als auch die Trigger und Views geladen. Eventuell vorhandene Daten innerhalb der Tabellen und Views werden erst abgefragt, wenn diese benötigt werden, sprich der Nutzer die Daten der jeweiligen Tabelle oder View einsehen möchte.

Sobald die Daten ausgelesen wurden, werden diese in der Analysephase weiterverarbeitet. Hierbei werden zunächst die Tabellenstrukturen analysiert und verarbeitet. Dazu werden zu jeder Tabelle die vorhandenen Spalten mit ihren Datentypen und weiteren Informationen aufgelistet. Desweiteren werden alle vorhandenen Constraints wie Indizes und Primärschlüssel ausgewertet.

An dieser Stelle ist vorab zu erwähnen, dass die Datenstrukturen, die später auch zur Darstellung der Graphen benutzt werden, in einer Hibernate-Datenbank lokal abgespeichert

werden. Dies gilt aufgrund von möglicherweise großen Datenmengen nicht für die Daten in den Tabellen und Views. Somit ist nach der ersten Verbindung zur Datenbank lediglich für die Einsicht der Daten eine Verbindung notwendig, die Datenstrukturen werden lokal abgelegt.

Im Anschluss an die Tabellenanalyse werden die Views untersucht. Zunächst werden auch hier die Strukturen analysiert und gespeichert. Nachfolgend müssen die Views auf mögliche Beziehungen zu anderen Tabellen oder Views untersucht werden. Zu diesem Zweck wurde ein Parser entwickelt, der die Definitionen der Views analysiert und die betroffenen Tabellen und Views herausfiltert.

Da innerhalb der Viewdefinitionen Kommentare erlaubt sind, werden diese und weitere nicht benötigte Abschnitte vorab aus den Definitionen entfernt. Anschließend beginnt die eigentliche Analyse, in der der Parser die Definitionen Stück für Stück auf Schlüsselwörter untersucht, um somit die Positionen der enthaltenen Tabellen- und View-Namen herauszufinden und diese speichern zu können.

Weiterführend ist zu erwähnen, dass der Parser, wie auch schon bei der Oracle-Anbindung implementiert, zusätzlich zu den Beziehungen, die Art der Beziehung herausfiltert. Damit ist gemeint, dass der Parser eine Unterscheidung zwischen positiven und negativen Beziehungen vornimmt.

Beim View-Parser wurde ein selbst geschriebener Parser verwendet. Aufgrund von Performance-Verbesserungen und vor allem möglicher Fehlerminimierung, wäre es denkbar, einen Parsergenerator wie beispielsweise JavaCup[19] einzusetzen. Dieser könnte anhand einer Grammatik den Parser automatisch generieren[20].

Im letzten Schritt der Analyse werden die Trigger analysiert. Zunächst werden hierbei auch die Datenstrukturen analysiert. Die geladenen Daten unterscheiden sich hierbei von denen der Tabellen und Views. In diesem Fall werden neben dem Namen des Triggers beispielsweise Informationen über die Aktion, auf die der Trigger reagiert, und der Ausführungszeitpunkt gespeichert.

Im Anschluss müssen auch hier die Beziehungen ermittelt werden. Bei diesem Vorgang werden ebenfalls die betroffenen Tabellen aus der Definition der Trigger gefiltert, wozu ein weiterer Parser benötigt wird. Anschließend muss jedoch noch betrachtet werden, ob die betroffenen Tabellen einen Trigger besitzen, der auf die Aktion des analysierten Triggers

[19] http://www2.cs.tum.edu/projects/cup/
[20] Siehe Kapitel 8.2 Ausblick – Backus-Naur-Form

reagiert. Erst dann liegt eine Beziehung zwischen diesen beiden Triggern vor, die schließlich gespeichert wird.

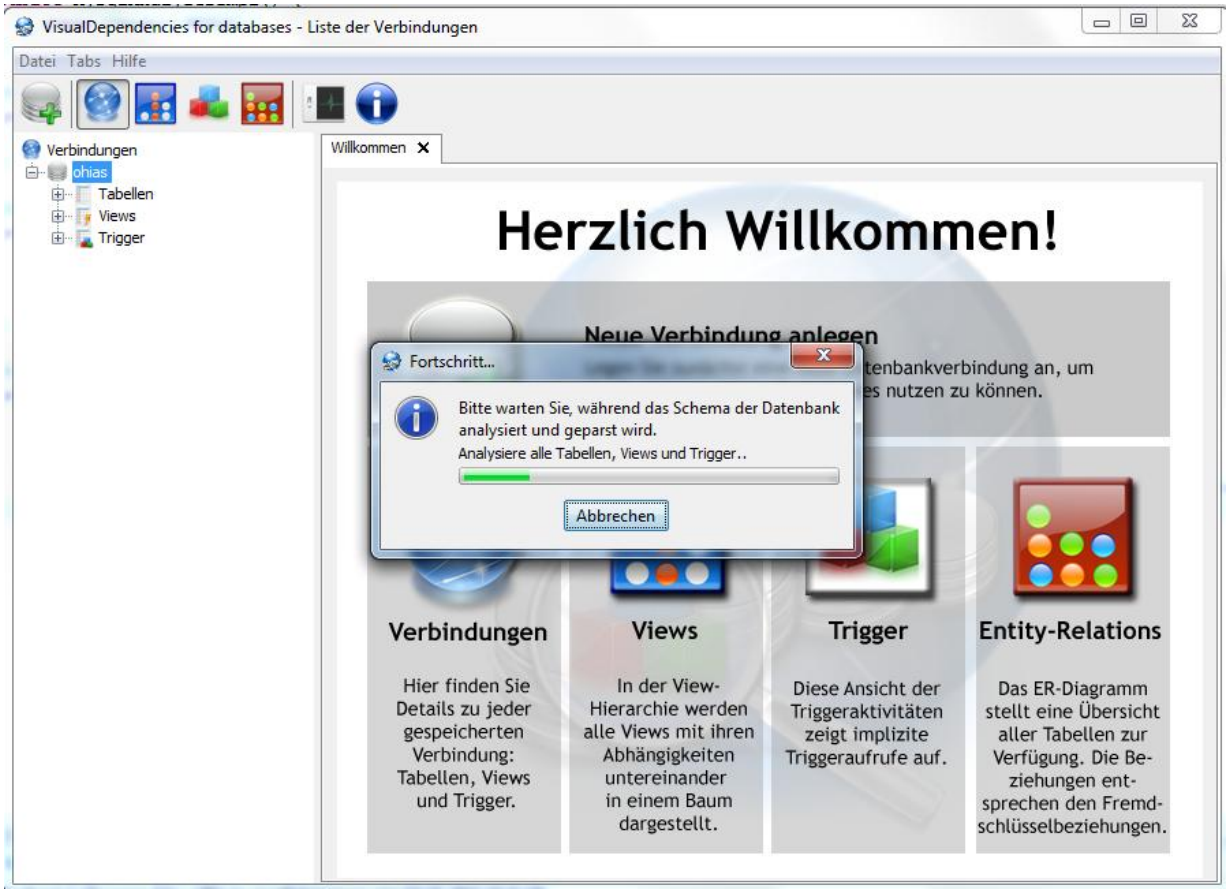

Abbildung 7: Laden der Datenstrukturen und Analyse der Tabellen, Views und Trigger

Graphische Darstellungen

Nachdem alle Datenstrukturen und Beziehungen zu den einzelnen Datenobjekten der Datenbank gespeichert wurden, können schon die grafischen Ansichten aufgerufen werden. Eine weitere Anpassung der GUI (Graphical User Interface), also der Benutzeroberfläche, ist nicht von Nöten, da die Applikation dank des MVC-Architekturmusters lediglich das für die Datenverwaltung zuständige Model austauschen muss. Die View-Implementierung bleibt die gleiche. Die Darstellung der Graphen übernimmt das schon implementierte Graphen-Framework „JUNG“. Diesem müssen lediglich die gespeicherten Beziehungen übergeben werden, damit die Graphen aufgebaut werden können.

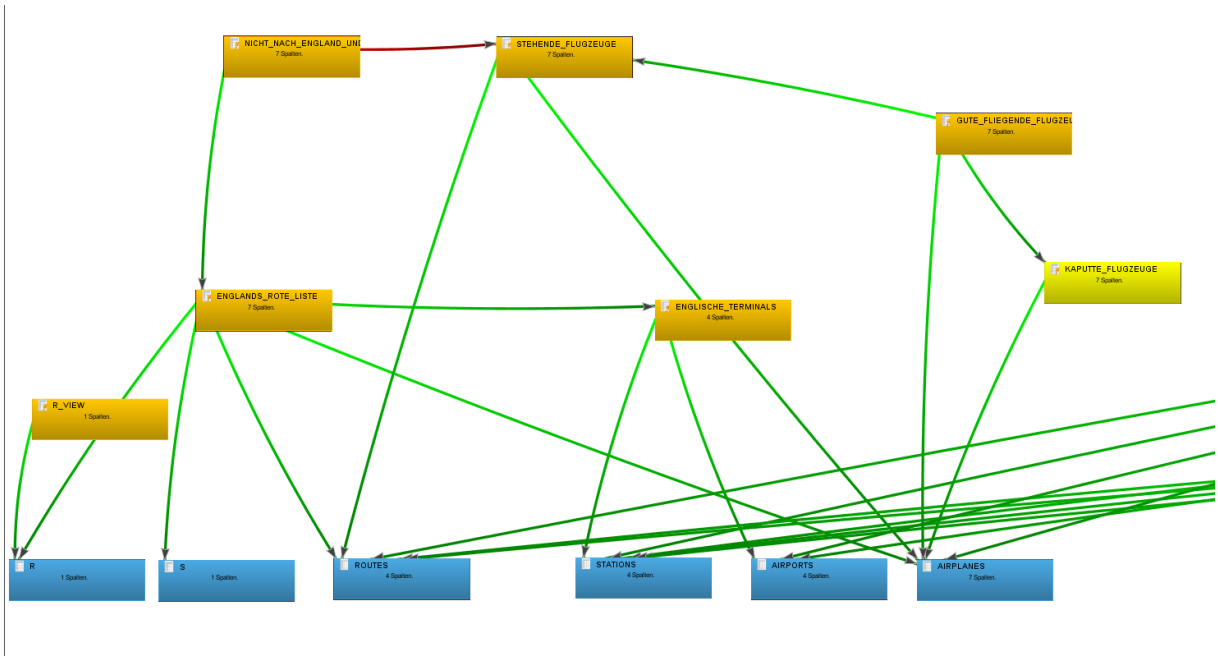

Abbildung 8: Ausschnitt aus der Ansicht der View-Tabellen-Beziehungen

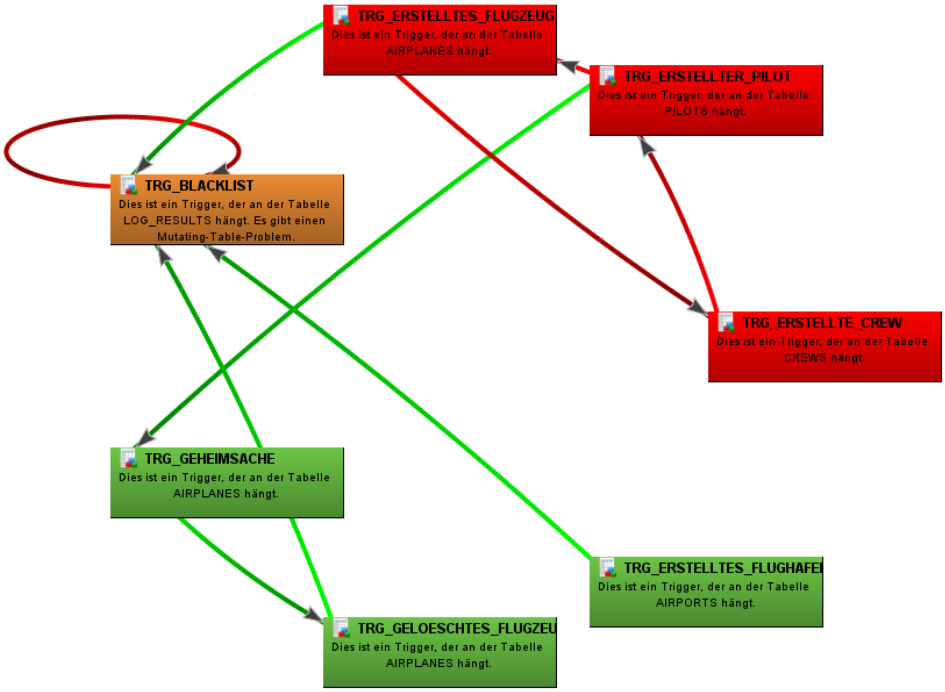

Abbildung 9: Ansicht der Trigger-Beziehungen

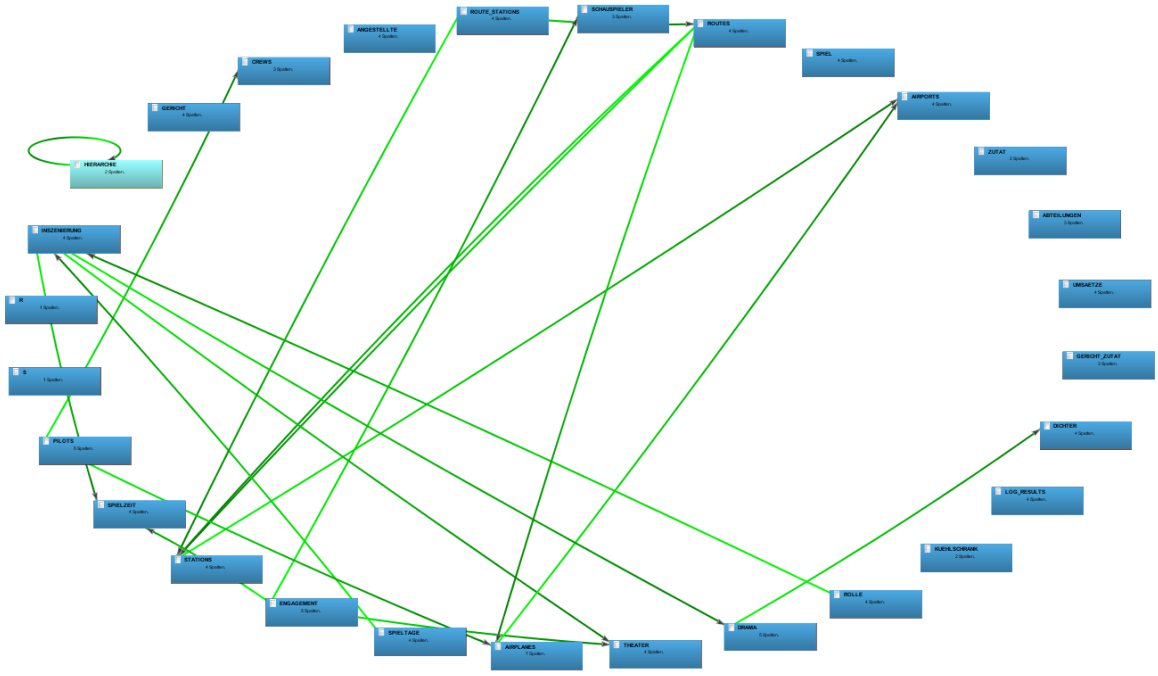

Abbildung 10: Ansicht eines komplexen Entity-Relationship-Diagramms

5. Arbeitspaket 2 – Einbindung der Funktionalitäten für Prozeduren und Funktionen

5.1 Zielsetzung

Das angestrebte Ziel in diesem Arbeitspaket ist die Erweiterung der Software um die Unterstützung von Prozeduren und Funktionen unter Oracle und die Implementierung einer neuen Ansicht.

Vor Beginn der Implementierung müssen zunächst ein paar Analysen und Planungen durchgeführt werden. Hierzu zählt, neben der Analyse des Data Dictionary von Oracle im Hinblick auf den Abruf der Daten über Prozeduren und Funktionen, auch die Planung der Erweiterung der Benutzeroberfläche. Es muss beispielsweise festgestellt werden, wo die Menüpunkte für neue Funktionalitäten platziert werden sollen und wie die neue Ansicht gestaltet sein soll.

Bei der Implementierung soll das Paket der einzufügenden Funktionen in zwei Teilpakete zerlegt werden. Der erste Teil befasst sich mit dem Laden der Daten und der Anzeige der Prozeduren und Funktionen in der Verbindungsansicht (der Ansicht, in der alle Datenbankobjekte in einer Baumdarstellung angezeigt werden). Im zweiten Teilschritt wird dann die neue grafische Ansicht zur Anzeige des neuen Graphen implementiert.

5.2 Laden der Datenstrukturen

Bevor mit der eigentlichen Implementierung des ersten Teilpakets begonnen werden kann, muss zunächst geklärt werden, wie und woher die relevanten Datenstrukturen geladen werden können. Wie alle Metadaten in Oracle, können auch diese Informationen aus dem Data Dictionary geladen werden. Es liegt nahe die Informationen aus der Tabelle „USER_PROCEDURES“ zu laden. Dabei wird jedoch schnell klar, dass diese Tabelle nicht alle geforderten Werte enthält. Sie gibt lediglich Auskunft über den Namen und den Typen (Prozedur oder Funktion) der Objekte. Zusätzlich muss noch der für die weitere Verarbeitung wichtige Ausführungscode geladen werden. Dieser Code befindet sich in der Tabelle „USER_SOURCE“. Die Besonderheit an dieser Tabelle ist die Tatsache, dass der Ausführungscode in Zeilen zerlegt in der Tabelle gespeichert ist. Jede Zeile des Codes besitzt einen Eintrag in der Tabelle zusammen mit der Nummer der Zeile. Es muss nun also der Code für jede Prozedur bzw. Funktion wieder zusammengefügt werden, um ihn in einem Feld speichern und später auch ausgeben zu können.

5.3 Analyse und Speicherung der Datenstrukturen

Um mit den geladenen Daten arbeiten zu können, muss zuerst geklärt werden, wie die Daten gespeichert und in der Software repräsentiert werden sollen. Dazu sind mehrere Klassen notwendig.
Wie schon erwähnt, unterscheiden sich Prozeduren und Funktionen kaum voneinander[21]. Dadurch können die beiden Datenbankobjekte von nur einer Klasse gehandhabt werden. Diese enthält dann die Information, ob es sich um eine Prozedur oder eine Funktion handelt. Um die Daten von Prozeduren und Funktionen innerhalb der Software verwalten zu können, wurde eine Klasse „ProcedureSchema“ implementiert, in der alle relevanten Informationen über die Datenbankobjekte gespeichert werden können. Dazu zählen neben dem Namen und dem Ausführungscode der Prozedur/Funktion auch die Listen mit den Tabellen, die bei der Ausführung des Codes benutzt oder verändert werden. Im Zuge der persistenten Speicherung der Daten über die Programmlaufzeit hinweg, implementiert die genannte Klasse die Schnittstelle „Serializable“. Die Daten können nun serialisiert werden und sind somit für die Speicherung in der Hibernate-Datenbank vorbereitet. Um die Kommunikation zwischen der Software und der Hibernate-Datenbank kümmert sich die Klasse „GenericHibernateDatabaseProcedureDAO“. Sie ist für das Speichern und das Laden der Objekte zuständig.

Nachdem die Grundlage für die Speicherung der Daten geschaffen wurde, kann nun die Analyse der Daten beginnen. Dieser Vorgang findet, wie auch die schon vorhandenen Analysen, beim Anlegen einer neuen Verbindung oder beim Aktualisieren einer bestehenden Verbindung statt. Weitere Aufrufe der Verbindung beziehen die Daten aus der lokalen Hibernate-Datenbank des Programms. Zunächst müssen die Daten aus der Datenbank, wie in Kapitel 5.2 beschrieben, geladen werden und im Anschluss im „ProcedureSchema“ gespeichert werden. Dies betrifft erst einmal nur die Grunddaten wie die Namen und die Ausführungscodes, die betroffenen Tabellen müssen erst noch ausgelesen werden.

Im Rahmen des Parse-Vorgangs sollen alle innerhalb der Prozeduren und Funktionen genutzten und geänderten Tabellen herausgefiltert und gespeichert werden. Da der Ausführungscode der Prozeduren und Funktionen dem der Triggerdefinitionen sehr ähnelt (beide Definitionen sind PL/SQL-Blöcke[22]), liegt es nahe, den bestehenden Trigger-Parser für Oracle genauer zu betrachten, da an dieser Stelle eventuell schon vorhandener Programmcode wiederverwendet werden kann.

[21] Siehe Kapitel 3.2 Datenbankobjekte
[22] PL/SQL ist eine Oracle-spezifische Programmiersprache

Leider ist im Rahmen dieser Prüfung aufgefallen, dass der Trigger-Parser nicht korrekt implementiert wurde. Daher muss ein neuer Parser entworfen werden, der die Anforderungen erfüllt.

5.4 (Neu-) Entwicklung des (Trigger-) Parsers

Bevor der neue Parser implementiert wird, soll noch einmal der bestehende Trigger-Parser genauer untersucht werden. In der Grundstruktur funktioniert der bisherige Parser nach folgendem Prinzip:

Zunächst wird der gesamte Ausführungscode eines Triggers auf Schlüsselwörter untersucht, die eine Änderung oder eine Selektion von einer oder mehreren Tabellen indizieren („SELECT“, „INSERT“, „UPDATE“ und „DELETE“). Bei einem gefundenen Schlüsselwort wird nun der gesamte weitere Code bis zum nächsten Semikolon in ein Array geschrieben, es existiert für jedes Schlüsselwort ein Array mit Statements.
Im Anschluss an diese Analyse werden die einzelnen Arrays in einer Schleife zerlegt und jedes Statement wird noch einmal gesondert untersucht, um die betroffenen Tabellen herauszufiltern. Es wird nun nach weiteren Schlüsselwörtern gesucht. Danach wird das auf ein Schlüsselwort folgende Wort im Statement geprüft, ob es sich hierbei um eine in der Datenbank vorhandene Tabelle handelt. Ist dies der Fall, so wird diese Tabelle der Liste der betroffenen Tabellen hinzugefügt, andernfalls wird an dieser Stelle abgebrochen und das nächste Statement untersucht. Für ein Standard-Statement mag dieses Vorgehen funktionieren, aber schon bei etwas komplexeren Statements treten hier Probleme auf.[23]

Die neue Implementierung behandelt nun auch die komplexeren Statements. Das Grundprinzip des Parsers ist erhalten geblieben, es wird weiterhin der Ausführungscode in die vier Statement-Bereiche unterteilt, jedoch wird ein Statement nicht mehr nur durch ein Semikolon abgeschlossen, sondern kann auch durch Schlüsselwörter beendet werden, um weitere Statement-Arten behandeln zu können[24]. Der nachfolgende Analyseteil wurde jedoch vollständig überarbeitet. Anhand der etwas komplexeren SELECT-Statements soll erklärt werden, wie dieser neue Parser arbeitet.

[23] Siehe Kapitel 2.2 Funktionen und Arbeitspakete - Abschnitt: Arbeitspaket 3
[24] Siehe Anhang E – Prozedur 1 als Beispiel. Das SELECT-Statement in der Schleife wird nicht von einem Semikolon geschlossen. Der neue Parser kann jedoch trotzdem damit umgehen.

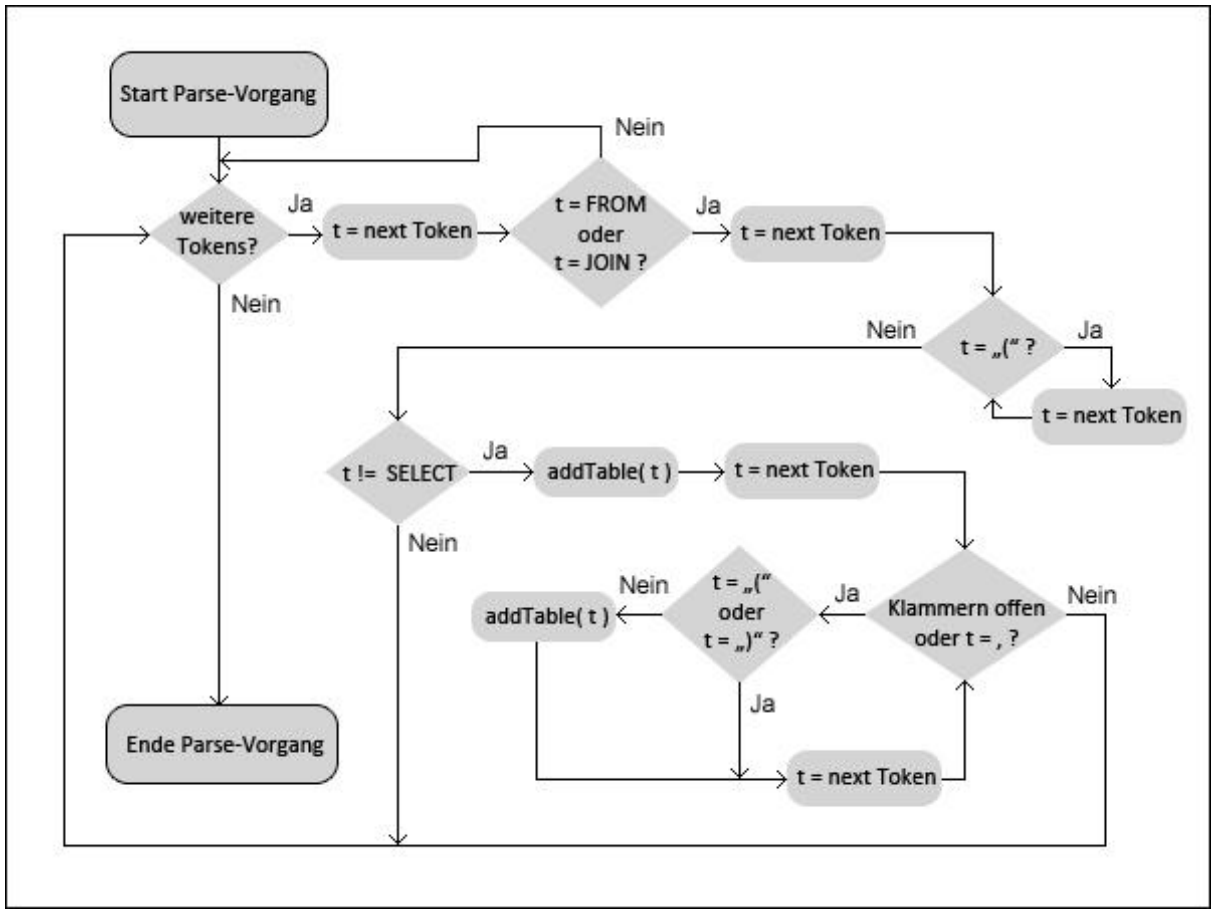

Abbildung 11: Stark vereinfachter Parse-Vorgang bei SELECT-Statements

Zu Beginn des Parse-Vorgangs wird eine Schleife aufgerufen, die prüft, ob noch weitere Token[25] im Statement vorhanden sind. Sind noch weitere Token vorhanden, so wird das nächste Token eingelesen und anschließend auf Übereinstimmung mit einem der beiden Schlüsselwörter „FROM“ und „JOIN“ geprüft. Diese beiden Wörter indizieren Vorkommnisse von Tabellen. Diese müssen jedoch nicht sofort nach den Schlüsselwörtern folgen, sie können ebenso durch eine Klammerung eingeschlossen sein. Desweiteren ist es möglich, dass anstatt eines Tabellennamens ein SUB-SELECT folgt, dies wird durch eine Abfrage nach dem Schlüsselwort „SELECT“ geprüft. Wenn es sich nicht um ein SUB-SELECT handelt und keine Klammern mehr folgen, so ist davon auszugehen, dass es sich bei dem Token um eine Tabelle handelt. Dies wird jedoch vorsichtshalber noch einmal gegen die Liste der in der Datenbank vorhandenen Tabellen geprüft. Existiert eine Tabelle mit dem angegebenen Namen, so kann diese zur Liste der betroffenen Tabellen hinzugefügt werden. Zwar hat der Parser nun eine Tabelle gefunden, jedoch darf noch nicht abgebrochen werden. Unter Oracle ist es möglich, Tabellennamen in einem SELECT kommasepariert

[25] Eingabesymbole bzw. Wörter, die vom Parser akzeptiert werden.

anzugeben. Dies muss ebenfalls geprüft werden, wobei auch hier wieder auf etwaige Klammern geprüft werden muss. Weiterführend könnte es ebenfalls vorkommen, dass ein SUBSELECT zu einem späteren Zeitpunkt im Statement auftaucht (z.B. in der WHERE-Klausel). Dies wird dadurch abgefangen, dass auch nach einem gefundenen „FROM“ das Statement nach diesem Wort untersucht wird, bis keine Tokens mehr vorhanden sind. Somit sollten alle gängigen Formulierungen von SELECT-Statements vom neu implementierten Parser abgedeckt sein.

Die zuvor in Listen gespeicherten Tabellen, werden im Anschluss an den Parse-Vorgang zusammen mit ihrer zugehörigen Prozedur abgespeichert.

Der Parse-Vorgang für die drei anderen Statement-Arten wurde ebenfalls verbessert, ist allerdings weniger kompliziert. Hierbei wird wieder nach Schlüsselwörtern gesucht, die das Vorkommen eines Tabellennamen indizieren (z.B. INSERT INTO <tabellenname>). Auch hier wurde im ursprünglichen Parser eine mögliche Klammersetzung nicht berücksichtigt, was nun im neuen Parser behoben ist.

Da die Ausführungscodes der Trigger und der Prozeduren / Funktionen sich stark ähneln und beide einen PL/SQL-Block verwenden, kann die Parse-Routine in beiden Fällen verwendet werden.

5.5 Entwurf und Implementierung des neuen Abschnitts in der Verbindungsansicht

Nachdem nun alle geforderten Daten zu den Prozeduren und Funktionen gesammelt, analysiert und abgespeichert wurden, kann die Entwicklung der Oberfläche begonnen werden. Hierbei sind vorerst nur der Entwurf und die Umsetzung der Erweiterung der Verbindungsansicht gemeint. Die Gestaltung der Oberfläche für die Ansicht des Graphen erfolgt zu einem späteren Zeitpunkt.

In erster Linie ist es vorgesehen, dass sich das Design an den schon vorhandenen Ansichten orientiert, um die neuen Funktionen bestmöglich in die Software zu integrieren. Da die Trigger-Detailansicht ebenso wie die Prozedur-Detailansicht nur einen Tab benötigt, um alle Informationen darzustellen, kann diese als Vorlage genutzt werden.

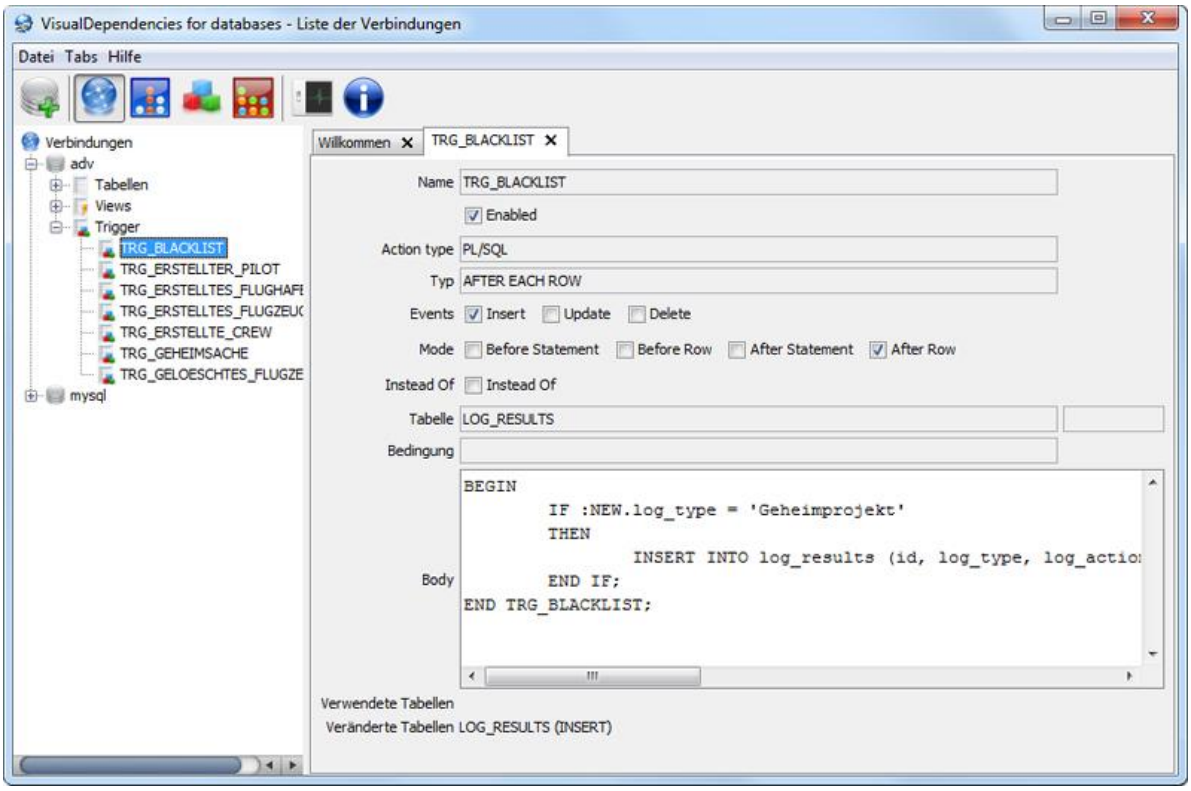

Abbildung 12: Detailansicht eines Triggers als Vorlage für die neue Verbindungsansicht

Anhand der in der Abbildung 12 gezeigten Ansicht wurde zunächst ein Wireframe-Modell, also ein Grundgerüst der neuen Oberfläche, entworfen, um einen groben Überblick über die Positionen und Größen der dargestellten Informationen zu erhalten. Die folgende Abbildung 13 zeigt das Wireframe-Modell:

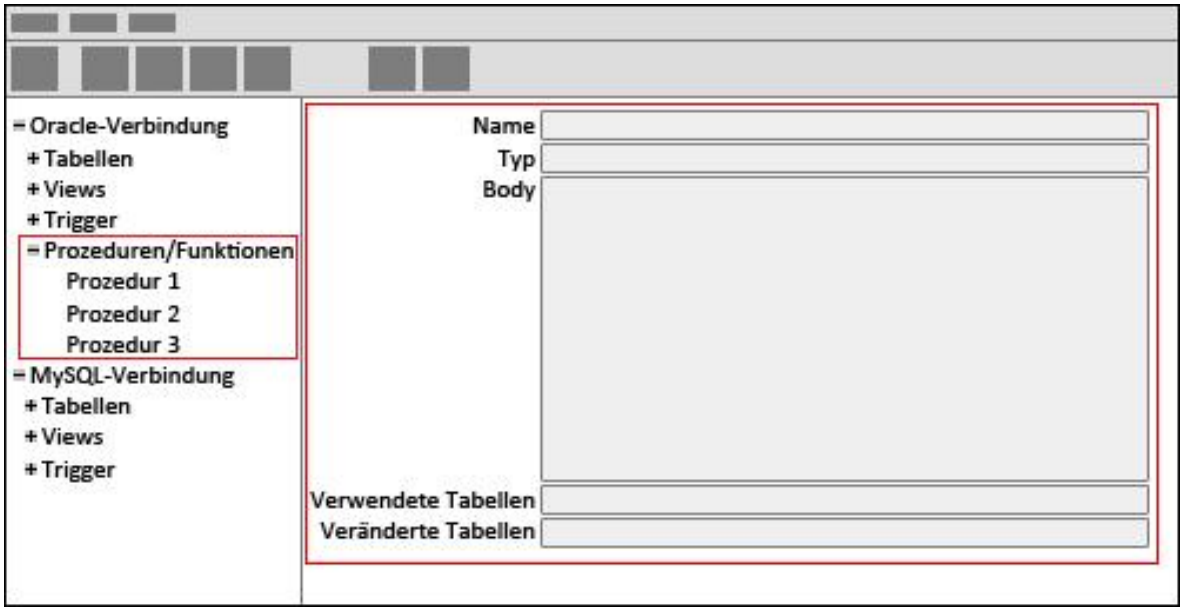

Abbildung 13: Wireframe-Modell der neuen Verbindungsansicht

Der in der linken Spalte gekennzeichnete Block stellt die Erweiterung des Verbindungsbaumes dar. Zusätzlich zu den vorhandenen Unterpunkten der Verbindungen (Tabellen, Views und Trigger) soll ein weiterer Punkt hinzugefügt werden, der die Prozeduren und Funktionen aufnimmt. Es ist zu beachten, dass dieser neue Navigationspunkt lediglich für Oracle-Verbindungen verfügbar sein soll. Eine Erweiterung der MySQL-Verbindungen um Prozeduren und Funktionen wäre denkbar, ist jedoch nicht Bestandteil dieser Arbeit.

Bei einem Klick auf eine Prozedur oder Funktion im Verbindungsbaum öffnet sich ein neuer Reiter, welcher die Detailansicht beinhaltet. Diese Ansicht ist im Wireframe-Modell auf der rechten Seite gekennzeichnet. Neben dem Namen, dem Typ (Prozedur oder Funktion) und dem Ausführungscode (Body) des Datenbankobjektes sollen an dieser Stelle auch die verwendeten und die veränderten Tabellen angezeigt werden. Wie bei der Trigger-Detailansicht soll auch hier für den Nutzer kenntlich gemacht werden, welche Veränderungen auf den Tabellen bei der Ausführung einer Prozedur oder Funktion vorgenommen werden (INSERT, UPDATE oder DELETE).

Durch den Entwurf des Wireframe-Modells liegt nun eine klare Definition der Oberfläche vor, anhand der nun zunächst die Baum-Ansicht auf der linken Seite erweitert wird. Dies gestaltet sich relativ einfach. Die Gestaltung des Baumes findet in der Klasse „ConnectionViewSidebar“ im Paket „ui.views.connections“ statt. Die Klasse besitzt eine Methode namens „buildNodes“, die verschiedene Methoden für die Generierung der Untergruppen einer Verbindung (Tabellen, Views und Trigger) aufruft. An dieser Stelle wird also der Baum für eine Verbindung erzeugt. Der Klasse muss nun eine weitere Methode „buildProcedureNodes“ hinzugefügt werden, die für die Generierung des neuen Abschnitts für Prozeduren und Funktionen zuständig ist. Hierbei wurde darauf geachtet, die Zweisprachigkeit der Anwendung fortzuführen: Die passende Übersetzung für den Oberpunkt wird aus einer Sprachdatei geladen. An der Stelle des Aufrufs der Methode „buildProcedureNodes“ in der Methode „buildNodes“ findet auch die Abfrage statt, die prüft, um welchen Datenbanktyp es sich handelt und ob der Punkt „Prozeduren und Funktionen“ überhaupt dargestellt werden soll. Die Prüfung und die etwaige Ausblendung des Zweiges im Baum sind aufgrund der fehlenden Unterstützung unter MySQL notwendig.

Bei Auswahl einer Prozedur oder Funktion aus der Baumdarstellung, soll sich ein neuer Tab mit der Detailansicht öffnen. Die Routine, die für die Öffnung eines neuen bzw. die Auswahl eines bestehenden Tabs zuständig ist, existiert schon in der Applikation und muss lediglich um eine zusätzliche konkrete Implementierung des Tabs erweitert werden. Hierzu muss in

der Klasse „ConnectionTabbedPane“, die für die Verwaltung der Tabs zuständig ist, eine Prüfung stattfinden, ob es sich bei dem zu öffnenden Tab um einen Tab für eine Prozedur bzw. Funktion handelt. Sollte diese Prüfung positiv ausfallen, so wird ein neues Objekt vom Typ „TabPanelProcedure“ erzeugt bzw., wenn der Tab schon geöffnet ist, das schon existierende Objekt zurückgegeben. Die Klasse „TabPanelProcedure“ erweitert die Klasse „JPanel“ der Java-Grafikbibliothek „Swing“ und enthält die Detailansicht der gewählten Prozedur oder Funktion, wie sie im Wireframe-Modell auf der rechten Seite dargestellt wird.

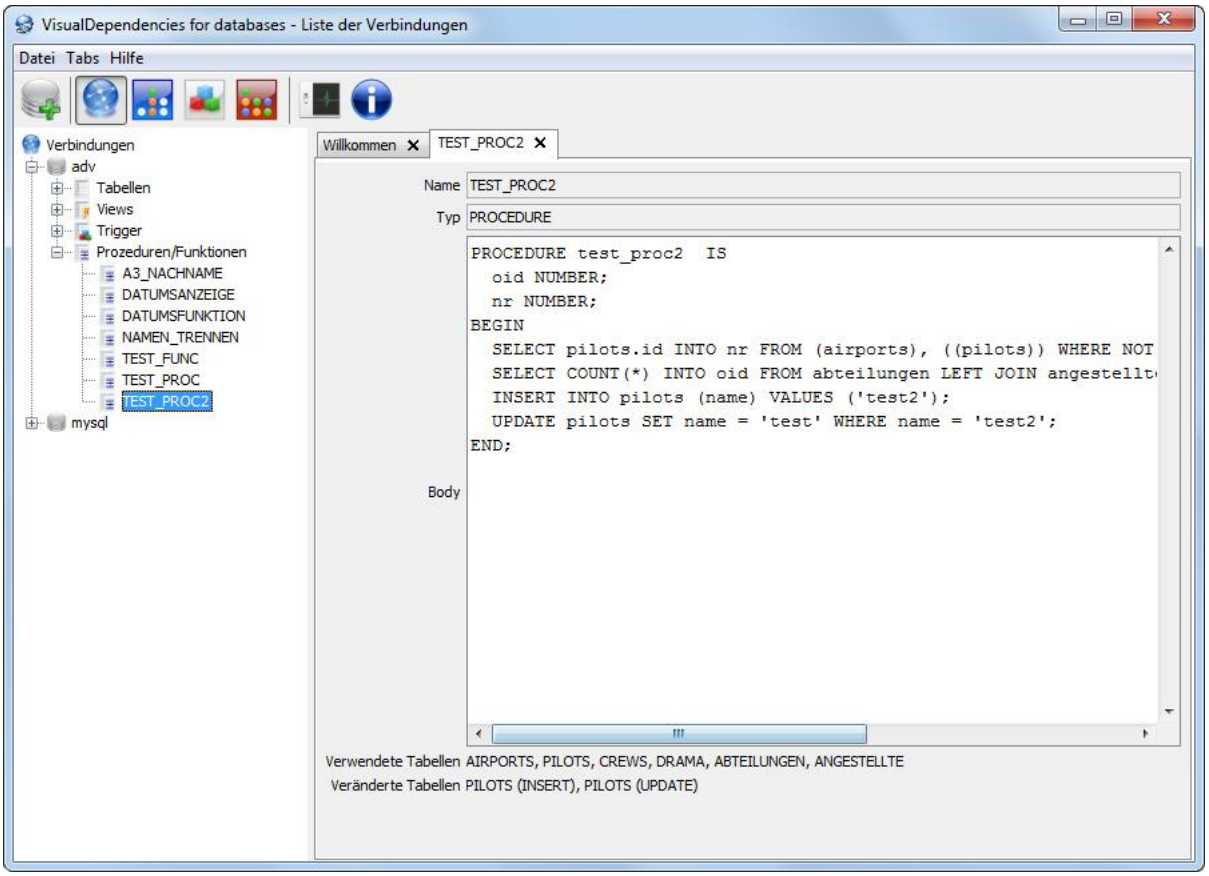

Abbildung 14: Neue Verbindungsansicht und Prozedur-Detailansicht

Im Rahmen der Implementierung der neuen Reiter für Prozeduren und Funktionen wurde viel mit der Verwaltung der Reiter gearbeitet. Die Verwaltung der Reiter meint beispielsweise das Schließen von einzelnen oder auch allen Tabs. Dabei ist aufgefallen, dass ein Fehler beim Schließen aller bzw. aller anderen Tabs entsteht. Die Tabs werden sowohl als Objekte im Tabbed Pane[26], als auch in einer weiteren internen Liste gespeichert. Diese interne Liste wird genutzt, um schon offene Tabs wiederzufinden. Wird ein einzelner Tab geschlossen, so

[26] Ein Container der Java-Grafikbibliothek „Swing“ zur Aufnahme von Tabs

werden das Objekt des Tabs und der Eintrag in der Liste gelöscht. Beim Schließen aller Tabs, sollten auch alle Tabs aus beiden Speicherorten gelöscht werden. Die Tabs werden auch aus der Ansicht entfernt, jedoch bleiben sie in der internen Liste gespeichert. Es ist nun nicht mehr möglich schon geschlossene Tabs wieder zu öffnen. Dies liegt an der Tatsache, dass versucht wird ein Objekt aufzurufen, da es sich noch in der Liste befindet. Das Objekt existiert jedoch nicht mehr. Um das Problem zu beheben wird nun beim Löschen aller Tabs die gesamte Liste der Objekte geleert. Zusätzlich dazu wird wieder der Tab mit dem Willkommensansicht geöffnet.

5.6 Entwurf und Implementierung der neuen Graphenansicht

Nachdem die Bearbeitung des ersten Teilpakets, also die Einbindung der neuen Datenbankobjekte in die Verbindungsansicht, abgeschlossen wurde, soll nun das zweite Teilpaket entworfen und implementiert werden. Dieses Teilpaket beschäftigt sich mit der Umsetzung der neuen Graphenansicht (Einzelansicht), in der die mit einer ausgewählten Tabelle in Verbindung stehenden Datenbankobjekte angezeigt werden. Diese Ansicht soll neben den bisherigen Objekten auch eine Repräsentation der Prozeduren und Funktionen enthalten.

Entwurf und Anforderungen

Bevor mit der Implementierung begonnen werden kann, ist zu überlegen, welche Funktionalitäten die Ansicht beinhalten soll, wie diese umsetzbar sind und wie die neue Ansicht grafisch dargestellt werden soll.

Da ein Graph angezeigt werden soll, der ein einzelnes Tabellenobjekt und zusätzlich die von dieser Tabelle abhängigen Datenbankobjekte enthält, ist es ersichtlich, dass alle Objekte in dem Graphen auf die Tabelle zeigen müssen. Aus dieser Aussage geht ebenfalls hervor, dass es dem Nutzer ermöglicht werden muss, eine Auswahl der Tabelle treffen zu können.

Weiterführend ist zu überlegen, ob alle Funktionen der bisherigen Graphenansichten übernommen werden oder ob Funktionalitäten wegfallen oder gar neue hinzukommen sollen. Dafür soll zunächst aufgelistet werden, welche Funktionalitäten bisher in den Graphenansichten verfügbar sind:

- Es bestehen diverse Funktionalitäten, um den Graphen optisch zu verändern. Hierzu zählen das Bewegen und Zoomen, genauso wie die Drag & Drop

Funktionalität. Außerdem ist es dem Benutzer möglich eines von mehreren verschiedenen Layouts zu wählen.

- Der Nutzer kann die aktuelle Ansicht als Bild im PNG-Format oder als DOT-Datei für die weitere Verwendung im Graphenframework Graphviz exportieren.
- Bei einem Rechtsklick auf einen Knotenpunkt des Graphen (ein Datenbankobjekt) erfährt der Nutzer in einem Kontextmenü weitere Informationen über das Objekt.
- In der linken Navigation kann der Benutzer ein Popup-Fenster öffnen, dass ihm die Möglichkeit gibt, Objekte mit ihren Relationen aus dem Graphen zu entfernen oder hinzuzufügen (einzelne Objekte können auch über das Kontextmenü gelöscht werden, was sich bei einem Rechtsklick auf das Objekt öffnet).

Wenn man die Punkte betrachtet, so wird schnell klar, dass dem Benutzer weiterhin die Möglichkeit geboten werden muss, den Graphen optisch zu verändern, um ihn an die eigenen Bedürfnisse anzupassen. Auch der Export des Graphen stellt eine wichtige Funktionalität der Graphenansicht dar und sollte nicht entfernt werden. Beispielsweise könnte somit eine Momentaufnahme des Graphen in einer Präsentation dargestellt werden, ohne die Software benutzen zu müssen. Ebenso wie die beiden genannten Funktionen ist die Funktion zur Anzeige von weiteren Informationen über die Datenbankobjekte von großer Bedeutung, da der Nutzer sonst die Ansicht wechseln müsste, um an die Informationen zu gelangen.
Betrachtet man nun die Funktionalität Objekte aus dem Graphen löschen zu können, so stellt sich die Frage, ob diese Funktion überhaupt noch relevant ist. Meiner Meinung nach ist diese Funktion für diesen Graphen nicht unbedingt von Nöten, da für den Graphen nur eine geringe Komplexität zu erwarten ist. Dies beruht auf der Tatsache, dass nur ein Tabellenobjekt mit seinen Beziehungen zu anderen Datenbankobjekten dargestellt wird, was dazu führt, dass jedes Objekt in dieser Ansicht nur eine einzige Relation besitzt, die auf die Tabelle zeigt. Daher kann diese Funktionalität in dieser Ansicht entfernt werden.

Da durch das Entfernen der genannten Funktion ein Platz in der linken Navigation frei geworden ist, kann dieser durch eine Auswahlmaske gefüllt werden. Diese Auswahlmaske soll alle Tabellenobjekte der aktuellen Datenbankverbindung in alphabetischer Reihenfolge beinhalten. Bei Auswahl eines Eintrags muss der Graph automatisch aktualisiert werden.

Zusätzlich zur neuen Ansicht selbst, muss es natürlich für den Benutzer auch eine Möglichkeit geben, die neuen Funktionalitäten aufzurufen. Dazu soll in der Menüleiste zur Auswahl der Ansichten ein weiteres Icon hinzugefügt werden, welches die neue Ansicht aufruft. Das neue Icon soll sich gestalterisch ebenfalls an der Ausgangssoftware orientieren.

Um sich die Positionierung der Elemente und das Design der neuen Ansicht besser vorstellen zu können, wurde hierbei ebenfalls ein Wireframe-Modell entworfen (siehe Abbildung 15).

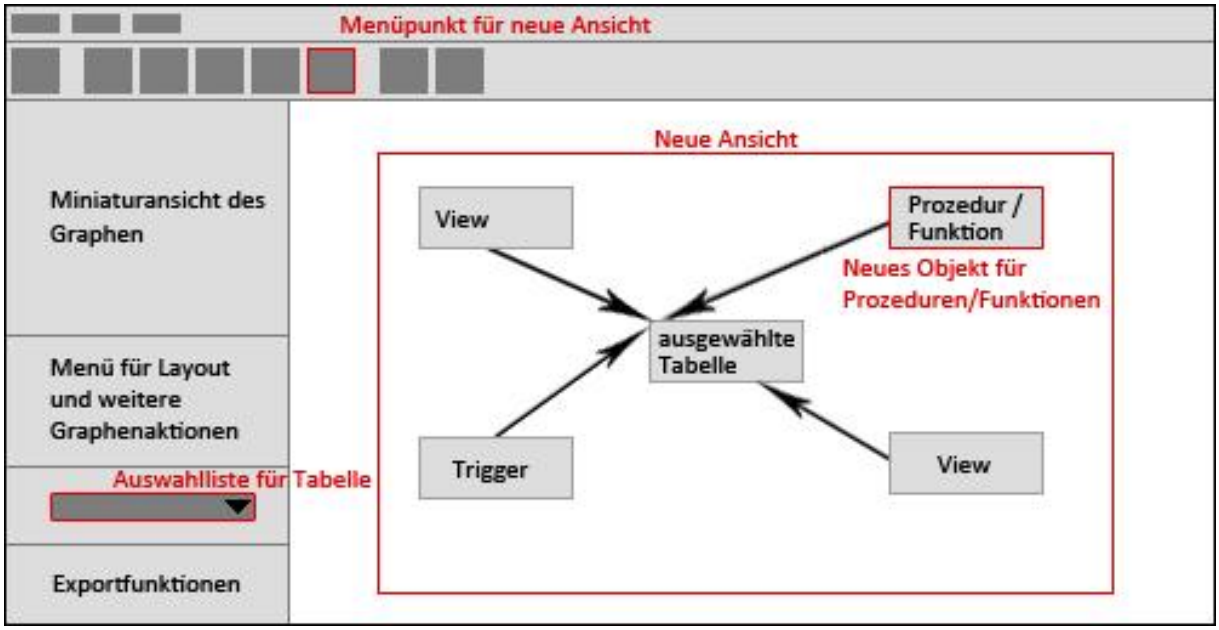

Abbildung 15: Wireframe-Modell der neuen Einzelansicht

In diesem Wireframe-Modell sind die neuen Funktionen und Darstellungen in rot gekennzeichnet. Über den neuen Menüpunkt in der oberen Toolbar soll später die neue Ansicht aufrufbar sein. Auf der linken Seite ist die Auswahlmaske für die Tabellen angedeutet, welche die vorherige Funktion, das Löschen und Hinzufügen von Datenbankobjekten, ersetzt. In der neuen Ansicht ist ebenfalls das neue Objekt für Prozeduren und Funktionen zu sehen.

Implementierung

Da sich die Ansicht nicht sehr stark von den vorhandenen Ansichten abgrenzt, liegt es nahe eine vorhandene Ansicht zu duplizieren, um ein Grundgerüst für die neue Ansicht zu schaffen. Hierzu wurden alle mit der Ansicht in Verbindung stehenden Pakete und Klassen dupliziert. Desweiteren wurden die Klassennamen neu benannt, dies muss sorgfältig in den betroffenen Dateien geprüft werden. Im Anschluss an diese Umbenennung kann die Klasse „ApplicationViewToolBar“ betrachtet werden. Diese Klasse beinhaltet die gesamte Darstellung der Menüleiste zur Auswahl der Ansichten, hier muss lediglich ein neuer Button mit einem Icon eingefügt werden.

Für die Darstellung der Inhalte sind in erster Linie zwei Klassen zuständig: Die Klasse „ProcViewData“ ist für die Verwaltung des Graphen und dessen Aktualisierung zuständig, die Klasse „ProcViewSidebar“ implementiert die linke Navigation. Da die linke Navigation fast identisch bleibt, wurden dort nur kleinere Änderungen vorgenommen. Die neue Auswahlliste muss mit der Liste der Tabellen der aktuellen Datenbankverbindung gefüllt werden. Aufgrund der schon bestehenden Programmierung ist innerhalb der Sidebar-Klasse die aktuelle Verbindung nicht immer verfügbar. Daher wurde eine Methode implementiert, die die Verbindung über das zugehörige ProcViewData-Objekt erfragt. Eine Aktualisierung der Auswahlliste sorgt dafür, dass der Graph aktualisiert wird. Die Klasse „ProcViewData“ ist prinzipiell nur für die Verwaltung des Graphen zuständig. Liegt eine Änderung vor, also wird z.B. ein neues Layout oder eine neue Tabelle gewählt, so sorgt die Klasse für das Neuzeichnen des Graphen. Der eigentliche Graph wird innerhalb der Klasse „DatabaseProcModelGraphTransformer“ erzeugt. Hierbei werden alle Listen mit Datenobjekten der aktuellen Verbindung auf ihre Beziehung zur gewählten Tabelle geprüft. Existiert eine Beziehung, so wird das Objekt zum Graphen hinzugefügt und eine Kante zwischen dem Objekt und der Tabelle erzeugt, die bei einer Berührung mit dem Mauszeiger anzeigt, welche Beziehung zwischen den Objekten besteht. Da bisher noch keine Repräsentation der Prozeduren und Funktionen im Graphen existiert, muss die Klasse „DatabaseObjectComponent“ um diesen Fall erweitert werden.

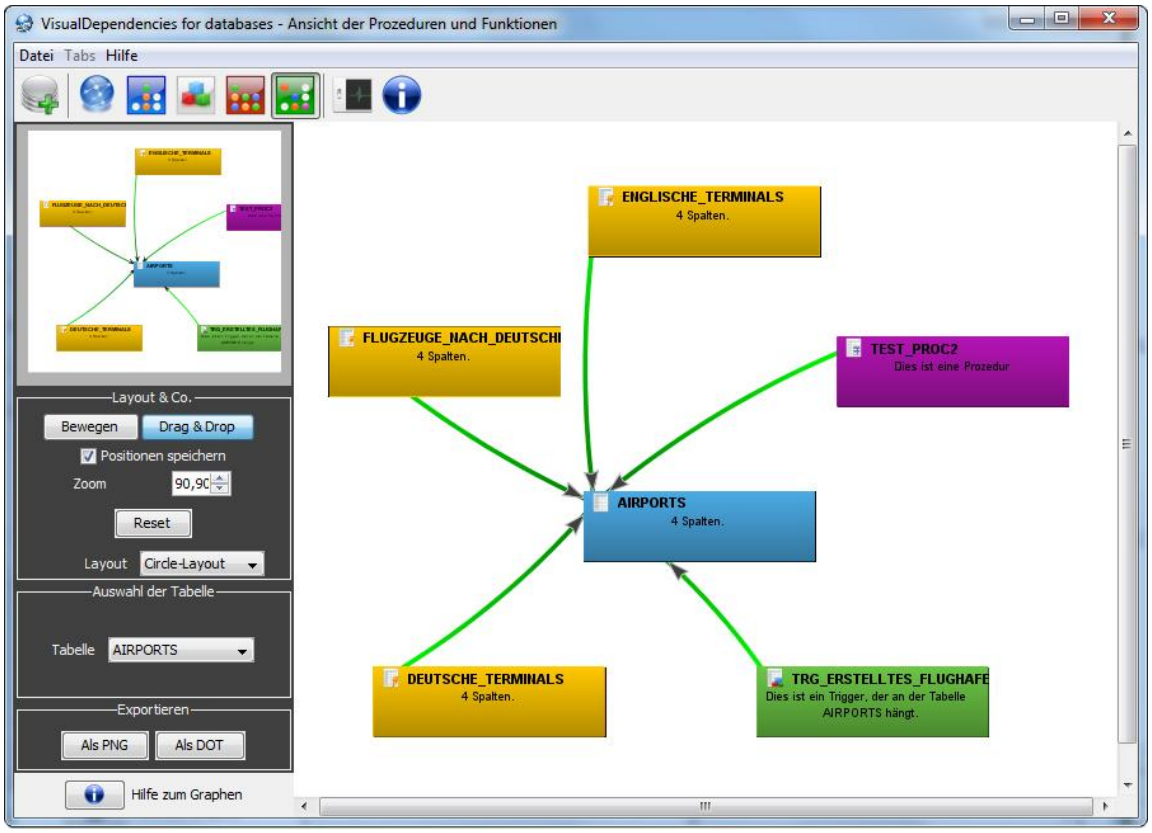

Abbildung 16: Fertige Einzelansicht

Nachdem die Implementierung für die neue Ansicht an sich abgeschlossen ist, kann nun noch eine weitere Funktionalität verbessert werden. Bisher öffnete sich bei einem Rechtsklick auf ein Trigger-Objekt ein Kontextmenü, das unter anderem den Unterpunkt „Betroffene Tabellen“ enthält. Dieser Unterpunkt zeigte bisher nur die im Trigger veränderten Tabellen, die verwendeten Tabellen wurden nicht angezeigt. Weiterführend ist aus den gegebenen Informationen nicht ersichtlich, welche Änderung bei Ausführung des Triggers an den Tabellen vorgenommen wird. Dies soll nun im Zuge der Weiterentwicklung ebenfalls geändert werden. Die Funktion wird ebenfalls für die Prozeduren und Funktionen implementiert, da diese ähnlich aufgebaut sind. Die Inhalte des Kontextmenüs werden View-spezifisch in der Klasse „PopupMenu“ generiert. Um die Inhalte zu laden, müssen erneut die Listen mit den betroffenen Tabellen geladen und untersucht werden. Hierbei wird gespeichert, welche Aktionen auf den Tabellen durchgeführt werden, was dann im Kontextmenü dargestellt werden kann.

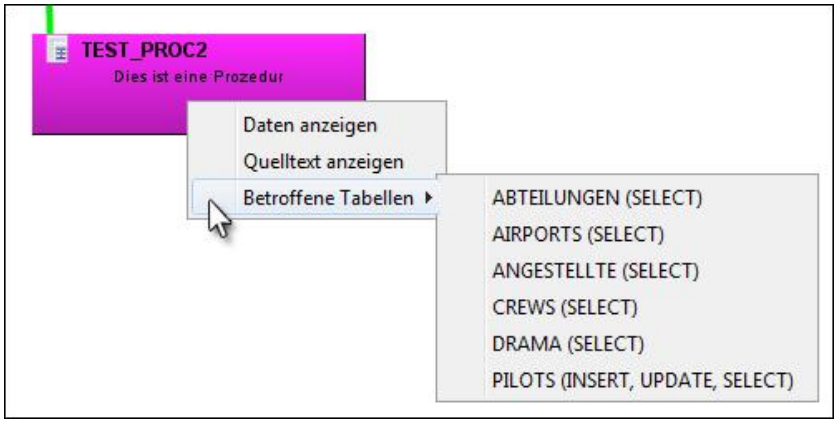

Abbildung 17: neues Kontextmenü bei einem Rechtsklick auf ein Prozedur-Objekt

Abbildung 18: Neue Willkommensansicht der Software

6. Arbeitspaket 3 – Graphenanalysen

Dieses Kapitel soll einen kurzen Ausblick auf Verbesserungsmöglichkeiten im Bereich der Graphendarstellungen bieten und Denkanstöße für die Weiterentwicklung der Software geben. Die Möglichkeiten sollen lediglich angedeutet werden, eine Implementierung der genannten Verbesserungen ist vorerst nicht vorgesehen.

Betrachtet man die vorliegenden Graphenansichten, so wird schnell klar, wo die derzeitigen Grenzen der Software liegen. Die vorgegebenen Layouts erfüllen nicht die Anforderungen, die man an eine ästhetische Darstellung von Graphennetzwerken stellt.
Es gibt viele ästhetische Kriterien, die man an Graphenlayouts stellen kann. Für den Benutzer von hohem Wert, ist die Minimierung von Kanten- und Knotenüberschneidungen. Durch eine Vielzahl an Überschneidungen wird das Schaubild unübersichtlich und ist somit von geringerem Nutzen. Eine weitere Möglichkeit den Graphen für das menschliche Auge ansehnlich zu machen, könnte das Finden und Darstellen von Symmetrien innerhalb des Graphen sein. Die Darstellung des Graphen innerhalb eines möglichst kleinen Bereichs wird ebenfalls als angenehm und übersichtlich empfunden. Es gibt noch weit mehr als diese Möglichkeiten um Graphen schöner zu gestalten, die Nennung der Kriterien sollte in erster Linie nur einen Anreiz geben darüber nachzudenken, wie solche Verbesserungen aussehen könnten.

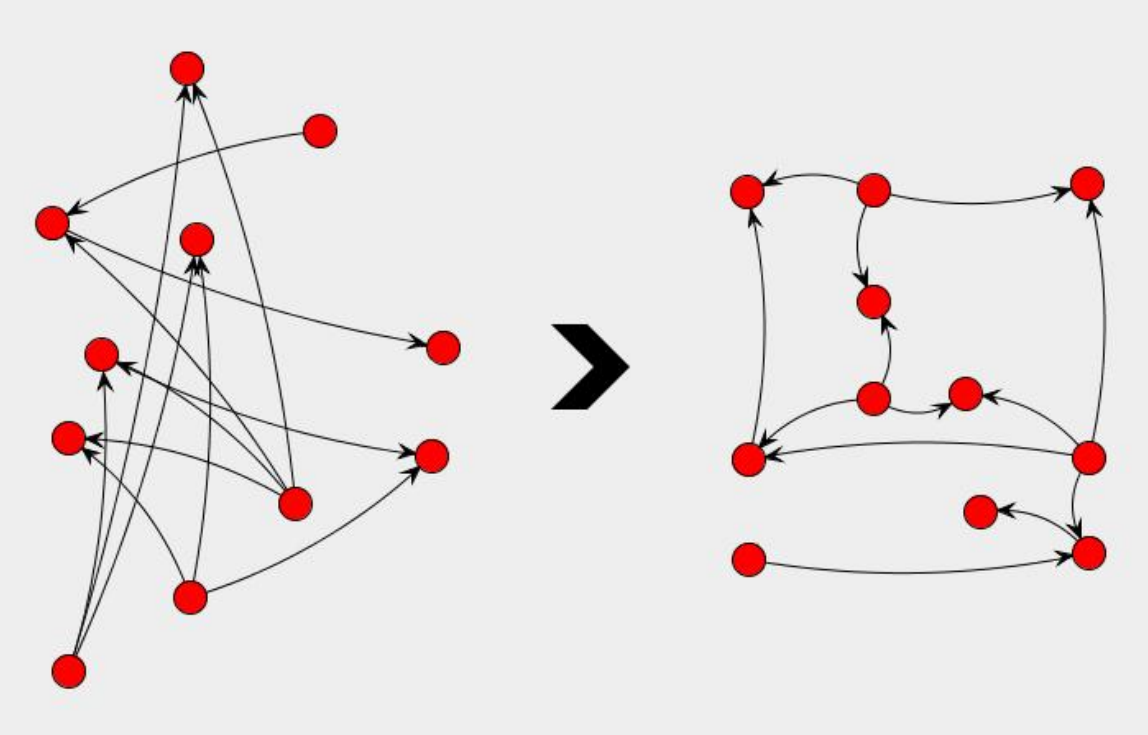

Abbildung 19: unästhetischer Graph (links) und seine planare und symmetrische Darstellung (rechts)[27]

Ich möchte an dieser Stelle zu dem Problem der Überschneidungen in Graphen zurückkommen. Ein Graph mit keinerlei Überschneidungen der Kanten wird als planar bezeichnet. Es existieren verschiedenste Algorithmen zur Untersuchung, ob ein Graph planar ist oder nicht. Dazu zählen der 1967 von Lempel, Even, Cederbaum entwickelte Algorithmus „vertex addition“ mit einer Laufzeit von $O(n^2)$ und der darauf basierende „path addition“-Algorithmus von Hopcroft und Tarjan von 1974 mit einer Laufzeit von $O(n)$. Der modernste Planaritätstest-Algorithmus, der auf der Tiefensuche basiert, wurde 2004 von Boyer und Myrvold veröffentlicht.[28]

Zwar gibt es einige (mittlerweile auch schnelle) Verfahren um herauszufinden, ob ein Graph planar ist, jedoch löst dies nicht das Problem der Darstellung, da die Verfahren lediglich prüfen, ob ein Graph planar ist, ihn jedoch nicht zeichnen. Zudem muss man sich wohl auch die Frage stellen, ob bei der Komplexität der gegebenen Graphen überhaupt ein planarer Graph möglich ist, was bei komplexeren Schemata wohl oftmals nicht der Fall ist. Demnach muss eine Lösung gefunden werden, wie die Graphen automatisch möglichst überschneidungsfrei gezeichnet werden können. Eine möglicherweise nutzbare Lösung bietet das so genannte „Tutte Embedding“-Verfahren von William Thomas Tutte. Es basiert

[27] Die beiden Graphen wurden mit Hilfe des JUNG-Frameworks erzeugt: http://jung.sourceforge.net/
[28] Vgl. [BM 04]

auf der Annahme, dass sich die Knoten eines Graphen gegenseitig anziehen. Die inneren Knoten sollen nun im „Gravitationszentrum“ positioniert werden, um eine möglichst überschneidungsfreie Ansicht zu gewährleisten. Durch diesen Algorithmus werden verwandte Knoten nah beieinander platziert, während die Überschneidungen der Kanten minimiert werden. Das Verfahren ist zwar ebenfalls grundsätzlich für planare Graphen ausgelegt, liefert aber meist für nicht planare Graphen ebenfalls gute Ergebnisse.

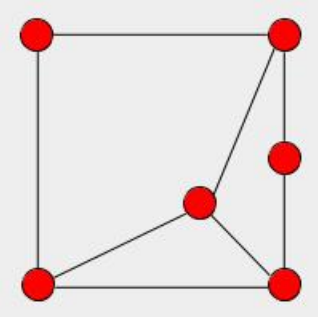

Abbildung 20: Tutte Embedding angewendet auf einen einfachen Graphen

6.1 Mögliche grafische Erweiterungen anhand eines aktuellen Beispiels

Zur Verdeutlichung der Wichtigkeit der Verbesserung der Graphendarstellungen möchte ich ein aktuelles Beispiel anführen:

Während sich die ersten Formen von sozialen Netzwerken im Internet noch auf einfache Bekanntschaftsbeziehungen zwischen den Personen beschränkten, gehen die aktuellen Internetportale, wie beispielsweise Facebook[29], StudiVZ[30] oder auch Xing[31], weit über dieses simple Modell hinaus und legen deutlich komplexere Datenmodelle zu Grunde. Um nur ein paar Features zu nennen, ist zu sagen, dass beispielsweise StudiVZ es dem Benutzer ermöglicht verschiedenen Gruppen beizutreten oder andere Personen auf Bildern zu markieren. Facebook geht auch darüber noch hinaus und bietet dem Nutzer die Möglichkeit weitere Bekannte zu finden, indem es sich beispielsweise mit dem Account des Nutzers im Chatprogramm Skype vernetzt und die dortigen Bekanntschaften herausfiltert. Nimmt man das Beispiel von Xing, einem Internetportal für vorwiegend geschäftliche Beziehungen, so kann der Nutzer sein Profil mit seinen persönlichen Fähigkeiten und beruflichen Erfahrungen füllen und Xing bietet dem Anwender dann Jobangebote potenzieller Arbeitgeber an, die auf

[29] http://www.facebook.com/
[30] http://www.studivz.net/
[31] http://www.xing.com/

sein Profil passen könnten. Die meisten modernen sozialen Netzwerke gehen getreu nach dem Motto: „Was jemandem mit ähnlichem Profil und Geschmack gefällt, könnte ihnen auch gefallen“.
Diese Fülle an komplexen Bekanntschaftsbeziehungen stellt die Entwickler der Datenmodelle vor große Herausforderungen, bei denen schnell der Überblick verloren werden kann. Das entwickelte Softwaretool unterstützt den Entwickler zwar schon jetzt bei seiner Arbeit, jedoch wird klar, dass die Software bei der Menge an Beziehungen und derart komplexen Datenmodellen schnell an ihre Grenzen stößt, da Überschneidungen nicht mehr vermieden werden können und die Übersichtlichkeit stark darunter leidet.

6.2 Ergänzungen im dreidimensionalen Raum

An dieser Stelle könnte ebenfalls eine mögliche Verbesserung der Graphen ansetzen, indem die Ansicht um eine weitere Dimension ergänzt wird. In diesem dreidimensionalen Raum ist eine überschneidungsfreie Ansicht möglich. Dem Nutzer sollte im Rahmen dieser Erweiterung die Möglichkeit gegeben werden, sich sozusagen frei im Raum zu bewegen. Hierzu müssen Zoom- und Drehfunktionalitäten implementiert werden. Darüber hinaus sollte es weiterhin die Drag & Drop Funktion zum Bewegen der Knoten geben. Außerdem wäre es weiterführend denkbar, dass eine Funktion hinzugefügt wird, die dem Nutzer die Möglichkeit gibt, einzelne Knoten im Graphen zu expandieren bzw. zu kontrahieren. So wäre es beispielsweise denkbar, dass zunächst das Entity-Relationship-Diagramm angezeigt wird. Wenn der Nutzer jedoch weitere Informationen zu einer Tabelle erhalten möchte, kann er den Tabellenknoten expandieren, wodurch weitere Datenbankobjekte wie Trigger, Views oder Prozeduren zum Vorschein kommen.

Zur Optimierung der Darstellung der Graphen eignen sich gerade im 3D-Raum die so genannten Kräftemodelle. Das eben schon erwähnte Tutte-Embedding ist eine Art dieser Modelle. Bei dreidimensionalen Darstellungen geht man jedoch noch einen Schritt weiter. Mit den hierbei verwendeten Algorithmen werden Verfahren implementiert, die „die Relationen zwischen den Einheiten eines Graphen als ein System von Kräften auffassen“[32]. Es wird in diesem Zusammenhang von zwei Arten von Kräften gesprochen: anziehenden und abstoßenden Kräften.

Die Knoten der Graphen repräsentieren sich gegenseitig abstoßende Pole. Sie versuchen sich also voneinander zu entfernen, was eine gute Verteilung der Knoten im Raum bewirkt. Die Stärke der abstoßenden Wirkung nimmt quadratisch mit der Entfernung der Knoten ab,

[32] [KRE 05], S. 103

damit sich die Knoten nicht bis ins Unendliche voneinander entfernen (Hier sind auch andere Kräfteverhältnisse denkbar, jedoch ist dies die üblich verwendete Größe).

Die Kanten stellen in diesem Kräftemodell sich anziehende Kräfte dar. Man kann sich die Beziehungen der Knoten untereinander als Federn vorstellen. Je weiter ein Knoten von einem mit ihm verbundenen Knoten entfernt ist, desto größer ist die anziehende Kraft der Feder zwischen den beiden Knoten. Es wird aufgrund der Ähnlichkeit zu der Kraft einer Feder hierbei von Spring Embeddern gesprochen.

Durch das Zusammenspiel dieser beiden Kräfte soll sich das System insgesamt ausgleichen, indem sich die Knoten zwar abstoßen, aber durch die an den Kanten wirkenden Kräfte im Rahmen gehalten werden. Somit befinden sich in Beziehung stehende Objekte näher beieinander als nicht zusammenhängende Objekte. Das System versucht bei korrekter Implementierung stets die Kräfte auf ein ausgeglichenes Niveau zu bringen, also dafür zu sorgen, dass die Kräfte, die auf die Knoten wirken, den Wert Null annehmen.

Neben der Standardimplementierung der Spring Embedder, die lediglich die beiden genannten Kräfte berücksichtigt, ist es auch denkbar, weitere Kräfte hinzuzufügen. Diese könnten beispielsweise für andere Gruppierungen unter den Objekten führen. Hier müsste zunächst analysiert werden, welche Gruppierungen einen Sinn ergeben.

Abbildung 21 zeigt, wie eine Implementierung einer dreidimensionalen Ansicht aussehen könnte.

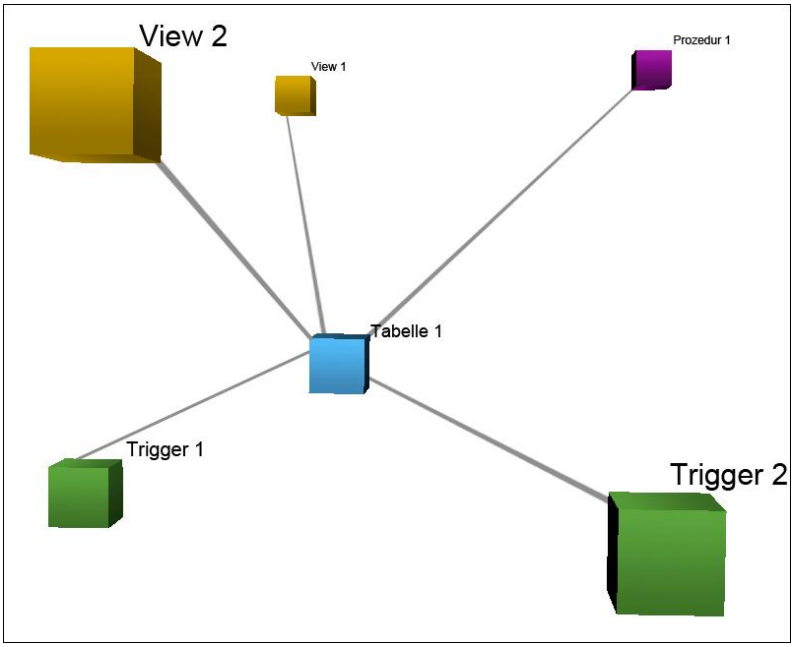

Abbildung 21: Mögliche Umsetzung der 3D-Ansicht

Der genannte Algorithmus könnte ebenso in der zweidimensionalen Ebene implementiert werden, bevor man schließlich eine weitere Dimension hinzufügt. Zwar werden dadurch nicht alle Überschneidungsprobleme gelöst, jedoch wird die Anzahl der Überschneidungen deutlich verringert und auch die Anordnung der Knoten wird verbessert. In Ansätzen hat diese Implementierung schon stattgefunden. Das verwendete Graphen-Framework „JUNG“ enthält bereits ein Layout namens „Spring Layout 2“, welches das Federprinzip grundsätzlich umsetzen sollte. Eine Einbindung in die Software zeigte jedoch, dass die Implementierung scheinbar nicht korrekt umgesetzt wurde bzw. für den speziellen Zweck nicht geeignet ist, da die wirkenden Kräfte sich nicht komplett ausgleichen, wodurch der Graph stetig in Bewegung ist. Erst eine Neu-Implementierung kann zeigen, ob sich das Kräftemodell für die Darstellung der Graphen eignet. Die Vorteile einer solchen Implementierung liegen jedenfalls klar auf der Hand:

Vorteile der Kräftemodelle[33]

- Durch die Flexibilität der Kräftemodell-Verfahren können die Algorithmen auf alle Typen von Graphen angewandt werden. Die Verfahren sind also nicht auf bestimmte Graphentypen spezialisiert.
- Es ist relativ einfach einen simplen Grundalgorithmus mit anziehenden und abstoßenden Kräften zu implementieren. Dieser muss dann natürlich noch an die spezifischen Eigenschaften der Graphen angepasst werden.
- Die Verfahren der Kräftemodelle erzeugen meist qualitativ hochwertige Zeichnungen, die Knoten sind gleichmäßig in Darstellungsraum verteilt und es werden Symmetrien angezeigt, da die meisten Berechnungsverfahren symmetrische Kräfteformeln verwenden.
- Da Eigenschaften von Objekten in die Berechnung einfließen können, ist es möglich, die Graphen an eigene Bedürfnisse anzupassen. Es wäre somit beispielsweise möglich, bestimmte Objekte näher beieinander zu gruppieren.

Wie bei fast jedem Verfahren, existieren auch hier Nachteile:

Nachteile der Kräftemodelle[32]

- Die Wahl geeigneter Eigenschaften bzw. Parameter für die Berechnungen der Kräfte ist nicht einfach. Um für jeden Graphen ein ähnlich gutes Ergebnis zu erzielen, müssten theoretisch die Parameter jedes Mal neu gewählt werden.

[33] Vgl. [NEU 06], S. 53f

- Es kommt vor, dass sich bei Auswahl vieler Eigenschaften zur Optimierung der Darstellung die Kräfte ausgleichen und somit keinen Erfolg bringen. Desweiteren wird die Laufzeit deutlich verlängert, je mehr Eigenschaften in die Berechnung einfließen.
- Die Laufzeit wird stark von der Auswahl des Verfahrens bestimmt. Verfahren mit einer geringen Laufzeit ergeben meist auch weniger gute Zeichnungen.
- Die Verfahren versuchen meist ein lokales Energieminimum zu erreichen, also einen Kräfteausgleich, was nicht immer eine optimale Zeichnung zur Folge hat. Hier sollten diese lokalen Minima unterbunden und ein globales Energieoptimum gefunden werden. Diese Optimierung beeinträchtigt jedoch stark die Laufzeit.

7. Anwendungsszenario

Ergänzend zu den Arbeitspaketen wurde im Rahmen der Bachelorarbeit ein Anwendungsszenario erstellt, welches sowohl als Überprüfung der implementierten Funktionalitäten als auch der Demonstration der Software dient. Das Anwendungsszenario wurde sowohl in einer Oracle- als auch in einer MySQL-Datenbank implementiert, um alle neuen Funktionalitäten abzudecken. Im folgenden Abschnitt wird jedoch lediglich auf die konkrete Implementierung unter Oracle eingegangen. Die MySQL-Version wurde entsprechend der spezifischen Syntax angepasst und enthält keine Prozeduren und Funktionen, weshalb nur die komplexere Oracle-Version beschrieben wird.

Zusätzlich zu dem angesprochenen Szenario soll außerdem untersucht werden, in wieweit die Software für größere Datenbanken bzw. -modelle geeignet ist bzw. an welchem Punkt sie an ihre Grenzen stößt, sowohl bei der Menge der Daten als auch bei der Übersichtlichkeit der Graphen. Was schon jetzt erkennbar ist, ist die Tatsache, dass bei stark vernetzten Datenbanken, die Übersichtlichkeit innerhalb der Graphen verloren geht. Hier können zukünftig bessere Graphenalgorithmen verwendet werden, die dafür sorgen, dass sich die dargestellten Datenbankobjekte möglichst nicht überschneiden[34].

7.1 Anwendungsszenario

Im folgenden Abschnitt wird zunächst das Szenario im groben Rahmen erklärt. Anschließend sollen die verschiedenen Ansichten getestet werden, um zu sehen, ob die Implementierung korrekt umgesetzt wurde.

Bei dem Anwendungsszenario handelt es sich um eine vereinfachte Verwaltungsdatenbank einer Software-Firma. Sie soll die firmeninternen Strukturen zur Bearbeitung von Projekten wiedergeben. Neben der Firmenhierarchie existiert eine Kundentabelle, welche alle Kunden der fiktiven Firma enthält. Jeder Kunde kann mehrere Projekte bei der Firma in Auftrag gegeben haben. Zusätzlich zu den aktuellen Projektdaten wird eine Historie geführt, die belegt, wann welcher Status im Projekt erreicht wurde. Jedes Projekt besteht aus Arbeitspaketen (packages), die wiederrum in Arbeitsschritte (tasks) unterteilt werden. Die Firmenhierarchie besteht aus Abteilungen mit jeweils einem Leiter. Jede Abteilung besteht wiederrum aus Teams mit jeweils einem Leiter. Eine weitere Tabelle erfasst die Mitarbeiter, die den Teams zugeordnet werden. Jeder Mitarbeiter kann in mehreren Teams

[34] Siehe Kapitel 6 Graphen

sein und mehrere Aufgaben aus Projekten übernehmen. Darüber hinaus besitzt jeder Mitarbeiter einen Benutzeraccount.

Die Umsetzung des Szenarios in einer Oracle-Datenbank kann im Anhang nachgelesen werden. Bei der Umsetzung ist bewusst keine optimale Lösung des Schemas gewählt worden, um aufzuzeigen, wie die Software Fehler im Datenmodell aufdecken kann. Beispielsweise wurde bewusst die „ON DELETE CASCADE“ Funktion bei Fremdschlüsseln missachtet. Stattdessen werden überflüssige Zeilen mit Triggern gelöscht, was einen Zyklus auslösen kann (wenn die Tabellen gegenseitig Zeilen löschen). Außerdem sind Trigger mit „Mutating Table“-Problemen erzeugt worden, also die Trigger greifen auf die eigene Tabelle zu. Zusätzlich wurde eine Prozedur programmiert, die möglichst viele Sonderfälle beinhaltet und auf viele Tabellen zugreift, um sehen zu können, ob der Parser für die Prozeduren und Funktionen korrekt funktioniert.
Zur Umsetzung des Szenarios wurden sowohl eine lokale Oracle-Express 10g, als auch eine MySQL5 Installation genutzt.

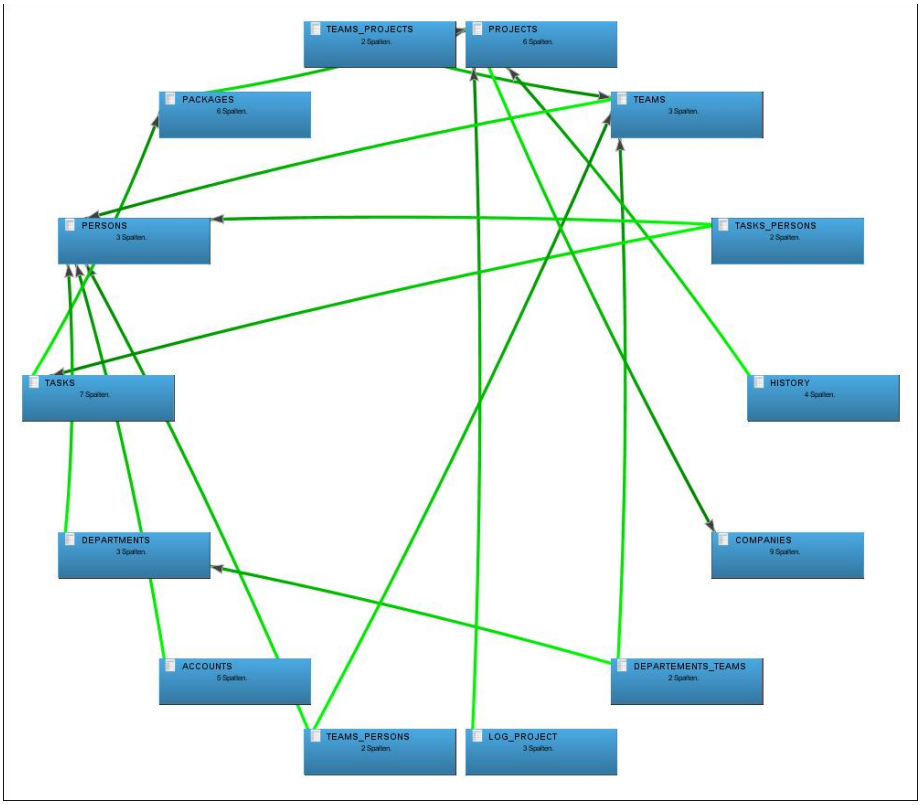

Abbildung 22: ER-Diagramm des Anwendungsszenario (Circle-Layout)

Abbildung 22 zeigt das Entity-Relationship-Diagramm des Anwendungsszenarios. Die Software löst alle Fremdschlüsselbeziehungen der Tabellen auf und stellt diese korrekt dar.

Jedoch ist sofort zu erkennen, dass die Darstellung noch deutliche Schwächen im Layout hat. Für die Ansicht wurde das derzeit beste in der Software vertretene Layout, das „Circle Layout“, genutzt, bei dem nicht darauf geachtet wird, dass es möglichst wenige Überschneidungen gibt. Der Graph wurde mittels Drag & Drop umgezeichnet, dies zeigt Abbildung 23. Wie zu sehen ist, handelt es sich um einen planaren also überschneidungsfrei zeichenbaren Graphen. Ein effizienterer auf einem Kräftemodell basierender Algorithmus könnte eine ähnliche vor allem planare Zeichnung ebenfalls automatisch erzeugen.

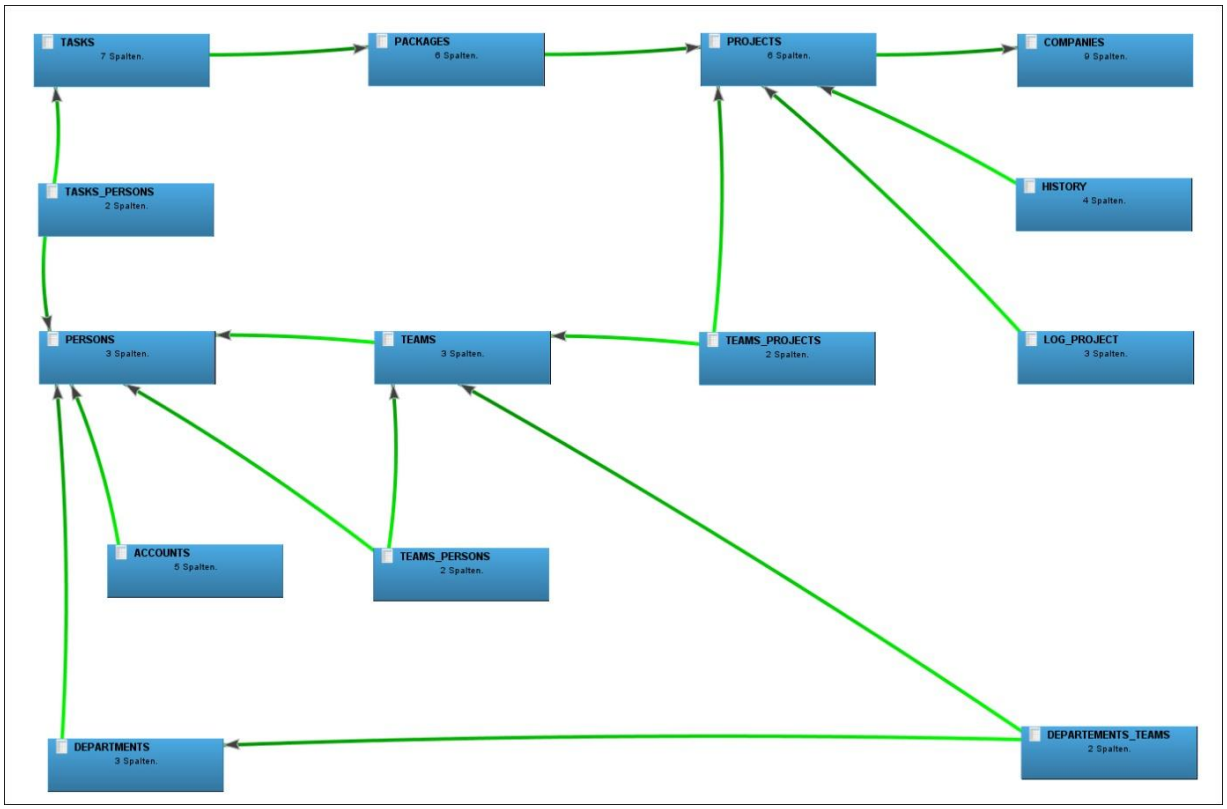

Abbildung 23: ER-Diagramm des Anwendungsszenarios (nach Drag & Drop)

Auch die View-Hierachie zeigt, dass eine Verbesserung der Graphenalgorithmen nötig ist, da auch hier einige Überschneidungen der Kanten vorhanden sind (siehe Abbildung 24).

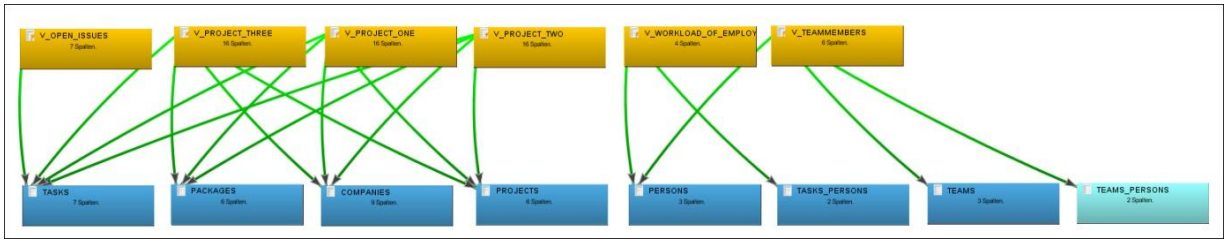

Abbildung 24: View-Hierarchie des Anwendungsszenarios

Die Notwendigkeit die Graphenansichten zu verbessern zeigt sich in den beiden Ansichten „Trigger-Ansicht“ und „Einzelansicht“ nicht so deutlich. Bei den Triggern ist dies darauf zurückzuführen, dass meist nicht so viele Trigger implementiert sind bzw. die vorhandenen Trigger nicht so viele Beziehungen untereinander besitzen.

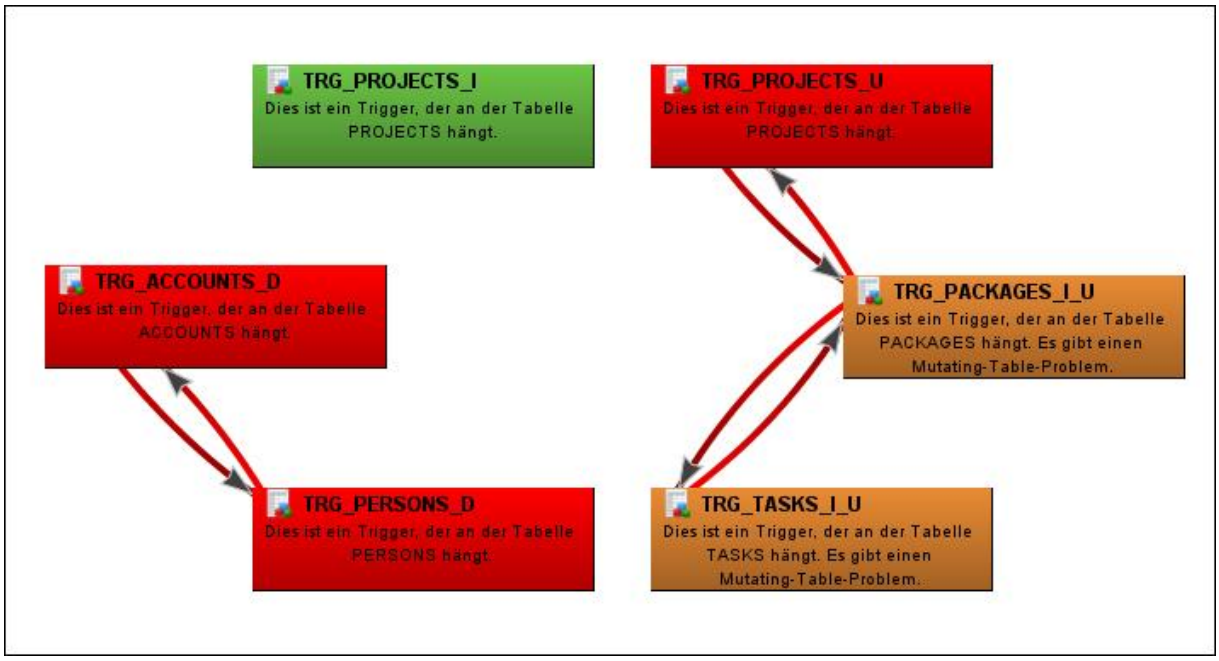

Abbildung 25: Trigger-Ansicht des Anwendungsszenarios

In Abbildung 25 ist gut zu erkennen, dass die „Mutating Table“-Probleme und die Zyklen bei den Triggern gut erkannt und dargestellt wurden.

Abbildung 26 zeigt die neu implementierte Einzelansicht, bei der die Tabellen im Zentrum der Betrachtung stehen sollen. Es werden für eine vom Nutzer gewählte Tabelle alle zugehörigen Objekte angezeigt. Da sich die Relationen auf einfache Beziehungen zwischen den Objekten und der Tabelle beschränken, ist es hierbei nicht zwingend notwendig, einen neuen Graphenalgorithmus zu implementieren. Denkbar wäre es jedoch, die Tabelle auch optisch in das Zentrum der Betrachtung zu rücken, also die Datenbankobjekte um die Tabelle herum zu gruppieren.

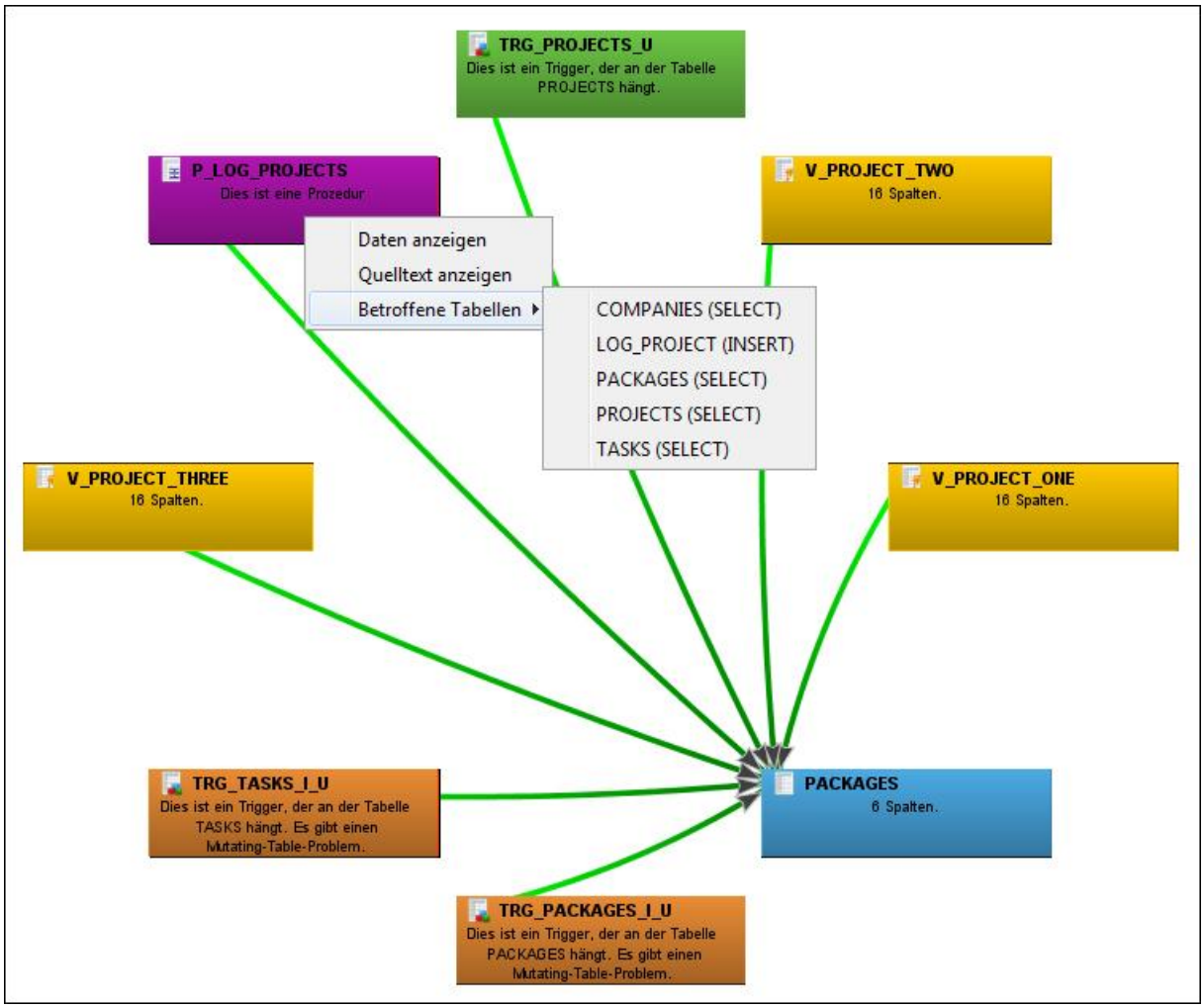

Abbildung 26: Die neue Einzelansicht in Einsatz beim Anwendungsszenario

Die Abbildung 26 zeigt darüber hinaus noch die neue Funktionalität, sich direkt im Diagramm anzeigen zu lassen, welche Tabelle von den Prozeduren, Funktionen oder Triggern betroffen sind und welche Aktionen an diesen Tabellen vorgenommen wurden. Hieran ist auch zu erkennen, dass der Prozedur-Parser auch mit komplizierteren Statements zurechtkommt.[35]

[35] siehe Anhang E: Prozedur **Fehler! Nur Hauptdokument**

7.2 Lasttest

Im Folgenden soll festgestellt werden, bis zu welcher Kapazität des Datenmodells die Software noch funktionsfähig bleibt bzw. bis zu welcher Anzahl an Datenbankobjekten ein sinnvolles Arbeiten noch möglich ist.
Hierzu ist angedacht, eine Vielzahl von Tabellen zu erzeugen, die jeweils eine ausgehende Beziehung zu einer anderen Tabelle besitzen. Hierzu wurde ein anonymer PL/SQL-Block konstruiert, mit dessen Hilfe die Tabellen angelegt werden können.

```
1  DECLARE
2  nametabelle VARCHAR2(20) := 'testtable';
3  BEGIN
4    FOR Lcntr IN 1..99
5    LOOP
6      EXECUTE IMMEDIATE 'CREATE TABLE ' || nametabelle || Lcntr || '
7      (id INT NOT NULL, CONSTRAINT testtable' || Lcntr || '_pk
8      PRIMARY KEY (id) ENABLE)';
9    END LOOP;
10 END;
```

Abbildung 27: Anonymer PL/SQL-Block zum automatischen Erzeugen von Tabellen

Um eine Einschätzung zu bekommen, ab welchem Bereich die Funktionalität der Software eingeschränkt wird, wurden die Tabellen in Blöcken von 100 Objekten angelegt.
Dabei war festzustellen, dass die Software die Tabellendaten erstaunlicherweise wenige Probleme mit großen Datenmengen hatte. Erst ab einer Anzahl von 800 Tabellen ließen sich leichte Performanceeinbußen erkennen. Dies machte sich jedoch lediglich bei den Graphen bemerkbar. Bei der „Drag & Drop“ - Funktionalität verzögerte die Ansicht leicht die Bewegungen des Nutzers. Ab einer Datenmenge von rund 1000 Tabellen war die Graphenansicht nur noch kaum nutzbar, da die Benutzereingaben sehr verzögert angenommen wurden. Der Verbindungsansicht bereitete die große Datenmenge keine Probleme, alle Tabellen wurden im Baum in üblicher Geschwindigkeit angezeigt und konnten schnell abgerufen werden.
Trotz der Möglichkeit eine solche Menge an Tabellen nutzen zu können, ist dies nicht praktikabel, da sich eine sinnvolle Darstellung im zweidimensionalen Raum aufgrund von mangelnden Platzverhältnissen als sehr schwierig bzw. unmöglich erweist.

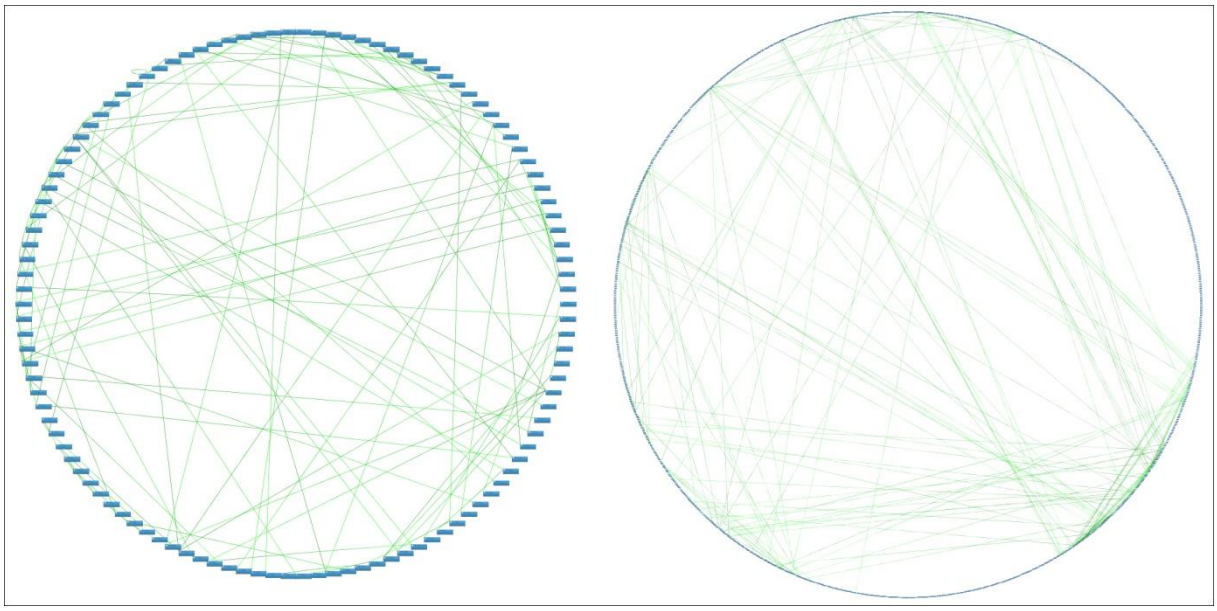

Abbildung 28: ER-Diagramm mit 100 (links) bzw. 1000 Tabellen (rechts)

Wie an Abbildung 28 deutlich zu erkennen ist, macht eine Darstellung von sehr vielen Datenbankobjekten in dieser Form wenig Sinn, da die Beziehungen nicht mehr erkennbar sind bzw. der Nutzer sich keine gute Übersicht mehr verschaffen kann.
Zusätzlich zu den Tabellen wurden im Anschluss Views erzeugt. Dies hat den Grund, dass diese beim Laden der Verbindung durch einen Parser geprüft werden und somit dort zu etwaigen Performanceengpässen führen könnten. Auch hier wurde ein anonymer PL/SQL-Block zur Erzeugung der Datenbankobjekte genutzt.

```
1  DECLARE
2  nametabelle VARCHAR2(20) := 'testview';
3  BEGIN
4    FOR Lcntr IN 1..99
5    LOOP
6      EXECUTE IMMEDIATE 'CREATE VIEW ' || nametabelle || Lcntr || '
7      AS SELECT * FROM persons';
8    END LOOP;
9  END;
```

Abbildung 29: Anonymer PL/SQL-Block zum automatischen Erzeugen von Views

Der Test zeigte auch hier, dass die Software an sich eine große Datenmenge aufnehmen kann. Lediglich das Laden der Verbindung dauerte einige Momente länger, da jede View vom Parser analysiert werden musste. Jedoch tritt auch hier erneut das Problem der unübersichtlichen Darstellung der Graphen auf.

8. Fazit

In diesem abschließenden Kapitel soll noch einmal der Erkenntnisstand dieser Arbeit zusammengefasst und ein Ausblick auf zusätzliche Erweiterungsmöglichkeiten gestellt werden.

8.1 Zusammenfassung

Die vorliegende Arbeit beschäftigt sich vorwiegend mit der Weiterentwicklung einer im Rahmen der Diplomarbeit „Visualisierung der Abhängigkeiten von Datenbankobjekten“[36] von Andre Kasper und Jan Philipp aus dem Jahr 2009 entstandenen Softwarelösung zur grafischen Veranschaulichung der komplexen Beziehungen von Objekten innerhalb von Datenbanksystemen. Schon die zu Grunde liegende Diplomarbeit zeigt einige Punkte auf, an denen eine Weiterentwicklung möglich wäre. Hierzu zählt neben der Erweiterung der Software um zusätzliche Datenbanksysteme (Bisher existiert nur eine Unterstützung von Oracle-Datenbanken) auch die Verbesserung bzw. Erweiterung der grafischen Darstellungen. Nach ausgiebiger Recherche ergab sich zunächst die Anforderung, die Software um das weit verbreitete Open Source Datenbanksystem MySQL zu erweitern. Hierzu wurde zunächst eine ausgiebige Analyse der Unterschiede zwischen der schon implementierten Oracle-Datenbank und dem MySQL-System erstellt. Aus dieser Analyse ergab sich, dass die Einbindung der aktuellen Version 5 von MySQL aufgrund der erfolgten Weiterentwicklung der Datenbank möglich geworden ist, da in der Version 5 erstmals ein dem Oracle Data Dictionary ähnelndes Metadaten-Verzeichnis implementiert wurde. Nachdem die grundlegenden Anforderungen an die Erweiterung erfüllt waren, wurde ein Konzept entwickelt, wie die neue Datenbank zu integrieren ist. Hierzu wurde die bestehende Systemarchitektur untersucht und die Schnittstellen festgelegt, an denen die Erweiterung ansetzen soll. Im Anschluss daran wurde der Weiterentwicklungsschritt umgesetzt und in groben Zügen in dieser Arbeit beschrieben.

Desweiteren wurde festgestellt, dass die ebenfalls wichtigen Datenbankobjekte „Prozeduren und Funktionen“ nicht in der Software vertreten sind, obwohl sie, genauso wie die Trigger, viele Beziehungen zu anderen Objekten beinhalten können. In einem weiteren Entwicklungsschritt wurde demnach die Software um die Anzeige der Prozeduren und Funktionen erweitert. Ebenfalls wurde daraufhin festgestellt, dass die drei Ansichten zwar alle bisher implementierten Datenbankobjekte anzeigen, jedoch gibt es keinen Graphen, der

[36] [KP 09]

alle Datenbankobjekte enthält und miteinander in Beziehung setzt. Zur Umsetzung dieser Anforderung wurde eine weitere Ansicht hinzugefügt, die darüber hinaus auch die neuen Objekte „Prozedur“ und „Funktion“ enthält.

Weiterführend wurden zusätzlich zu den zwei genannten Arbeitspaketen einige Fehler in der Software behoben. Hierzu zählt unter anderem die Weiterentwicklung der Routine zum Parsen der Trigger-Definitionen, welche zuvor nicht alle Abhängigkeiten zu Tabellen oder Views lieferte.

Zusätzlich zu den Erweiterungen der Software ergab sich beim Umgang mit dem Programm die grundsätzliche Anforderung, die Graphenansichten zu verbessern, da diese keine optimalen Ergebnisse liefern. Da eine Implementierung den Rahmen dieser Arbeit übersteigen würde, beschränkt sich die Ausarbeitung lediglich auf die Analyse der vorhandenen Graphen und eine Erläuterung der möglichen Verbesserungen. Schnell wurde auch hier klar, dass es sich bei dem Thema „Graphen“ um ein sehr komplexes Thema handelt, weshalb diese Arbeit nur eine grobe Analyse bietet.

Im Anschluss an die Implementierungen und Analysen wurde zusätzlich noch ein Anwendungsszenario entworfen, welches als Test der Funktionalitäten der Software dienen und außerdem aufzeigen sollte, bis zu welcher Datenbankgröße bzw. Menge von Datenbankobjekten, die Software handhabbar bleibt.

8.2 Ausblick

Schlussendlich wurde die Software zwar stark weiterentwickelt, jedoch gerade aufgrund der schnell voranschreitenden Entwicklung der Technik ergeben sich immer wieder neue Möglichkeiten und Ideen zur Verbesserung der Software. Dieser letzte Abschnitt soll noch einmal aufzeigen, an welchen Stellen die Software noch Entwicklungsmöglichkeiten offen hält.

Zunächst ist zu sagen, dass natürlicherweise nicht alle verfügbaren Datenbanksysteme integriert wurden. Daher ist es immer möglich, weitere Systeme wie beispielsweise die Datenbank DB2 von IBM, den MS SQL Server oder auch PostgreSQL in die Software einzubinden. Hierfür muss jedoch für jedes einzubindende Datenbanksystem zuvor eine Analyse erstellt werden, ob eine Anbindung sinnvoll möglich ist.

Backus-Naur-Form

Zur möglichen Performance-Steigerung und vor allem zum Zweck der Fehlerminimierung könnten in Zukunft sowohl die Oracle-Parser für Views, Trigger und Prozeduren/Funktionen als auch die MySQL-Implementierungen selbiger mit Hilfe der Backus-Naur-Form umgesetzt werden.
Zunächst soll diesbezüglich erklärt werden, worum es sich bei der Backus-Naur-Form überhaupt handelt:

Die Backus-Naur-Form (auch Backusnormalform, kurz BNF, genannt) ist eine formale Metasprache, die zur Darstellung kontextfreier Grammatiken dient.[37] Hauptsächlich wird die Backus-Naur-Form zur Beschreibung der Syntax höherer Programmiersprachen verwendet. Der Quelltext dieser Programmiersprachen besteht aus Terminalsymbolen, dies können Buchstaben, Ziffern, Leerzeichen oder auch Satzzeichen sein. Um nun diesen Quelltext formal beschreiben zu können, werden Produktionsregeln gebildet, die einzelnen Symbolen bzw. meist Symbolfolgen Nichtterminalsymbole zuordnen. Somit können Ketten von Symbolen gebildet werden, die den Quelltext beschreiben, indem Regeln aneinander gehängt werden.

Beispiel für die formale Beschreibung einer 2-stelligen Zahl mittels der Backus-Naur-Form:

<Ziffer ohne Null> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<Ziffer> ::= 0 | <Ziffer ohne Null>
<Zweistellige Zahl> ::= <Ziffer ohne Null> <Ziffer>

Wie zu erkennen ist, so wird die 2-stellige Zahl aus mehreren Produktionsregeln zusammen gesetzt. Nichtterminalsymbole werden in spitzen Klammern geschrieben, eine Zuordnung von Symbolfolgen zur Nichtterminalsymbolen erfolgt mit den Zeichen „::=“. Der senkrechte Strich symbolisiert eine „oder“-Trennung für Alternativen.

Konkret auf das Projekt bezogen, würde dies bedeuten, dass die mögliche Syntax der Definitionen der Trigger und Views mittels der Backus-Naur-Form formal beschrieben wird. Diese Beschreibung soll schließlich dazu dienen, mittels eines Parser-Generators wie beispielsweise JavaCup automatisch einen Parser zur Analyse der Definitionen zu erzeugen.

Betrachtet man die Backus-Naur-Form genauer, so fällt schnell auf, dass diese Form der formalen Beschreibung der Syntax schnell an ihre Grenzen stößt. Beispielsweise existieren

[37] Vgl. [ITW 10]

keine Regeln, um einen optionalen Term oder eine Wiederholung einer bestimmten Zeichenfolge darzustellen. Abhilfe schafft hier die erweiterte Backus-Naur-Form, die von der Internationalen Organisation für Normung (ISO) standardisiert wurde und alle benötigten Funktionalitäten enthält, um die Definitionen von Triggern und Views formal ausreichend zu beschreiben.

8.3 Schlusswort

Abschließend ist zu sagen, dass es noch sehr viele Möglichkeiten und Ideen gibt, um die entstandene Software zu erweitern, jedoch schafft der bisherige Stand der Software schon ein sehr hilfreiches Werkzeug, mit dessen Hilfe nicht nur die Fremdschlüsselabhängigkeiten mit einem Entity-Relationship-Diagramm, sondern auch weitere Abhängigkeiten von Datenbankobjekten untereinander dargestellt werden können. In Anbetracht der schnellen Entwicklung der Technik, wird es immer wichtiger durch gute Werkzeuge bei der Arbeit unterstützt zu werden. Gerade der stetig wachsende Bedarf an Datenspeicherungen erfordert optimierte Datenbanksysteme bzw. Datenmodelle. Die Software stellt ein gutes Fundament zur Hilfe bei der Optimierung dar, ist jedoch noch ausbaufähig. Besonders die Verbesserung der Graphenalgorithmen ist hierbei wichtig, da gerade bei komplexeren Datenmodellen die Darstellung aufgrund von vielen Überschneidungen und mangelndem Platz im zweidimensionalen Raum leidet. Ansätze, wie die Nutzung einer weiteren Dimension, wurden im Kapitel „Graphenanalysen“ angeschnitten[38]. Zusätzlich hierzu sind auch andere Erweiterungen denkbar, die durch die flexible und demnach zukunftssichere Struktur der Software problemlos implementiert werden könnten. Welche nützlichen Erweiterungen das sein werden ist noch offen und richtet sich nach den zukünftigen Ideen und Anforderungen.

[38] siehe Kapitel 6 Arbeitspaket 3 – Graphenanalysen

9. Literaturverzeichnis

[BM 04] J. M. Boyer, W. J. Myrvold
„On the Cutting Edge: Simplified *O*(*n*) Planarity by Edge Addition“, 2004
Journal of Graph Algorithms and Applications
http://jgaa.info/ Vol. 8, Nr. 3, S. 241‑273
http://jgaa.info/accepted/2004/BoyerMyrvold2004.8.3.pdf
im Internet abgerufen im September 2010

[DBS 07] H. Faeskorn-Woyke, B. Bertelsmeier, P. Riemer, E. Bauer
„Datenbanksysteme – Theorie und Praxis mit SQL2003, Oracle und MySQL“
Pearson Studium, 2007

[ITW 10] Autor unbekannt
„Artikel über die Backus-Naur-Form“
http://www.itwissen.info/definition/lexikon/Backus-Naur-Form-BNF.html
im Internet abgerufen im August 2010

[KP 09] A. Kasper, J. Phillip
Visualisierung der Abhängigkeiten von Datenbankobjekten, August 2009
http://drop.io/visualdependencies/asset/da-2009-philipp-kasper-v1-1-pdf
im Internet abgerufen im August 2009

[KRE 05] Krempel, Lothar
„Visualisierung komplexer Strukturen: Grundlagen der Darstellung mehrdimensionaler Netzwerke“, August 2005
Campus Verlag, 1. Sonderband Auflage, S.103 – 118

[NEU 06] K. Neubauer
„Informationsvisualisierung - Entwicklung eines Tools zur expressiven und effektiven Darstellung komplexer, relationaler Daten“, Juni 2006
http://theses.fh-hagenberg.at/system/files/pdf/Neubauer06.pdf
im Internet abgerufen im September 2010

[SON 09] A. Bluhm, J. Eickmeyer, T. Feith, N. Mattar, und T. Pfeiffer
„Exploration von sozialen Netzwerken im 3D Raum am Beispiel von SONAR für Last.fm“, 2009
http://www.techfak.uni-bielefeld.de/~tpfeiffe/pubs/2009_-_Bluhm_et_al_SONAR.pdf
im Internet abgerufen im September 2010

A. Anhang – Tabellen des Anwendungsszenarios

```
1  CREATE TABLE COMPANIES (
2    COMPANY_ID INT NOT NULL,
3    COMPANY_CONTACT_LASTNAME VARCHAR2(100) NOT NULL,
4    COMPANY_NAME VARCHAR2(100) NOT NULL,
5    COMPANY_CONTACT_NAME VARCHAR2(100) NOT NULL,
6    COMPANY_CONTACT_MAIL VARCHAR2(100) NOT NULL,
7    COMPANY_STREET VARCHAR2(100),
8    COMPANY_HOUSE_NUMBER VARCHAR2(10),
9    COMPANY_POSTCODE VARCHAR2(10),
10   COMPANY_CITY VARCHAR2(50),
11   CONSTRAINT COMPANIES_PK PRIMARY KEY (COMPANY_ID) ENABLE
12 );
13 ALTER TABLE COMPANIES
14   ADD CONSTRAINT COMPANIES_UK1 UNIQUE (COMPANY_NAME) ENABLE;
```

Tabelle 1: „COMPANIES“ – Enthält die Kundendaten

```
1  CREATE TABLE PERSONS (
2    PERSON_ID INT NOT NULL,
3    PERSON_LASTNAME VARCHAR2(50) NOT NULL,
4    PERSON_NAME VARCHAR2(50) NOT NULL,
5    CONSTRAINT PERSONS_PK PRIMARY KEY (PERSON_ID) ENABLE
6  );
```

Tabelle 2: „PERSONS“ – Enthält die Mitarbeiterdaten

```
1  CREATE TABLE ACCOUNTS (
2    ACCOUNT_ID INT NOT NULL,
3    ACCOUNT_PERSON_ID INT NOT NULL,
4    ACCOUNT_USERNAME VARCHAR2(20) NOT NULL,
5    ACCOUNT_PASSWORD VARCHAR2(32) NOT NULL,
6    ACCOUNT_CREATION_DATE DATE NOT NULL,
7    CONSTRAINT ACCOUNTS_PK PRIMARY KEY (ACCOUNT_ID) ENABLE
8  );
9  ALTER TABLE ACCOUNTS
10   ADD CONSTRAINT ACCOUNTS_PERSONS_FK1 FOREIGN KEY
11 (ACCOUNT_PERSON_ID)
12   REFERENCES PERSONS (PERSON_ID) ON DELETE SET NULL ENABLE;
```

Tabelle 3: „ACCOUNTS“ – Enthält die Benutzerdaten der Mitarbeiter

```
CREATE TABLE TEAMS (
  TEAM_ID INT NOT NULL,
  TEAM_NAME VARCHAR2(30) NOT NULL,
  TEAM_LEADER INT NOT NULL,
  CONSTRAINT TEAMS_PK PRIMARY KEY (TEAM_ID) ENABLE
);
ALTER TABLE TEAMS
  ADD CONSTRAINT TEAMS_UK1 UNIQUE (TEAM_NAME) ENABLE;
ALTER TABLE TEAMS
  ADD CONSTRAINT TEAMS_PERSONS_FK1 FOREIGN KEY (TEAM_LEADER)
  REFERENCES PERSONS (PERSON_ID) ON DELETE SET NULL ENABLE;
```

Tabelle 4: „TEAMS“ – Enthält die Teamdaten

```
CREATE TABLE DEPARTMENTS (
  DEPARTMENT_ID INT NOT NULL,
  DEPARTMENT_NAME VARCHAR2(30) NOT NULL,
  DEPARTMENT_LEADER INT NOT NULL,
  CONSTRAINT DEPARTMENTS_PK PRIMARY KEY (DEPARTMENT_ID) ENABLE
);
ALTER TABLE DEPARTMENTS
  ADD CONSTRAINT DEPARTMENTS_UK1 UNIQUE (DEPARTMENT_NAME) ENABLE;
ALTER TABLE DEPARTMENTS
  ADD CONSTRAINT DEPARTMENTS_PERSONS_FK1 FOREIGN KEY
(DEPARTMENT_LEADER)
  REFERENCES PERSONS (PERSON_ID) ON DELETE SET NULL ENABLE;
```

Tabelle 5: „DEPARTMENTS“ – Enthält die Daten der Abteilungen

```
CREATE TABLE DEPARTEMENTS_TEAMS (
  DEPARTMENT_ID INT NOT NULL,
  TEAM_ID INT NOT NULL,
  CONSTRAINT DEPARTEMENTS_TEAMS_PK PRIMARY KEY
    (DEPARTMENT_ID, TEAM_ID) ENABLE
);
ALTER TABLE DEPARTEMENTS_TEAMS
  ADD CONSTRAINT DEPARTEMENTS_TEAMS_TEAMS_FK1 FOREIGN KEY (TEAM_ID)
  REFERENCES TEAMS (TEAM_ID) ON DELETE CASCADE ENABLE;
ALTER TABLE DEPARTEMENTS_TEAMS
  ADD CONSTRAINT DEPARTEMENTS_TEAMS_DEPART_FK1 FOREIGN KEY
    (DEPARTMENT_ID)
  REFERENCES DEPARTMENTS (DEPARTMENT_ID) ON DELETE CASCADE ENABLE;
```

Tabelle 6: „DEPARTMENTS_TEAMS“ – Zuordnung Teams zu Abteilungen

```
CREATE TABLE TEAMS_PERSONS (
  TEAM_ID INT NOT NULL,
  PERSON_ID INT NOT NULL,
  CONSTRAINT TEAMS_PERSONS_PK PRIMARY KEY (TEAM_ID, PERSON_ID)
  ENABLE
);
ALTER TABLE TEAMS_PERSONS
  ADD CONSTRAINT TEAMS_PERSONS_TEAMS_FK1 FOREIGN KEY (TEAM_ID)
  REFERENCES TEAMS (TEAM_ID) ON DELETE CASCADE ENABLE;
ALTER TABLE TEAMS_PERSONS
  ADD CONSTRAINT TEAMS_PERSONS_PERSONS_FK1 FOREIGN KEY (PERSON_ID)
  REFERENCES PERSONS (PERSON_ID) ON DELETE CASCADE ENABLE;
```

Tabelle 7: „TEAMS_PERSONS“ – Zuordnung Mitarbeiter zu Teams

```
CREATE TABLE PROJECTS (
  PROJECT_ID INT NOT NULL,
  PROJECT_NAME VARCHAR2(50) NOT NULL,
  PROJECT_COMPANY_ID INT NOT NULL,
  PROJECT_DESCRIPTION VARCHAR2(255),
  PROJECT_STATE INT DEFAULT 0 NOT NULL,
  PROJECT_LAST_UPDATE DATE NOT NULL,
  CONSTRAINT PROJECTS_PK PRIMARY KEY (PROJECT_ID) ENABLE
);
ALTER TABLE PROJECTS
  ADD CONSTRAINT PROJECTS_COMPANIES_FK1 FOREIGN KEY
(PROJECT_COMPANY_ID)
  REFERENCES COMPANIES (COMPANY_ID) ON DELETE CASCADE ENABLE;
```

Tabelle 8: „PROJECTS“ – Enthält die Projektdaten

```
CREATE TABLE TEAMS_PROJECTS (
  TEAM_ID INT NOT NULL,
  PROJECT_ID INT NOT NULL,
  CONSTRAINT TEAMS_PROJECTS_PK PRIMARY KEY (TEAM_ID, PROJECT_ID)
  ENABLE
);
ALTER TABLE TEAMS_PROJECTS
  ADD CONSTRAINT TEAMS_PROJECTS_PROJECTS_FK1
  FOREIGN KEY (PROJECT_ID)
  REFERENCES PROJECTS (PROJECT_ID) ON DELETE CASCADE ENABLE;
ALTER TABLE TEAMS_PROJECTS
  ADD CONSTRAINT TEAMS_PROJECTS_TEAMS_FK1 FOREIGN KEY (TEAM_ID)
  REFERENCES TEAMS (TEAM_ID) ON DELETE CASCADE ENABLE;
```

Tabelle 9: „TEAMS_PROJECTS“ – Zuordnung Teams zu Projekten

```
CREATE TABLE HISTORY (
  HISTORY_ID INT NOT NULL,
      HISTORY_PROJECT_ID INT NOT NULL,
      HISTORY_STATE INT DEFAULT 0 NOT NULL,
      HISTORY_DATE DATE NOT NULL,
      CONSTRAINT HISTORY_PK PRIMARY KEY (HISTORY_ID) ENABLE
);
ALTER TABLE HISTORY
  ADD CONSTRAINT HISTORY_PROJECTS_FK1 FOREIGN KEY
(HISTORY_PROJECT_ID)
  REFERENCES PROJECTS (PROJECT_ID) ON DELETE CASCADE ENABLE;
CREATE SEQUENCE SEQ_HISTORY INCREMENT BY 1 START WITH 1;
```

Tabelle 10: „HISTORY“ – Enthält die Verlaufsdaten der Projekte

```
CREATE TABLE PACKAGES (
  PACKAGE_ID INT NOT NULL,
  PACKAGE_NAME VARCHAR2(50) NOT NULL,
  PACKAGE_PROJECT_ID INT NOT NULL,
  PACKAGE_DESCRIPTION VARCHAR2(255),
  PACKAGE_STATE INT DEFAULT 0 NOT NULL,
  PACKAGE_LAST_UPDATE VARCHAR2(20),
  CONSTRAINT PACKAGES_PK PRIMARY KEY (PACKAGE_ID) ENABLE
);
ALTER TABLE PACKAGES
  ADD CONSTRAINT PACKAGES_PROJECTS_FK1 FOREIGN KEY
(PACKAGE_PROJECT_ID)
  REFERENCES PROJECTS (PROJECT_ID) ON DELETE CASCADE ENABLE;
```

Tabelle 11: „PACKAGES“ – Enthält die Arbeitspakete der Projekte

```
CREATE TABLE TASKS (
  TASK_ID INT NOT NULL,
  TASK_NAME VARCHAR2(50) NOT NULL,
  TASK_PACKAGE_ID INT NOT NULL,
  TASK_DESCRIPTION VARCHAR2(255),
  TASK_STATE INT DEFAULT 0 NOT NULL,
  TASK_CREATION_DATE DATE NOT NULL,
  TASK_LAST_UPDATE DATE NOT NULL,
  CONSTRAINT TASKS_PK PRIMARY KEY (TASK_ID) ENABLE
);
ALTER TABLE TASKS
  ADD CONSTRAINT TASKS_PACKAGES_FK1 FOREIGN KEY (TASK_PACKAGE_ID)
  REFERENCES PACKAGES (PACKAGE_ID) ON DELETE CASCADE ENABLE;
```

Tabelle 12: „PERSONS“ – Enthält die Arbeitsschritte der Arbeitspakete

```
1  CREATE TABLE TASKS_PERSONS (
2    TASK_ID INT NOT NULL,
3    PERSON_ID INT NOT NULL,
4    CONSTRAINT TASKS_PERSONS_PK PRIMARY KEY (TASK_ID, PERSON_ID)
5    ENABLE
6  );
7  ALTER TABLE TASKS_PERSONS
8    ADD CONSTRAINT TASKS_PERSONS_TASKS_FK1 FOREIGN KEY (TASK_ID)
9    REFERENCES TASKS (TASK_ID) ENABLE;
10 ALTER TABLE TASKS_PERSONS
11   ADD CONSTRAINT TASKS_PERSONS_PERSONS_FK1 FOREIGN KEY (PERSON_ID)
12   REFERENCES PERSONS (PERSON_ID) ENABLE;
```

Tabelle 13: „PERSONS“ – Enthält die Zuordnung Mitarbeiter zu Arbeitsschritten

```
1  CREATE TABLE LOG_PROJECT (
2    LOG_ID INT NOT NULL,
3    LOG_PROJECT_ID INT NOT NULL,
4    LOG_WARNING VARCHAR2(50) NOT NULL,
5    CONSTRAINT LOG_PROJECT_PK PRIMARY KEY (LOG_ID) ENABLE
6  );
7  ALTER TABLE LOG_PROJECT
8    ADD CONSTRAINT LOG_PROJECT_PROJECTS_FK1 FOREIGN KEY
9  (LOG_PROJECT_ID)
10   REFERENCES PROJECTS (PROJECT_ID) ENABLE;
11 CREATE SEQUENCE SEQ_LOG_PROJECT INCREMENT BY 1 START WITH 1;
```

Tabelle 14: „LOG_PROJECT“ – Enthält Fehlermeldungen über die Tabelle „PROCJECTS“

B. Anhang – Inhalt der Tabellen

```
INSERT INTO COMPANIES (COMPANY_ID, COMPANY_CONTACT_LASTNAME,
 COMPANY_NAME, COMPANY_CONTACT_NAME, COMPANY_CONTACT_MAIL,
 COMPANY_STREET, COMPANY_HOUSE_NUMBER, COMPANY_POSTCODE,
 COMPANY_CITY) VALUES ('1', 'Mustermann', 'Autohaus Mustermann',
 'Max', 'max.mustermann@autohaus-mustermann.de', 'Musterstraße',
 '7', '55473', 'Musterhausen');
INSERT INTO COMPANIES (COMPANY_ID, COMPANY_CONTACT_LASTNAME,
 COMPANY_NAME, COMPANY_CONTACT_NAME, COMPANY_CONTACT_MAIL,
 COMPANY_STREET, COMPANY_HOUSE_NUMBER, COMPANY_POSTCODE,
 COMPANY_CITY) VALUES ('2', 'Hansen', 'Hansen Versicherungen',
 'Harmut', 'harmut.hansen@hansen.de', 'Hansaweg', '8', '90237',
 'Hanshausen');
INSERT INTO COMPANIES (COMPANY_ID, COMPANY_CONTACT_LASTNAME,
 COMPANY_NAME, COMPANY_CONTACT_NAME, COMPANY_CONTACT_MAIL,
 COMPANY_STREET, COMPANY_HOUSE_NUMBER, COMPANY_POSTCODE,
 COMPANY_CITY) VALUES ('3', 'Meier', 'Bäckerei Meier', 'Marianne',
 'marianne.meier@baeckerei-meier.de', 'Hauptstraße', '92', '98327',
 'Backstadt');

INSERT INTO PERSONS (PERSON_ID, PERSON_LASTNAME, PERSON_NAME)
 VALUES ('1', 'Kastleiner', 'Marc');
INSERT INTO PERSONS (PERSON_ID, PERSON_LASTNAME, PERSON_NAME)
 VALUES ('2', 'Müller', 'Anne');
INSERT INTO PERSONS (PERSON_ID, PERSON_LASTNAME, PERSON_NAME)
 VALUES ('3', 'Werner', 'Heinrich');
INSERT INTO PERSONS (PERSON_ID, PERSON_LASTNAME, PERSON_NAME)
 VALUES ('4', 'Pahl', 'Christian');
INSERT INTO PERSONS (PERSON_ID, PERSON_LASTNAME, PERSON_NAME)
 VALUES ('5', 'Doppstadt', 'David');
INSERT INTO PERSONS (PERSON_ID, PERSON_LASTNAME, PERSON_NAME)
 VALUES ('6', 'Schönfeld', 'Gerrit');
INSERT INTO PERSONS (PERSON_ID, PERSON_LASTNAME, PERSON_NAME)
 VALUES ('7', 'Stachowski', 'Vitali');
INSERT INTO PERSONS (PERSON_ID, PERSON_LASTNAME, PERSON_NAME)
 VALUES ('8', 'Morkens', 'Julia');


INSERT INTO PROJECTS (PROJECT_ID, PROJECT_NAME, PROJECT_COMPANY_ID,
 PROJECT_DESCRIPTION, PROJECT_STATE, PROJECT_LAST_UPDATE)
 VALUES ('1', 'Automobile Bestandsverwaltung', '1', 'Es soll eine
 Webapplikation entwickelt werden, worüber die Automobilbestände
 verwaltet werden können.', '0', TO_DATE('17.09.10', 'DD.MM.RR'));
INSERT INTO PROJECTS (PROJECT_ID, PROJECT_NAME, PROJECT_COMPANY_ID,
 PROJECT_DESCRIPTION, PROJECT_STATE, PROJECT_LAST_UPDATE)
 VALUES ('2', 'Kundendatenbank', '2', 'Eine Desktop-Anwendung zur
```

```
 Verwaltung der Kundendaten', '0', TO_DATE('07.09.10', 'DD.MM.RR'));

INSERT INTO PROJECTS (PROJECT_ID, PROJECT_NAME, PROJECT_COMPANY_ID,
 PROJECT_DESCRIPTION, PROJECT_STATE, PROJECT_LAST_UPDATE) VALUES
 ('3', 'Webauftritt', '3', 'Es soll eine neue Homepage mit
 Bewertungsportal entworfen werden.', '0', TO_DATE('14.09.10',
 'DD.MM.RR'));

INSERT INTO ACCOUNTS (ACCOUNT_ID, ACCOUNT_PERSON_ID,
 ACCOUNT_USERNAME, ACCOUNT_PASSWORD, ACCOUNT_CREATION_DATE) VALUES
 ('1', '1', 'marc.kastleiner', '12345', TO_DATE('06.09.2005',
 'DD.MM.RR'));
INSERT INTO ACCOUNTS (ACCOUNT_ID, ACCOUNT_PERSON_ID,
 ACCOUNT_USERNAME, ACCOUNT_PASSWORD, ACCOUNT_CREATION_DATE)
 VALUES ('2', '2', 'anne.mueller', '12345', TO_DATE('06.09.2005',
 'DD.MM.RR'));
INSERT INTO ACCOUNTS (ACCOUNT_ID, ACCOUNT_PERSON_ID,
 ACCOUNT_USERNAME, ACCOUNT_PASSWORD, ACCOUNT_CREATION_DATE)
 VALUES ('3', '3', 'heinrich.werner', '12345', TO_DATE('06.09.2005',
 'DD.MM.RR'));
INSERT INTO ACCOUNTS (ACCOUNT_ID, ACCOUNT_PERSON_ID,
 ACCOUNT_USERNAME, ACCOUNT_PASSWORD, ACCOUNT_CREATION_DATE)
 VALUES ('4', '4', 'christian.pahl', '12345', TO_DATE('06.09.2005',
 'DD.MM.RR'));
INSERT INTO ACCOUNTS (ACCOUNT_ID, ACCOUNT_PERSON_ID,
 ACCOUNT_USERNAME, ACCOUNT_PASSWORD, ACCOUNT_CREATION_DATE)
 VALUES ('5', '5', 'david.doppstadt', '12345', TO_DATE('06.09.2005',
 'DD.MM.RR'));
INSERT INTO ACCOUNTS (ACCOUNT_ID, ACCOUNT_PERSON_ID,
 ACCOUNT_USERNAME, ACCOUNT_PASSWORD, ACCOUNT_CREATION_DATE)
 VALUES ('6', '6', 'gerrit.schoenfeld', '12345',
 TO_DATE('06.09.2005',
 'DD.MM.RR'));
INSERT INTO ACCOUNTS (ACCOUNT_ID, ACCOUNT_PERSON_ID,
 ACCOUNT_USERNAME, ACCOUNT_PASSWORD, ACCOUNT_CREATION_DATE)
 VALUES ('7', '7', 'vitali.stachowski', '12345',
 TO_DATE('06.09.2005', 'DD.MM.RR'));
INSERT INTO ACCOUNTS (ACCOUNT_ID, ACCOUNT_PERSON_ID,
 ACCOUNT_USERNAME, ACCOUNT_PASSWORD, ACCOUNT_CREATION_DATE)
 VALUES ('8', '8', 'julia.morkens', '12345', TO_DATE('06.09.2005',
 'DD.MM.RR'));

INSERT INTO DEPARTMENTS (DEPARTMENT_ID, DEPARTMENT_NAME,
 DEPARTMENT_LEADER) VALUES (1, 'Designabteilung', 8);
INSERT INTO DEPARTMENTS (DEPARTMENT_ID, DEPARTMENT_NAME,
 DEPARTMENT_LEADER) VALUES (2, 'Softwareentwicklung', 1);

```

```
93  INSERT INTO TEAMS (TEAM_ID, TEAM_NAME, TEAM_LEADER)
94   VALUES ('1', 'Planung Software', '2');
95  INSERT INTO TEAMS (TEAM_ID, TEAM_NAME, TEAM_LEADER)
96   VALUES ('2', 'Planung Design', '8');
97  INSERT INTO TEAMS (TEAM_ID, TEAM_NAME, TEAM_LEADER)
98   VALUES ('3', 'Umsetzung Software', '1');
99  INSERT INTO TEAMS (TEAM_ID, TEAM_NAME, TEAM_LEADER)
100  VALUES ('4', 'Umsetzung Datenbanken', '3');
101 INSERT INTO TEAMS (TEAM_ID, TEAM_NAME, TEAM_LEADER)
102  VALUES ('5', 'Umsetzung Design', '8');
103
104 INSERT INTO DEPARTEMENTS_TEAMS (DEPARTMENT_ID, TEAM_ID)
105  VALUES ('2', '1');
106 INSERT INTO DEPARTEMENTS_TEAMS (DEPARTMENT_ID, TEAM_ID)
107  VALUES ('1', '2');
108 INSERT INTO DEPARTEMENTS_TEAMS (DEPARTMENT_ID, TEAM_ID)
109  VALUES ('2', '3');
110 INSERT INTO DEPARTEMENTS_TEAMS (DEPARTMENT_ID, TEAM_ID)
111  VALUES ('2', '4');
112 INSERT INTO DEPARTEMENTS_TEAMS (DEPARTMENT_ID, TEAM_ID)
113  VALUES ('1', '5');
114
115 INSERT INTO TEAMS_PERSONS (TEAM_ID, PERSON_ID) VALUES ('1', '2');
116 INSERT INTO TEAMS_PERSONS (TEAM_ID, PERSON_ID) VALUES ('1', '3');
117 INSERT INTO TEAMS_PERSONS (TEAM_ID, PERSON_ID) VALUES ('1', '1');
118 INSERT INTO TEAMS_PERSONS (TEAM_ID, PERSON_ID) VALUES ('2', '8');
119 INSERT INTO TEAMS_PERSONS (TEAM_ID, PERSON_ID) VALUES ('2', '7');
120 INSERT INTO TEAMS_PERSONS (TEAM_ID, PERSON_ID) VALUES ('5', '8');
121 INSERT INTO TEAMS_PERSONS (TEAM_ID, PERSON_ID) VALUES ('5', '7');
122 INSERT INTO TEAMS_PERSONS (TEAM_ID, PERSON_ID) VALUES ('4', '3');
123 INSERT INTO TEAMS_PERSONS (TEAM_ID, PERSON_ID) VALUES ('4', '4');
124 INSERT INTO TEAMS_PERSONS (TEAM_ID, PERSON_ID) VALUES ('3', '1');
125 INSERT INTO TEAMS_PERSONS (TEAM_ID, PERSON_ID) VALUES ('3', '5');
126 INSERT INTO TEAMS_PERSONS (TEAM_ID, PERSON_ID) VALUES ('3', '4');
127 INSERT INTO TEAMS_PERSONS (TEAM_ID, PERSON_ID) VALUES ('3', '6');
128
129 INSERT INTO TEAMS_PROJECTS (PROJECT_ID, TEAM_ID) VALUES ('1', '1');
130 INSERT INTO TEAMS_PROJECTS (PROJECT_ID, TEAM_ID) VALUES ('1', '2');
131 INSERT INTO TEAMS_PROJECTS (PROJECT_ID, TEAM_ID) VALUES ('1', '3');
132 INSERT INTO TEAMS_PROJECTS (PROJECT_ID, TEAM_ID) VALUES ('1', '4');
133 INSERT INTO TEAMS_PROJECTS (PROJECT_ID, TEAM_ID) VALUES ('1', '5');
134 INSERT INTO TEAMS_PROJECTS (PROJECT_ID, TEAM_ID) VALUES ('2', '1');
135 INSERT INTO TEAMS_PROJECTS (PROJECT_ID, TEAM_ID) VALUES ('2', '2');
136 INSERT INTO TEAMS_PROJECTS (PROJECT_ID, TEAM_ID) VALUES ('2', '3');
137 INSERT INTO TEAMS_PROJECTS (PROJECT_ID, TEAM_ID) VALUES ('2', '4');
138 INSERT INTO TEAMS_PROJECTS (PROJECT_ID, TEAM_ID) VALUES ('2', '5');
139 INSERT INTO TEAMS_PROJECTS (PROJECT_ID, TEAM_ID) VALUES ('3', '1');
```

```
INSERT INTO TEAMS_PROJECTS (PROJECT_ID, TEAM_ID) VALUES ('3', '2');
INSERT INTO TEAMS_PROJECTS (PROJECT_ID, TEAM_ID) VALUES ('3', '3');
INSERT INTO TEAMS_PROJECTS (PROJECT_ID, TEAM_ID) VALUES ('3', '5');

INSERT INTO PACKAGES (PACKAGE_ID, PACKAGE_NAME, PACKAGE_PROJECT_ID,
 PACKAGE_DESCRIPTION, PACKAGE_STATE, PACKAGE_LAST_UPDATE)
 VALUES ('1', 'Planung und Umsetzung Designkonzept', '2', '-', '3',
 '16.09.2010');
INSERT INTO PACKAGES (PACKAGE_ID, PACKAGE_NAME, PACKAGE_PROJECT_ID,
 PACKAGE_DESCRIPTION, PACKAGE_STATE, PACKAGE_LAST_UPDATE)
 VALUES ('2', 'Planung Struktur der Software', '2', '-', '2',
 '14.09.2010');
INSERT INTO PACKAGES (PACKAGE_ID, PACKAGE_NAME, PACKAGE_PROJECT_ID,
PACKAGE_DESCRIPTION, PACKAGE_STATE, PACKAGE_LAST_UPDATE)
 VALUES ('3', 'Planung der Datenbank', '2', '-', '2', '13.09.2010');
INSERT INTO PACKAGES (PACKAGE_ID, PACKAGE_NAME, PACKAGE_PROJECT_ID,
 PACKAGE_DESCRIPTION, PACKAGE_STATE, PACKAGE_LAST_UPDATE)
 VALUES ('4', 'Entwurf der Designvorlagen', '2', '-', '1',
 '15.09.2010');
INSERT INTO PACKAGES (PACKAGE_ID, PACKAGE_NAME, PACKAGE_PROJECT_ID,
 PACKAGE_DESCRIPTION, PACKAGE_STATE, PACKAGE_LAST_UPDATE)
 VALUES ('5', 'Umsetzung der Benutzeransicht', '2', '-', '0',
 '14.09.2010');
INSERT INTO PACKAGES (PACKAGE_ID, PACKAGE_NAME, PACKAGE_PROJECT_ID,
 PACKAGE_DESCRIPTION, PACKAGE_STATE, PACKAGE_LAST_UPDATE)
 VALUES ('6', 'Umsetzung des Datenbankkonzeptes', '2', '-', '0',
 '14.09.2010');
INSERT INTO PACKAGES (PACKAGE_ID, PACKAGE_NAME, PACKAGE_PROJECT_ID,
 PACKAGE_DESCRIPTION, PACKAGE_STATE, PACKAGE_LAST_UPDATE)
 VALUES ('7', 'Umsetzung der Geschäftslogik', '2', '-', '0',
 '14.09.2010');
INSERT INTO PACKAGES (PACKAGE_ID, PACKAGE_NAME, PACKAGE_PROJECT_ID,
 PACKAGE_DESCRIPTION, PACKAGE_STATE, PACKAGE_LAST_UPDATE)
 VALUES ('8', 'Abschließende Tests', '2', '-', '0', '14.09.2010');
INSERT INTO PACKAGES (PACKAGE_ID, PACKAGE_NAME, PACKAGE_PROJECT_ID,
 PACKAGE_DESCRIPTION, PACKAGE_STATE, PACKAGE_LAST_UPDATE)
 VALUES ('9', 'Planung', '1', '-', '0', '17.09.2010');
INSERT INTO PACKAGES (PACKAGE_ID, PACKAGE_NAME, PACKAGE_PROJECT_ID,
 PACKAGE_DESCRIPTION, PACKAGE_STATE, PACKAGE_LAST_UPDATE)
 VALUES ('10', 'Planung', '3', '-', '0', '17.09.2010');
INSERT INTO PACKAGES (PACKAGE_ID, PACKAGE_NAME, PACKAGE_PROJECT_ID,
 PACKAGE_DESCRIPTION, PACKAGE_STATE, PACKAGE_LAST_UPDATE)
 VALUES ('11', 'Umsetzung', '1', '-', '0', '17.09.2010');
INSERT INTO PACKAGES (PACKAGE_ID, PACKAGE_NAME, PACKAGE_PROJECT_ID,
 PACKAGE_DESCRIPTION, PACKAGE_STATE, PACKAGE_LAST_UPDATE)
 VALUES ('12', 'Umsetzung', '3', '-', '0', '17.09.2010');
INSERT INTO PACKAGES (PACKAGE_ID, PACKAGE_NAME, PACKAGE_PROJECT_ID,
```

```
 PACKAGE_DESCRIPTION, PACKAGE_STATE, PACKAGE_LAST_UPDATE)
 VALUES ('13', 'Test', '3', '-', '0', '17.09.2010');

INSERT INTO TASKS (TASK_ID, TASK_NAME, TASK_PACKAGE_ID,
 TASK_DESCRIPTION, TASK_STATE, TASK_CREATION_DATE, TASK_LAST_UPDATE)
 VALUES ('1', 'Skizzen für neues Logo', '1', '-', '5',
 TO_DATE('10.09.2010', 'DD.MM.RR'), TO_DATE('17.09.2010',
 'DD.MM.RR'));
INSERT INTO TASKS (TASK_ID, TASK_NAME, TASK_PACKAGE_ID,
 TASK_DESCRIPTION, TASK_STATE, TASK_CREATION_DATE, TASK_LAST_UPDATE)
 VALUES ('2', 'Entscheidung Farbmodell', '1', '-', '5',
 TO_DATE('10.09.2010', 'DD.MM.RR'), TO_DATE('17.09.2010',
 'DD.MM.RR'));
INSERT INTO TASKS (TASK_ID, TASK_NAME, TASK_PACKAGE_ID,
 TASK_DESCRIPTION, TASK_STATE, TASK_CREATION_DATE, TASK_LAST_UPDATE)
 VALUES ('3', 'Grundaufbau der Benutzeroberfläche', '1', '-', '3',
 TO_DATE('10.09.2010', 'DD.MM.RR'), TO_DATE('17.09.2010',
 'DD.MM.RR'));
INSERT INTO TASKS (TASK_ID, TASK_NAME, TASK_PACKAGE_ID,
 TASK_DESCRIPTION, TASK_STATE, TASK_CREATION_DATE, TASK_LAST_UPDATE)
 VALUES ('4', 'Festlegung der Technologie (Abschätzung)', '2', '-',
 '4', TO_DATE('10.09.2010', 'DD.MM.RR'), TO_DATE('17.09.2010',
 'DD.MM.RR'));
INSERT INTO TASKS (TASK_ID, TASK_NAME, TASK_PACKAGE_ID,
 TASK_DESCRIPTION, TASK_STATE, TASK_CREATION_DATE, TASK_LAST_UPDATE)
 VALUES ('5', 'Festlegung des Datenbanktyps', '3', '-', '5',
 TO_DATE('10.09.2010', 'DD.MM.RR'), TO_DATE('17.09.2010',
 'DD.MM.RR'));
INSERT INTO TASKS (TASK_ID, TASK_NAME, TASK_PACKAGE_ID,
 TASK_DESCRIPTION, TASK_STATE, TASK_CREATION_DATE, TASK_LAST_UPDATE)
 VALUES ('6', 'Entwurf der Softwarestruktur (UML)', '2', '-', '2',
 TO_DATE('10.09.2010', 'DD.MM.RR'), TO_DATE('17.09.2010',
 'DD.MM.RR'));
INSERT INTO TASKS (TASK_ID, TASK_NAME, TASK_PACKAGE_ID,
 TASK_DESCRIPTION, TASK_STATE, TASK_CREATION_DATE, TASK_LAST_UPDATE)
 VALUES ('7', 'Entwurf des ER-Diagramms', '3', '-', '3',
 TO_DATE('10.09.2010', 'DD.MM.RR'), TO_DATE('17.09.2010',
 'DD.MM.RR'));
INSERT INTO TASKS (TASK_ID, TASK_NAME, TASK_PACKAGE_ID,
 TASK_DESCRIPTION, TASK_STATE, TASK_CREATION_DATE, TASK_LAST_UPDATE)
 VALUES ('8', 'Umsetzung des grundsätzlichen Designaufbaus', '4',
 '-', '4', TO_DATE('10.09.2010', 'DD.MM.RR'), TO_DATE('17.09.2010',
 'DD.MM.RR'));
INSERT INTO TASKS (TASK_ID, TASK_NAME, TASK_PACKAGE_ID,
 TASK_DESCRIPTION, TASK_STATE, TASK_CREATION_DATE, TASK_LAST_UPDATE)
 VALUES ('9', 'Verbesserung des Designmodells', '4', '-', '2',
 TO_DATE('10.09.2010', 'DD.MM.RR'), TO_DATE('17.09.2010',
```

```
 'DD.MM.RR'));

INSERT INTO TASKS (TASK_ID, TASK_NAME, TASK_PACKAGE_ID,
 TASK_DESCRIPTION, TASK_STATE, TASK_CREATION_DATE, TASK_LAST_UPDATE)
 VALUES ('10', 'Entwurf der Hauptansicht', '5', '-', '2',
 TO_DATE('10.09.2010', 'DD.MM.RR'), TO_DATE('17.09.2010',
 'DD.MM.RR'));
INSERT INTO TASKS (TASK_ID, TASK_NAME, TASK_PACKAGE_ID,
 TASK_DESCRIPTION, TASK_STATE, TASK_CREATION_DATE, TASK_LAST_UPDATE)
 VALUES ('11', 'Entwurf der Tabellenanzeige', '5', '-', '1',
 TO_DATE('10.09.2010', 'DD.MM.RR'), TO_DATE('17.09.2010',
 'DD.MM.RR'));
INSERT INTO TASKS (TASK_ID, TASK_NAME, TASK_PACKAGE_ID,
 TASK_DESCRIPTION, TASK_STATE, TASK_CREATION_DATE, TASK_LAST_UPDATE)
 VALUES ('12', 'Anlegen der Tabellensturktur', '6', '-', '1',
 TO_DATE('10.09.2010', 'DD.MM.RR'), TO_DATE('17.09.2010',
 'DD.MM.RR'));
INSERT INTO TASKS (TASK_ID, TASK_NAME, TASK_PACKAGE_ID,
 TASK_DESCRIPTION, TASK_STATE, TASK_CREATION_DATE, TASK_LAST_UPDATE)
 VALUES ('13', 'Anlegen der Trigger, Prozeduren und Views', '6',
 '-', '1', TO_DATE('10.09.2010', 'DD.MM.RR'), TO_DATE('17.09.2010',
 'DD.MM.RR'));
INSERT INTO TASKS (TASK_ID, TASK_NAME, TASK_PACKAGE_ID,
 TASK_DESCRIPTION, TASK_STATE, TASK_CREATION_DATE, TASK_LAST_UPDATE)
 VALUES ('14', 'Integration der Funktion "neuen Kunden anlegen"',
 '7', '-', '0', TO_DATE('10.09.2010', 'DD.MM.RR'),
 TO_DATE('17.09.2010', 'DD.MM.RR'));
INSERT INTO TASKS (TASK_ID, TASK_NAME, TASK_PACKAGE_ID,
 TASK_DESCRIPTION, TASK_STATE, TASK_CREATION_DATE, TASK_LAST_UPDATE)
 VALUES ('15', 'Umsetzung der Funktion "neues Versicherungsmodell"',
 '7', '-', '0', TO_DATE('10.09.2010', 'DD.MM.RR'),
 TO_DATE('17.09.2010', 'DD.MM.RR'));
INSERT INTO TASKS (TASK_ID, TASK_NAME, TASK_PACKAGE_ID,
 TASK_DESCRIPTION, TASK_STATE, TASK_CREATION_DATE, TASK_LAST_UPDATE)
 VALUES ('16', 'Umsetzung Berarbeitungsfunktion', '7', '-', '0',
 TO_DATE('10.09.2010', 'DD.MM.RR'), TO_DATE('17.09.2010',
 'DD.MM.RR'));
 INSERT INTO TASKS (TASK_ID, TASK_NAME, TASK_PACKAGE_ID,
 TASK_DESCRIPTION, TASK_STATE, TASK_CREATION_DATE, TASK_LAST_UPDATE)
 VALUES ('17', 'Umsetzung Fkt "Zuordnung Kunde - Versicherung"',
 '7', '-', '0', TO_DATE('10.09.2010', 'DD.MM.RR'),
 TO_DATE('17.09.2010', 'DD.MM.RR'));
INSERT INTO TASKS (TASK_ID, TASK_NAME, TASK_PACKAGE_ID,
 TASK_DESCRIPTION, TASK_STATE, TASK_CREATION_DATE, TASK_LAST_UPDATE)
 VALUES ('18', 'Umsetzung Menüstruktur', '7', '-', '0',
 TO_DATE('10.09.2010', 'DD.MM.RR'), TO_DATE('17.09.2010',
 'DD.MM.RR'));
```

```
281 INSERT INTO TASKS (TASK_ID, TASK_NAME, TASK_PACKAGE_ID,
282  TASK_DESCRIPTION, TASK_STATE, TASK_CREATION_DATE, TASK_LAST_UPDATE)
283  VALUES ('19', 'Test des Datenbankschemas', '8', '-', '0',
284  TO_DATE('10.09.2010', 'DD.MM.RR'), TO_DATE('17.09.2010',
285  'DD.MM.RR'));
286 INSERT INTO TASKS (TASK_ID, TASK_NAME, TASK_PACKAGE_ID,
287  TASK_DESCRIPTION, TASK_STATE, TASK_CREATION_DATE, TASK_LAST_UPDATE)
288  VALUES ('20', 'Test der Software', '8', '-', '0',
289  TO_DATE('10.09.2010', 'DD.MM.RR'), TO_DATE('17.09.2010',
290  'DD.MM.RR'));
291 INSERT INTO TASKS (TASK_ID, TASK_NAME, TASK_PACKAGE_ID,
292  TASK_DESCRIPTION, TASK_STATE, TASK_CREATION_DATE, TASK_LAST_UPDATE)
293  VALUES ('21', 'Grundkozept Design', '9', '-', '0',
294  TO_DATE('17.09.10', 'DD.MM.RR'), TO_DATE('17.09.10', 'DD.MM.RR'));
295 INSERT INTO TASKS (TASK_ID, TASK_NAME, TASK_PACKAGE_ID,
296  TASK_DESCRIPTION, TASK_STATE, TASK_CREATION_DATE, TASK_LAST_UPDATE)
297  VALUES ('22', 'Grundkozept Design', '10', '-', '0',
298  TO_DATE('17.09.10', 'DD.MM.RR'), TO_DATE('17.09.10', 'DD.MM.RR'));
299 INSERT INTO TASKS (TASK_ID, TASK_NAME, TASK_PACKAGE_ID,
300  TASK_DESCRIPTION, TASK_STATE, TASK_CREATION_DATE, TASK_LAST_UPDATE)
301  VALUES ('23', 'Grundkonzept Software', '9', '-', '0',
302  TO_DATE('17.09.10', 'DD.MM.RR'), TO_DATE('17.09.10', 'DD.MM.RR'));
303 INSERT INTO TASKS (TASK_ID, TASK_NAME, TASK_PACKAGE_ID,
304  TASK_DESCRIPTION, TASK_STATE, TASK_CREATION_DATE, TASK_LAST_UPDATE)
305  VALUES ('24', 'Grundkozept Design', '10', '-', '0',
306  TO_DATE('17.09.10', 'DD.MM.RR'), TO_DATE('17.09.10', 'DD.MM.RR'));
307 INSERT INTO TASKS (TASK_ID, TASK_NAME, TASK_PACKAGE_ID,
308  TASK_DESCRIPTION, TASK_STATE, TASK_CREATION_DATE, TASK_LAST_UPDATE)
309  VALUES ('25', 'Entwicklung Frontend', '11', '-', '0',
310  TO_DATE('17.09.10', 'DD.MM.RR'), TO_DATE('17.09.10', 'DD.MM.RR'));
311 INSERT INTO TASKS (TASK_ID, TASK_NAME, TASK_PACKAGE_ID,
312  TASK_DESCRIPTION, TASK_STATE, TASK_CREATION_DATE, TASK_LAST_UPDATE)
313  VALUES ('26', 'Entwicklung Backend', '11', '-', '0',
314  TO_DATE('17.09.10', 'DD.MM.RR'), TO_DATE('17.09.10', 'DD.MM.RR'));
315 INSERT INTO TASKS (TASK_ID, TASK_NAME, TASK_PACKAGE_ID,
316  TASK_DESCRIPTION, TASK_STATE, TASK_CREATION_DATE, TASK_LAST_UPDATE)
317  VALUES ('27', 'Entwicklung Webseite', '12', '-', '0',
318  TO_DATE('17.09.10', 'DD.MM.RR'), TO_DATE('17.09.10', 'DD.MM.RR'));
319 INSERT INTO TASKS (TASK_ID, TASK_NAME, TASK_PACKAGE_ID,
320  TASK_DESCRIPTION, TASK_STATE, TASK_CREATION_DATE, TASK_LAST_UPDATE)
321  VALUES ('28', 'Installation Conten Management System', '12', '-',
322  '0', TO_DATE('17.09.10', 'DD.MM.RR'), TO_DATE('17.09.10',
323  'DD.MM.RR'));
324 INSERT INTO TASKS (TASK_ID, TASK_NAME, TASK_PACKAGE_ID,
325  TASK_DESCRIPTION, TASK_STATE, TASK_CREATION_DATE, TASK_LAST_UPDATE)
326  VALUES ('29', 'Test Frontend', '13', '-', '0', TO_DATE('17.09.10',
327  'DD.MM.RR'), TO_DATE('17.09.10', 'DD.MM.RR'));
```

```
328 INSERT INTO TASKS (TASK_ID, TASK_NAME, TASK_PACKAGE_ID,
329  TASK_DESCRIPTION, TASK_STATE, TASK_CREATION_DATE, TASK_LAST_UPDATE)
330  VALUES ('30', 'Test Backend', '13', '-', '0', TO_DATE('17.09.10',
331  'DD.MM.RR'), TO_DATE('17.09.10', 'DD.MM.RR'));
332
333 INSERT INTO TASKS_PERSONS (TASK_ID, PERSON_ID) VALUES ('1', '8');
334 INSERT INTO TASKS_PERSONS (TASK_ID, PERSON_ID) VALUES ('2', '7');
335 INSERT INTO TASKS_PERSONS (TASK_ID, PERSON_ID) VALUES ('2', '8');
336 INSERT INTO TASKS_PERSONS (TASK_ID, PERSON_ID) VALUES ('3', '8');
337 INSERT INTO TASKS_PERSONS (TASK_ID, PERSON_ID) VALUES ('3', '7');
338 INSERT INTO TASKS_PERSONS (TASK_ID, PERSON_ID) VALUES ('4', '1');
339 INSERT INTO TASKS_PERSONS (TASK_ID, PERSON_ID) VALUES ('4', '2');
340 INSERT INTO TASKS_PERSONS (TASK_ID, PERSON_ID) VALUES ('4', '3');
341 INSERT INTO TASKS_PERSONS (TASK_ID, PERSON_ID) VALUES ('5', '1');
342 INSERT INTO TASKS_PERSONS (TASK_ID, PERSON_ID) VALUES ('5', '3');
343 INSERT INTO TASKS_PERSONS (TASK_ID, PERSON_ID) VALUES ('6', '1');
344 INSERT INTO TASKS_PERSONS (TASK_ID, PERSON_ID) VALUES ('6', '2');
345 INSERT INTO TASKS_PERSONS (TASK_ID, PERSON_ID) VALUES ('7', '3');
346 INSERT INTO TASKS_PERSONS (TASK_ID, PERSON_ID) VALUES ('8', '7');
347 INSERT INTO TASKS_PERSONS (TASK_ID, PERSON_ID) VALUES ('9', '7');
348 INSERT INTO TASKS_PERSONS (TASK_ID, PERSON_ID) VALUES ('9', '8');
349 INSERT INTO TASKS_PERSONS (TASK_ID, PERSON_ID) VALUES ('10', '5');
350 INSERT INTO TASKS_PERSONS (TASK_ID, PERSON_ID) VALUES ('11', '4');
351 INSERT INTO TASKS_PERSONS (TASK_ID, PERSON_ID) VALUES ('12', '4');
352 INSERT INTO TASKS_PERSONS (TASK_ID, PERSON_ID) VALUES ('13', '3');
353 INSERT INTO TASKS_PERSONS (TASK_ID, PERSON_ID) VALUES ('14', '5');
354 INSERT INTO TASKS_PERSONS (TASK_ID, PERSON_ID) VALUES ('15', '6');
355 INSERT INTO TASKS_PERSONS (TASK_ID, PERSON_ID) VALUES ('16', '1');
356 INSERT INTO TASKS_PERSONS (TASK_ID, PERSON_ID) VALUES ('17', '1');
357 INSERT INTO TASKS_PERSONS (TASK_ID, PERSON_ID) VALUES ('18', '4');
358 INSERT INTO TASKS_PERSONS (TASK_ID, PERSON_ID) VALUES ('19', '3');
359 INSERT INTO TASKS_PERSONS (TASK_ID, PERSON_ID) VALUES ('19', '4');
360 INSERT INTO TASKS_PERSONS (TASK_ID, PERSON_ID) VALUES ('20', '1');
361 INSERT INTO TASKS_PERSONS (TASK_ID, PERSON_ID) VALUES ('20', '5');
362 INSERT INTO TASKS_PERSONS (TASK_ID, PERSON_ID) VALUES ('20', '4');
```

Inhalt der Tabellen 1: Die Testdaten für das Anwendungsszenario

C. Anhang – Views des Anwendungsszenarios

```
1  CREATE VIEW v_teammembers AS
2    SELECT t.team_id, t.team_name, p.person_id,
3      p.person_name, p.person_lastname,
4      is_teamleader(p.person_lastname, p.person_name, t.team_id) AS
5      teamleader
6    FROM teams t, teams_persons tp, persons p
7    WHERE t.team_id = tp.team_id
8    AND tp.person_id = p.person_id
9    ORDER BY t.team_id, teamleader;
```

View 2: „V_TEAMMEMBERS“ – Enthält die Mitglieder der Teams und gibt Auskunft über den Leiter

```
1  CREATE VIEW v_project_one AS
2    SELECT pr.project_id, pr.project_name, c.company_name,
3      pr.project_description, state_to_string(pr.project_state) AS
4      project_state, pr.project_last_update,
5      p.package_id, p.package_name, p.package_description,
6      state_to_string(p.package_state) AS package_state,
7      p.package_last_update,
8      t.task_id, t.task_name, t.task_description,
9      state_to_string(t.task_state) AS task_state, t.task_last_update
10   FROM projects pr, packages p, tasks t, companies c
11   WHERE pr.project_id = 1
12   AND pr.project_company_id = c.company_id
13   AND pr.project_id = p.package_project_id
14   AND p.package_id = t.task_package_id;
```

View 3: „V_PROJECT_ONE“ – Enthält alle Arbeitsschritte des ersten Projekts

```
1  CREATE VIEW v_project_two AS
2    SELECT pr.project_id, pr.project_name, c.company_name,
3      pr.project_description, state_to_string(pr.project_state) AS
4      project_state, pr.project_last_update,
5      p.package_id, p.package_name, p.package_description,
6      state_to_string(p.package_state) AS package_state,
7      p.package_last_update,
8      t.task_id, t.task_name, t.task_description,
9      state_to_string(t.task_state) AS task_state, t.task_last_update
10   FROM projects pr, packages p, tasks t, companies c
11   WHERE pr.project_id = 2
12   AND pr.project_company_id = c.company_id
13   AND pr.project_id = p.package_project_id
14   AND p.package_id = t.task_package_id;
```

View 4: „V_PROJECT_TWO“ – Enthält alle Arbeitsschritte des zweiten Projekts

```
1  CREATE VIEW v_project_three AS
2  SELECT pr.project_id, pr.project_name, c.company_name,
3      pr.project_description, state_to_string(pr.project_state) AS
4      project_state, pr.project_last_update,
5      p.package_id, p.package_name, p.package_description,
6      state_to_string(p.package_state) AS package_state,
7      p.package_last_update,
8      t.task_id, t.task_name, t.task_description,
9      state_to_string(t.task_state) AS task_state, t.task_last_update
10   FROM projects pr, packages p, tasks t, companies c
11   WHERE pr.project_id = 3
12   AND pr.project_company_id = c.company_id
13   AND pr.project_id = p.package_project_id
14   AND p.package_id = t.task_package_id;
```

View 5: „V_PROJECT_THREE“ – Enthält alle Arbeitsschritte des dritten Projekts

```
1  CREATE VIEW v_open_issues AS
2    SELECT * FROM tasks
3    WHERE lower(state_to_string(task_state)) = 'offen';
```

View 6: „V_OPEN_ISSUES“ – Enthält alle offenen Arbeitsschritte

```
1  CREATE VIEW v_workload_of_employees AS
2    SELECT p.person_id, p.person_name, p.person_lastname, COUNT(*) AS
3      workload
4    FROM tasks_persons tp LEFT JOIN persons p
5    ON tp.person_id = p.person_id
6    GROUP BY p.person_id, p.person_name, p.person_lastname
7    ORDER BY p.person_lastname, p.person_name;
```

View 7: „V_WORKLOAD_OF_EMPLOYEES“ –Auslastung der Mitarbeiter (Anzahl zugeordneter Arbeitsschritte)

D. Anhang – Trigger des Anwendungsszenarios

```
1  CREATE OR REPLACE
2  TRIGGER TRG_TASKS_I_U
3  AFTER INSERT OR UPDATE ON TASKS
4  FOR EACH ROW
5  DECLARE
6    CURSOR counter IS
7      SELECT COUNT(*) FROM TASKS
8      WHERE task_state != 5;
9    num_other_states INTEGER;
10 BEGIN
11   OPEN counter;
12   FETCH counter INTO num_other_states;
13   IF (num_other_states = 0) THEN
14     UPDATE packages SET package_state = 5 WHERE package_id =
15     :NEW.task_package_id;
16   END IF;
17 END;
```

Trigger 1: „TRG_TASKS_I_U“

```
1  CREATE OR REPLACE
2  TRIGGER TRG_PACKAGES_I_U
3  AFTER INSERT OR UPDATE ON PACKAGES
4  FOR EACH ROW
5  DECLARE
6    CURSOR counter IS
7      SELECT COUNT(*) FROM PACKAGES
8      WHERE package_project_id = :NEW.package_project_id
9      AND package_state != 5;
10   num_other_states INTEGER;
11 BEGIN
12   OPEN counter;
13   FETCH counter INTO num_other_states;
14   IF (num_other_states = 0) THEN
15     UPDATE projects SET project_state = 5 WHERE project_id =
16     :NEW.package_project_id;
17   END IF;
18
19   IF UPDATING THEN
20     IF (:NEW.package_state = 5) THEN
21       UPDATE tasks SET task_state = 5 WHERE task_package_id =
22 :NEW.package_id;
23     END IF;
24   END IF;
25 END;
```

Trigger 2: „TRG_PACKAGES_I_U “

```
1  CREATE OR REPLACE
2  TRIGGER TRG_PROJECTS_I
3  AFTER INSERT ON PROJECTS
4  FOR EACH ROW
5  BEGIN
6    INSERT INTO history (history_id, history_project_id, history_date)
7    VALUES (SEQ_HISTORY.nextval, :NEW.project_id, SYSDATE());
8  END;
```

Trigger 3: „TRG_PROJECTS_I “

```
1  CREATE OR REPLACE
2  TRIGGER TRG_PROJECTS_U
3  AFTER UPDATE ON PROJECTS
4  FOR EACH ROW
5  BEGIN
6    INSERT INTO history (history_id, history_project_id, history_date)
7    VALUES (SEQ_HISTORY.nextval, :NEW.project_id, SYSDATE());
8    IF (:NEW.project_state = 5) THEN
9      UPDATE packages SET package_state = 5 WHERE package_project_id =
10     :NEW.project_id;
11   END IF;
12 END;
```

Trigger 4: „TRG_PROJECTS_U “

```
1  CREATE OR REPLACE
2  TRIGGER TRG_PERSONS_D
3  AFTER DELETE ON PERSONS
4  FOR EACH ROW
5  BEGIN
6    DELETE FROM accounts WHERE account_person_id = :OLD.person_id;
7  END;
```

Trigger 5: „TRG_PERSONS_D “

```
1  CREATE OR REPLACE
2  TRIGGER TRG_ACCOUNTS_D
3  AFTER DELETE ON ACCOUNTS
4  FOR EACH ROW
5  BEGIN
6    DELETE FROM persons WHERE person_id = :OLD.account_person_id;
7  END;
```

Trigger 6: „TRG_ACCOUNTS_D “

E. Anhang – Prozeduren/Funktionen des Anwendungsszenarios

```
1  CREATE OR REPLACE
2  FUNCTION is_teamleader (lastname in VARCHAR2, name in varchar2,
3  team_id2 in INTEGER)
4  RETURN VARCHAR2 is
5    id_person INTEGER;
6    id_teamleader INTEGER;
7  BEGIN
8    SELECT person_id INTO id_person
9    FROM persons
10   WHERE LOWER(person_lastname) = LOWER(lastname)
11   AND LOWER(person_name) = LOWER(name);
12
13   SELECT team_leader INTO id_teamleader
14   FROM teams  WHERE team_id = team_id2;
15
16   IF (id_person = id_teamleader) THEN
17     return 'Ja';
18   ELSE
19     return 'Nein';
20   END IF;
21 END;
```

Funktion 1: „IS_TEAMLEADER“

```
1  CREATE OR REPLACE
2  FUNCTION state_to_string(state in INTEGER)
3  RETURN VARCHAR2 IS
4  BEGIN
5   CASE state
6     WHEN 5 THEN RETURN 'Fertiggestellt';
7     WHEN 4 THEN RETURN 'Nachbearbeitung';
8     WHEN 3 THEN RETURN 'Kontrolle';
9     WHEN 2 THEN RETURN 'Bearbeitung';
10    WHEN 1 THEN RETURN 'Entwurf';
11    ELSE RETURN 'Offen';
12  END CASE;
13 END;
```

Funktion 2: „STATE_TO_STRING“

```
1  CREATE OR REPLACE
2  PROCEDURE p_log_projects IS
3    i_count INTEGER;
4  BEGIN
5    FOR v_item IN
6      (SELECT fs.project_id as project_id, COUNT(*) as counter
7       FROM
8         (SELECT * FROM ((projects pr)), (packages p), tasks t
9          WHERE pr.project_id = p.package_project_id
10         AND p.package_id = t.task_package_id) fs
11      LEFT JOIN companies c
12      ON fs.project_company_id = c.company_id
13      GROUP BY fs.project_id)
14   LOOP
15     IF (v_item.counter > 50) THEN
16       INSERT INTO log_project
17       (log_id, log_project_id, log_warning)
18       VALUES
19       (seq_log_project.nextval, v_item.project_id, 'ACHTUNG');
20     END IF;
21   END LOOP;
22 END;
```

Prozedur 1: „P_LOG_PROJECT“

Erklärung

Ich versichere, die von mir vorgelegte Arbeit selbständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

Rösrath, 17.09.2010

Marc Kastleiner