INSTITUT FÜR INFORMATIK
Datenbanken und Informationssysteme

Universitätsstr. 1 D–40225 Düsseldorf

Entwicklung eines Tools zur Generierung von Testdaten für Data Warehouses

Thomas Peter Schulenberg

Bachelorarbeit

Beginn der Arbeit:	26. Juni 2007
Abgabe der Arbeit:	10. Oktober 2007
Gutachter:	Prof. Dr. Stefan Conrad
	Prof. Dr. Jörg Rothe

Erklärung

Hiermit versichere ich, dass ich diese Bachelorarbeit selbstständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Düsseldorf, den 10. Oktober 2007

Thomas Peter Schulenberg

Zusammenfassung

Thema meiner Arbeit war die Entwicklung eines Testdaten Generators für Data Warehouse Anwendungen. Der Data Generator sollte dabei SQL-konforme Daten erzeugen. Diese Daten sollen später im Lehrstuhl "Datenbanken und Informationssysteme" von Professor Dr. Stefan Conrad zum Testen und Entwickeln von Data Warehouses und Datenbankanwedungen im allgemeinen und zum Anlernen von Studenten an solcherart Systemen im besonderen verwendet werden. Für diese Systeme ist es erforderlich, dass der Data Generator bei einem Befüllungsvorgang mehere zehntausend Datensätze erzeugen kann. Diese Datensätze sollen eine möglichst realistische Abbildung echter Datensätze aus Kunden- und Geschäftsprozessdaten sein. Zusätzlich sollte ein einfaches Tabellenmanagement über das Programm möglich sein, welches die normalen SQL-Befehle `alter`, `drop` und `create` unterstützt und es zusätzlich ermöglicht über `delete` den Inhalt aus einer Tabelle zu löschen.
Für das eigentliche Generieren von Testdaten gibt es drei Oberflächen. Davon sind zwei als reine Informationsquelle für den Nutzer gedacht. Eine dieser Oberflächen ist eine normale Programmhilfe mit generellen Informationen zu allen Funktionen und Befüllungsfunktionen des Programms. Die andere Oberfläche bietet Informationen über eine ausgewählte Tabelle.
Das entstandene Programm kann in DB2-Datenbanktabellen Testdaten einfügen und sich alle benötigten Informationen dafür vorher von diesem DB2-Datenbankserver erfragen.
Das bei dieser Arbeit entstandene Programm bietet Lösungen zu vielen verschiedenen benötigten Funktionalitäten eines Data Generators und unterstützt alle gängigen SQL-Datentypen. Doch gerade durch diese große Anzahl an Konzepten in und um diese Datenbanken kann und darf dieser Data Generator keinen zu strikten Regeln folgen, da er sonst diese Vielfalt beschneiden würde. Das heißt aber im Gegenzug, dass die Benutzung dieses Programms einen Nutzer voraussetzt, der gute Grundkenntnisse in SQL mitbringt und zwar nicht nur theoretische sondern sehr konkrete praktische Grundkenntnisse über das benutzte Datenbanksystem, in diesem Falle DB2. Nur so kann er die verschiedenen Funktionen richtig einschätzen und in den vielen Programmdialogen die richtigen Auswahlen treffen. Auch bei der Identifizierung der Fehler aus der riesigen Masse an möglichen Fehlern muss auf die Mithilfe des Nutzers gebaut werden. Diesem werden dafür alle Informationen über den Fehler zur Verfügung gestellt. Dieses Vorgehen wird auch mit den beiden reinen Informationsoberflächen fortgesetzt, die viele Fehler verhindern können, sofern sie vorab gelesen werden.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Datenbanken werden heute universell und in allen Bereichen des Lebens eingesetzt. Gerade in den Bereichen Web- oder Computeranwendungen spielen Daten eine große Rolle und müssen verwaltet werden. Aber vor allem in den Bereichen Forschung und Wissenschaft sind riesige Datenbanken unumgänglich. Praktisch jedes Forschungsgebiet sammelt unaufhörlich neues Wissen und es entstehen täglich neue Publikationen. Ohne eine Katalogisierung dieses Wissens würde vieles doppelt erforscht werden und es würde seltener Synergieeffekte zwischen Forschergruppen geben. Noch zu nennen in diesem Zusammenhang ist die Bioinformatik und dort die Genomforschung. Bei der Entschlüsselung der Genome von Lebewesen fallen die Daten abertausender DNA-Sequenzen an. Diese können nur noch mittels Datenbanken verarbeitet werden.
Um große Datenmengen maschinell bewältigen und direkt innerhalb einer Umgebung analysieren zu können, wurden Data Warehouses entwickelt. Kundenadressdatensätze mit möglichst vielen Kontaktdaten, sowie Kundenprofile, die Aufschluss über die Vorlieben und Bedürfnisse des Kunden geben, stehen dabei klar im Vordergrund gegenüber zum Beispiel den Lagerbestandsdaten einer Firma. Denn dieses Wissen über den Kunden ist mittlerweile ein Kapital der Firma, mit dem sich direkt der Umsatz beeinflussen lässt. Weiterhin werden Data Warehouses zur statistischen Erfassung von Geschäftsprozessen benutzt, vor allem für das Auswerten von Gewinn und Umsatz heruntergebrochen auf einzelne Regionen oder Filialen. Daten werden dafür in großen Mengen erhoben, gespeichert und sogar gehandelt, auch wenn das Handeln mit Daten oft in rechtlichen Grauzonen geschieht.
Für die Veranstaltung Data Warehouses des Lehrstuhls “Datenbanken und Informationssysteme” von Professor Dr. Stefan Conrad besteht nun der Bedarf an großen und realistischen Testdatensätzen, einerseits zum Testen fertiger Data Warehouse Software und andererseits zum Testen eigener Implementierungen, aber auch zum Üben des Umganges mit den Funktionen eines fertigen Data Warehouses.
Dabei gibt es nicht die Möglichkeit auf Adressdaten oder noch besser vollständige Kundendatensätze größerer Firmen zurückzugreifen, da diese Daten mittlerweile zum Kapital eines Unternehmens dazugerechnet werden und auch in Hinsicht auf die Konkurrenz nicht firmenextern zur Verfügung gestellt werden. Außerdem gibt es rechtliche Probleme aus Datenschutzgründen, die zum Beispiel verhindern mit den original Studentenadressdatensätzen zu arbeiten.
Vorhandene Datengeneratoren bieten oft nur das Befüllen mit Zufallszahlen und Buchstabenkombinationen, was aus verschiedenen Gründen nicht ausreichend ist.
Hinzu kommt, dass realistische Adress- und Kundendatensätze viele Besonderheiten haben, mit denen eine Verwaltungssoftware egal aus welchen Bereichen umgehen können muss. Dafür müssen in der Entwicklungsphase dann auch entsprechend realistische Testdaten vorhanden sein oder generiert werden können.
Insgesamt soll die hier entwickelte Software aber universell zum Befüllen und Verwalten von Testdaten(-banken) nutzbar sein. Allerdings wird der Datagenerator vorerst nur eine Verbindung zu DB2 Datenbanken von IBM ermöglichen.

1.2 Problemstellung

Der Anspruch an die von meinem Programm generierten Datensätze ist es, so gut wie möglich die reale Welt der Kundendaten und Geschäftsprozessdaten abzubilden.
Dabei soll es die Möglichkeit geben, dass es zu großen Übereinstimmungen zwischen zwei verschiedenen Tupeln kommen kann, wie dies auch bei echten Daten der Fall ist. Insgesamt soll das Programm mehrere zehntausend Tupel generieren können.
Zusätzlich muss das Programm die Kommunikation mit DB2-Datenbankservern beherrschen und soll in der Lage sein, neue Tabellen anzulegen und vorhandene zu ändern und zu löschen.
Für die Datensätze sollen die typischen Kundendatenfelder unterstützt werden wie: Vorname, Nachname, Straße, Hausnummer, Stadt, PLZ, Land. Es soll die Möglichkeit geben, weitere Datenfelder mit Zufallsbuchstaben- oder Zufallszahlenkombinationen zu füllen.
Es wird aber nicht verlangt, dass das Programm von sich aus mit jeglichem spezifizierten SQL-Datentyp umgehen kann, sondern der Nutzer muss mit eigenem SQL-Wissen für die valide Befüllung sorgen, da die Funktionalität des Programms sonst zu sehr einschränkt werden würde.
Zum Generieren der Tupel wird dem Programm eine Basis an XML basierten Rohdaten mitgeliefert, die dann immer wieder neu kombiniert werden, was die nötige Vielfalt bietet und doch Übereinstimmungen durch die hohe Anzahl der Tupel nicht ausschließt.
Auf die Besonderheiten und funktionalen Abhängigkeiten realer Daten soll durch die Funktionalität des Programms und die Form der XML-Daten Rücksicht genommen werden, so dass auch bei den simulierten Datensätzen PLZ und Ort in einer Beziehung stehen.

1.3 Struktur der Arbeit

Im zweiten Kapitel dieser Arbeit wird die Theorie eines Data Warehouses besprochen. Außerdem wird die verwendete Datenbanksoftware DB2 von IBM kurz vorgestellt und es wird in die Grundlagen von SQL eingeführt. Danach wird die Programmiersprache Java, sowie die in Java verwendete graphische Oberfläche Swing und die Datenbankschnittstelle JDBC vorgestellt. Anschließend wird das Konzept von XML kurz erläutert. Als letzten Punkt gibt es einen Abschnitt über Data Generatoren, welche auf diesen Grundlagen basieren.
Das dritte Kapitel beschäftigt sich mit der Umsetzung des Projektes. Hierbei wird erst einmal auf die Konzeptionierung des Programms in Hinsicht auf den Funktionsumfang sowie auf die letztendlich abbildbaren Datentypen eingegangen.
Danach wird die Implementierung der einzelnen Klassen besprochen.
Im vierten Kapitel wird anhand von Bildern der jeweiligen Oberfläche durch das fertige Programm geführt. Dabei wird anhand eines Beispiels das eigentliche Generieren der Testdaten vorgestellt.
Das fünfte und letzte Kapitel beinhaltet ein Fazit sowie einen Ausblick auf mögliche sinnvolle Erweiterungen des Programms.

2 Grundlagen

2.1 Data Warehouses

Der Begriff Data Warehouse kommt aus dem Bereich der Business Intelligence wie zum Beispiel auch das Customer Knowledge Management. Im Bereich der Business Intelligence geht es darum, die Möglichkeiten der modernen Technik zu nutzen, um aus den riesigen Mengen der in einem Unternehmen anfallenden Daten einen weiteren Nutzen zu ziehen. Dafür müssen diese zentral gesammelt, analysiert und ausgewertet werden. Dabei werden Daten aus vielen verschiedenen Quellen in ein System eingepflegt. Von dort werden sie dann vielen verschiedenen Anwendern entsprechend ihrer unterschiedlichen Bedürfnisse zur Verfügung gestellt. Ein Nebeneffekt ist auch eine Archivierung aller Geschäftsdaten an einem Ort. In einem Data Warehouse werden die Daten losgelöst von ihrer Herkunft betrachtet und können je nach Zweck der Analyse temporär neu zusammengeführt werden. Eine Veränderung der eigentlichen Rohdaten ist meist nicht mehr erforderlich, sondern der Datenbestand wird kontinuierlich um weitere Rohdaten erweitert und alle Daten zusammen werden aus immer wieder neuen Blickwinkeln betrachtet. Dadurch handelt es sich bei einem Data Warehouse oft um eine Nur-Lese-Datenbank, die von außen erweitert wird [PR97].
Generell sollte man zwischen den Begriffen Data Warehouse und Data Warehouse System unterscheiden, wobei das eigentliche Data Warehouse nur die Datenspeicherung meint und erst im Data Warehouse System die Managementsoftware integriert ist. Eine weitere Begrifflichkeit in diesem Zusammenhang ist OLAP, was für "Online Analytical Prozessing" steht und das Verdichten von Daten zu einem mehrdimensionalen sogenannten "Würfel" meint. OLAP beinhaltet Funktionen wie: Summe, Maximum und Minimum, welche auch von SQL selbst bereitgestellt werden, aber auch Funktionen wie die Berechnung von Median und Standardabweichung, welche den Funktionsumfang von SQL deutlich überschreiten. OLAP ist damit eine Weiterentwicklung von Tabellenkalkulationsprogrammen ins Mehrdimensionale [HS00]. Ein weiterer Begriff, der auftaucht, ist Data Mining, darunter versteht man, dass der Datenbestand mit statistisch mathematischen Methoden auf erkennbare Muster durchsucht wird [HS00]. Ein Data Warehouse System ist somit ein Entscheidungssystem, denn anhand der Analyse der Daten können Trends abgelesen werden und auf diese Trends kann bei Bedarf reagiert werden.
Ein sehr beliebtes Beispiel für den Nutzen von Data Warehouses ist die Auswertung von Supermarktkassendaten. Dies ist erst durch die flächendeckende Einführung von Scanner-Kassen möglich, denn mit ihren Daten können Einkäufe nach Produkten aufgeschlüsselt werden, da nun jedes Produkt dem eingescannten Strichcode zugeordnet werden kann, während früher nur die Preise eingetippt wurden. Dabei stellte sich heraus, dass es viele Einkäufe gibt, die unter anderem Windeln und Bier beinhalten, also den Bedarf junger Väter. Daraufhin ordneten die Supermärkte diese beiden Produkte so weit wie möglich entfernt voneinander in den Regalen an, so dass der Käufer an vielen Dingen vorbei gehen muss, bis er beides erreicht hat. Danach konnte anhand von Statistiken gezeigt werden, dass der Absatz jener Produkte stieg, die auf dem Weg zwischen Windeln und Bier angeordnet wurden.
Primär geht es hier um den Umsatz des Unternehmens und nicht um einen Servicege-

winn für den Kunden. Dies zeigt das vorangegangene Beispiel, die Väter hätten sich sicher über kürzere Wege zwischen Bier und Windeln gefreut.
Um die Einkäufe darüber hinaus noch einzelnen Kunden zuordnen zu können, wurden sogenannte Kundenkarten entwickelt, welche bei einen Einkauf mitgescannt werden. Da nun alle Daten über einen Kunden an einem Ort gespeichert werden können, können diese natürlich sehr einfach zur Erstellung von Kundenprofilen genutzt werden, die einer Marketingabteilung Wissen über den Kunden liefern und es so ermöglichen, gezieltes Wissen für den Kunden zu generieren. So kann ein Unternehmen einem Kunden nur noch seinem Profil entsprechende Werbung zukommen lassen. Dadurch wird das firmeneigene Marketing effizienter und entwickelt sich vom mittlerweile üblichen Kundenbelästigungssytem zu einem wirklichen Kundenbenachrichtigungssystem. Auch die Benachrichtigung bei Rückrufen und Umtauschaktionen kann auf die Kunden beschränkt werden, die es wirklich betrifft.
Generell steht diese Art des Datensammelns durch Firmen in der öffentlichen Kritik, maginär wegen der oft mangelhaften Absicherung der Daten gegen den Datenklau. Aber der Hauptgrund ist die Vorratsdatenspeicherung an sich, Stichwort “gläserner Mensch” und die Verlockung, an diesen Daten durch einen Weiterverkauf noch einmal zu verdienen. Um die Akzeptanz beim Kunden trotzdem aufrechtzuerhalten, geben die Händler deshalb dem Kunden kleine Rabatte oder andere Belohnungen, wenn er im Gegenzug eine Karte zum Datensammeln benutzt. Das bekannteste deutsche Beispiel ist das Payback-System, bei dem sich ungewöhlich viele Firmen aus den unterschiedlichsten Branchen auf ein System geeinigt haben. Nun kann eine Firma nicht nur das Zusammenspiel einzelner Produkte auf verschiedenen Quittungen analysieren, sondern kann jedem Einkauf ein genaues Kundenprofil, mit Alter und Geschlecht zuordnen, und so für jedes Produkt seine Käufergruppe ermitteln, die dann gezielt beworben werden kann.

2.1.1 Die theoretischen Grundlagen von Data Warehouses

Im technischen Bereich gibt es mehrere Ansätze zur Datenbankgestaltung als Grundlage eines Data Warehouses. Gesucht ist eine Art der Datenbevorratung, die wenig Platz verbraucht, aber eine schnelle dynamische Analyse der Daten ermöglicht. Um die Analyse zu vereinfachen ist ein multidimensionales Datenmodell besonders gut geeignet. Man arbeitet mit Fakten, Dimensionen und Würfeln (’Cubes’). Die Fakten bezeichne ich als Rohdaten. Diese Rohdaten sind die in den Geschäftsprozessen direkt gewonnenen Daten, die mit dem Data Warehouse nun näher analysiert werden sollen. Über diese Fakten hinaus gibt es zum Beispiel Dimensionen bezüglich der Zeit, der geographischen Region, oder der Produktkategorie. Diese Dimensionen können noch einmal in Klassenhierarchien wie Elektronik, Kleinelektronik, Handys unterteilt sein. Die Dimensionen sind dabei wie unterschiedliche Sichten auf die Fakten zu sehen, die für eine orthogonale Strukturierung des Datenraums sorgen. Diese Dimensionen bilden die Kanten der Würfel, die Würfel als Ganzes wiederum repräsentieren das Ergebnis einer Anfrage (siehe [Con07] und [PR97]).

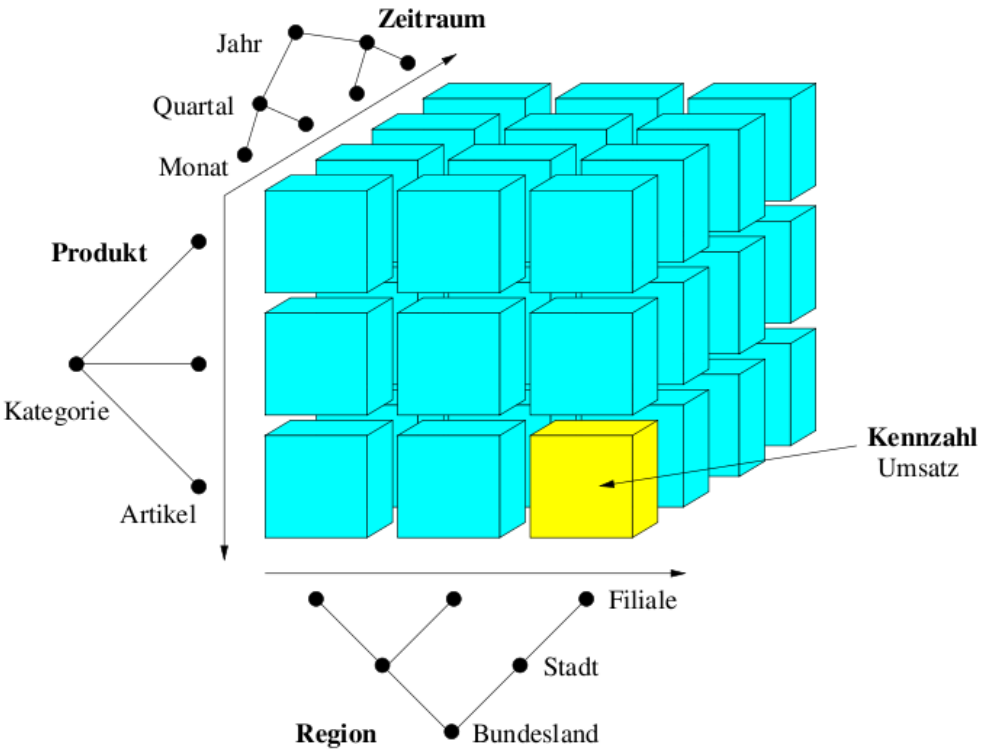

Abbildung 1: Ein solcher Würfel (entnommen aus [Con07])

Für die relationale Speicherung eines solchen multidimensionalen Datenmodells bieten sich zwei verschiedene Ansätze an.
Ein Ansatz zur Datenbankgestaltung ist das Star-Schema (deutsch: Sternschema). Jede Dimension wird in einer eigenen Dimensionstabelle dargestellt, wo sie über eine Id erreichbar ist. Betrachtet man zum Beispiel die Dimension Zeit, so ist jeder Zeitpunkt, den man betrachten möchte, ein Tupel in der Dimensionstabelle Zeit. Jeder Zeitpunkt, der dort verzeichnet ist, hat eine eindeutige Zeit_ID und seine Attribute wie Jahr, Monat und Tag. Damit bilden die Klassifikationshierarchien einer Dimension die Spalten in einer Dimensionstabelle.
In einer Faktentabelle steht nun zum Beispiel ein Produkt aus dem eigenen Warenangebot mit seiner Zeit_ID, seiner Regions_ID und seiner Produkt_ID sowie als nicht Id-Attribut die Anzahl der verkauften Exemplare. Wobei die Ids der Dimensionen als Fremdschlüssel, gemeinsam den Primärschlüssel der Faktentabelle bilden.

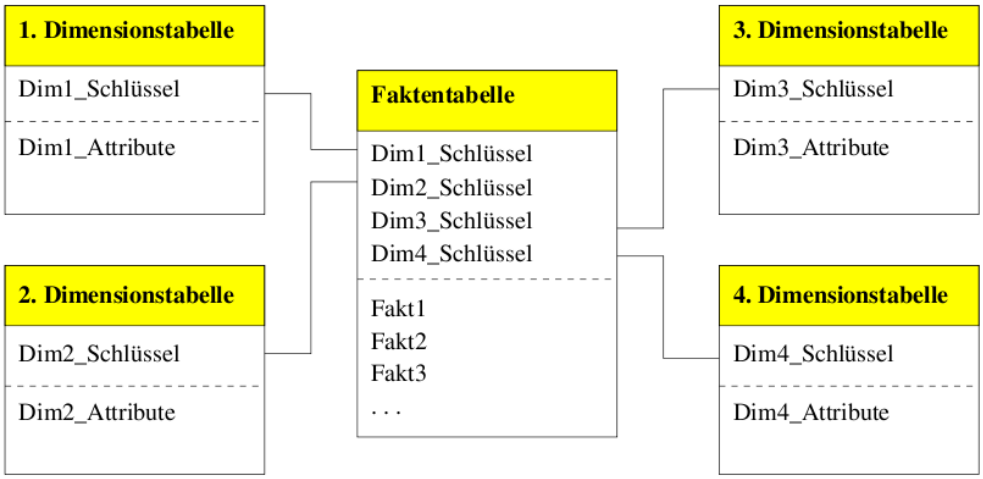

Abbildung 2: Tabellen im Star-Schema (entnommen aus [Con07])

Führt man bei einer Anfrage über die Ids alle Daten zusammen, so sieht man, wie viel von einem bestimmten Produkt (aus der Faktentabelle), das zu einer bestimmten Produktfamilie gehört (Information über die Produkt_ID), in einer bestimmten Absatzregion oder auch nur einer bestimmten Filiale (über die Regions_ID), in einem bestimmten Quartal oder auch nur an einem bestimmten Tag (über die Zeit_ID), verkauft wurde. Die einzelnen Informationen stecken dabei in vielen verschiedenen aber sehr gut strukturierten und übersichtlichen Tabellen.
Der zweite Ansatz ist das Snowflake-Schema (deutsch: Schneeflockenschema), bei diesem Ansatz ist die Faktentabelle wie beim Star-Schema aufgebaut. Sie beinhaltet also zum Beispiel konkrete Verkaufszahlen eines Produktes, und über verschiedene Ids kann man herausfinden, aus welchem Zeitraum die Verkaufszahlen sind, von welcher Filiale und zu welcher Ware sie genau gehören. Auch hier bilden diese Ids zusammen wieder den Primärschlüssel der Faktentabelle. Aber die Dimensionstabellen sind nicht so kompakt wie beim Star-Schema, sondern beinhalten immer nur eine Klassenhierarchieebene. Das heißt, dass in der Faktentabelle eine Filial_ID eingetragen ist. Diese führt aber nur zu einem Filialnamen in der Dimensionstabelle Filiale, sowie einer Stadt_ID. Diese Stadt_ID führt zu der Dimensionstabelle Stadt, die damit die nächste Klassenhierarchieebene abbildet. Hier gibt es das Attribut Stadtname sowie eine Landes_ID, also muss man weiter zur Dimensionstabelle Land, erst dort findet man den Landesnamen zu der Filiale, deren Verkaufszahlen in der Faktentabelle stehen.

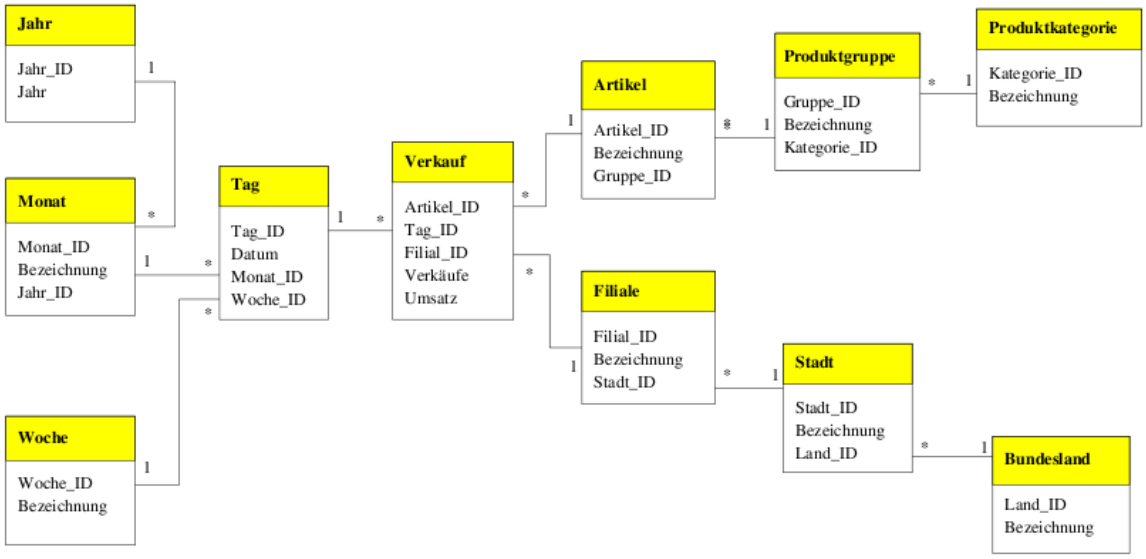

Abbildung 3: Tabellen im Snowflake-Schema (entnommen aus [Con07])

Das Snowflake-Schema ist durch diese starke Zergliederung normalisiert, was den Vorteil hat, weniger redundante Informationen zu speichern, und somit Update Anomalien vorbeugt. Aber sein Nachteil besteht darin, dass man bei jeder Anfrage über eine große Anzahl von Tabellen auch innerhalb einer Dimension joinen muss (siehe [Con07] und [PR97]).

2.2 DB2

DB2 ist ein Datenbanksystem von IBM, welches aktuell in Version 9 vorliegt. Mir wurde vom Lehrstuhl DB2 Personal 9.1 zur Verfügung gestellt. Als eine professionelle Software unterstützt DB2 verschiedene Betriebssysteme problemlos und ist in Version 9 als Hybriddatenbankserver ausgelegt, welcher sowohl mit relationalen Daten (SQL), als auch mit XML basierten Daten umgehen kann. Es gibt auch eine Data Warehouse Edition, welche neben einer DB2-Datenbank den kompletten Funktionsumfang eines Data Warehouse Systems bietet.

2.3 SQL

Bei Datenbanken gibt es sehr viele verschiedene Ansätze zur Organisation, mehrheitlich durchgesetzt haben sich dabei relationale Datenbanken vor allem mit SQL als verwendete Sprache. SQL, was für “Structured Querry Language” steht, ist dabei aber viel mehr als eine reine Anfragesprache. SQL wird generell noch einmal in die beiden Bereiche “Data Manipulation Language” (DML) und “Data Definition Language” (DDL)

unterteilt [Bor02].
Die DDL umfasst primär das Erstellen, Verändern und Löschen von Tabellen. Im Programm verwende ich die englischen Befehlswörter `create`, `alter` und `drop` als Funktionsnamen. Auch Indizes und Views können mit diesen Befehlen definiert werden, was aber für mein Programm irrelevant ist. Wichtig hingegen ist, dass auf der Ebene der DDL funktionale Abhängigkeiten zwischen Tabellen durch das Setzen von `Foreign Keys` (deutsch: Fremdschlüssel) niedergeschrieben werden. Diese Fremdschlüssel verweisen auf `Primary Keys` (deutsch: Primärschlüssel) in anderen Tabellen. Das Ganze hat zum Hintergrund die Vermeidung von redundanten Informationen. Was wiederum eine Vermeidung von verschiedenen Anomalien, ein vereinfachtes Updaten und eine Verminderung des Speicherbedarfs bedeutet. Was sind nun redundante Informationen? Speichert man zum Beispiel die Bestandsdaten einer Bibliothek, so gehört zu jedem Verlagsnamen ein fester Haupsitz. Wird dieses Wertepaar in der gleichen Tabelle gespeichert wie auch der Titel, der Autor, etc. so ist das redundant. Da man über den Verlagsnamen immer auf einen eindeutigen Ort schließen kann, kann man die Spalte Verlagsort sparen, wenn es statt dessen eine kleine Tabelle gibt, wo jedem Verlag einmal ein Ort zugewiesen wird. Nun macht es Sinn diese beiden Tabellen zu verbinden. Da in der neuen Tabelle jeder Verlagsname nur noch einmal vorkommen muss, eignet sich der Name sehr gut als `Primary Key`, denn dieser muss eindeutig sein, und darf damit nur einmal vorkommen. Nachdem man dies so gesetzt hat, kann man in der eigentlichen Buchtabelle nun die Verlagsnamen als `Foreign Keys` eintragen, und die Bücher über die zweite Tabelle mit dem Verlagsort verbinden. Da jeder Verlagsort nur noch ein einziges Mal in der Datenbank vorkommt, ist es nun auch neben dem Sparen von Speicherplatz viel einfacher bei einem Umzug des Verlages, dessen Ort zu ändern. Dies muss nur noch einmal in der zweiten Tabelle geschehen, statt mehrmals in der ursprünglichen Tabelle.
Die DML beschäftigt sich nicht mit der Definition von Tabellen, sondern mit dem Manipulieren einzelner Tupel oder Werte. Die Befehlsworte lauten hier `insert`, `update` und `delete`, sowie die einfache Datenbankanfrage von Tupeln mit einem `select`. Hierbei gibt es eine unendliche Fülle von Möglichkeiten, die SQL bietet. Fast alle bekannten Operatoren sowie verschiedene Aggregatsfunktionen und das Joinen von Tabellen miteinander sind vorhanden.
Diese große Vielfalt wird aber vom Data Generator kaum genutzt, und vom Nutzer werden hauptsächlich Kenntnisse des `alter`- und `create`-Statements erwartet.
SQL wurde im Laufe der Zeit immer wieder als neuer Standard gefasst, wobei das Hauptaugenmerk aber auf der Erweiterung des Standards lag, aktuellste Version ist SQL2003.
Probleme bereitet generell, dass jede Datenbanksoftware ihren eigenen SQL-Dialekt benutzt. Dieser Dialekt unterstützt manchmal Befehle nicht, die im eigentlichen Standard vorkommen, oder aber die Befehlstruktur weicht von der im Standard festgelegten ab. Auch haben die verschiedenen Hersteller oft noch eigene Funktionen eingebaut, die nicht oder noch nicht im Standard enthalten sind.

2.4 Java

Java wurde 1991, damals noch unter dem Namen Oak, von einem Team um Patrick Naughton, James Gosling und Mike Sheridan bei Sun Microsystems entwickelt [CL04]. Die Programmiersprache Java bildet mit der Java-Laufzeitumgebung (oder auch Java-Plattform) die Java-Technologie. Java wurde aber erst 1995 der Öffentlichkeit vorgestellt, nachdem sich mehrmals der Name und der Verwendungszweck geändert hatten.
Die großen Vorteile der endgültigen Java-Technologie sind die Objektorientiertheit der Programmiersprache und die Plattformunabhängigkeit der Laufzeitumgebung. Wobei zu berücksichtigen ist, dass die Java Laufzeitumgebung nur auf Systemen läuft, für die es eine Java Virtual Machine gibt. Ein weiteres Feature sind die frei verfügbaren Klassenbibliotheken mit schon fertigen Klassen, sowie die Möglichkeit seinen eigenen Code in solche einzupflegen und so in anderen Projekten ohne "copy and paste" wiederverwerten zu können.
In dieser Arbeit wird die neueste Version der Java-Technologie J2SE 5.0 verwendet.

2.4.1 Swing

Java bietet mittlerweile drei Toolsets zum Programmieren grafischer Oberflächen. Diese sind AWT, Swing und SWT. Für dieses Projekt habe ich mich für Swing entschieden, denn nur dieses Toolset bietet eine große Menge an Dialogelementen, welche ich benötigen werde, um den Nutzern eine größtmögliche Funktionalität zu bieten. Diese Dialogelemente sind im Gegensatz zu SWT alle im Standardumfang enthalten, was die Kompatibilität plattformübergreifend erhöht. Außerdem gibt es die Möglichkeit sich mit Swing dem sogenannten "Look and Feel" des jeweiligen Betriebssystems anzupassen, was AWT nicht beherrscht [CL04].

2.4.2 JDBC

Für datenbankbasierte Java-Anwendungen ist im Java Standard Paket die JDBC-Schnittstelle seit JDK 1.1 integriert. Hierbei handelt es sich um ein an ODBC angelehntes Datenbank-Interface für SQL-Datenbanken. Bei der Kommunikation zwischen einer Java-Applikation und einer SQL-Datenbank übernimmt JDBC die Aufgabe eines "Transportprotokolls".
Das Grundgerüst für den Verbindungsaufbau zum Datenbankserver ist dabei immer das gleiche, bestehend aus Methoden des `Java.sql` Paketes. Es wird ein Connection Object erzeugt, welchem ein JDBC-DB2-Treiber sowie die entsprechenden Verbindungsdaten zur Verfügung gestellt werden. Danach wird noch ein Object vom Typ `Statement` erzeugt, welches die SQL-Anfrage des Programms auf verschiedene Weise verarbeitet, je nachdem, ob es eine `select`-Anfrage oder etwas anderes ist. Für die `select`-Anfrage erhält man einen Rückgabewert vom Typ `ResultSet`, `ResultSetMetaData` oder `DatabaseMetaData`, welcher noch bei geöffneter Verbindung zeilenweise verarbeitet werden muss. Ist dies geschehen, sollte man das `Statement` und die `Connection` schließen. Die ganze Verbindung findet in einem Try-Catch-Block statt, um Exceptions abfangen und behandeln zu können. Eventuell auftretende Fehlermeldungen gibt das

Programm dem Nutzer zur Korrektur seiner Eingaben aus [CL04].

```
Connection conn;
Class.forName("com.ibm.db2.jcc.DB2Driver");
conn = DriverManager.getConnection(connectionString, userString, pwString);
Statement stmt = conn.createStatement();
.
.
.
stmt.close();
conn.close();
```

2.5 XML

XML steht für "Extensible Markup Language" und ist eine Meta-Auszeichnungssprache, um Daten strukturiert zu beschreiben. Genauso wie Java ist XML plattformunabhängig, solange ein nötiger Parser vorliegt, um die Informationen auszulesen. XML ist ebenso wie HTML eine Teilmenge von SGML "Standard Generalized Markup Language" , ein Standard von 1986 , um Daten unabhängig von Plattform und Anwendung in ihrer logischen Struktur zu speichern. Bei SGML kann man sich völlig frei Strukturelemente erstellen, solange man diese standardkonform definiert. Grob gesehen waren vielen Anwendern die Möglichkeiten von SGML zu umfangreich, und man einigte sich vor allem für Webanwendungen auf HTML, was nur aus im Standard vorher festgelegten Strukturelementen wählen kann. HTML wiederum wurde mit dem steigenden Nutzen und den steigenden Nutzern des Internets zu eingeschränkt und man ging mit XML einen Mittelweg aus SGML und HTML [Bor01].

2.6 Data Generatoren

Unter Data Generatoren versteht man allgemein Programme, die Daten in eine wie auch immer geartete Datenbank füllen. Diese Daten werden generiert und sind somit immer nur der Versuch einer Abbildung von Realdaten. Die einfachste Methode ist es, mit Zufallszahlen und -buchstaben zu arbeiten, was aber eigentlich nur ausreicht, um die Funktionalität einer Datenbank an sich zu prüfen. Will man nun aber eine Software, die Realdaten aus einer Datenbank verarbeiten soll, testen und stehen diese Daten noch nicht zur Verfügung, so braucht man leistungsfähigere Data Generatoren. Die Implementierung eines solchen ist Ziel dieser Arbeit.
Um den Problemen bei zufälligen Nonsens-Wörtern zu entgehen, ist der Grundgedanke mit XML-basierten Listen zu arbeiten. Diese bestehen aus einer Menge an einzelnen Namen, zum Beispiel Straßennamen. Zusätzlich erhält jeder Name einen Wert "Anzahl", um damit verschieden häufiges Auftreten zu simulieren. Dies geschieht dadurch, dass der XML-Parser jeden Namen so oft in die entsprechende `ArrayList` einfügt, wie seine Anzahl das vorgibt. Im Data Generator findet dann mithilfe der Javaklasse `Random` ein zufälliger Zugriff auf einen Wert der `ArrayList` statt. Werte, die mehrmals in dieser

`ArrayList` stehen, werden so auch häufiger verwendet. Auf diesem Weg kann man eine natürliche Verteilung simulieren.
Drei große Problematiken ergeben sich dabei:
Die erste Problematik entsteht durch die Vielzahl der möglichen Datentypen, die allein schon aus den SQL-Standards resultieren und eine unterschiedliche Behandlung bei dem Erstellen des `insert`-Statements erfordern. Hier ergibt sich sogar zusätzlich noch die Problematik, dass jeder Datenbankanbieter diese Standards unterschiedlich umsetzt und unterstützt. Weiterhin erweitern die Datenbankanbieter auch noch die Befehlssätze ihrer Software über den SQL-Standard hinaus. Theoretisch müsste man versuchen zu verhindern, dass ein Nutzer ein Attribut mit einem falschen Datentyp befüllt, oder sich mit den zur Verfügung gestellten Funktionen Daten generieren lässt, die zu groß sind für die Breite des Feldes, in das sie eingefüllt werden sollen. Doch hier kann man nicht eindeutig von falsch sprechen, also muss man andere Lösungen finden. Zum Beispiel muss man abfangen, wenn Zahlen, die einen SQL-Integerwert darstellen, in ein Feld des Typen "Char" oder "Varchar" eingefügt werden sollen. Es ist zwar möglich Ziffern in diese Felder zu befüllen, aber der ganze Wert muss bei diesen Feldtypen mit Hochkommata umschlossen sein.
Die zweite Problematik ergibt sich bei der Anbindung des Programms über die JDBC-Schnittstelle an unterschiedliche Datenbanken. Das Abfragen von Strukturinformationen wie zum Beispiel eine Tabellenübersicht, gestaltet sich immer unterschiedlich, da jede Datenbanksoftware ihre eigene interne Struktur besitzt. Dies führt zusammen mit der ersten Problematik zu der Entscheidung, den entstehenden Data Generator nur für eine Datenbanksoftware zu optimieren, dies ist DB2 von IBM.
Drittens kann es in Datenbanken interne funktionale Abhängigkeiten zwischen verschiedenen Tabellen mittels `Primary Keys` und `Foreign Keys` geben. Dies ist sogar einer der Hauptgründe für den Siegeszug der relationalen Datenbanken in der Wirtschaft, also absolut erwünscht und erforderlich. Doch um solche voneinander abhängigen Tabellen mit einem Data Generator befüllen zu können, müssen auch alle Informationen über solche Beziehungen vorab angefragt und im Programm richtig verarbeitet werden. Dadurch wächst die Komplexität des Programms enorm, da für jede Tabelle vor dem Befüllen abgefragt werden muss, ob es Beziehungen zu anderen Tabellen gibt. Danach stellt sich die Frage nach dem Umgang mit diesem Wissen. Es muss gewählt werden, ob man durch feste Regeln einen "richtigen" Weg vorgibt oder dem Nutzer eine gewisse Freiheit lässt, selber auf die Richtigkeit seiner Aktionen zu achten, und ihm somit mehr Möglichkeiten eröffnet. Das erfordert dann allerdings ein gewisses Vorwissen vom Nutzer.

3 Implementierung

3.1 Konzeption

Es soll ein plattformunabhängiges grafisches Programm entstehen, welches insgesamt aber auf Funktionalität und nicht auf Performance optimiert werden soll. Um trotzdem nicht zu viele Ressourcen zu verbrauchen, wird für jede Interaktion mit dem Datenbankserver eine neue Verbindung geöffnet und nach Ausführen der Aktion oder aber einer Fehlermeldung durch den Datenbankserver wieder beendet. Gestartet werden soll das Programm über eine eigene Klasse, welche neben der ersten Oberfläche noch den XML-Parser initialisiert. Dadurch ist es möglich, dass der XML-Parser nur einmal zu Beginn des Programms aufgerufen werden muss. Somit werden die XML-Daten am Anfang des Programms einmal in `ArrayLists` eingelesen und stehen danach der Befüllungsfunktion permanent zur Verfügung. Alle Arten von Daten, die von mehreren Klassen benutzt werden, somit auch die `ArrayLists` mit den XML-Daten, werden zentral in einer Klasse `DataHaendler` gespeichert.
Nachdem man die Verbindungsdaten zum Server eingegeben hat, steht dem Nutzer auf der zentralen Oberfläche eine Liste aller Tabellen des angemeldeten Datenbankusers zur Verfügung. Von hier aus wählt man erst eine Tabelle und dann eine Aktion, die mit dieser Tabelle durchgeführt werden soll. Die möglichen Aktionen sind Befüllen, Verändern und Löschen einer Tabelle, sowie das Löschen des gesamten Inhaltes aus einer Tabelle. Erfordern diese Aktionen viele Nutzereingaben, so öffnet sich für sie eine neue Oberfläche. Dies ist beim Verändern und Befüllen von Tabellen der Fall. Die beiden verschiedenen Versionen vom Löschen hingegen lösen einen Nachfragedialog aus, damit nicht durch das unbeabsichtigte Klicken der Buttons aus Versehen gelöscht wird. Zusätzlich gibt es noch die Aktion “Erstellen einer Tabelle”, welche natürlich nicht der vorherigen Auswahl einer Tabelle bedarf. Auch diese Aktion findet der Übersicht halber auf einer eigenen Oberfläche statt. Außerdem gibt es noch zwei Oberflächen, die verschiedene Informationen für den Nutzer bereithalten. Anders als alle anderen Oberflächen schließen diese beiden nicht das aktuelle Fenster, sondern werden zusätzlich geöffnet. Die eine dieser Oberflächen ist eine generelle Hilfe, welche über alle möglichen Aktionen sowie alle möglichen Befüllungsfunktionen des Programms informiert. Als zweites gibt es eine Oberfläche, die über die zum Befüllen oder zum Ändern vorgesehenen Tabellen zusätzliche Informationen bereithält. Dies sind Informationen über die Primärattribute, die Beziehungen zu anderen Tabellen und die Anzahl schon vorhandener Tupel in der Tabelle. Außerdem erhält der Nutzer hier eine Benachrichtigung über mögliche `auto_increment`-Spalten, welche vorher aus dem Befüllungsvorgang entfernt wurden. Als letztes werden noch die möglichen Spalten aufgelistet, welche keine “null”-Werte erlauben.
Die Informationen haben den Zweck, dem Nutzer die Möglichkeit zu geben, die richtigen Funktionen zu wählen, da es einfach nicht möglich ist, alle falschen Eingaben vorherzusehen und abzufangen, ohne den versierten Nutzer mitunter stark einzuschränken.
Zusätzlich wird es eine kleine Menübar geben, in der zum Beispiel der Hilfe-Button untergebracht ist, sowie zwei weitere Buttons, um das Programm immer wieder von vorn beginnen zu können. Der eine Button führt zurück zur Verbindungsoberfläche der

andere zur Tabellenübersicht.
Um auf Nutzereingaben reagieren oder den Nutzer über den Ausgang von Aktionen benachrichtigen zu können, werden verschiedenste kleine Nachrichtenfenster genutzt. Dies dient auch dazu, durch Auswahl- und Eingabemöglichkeiten bei den einzelnen Funktionen den Gesamtfunktionsumfang des Programms noch zu steigern. Alle Fehlermeldungen bei der Kommunikation mit dem Server werden ebenfalls in Rohform samt Fehlercodes dem Nutzer über Dialogfenster gezeigt. Diese Fehlercodes geben dem versierten Nutzer die Möglichkeit über die Dokumentation von DB2 auf das Problem zu schließen, vor allem dann, wenn die eigentliche Fehlermeldung von sich aus nicht schlüssig ist. Generell bleiben seine Eingaben und/oder Auswahlen innerhalb der Oberflächen dabei erhalten. Dadurch kann der Nutzer seine Eingaben nach einem Fehler noch einmal verändern oder über die Menübar an eine andere Stelle des Programms gehen.
Bei zwei Aktionen ist es nötig dem Nutzer einige Informationen über die ausgewählte Tabelle zu geben. Dies sind die Befüllung und die Veränderung einer Tabelle. Dazu dient die weiter oben vorgestellte Oberfläche. Auch für die Tabellenübersicht, die zwischen allen Aktionen bei Bedarf aktualisiert wird, und für die Funktionen, die Daten aus anderen Tabellen für das Befüllen verfügbar machen, braucht man eine Klasse, die über `select`-Befehl Daten von dem Datenbankserver abfragt. Dies erledigt die Klasse `ConnectionDB2`. Für alle Operationen auf dem Datenbankserver, die keine Daten zurückliefern, also das Löschen sowie das Verändern oder Erstellen von Tabellen, nutze ich die Klasse `DataUpdater`. Dies liegt daran, dass die Statements, die von den beiden verschiedenen Klassen jeweils verarbeitet werden können, auch mit zwei verschiedenen Methoden an den Datenbankserver übergeben werden müssen. Es macht also Sinn, diese beiden unterschiedlichen Funktionalitäten zu trennen. Denn nach einem `select` muss die Klasse `ConnectionDB2` die Daten von der Datenbank direkt verarbeiten, da ein solches `ResultSet` zusammen mit der `Connection` und dem `Statement` (siehe Kapitel 2) geschlossen werden muss. Daher ist es nicht möglich es an eine andere Klasse weiterzureichen. Das `ResultSet` muss also in der Klasse `ConnectionDB2` ausgelesen und in Variablen der Klasse `DataHaendler` geschrieben werden. Hingegen erhält die Klasse `DataUpdater` außer eventuellen Exceptions keine Informationen von der Datenbank.
Um der unterschiedlichen Anzahl von Attributen innerhalb einer Tabelle sowie der vielen Fremdschlüssel, bedingt durch die beiden Daten-Schemata aus Kapitel 2 gerecht zu werden, müssen meine Oberflächen dynamisch in ihrer Größe sein.
Für den eigentlichen Befüllungsvorgang müssen die zur Verfügung gestellten Funktionen nun vorsortiert den SQL-Datentypen zugeordnet werden, um so schon einige Fehler durch den Nutzer zu unterdrücken. Zusätzlich muss über Validitätsabfragen und Defaultwerte weiteren Fehlern vorgebeugt werden.
Der Befüllungsvorgang startet nach der Auswahl einer Funktion für jede Spalte durch den Nutzer mit einigen Abfragen. Werden alle Abfragen gültig beantwortet, ist die Vorbereitung der Daten und des `insert`-Statements abgeschlossen. Nun wird eine Verbindung zum Datenbankserver aus der Klasse `FillTableGui` geöffnet und erst nach Erstellen aller Tupel oder einem Abbruch geschlossen. Eine erste `for`-Schleife läuft über die Anzahl der Tupel, die erstellt werden sollen. Diese Anzahl wurde vorher vom Nutzer abgefragt. Innerhalb dieser `for`-Schleife gibt es eine weitere `for`-Schleife, die über die Anzahl der zu befüllenden Attribute geht. In ihr wird nacheinander der durch

den Nutzer selektierte Wert der Dropdownliste abgefragt, und es wird der dementsprechende Einfüllwert erstellt. Dabei müssen nicht zu befüllende Spalten berücksichtigt werden. Auch muss bei einer Fehlermeldung, zum Beispiel durch doppelt erstellte `Primary Keys`, richtig reagiert werden. Am Ende muss der Nutzer über Erfolg oder Misserfolg der Aktion benachrichtigt werden.

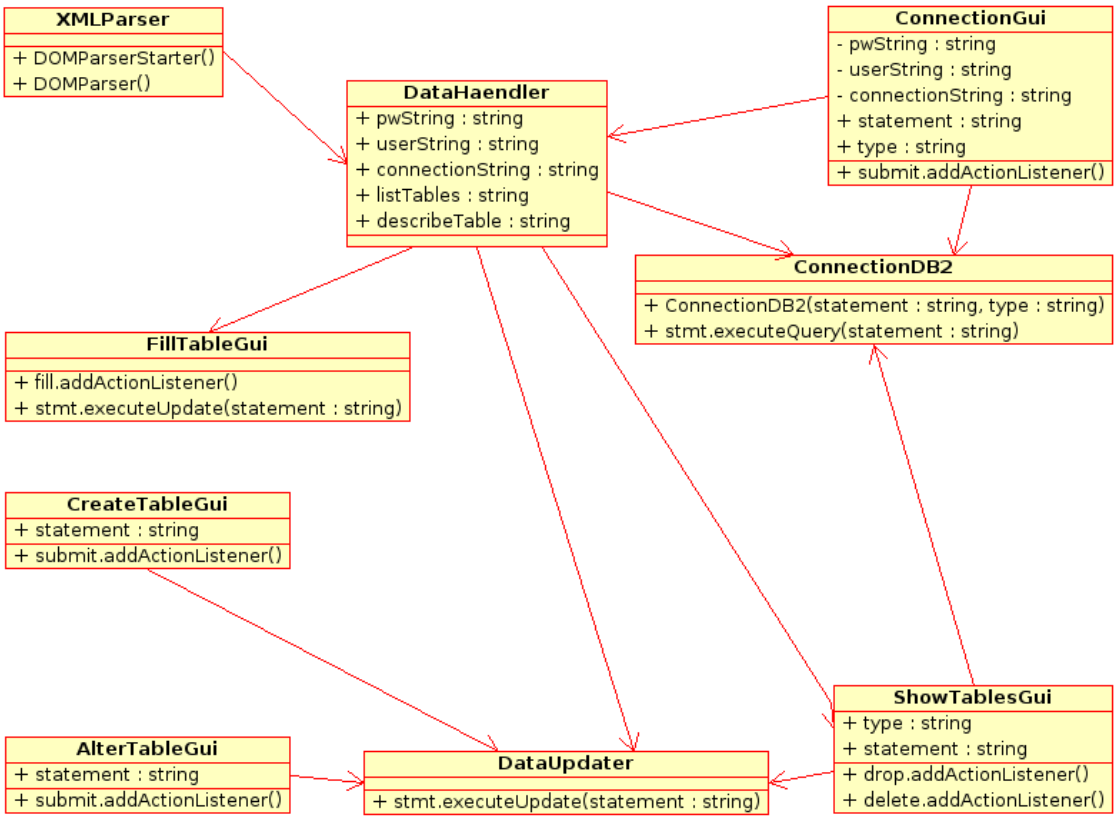

Abbildung 4: UML-Diagramm

3.2 Die Klasse `ProgrammStarter`

Die erste Klasse des Data Generators ist die Klasse `ProgrammStarter`. Sie initialisiert neben der ersten Oberfläche noch den XML-Parser, damit dieser einmal zu Beginn des Programms die aktuellsten XML-Dateien auswertet und ihren Inhalt als Grunddatenbasis zur Verfügung stellt.

```
public class ProgrammStarter
{
        public static void main(String[] args)
        {
                XMLParser parse = new XMLParser ();
                parse.DOMParserStarter();

                ConnectionGui gui = new ConnectionGui();
                gui.setVisible(true);

        }
}
```

3.3 XML

Über XML-Dateien werden dem Programm Listen zum Beispiel mit Vornamen oder Nachnamen zur Verfügung gestellt. Dabei ist es möglich, in die daraus resultierende Grunddatenbasis des Programms Häufigkeiten einzubauen. Zum einem durch ein mehrmaliges Einfügen eines Wertes an verschiedenen Stellen der Liste, aber zum zweiten auch durch die Angabe "Anzahl", die man im XML-Schema jedem Wert zuordnen muss. Will man nicht mit ihr arbeiten, setzt man sie gleich 1.
Als Beispiel steht hier der Inhalt der XML-Datei currency.xml, wo beide Möglichkeiten kombiniert werden:

```
<?xml version="1.0" encoding="UTF–8"?>
<currency>
        <name Anzahl="5">EUR
        </name>
        <name Anzahl="1">USD
        </name>
        <name Anzahl="5">EUR
        </name>
        <name Anzahl="1">GBP
        </name>
        <name Anzahl="5">EUR
        </name>
        <name Anzahl="1">CNY
        </name>
        <name Anzahl="5">EUR
        </name>
        <name Anzahl="1">JPY
        </name>
</currency>
```

Im späteren Programm erlaubt mir diese Datei eine große Menge an möglichen Häufigkeitsverteilungen zwischen den Währungen. Allein dadurch, dass ich das Programm bei der Auswahl eines Wertes nur auf einen Bereich der `ArrayList` zugreifen lasse, kann man aus einer Liste verschiedene Unterfunktionen realisieren. Das mehrfache Vorkommen des Wertes "EUR" sorgt dafür, dass es mit der Begrenzung der Zufallszahlen auf jeweils nur ein Intervall die Möglichkeit "1:1 EUR: eine der anderen Währungen" gibt. Das Setzen des Wertes Anzahl auf 5 bei jedem Euroeintrag erlaubt mir zusätzlich "5:1 EUR: eine der anderen Währungen" anzubieten. Die Nutzung aller Listeneinträge hingegen ergibt ein Verhältnis von: "5:1 EUR: alle anderen Währungen zusammen".

3.3.1 Die Klasse `XMLParser`

Jede XML-Datei muss einmal in die Klasse `XMLParser` eingetragen werden, damit ihre Werte direkt nach dem Programmstart in eine `ArrayList` geparst werden können. Dieser Eintrag steht in der ersten Instanzmethode des XML-Parsers und sieht so aus:

```
public void DOMParserStarter(){
        DOMParser(new File("xmldata/surname.xml"),DataHaendler.surname);
```

Über diesen Eintrag übergibt die Methode `DOMParserstarter` jeweils eine XML-Datei und die dazugehörige ArrayList an die Methode `DOMParser` und die Methode `DOMParserStarter` selbst wird zu Beginn des Programms einmal von der Klasse `ProgrammStarter` aufgerufen.
Der XML-Parser füllt jeden Wert so oft in die entsprechende `ArrayList` ein, wie der jeweilige Wert des Attributes Anzahl dies vorgibt.

3.4 Die Klasse `DataHaendler`

In der Klasse `DataHaendler` werden die `ArrayLists`, die mit den XML-Daten befüllt worden sind, gespeichert. Zusätzlich werden hier die Daten, die für die Verbindung zum Datenbankserver benötigt werden, solange gespeichert, bis der Nutzer eine neue Verbindung startet oder aber das Programm beendet. Auch einige boolesche Variablen, die bei Auftreten von Exceptions in den beiden Klassen, die zum Datenbankserver verbinden, auf `true` gesetzt werden, werden hier global verwaltet. Denn die beiden Verbindungsklassen werden jeweils von mehreren verschiedenen Oberflächen aufgerufen, welche durch Abfragen dieser Variablen richtig auf den Fehler reagieren können. Auch die Liste aller Tabellen des angemeldeten Datenbankusers wird hier immer in ihrer aktuellsten Version bevorratet. Dadurch kann man mit dem Button in der Menübar entsprechend zurück zur Tabellenübersicht, ohne diese Liste neu anfordern zu müssen.
Außerdem werden noch zwei weitere Arten von Variablen in der Klasse `DataHaendler` gespeichert. Zum einem wird in dieser Klasse festgelegt, welche der vorhandenen Funktionen zum Erstellen von Testdaten bei welchem SQL-Datentyp angeboten werden. Somit lässt sich eine Menge Benutzungsfehler von vornherein verhindern. Zum anderen werden hier drei `HashMaps` verwaltet, die man braucht, um die Funktionen zum Befüllen beliebig oft bei einem Befüllungsvorgang verwendbar zu machen. Wird zum Beispiel die Funktion "(house)number" für Hausnummern genutzt, fragt das Programm

nach einer Start- und einer Endzahl für das Intervall, aus dem die Hausnummern erstellt werden sollen. Nun muss es die Möglichkeit geben, diese Funktion für die gleiche Tabelle noch einmal zu verwenden, um zum Beispiel die absolute Anzahl der Einkäufe dieser Kunden zu generieren. Natürlich möchte man dabei zwei unterschiedliche Intervalle. Da man Variablen in Java aber nicht flexibel benennen kann, wird hier die erste eingegebene Startzahl als Wert in eine `HashMap` geschrieben. Als Schlüssel erhält sie ihren Namen "startNumber" plus die Zahl des Schleifendurchlaufs, bei dem sie erstellt wurde. Beim einem zweiten Abfragedialog wird die zweite eingegebene Startzahl in die gleiche Variable gespeichert, aber der Schlüssel im `HashMap`-Eintrag ist nun ein anderer, so dass beim Befüllen zwei unterschiedliche Intervalle als Grundlage dienen.

```
DataHaendler.numberMap.put("endNumber"+j, endNumber);
DataHaendler.numberMap.put("startNumber"+j, startNumber);
```

Genauso wird auch später vorgegangen um die Befüllungsoberfläche dynamisch der Anzahl der zu befüllenden Spalten anzupassen. Dies wird in einem späteren Abschnitt anhand des Sourcecodes genauer ausgeführt.

3.5 Die Klassen zur Kommunikation mit dem Server

Es gibt zwei Klassen, die nur für die Kommunikation mit dem Datenbankserver verantwortlich sind. Die von ihnen benötigten Verbindungsinformationen wie Host, Port, Alias, User und Passwort werden allgemein in der Klasse `DataHaendler` bevorratet und von dort aus zur Verfügung gestellt.
Die Klasse `ConnectionDB2` ist recht klein und allgemein gehalten, sie bekommt ein Statement und einen Typ beim Aufruf übergeben. Dies ist bei dieser Klasse immer ein `select`-Statement, welches über `rset = stmt.executeQuery(statement)` übergeben wird, und so ein `ResultSet` zurückgibt. Das `ResultSet` wird dann je nach Typ, der bei Aufruf der Klasse übergeben wurde, ausgewertet, und die gewonnenen Informationen werden in Variablen der Klasse `DataHaendler` gespeichert.
Erst bei der Verarbeitung der `ResultSets` wird die Klasse je nach vorher übergebenen Typen spezieller. Alle Daten müssen noch in ihr verarbeitet werden, da beim Schließen der Verbindung auch das `ResultSet` geschlossen wird. Deshalb werden alle Daten aufgearbeitet und in die Klasse `DataHaendler` gespeichert und von den anderen Klassen dort abgefragt.
Ist der an `ConnectionDB2` übergebene Typ "table overview", so wird in der DB2-Systemtabelle "syscatTables" nach allen Tabellen des angemeldeten Datenbankusers gesucht. Ist der Typ hingegen "foreign value", so werden aus einer Tabelle die Werte einer einzelnen Spalte für die Funktion "foreign key" oder "foreign value" abgerufen.
Ist der Typ gleich "describe table", so werden von der Klasse `FillTableGui` oder `AlterTableGui` Informationen über eine Tabelle benötigt. Es gibt noch zwei weitere Typen "Columns " und "Primary Key ", sie werden gebraucht um für die Funktionen "foreign key" oder "foreign value" die Spaltennamen für den Nachrichtendialog dieser beiden Funktionen abzufragen.
Nachdem ein einfacher `select` auf die Tabelle ausgeführt wurde, werden über die Methode `ResultSetMetaData` die Namen, Datentypen und Feldbreiten der Spalten so-

wie die Information, ob sie "auto_increment" oder "notnull" sind, ausgelesen. Anschließend wird die Anzahl der schon vorhandenen Tupel durch das Zählen der Zeilen im `ResultSet` berechnet. Danach wird über die Methode `DatabaseMetaData` nach Beziehungen zu anderen Tabellen gesucht.
Die zweite Klasse zur Kommunikation mit dem Datenbankserver `DataUpdater` bekommt nur ein Statement übergeben, da sie lediglich mit `stmt.executeUpdate (statement)` Befehle wie `drop`, `delete`, `create` und `alter` ausführt. Dabei fallen keine zu verarbeitenden Informationen außer den Exceptions an. Die Exceptions werden aber sowieso im Try-Catch-Block verarbeitet.

3.6 Das gemeinsame Layout der Oberflächen

Um die nötige Übersicht im Programmcode zu behalten und den unterschiedlichen Funktionen des Programms gerecht zu werden, wurde für fast jede Funktion eine eigene Oberfläche angelegt. Da ich zuvor keine grafischen Oberflächen mit Java programmiert hatte, habe ich vor Beginn der Umsetzung erst einmal alle benötigten Oberflächen auf Papier skizziert. So konnte ich ein gemeinsames Layout entwickeln und entscheiden, welche der Layoutmöglichkeiten von Java zu meinen Ansprüchen und den Anforderungen an das Programm am besten passen. Aus den verschiedenen möglichen Techniken habe ich mich für Swing und dort für das `GridBagLayout` entschieden. Meine Beweggründe für Swing habe ich schon in Kapitel 2 kurz dargelegt. Innerhalb von Swing gab es wieder verschiedene Umsetzungsmöglichkeiten, nämlich mit einem `JFrame` oder einem `JWindow` zu arbeiten. Ein `JFrame` bietet mir für mein Projekt den Vorteil, dass sich das entstandene Fenster wie ein ganz normales Betriebssystemfenster verhält und so zum Beispiel über einen Minimieren-, Maximieren- und Schließen-Button verfügt.
Da durch die starke Unterteilung des Programms auf insgesamt sieben verschiedene Oberflächen jede einzelne nur wenige Komponenten darstellen musste, bot sich das `GridBagLayout` an. Denn mit diesem konnten die Komponenten sehr gut strukturiert platziert werden. Das `GridBagLayout` legt ein Gitternetz über das gesamte `JFrame`, also das später sichtbare Fenster. Man verteilt die benötigten Komponenten wie Buttons, Lables oder Textfelder auf die durchnummerierten Zellen in dem Gitter. Will man nun in der einen Zeile drei Buttons nebeneinander darstellen, und hat man nirgendwo sonst mehr als drei Komponenten in einer Zeile, so hat man automatisch auf der ganzen Fensterfläche drei Zellen nebeneinander pro Zeile. Will man diese in einer anderen Zeile nicht nutzen, so kann man einzelne Zellen einfach nicht belegen oder anderen Komponenten mehrere Zellen zuweisen.

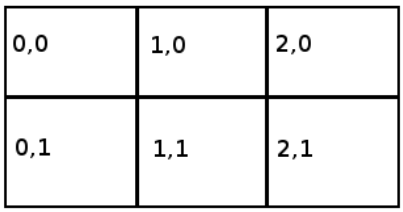

Abbildung 5: Bezeichnungen im `GridBagLayout`

Üblicherweise müsste man nun jeder Komponente aufwendig sechs Werte, über ihren Platz und ihre Größe zuweisen. Um dies zu vereinfachen und pro Komponente in eine Zeile schreiben zu können, benutze ich die Methode `buildConstraints`, die ich so in [CL04] gefunden habe:

```
void buildConstraints
(GridBagConstraints gbc, int gx, int gy, int gw, int gh, int wx, int wy){
        gbc.gridx = gx;
        gbc.gridy = gy;
        gbc.gridwidth = gw;
        gbc.gridheight = gh;
        gbc.weightx = wx;
        gbc.weighty = wy;
}
```

Danach sieht der komplette Code zum Platzieren einer Komponente so aus:

```
buildConstraints(constraints, 0, 5, 2, 1, 0, 20);
constraints.fill = GridBagConstraints.BOTH;
connectionForm.setConstraints(submit, constraints);
pane.add(submit)
```

In der ersten Zeile wird der Komponente die Zelle (0/5) zugewiesen und danach wird festgelegt, dass die Komponente über zwei Spalten und eine Zeile geht. Die beiden letzten Werte sind für die Höhe und die Breite des vorher beschriebenen Feldes. Setzt man sie auf Null, hängen die wirklichen Werte von der Größe des gesamten Fensters und der Anzahl aller Komponenten ab. Setzt man stattdessen andere Werte, so legt man für dieses Feld eine Größe fest, aber auch diese Größe ist noch abhängig von der Gesamtgröße des Fensters. In der vierten Zeile des Codes fügt man die Komponenten dem eigentlichen Container des Fensters hinzu.
Über die Einstellung `setLookandFeel` kann man auswählen, ob die Oberfläche auf jedem System ein bestimmtes Aussehen annehmen soll oder sich jeweils dem Systemaussehen anpasst. Dabei ist aber zu beachten, dass nicht jedes Betriebssystem jedes der in Swing vorhandenen Layouts umsetzen kann. Deshalb habe ich hier die Anpassung an das jeweilige Betriebssystem des Benutzers gewählt.
Hierbei geht es aber auch nur um ein generelles Layout des Fensters also der Maximieren-, Minimieren- und Schließen- Buttons sowie der äußersten Rahmen. Dem sowieso vorhandenen Schließen-Button in der äußersten rechten Ecke des Fensters ha-

be ich noch die Funktionalität zugeordnet, dass das Programm beendet wird, wenn der Nutzer darauf klickt. Alternativ ist auch möglich, dass nur das aktuelle Fenster geschlossen wird. Diese Einstellung benutze ich bei den beiden Informationsfenstern.
Im nächsten Absatz wird die dynamische Größe einiger Oberflächen besprochen. Der restliche Aufbau der Oberflächen erfolgt mit den Swingstandardkomponenten und bedarf hier keiner weiteren Erwähnung, da alle Oberflächen noch einmal in Kapitel 4 ausführlich gezeigt und besprochen werden.

3.7 Die dynamische Größenanpassung von Oberflächen

Es war für mich eine besondere Herausforderung auf die große mögliche Anzahl von Attributen einer Tabelle zu reagieren. Dabei bestand nicht die Möglichkeit, ein großes Feld für die Spalten, in denen jeweils eine Funktion für ein zu befüllendes Attribut gewählt wird, zu reservieren. Denn die Größe der Spalten würde von Swing an den vorhandenen Platz angepasst, so dass bei nur zwei Attributen riesige Komponenten angezeigt werden würden. Auch die Oberfläche so groß wie möglich zu machen und über eine `JScrollPane` einen Scrollbalken zu generieren, falls dies nötig ist, sieht aus dem gleichen Grund für Tabellen mit wenigen Attributen sehr unschön aus. Also bin ich bei der Befüllungsoberfläche zweigleisig gefahren. Erst einmal wird die Größe der gesamten Oberfläche in Bezug zu der Anzahl der Attribute gesetzt. Zusätzlich werden die einzelnen Spalten pro Attribut alle zusammen in einem zusätzlichen `JPanel` erzeugt, welches wiederum in einem `JScrollPane` liegt. Dies geschieht, da ein `JScrollPane` nur ein Objekt als Inhalt akzeptiert. Dieses `JScrollPane` liegt wiederum im gleichen `JPanel` wie die sonstigen Felder und Buttons. Dadurch ist der Gesamteindruck immer gleich, während der Spaltenbereich den Bedürfnissen dynamisch angepasst wird.
Auch die Oberfläche zum Verändern vorhandener Tabellen und die Oberfläche mit den Strukturinformationen einer Tabelle müssen in ihrer Größe dynamisch sein. Denn gerade bei Data Warehouses, die mit dem Snowflake-Schema arbeiten, werden viele Fremdschlüssel verwendet, wodurch für die Oberfläche mit den Strukturinformationen die benötigte Größe des Textfeldes vorher nicht absehbar ist. Doch bei dieser Oberfläche wähle ich einen einfacheren Weg und nehme ein recht großes `JTextArea`. Dieses ist in seiner Größe an der durchschnittlich zu erwartenden Menge an Informationen ausgelegt und wird bei Bedarf um eine Scrollbar erweitert, da es direkt in einem `JScrollPane` liegt und dieses `JScrollPane` die komplette Oberfläche bildet, da keine anderen Komponenten benötigt werden. Für die Oberfläche zum Verändern der Tabelle ist nur das Textfeld mit der Auflistung aller Spalten in einem extra `JScrollPane` untergebracht. Auch hier ist eine recht große Größe vorgegeben und erst, wenn diese nicht reicht, erscheint der Scrollbalken.

3.8 Die interne Struktur der Befüllungsfunktion

Da die Oberfläche zum Befüllen dynamisch aufgebaut werden muss, müssen auch die Namen der Komponenten aus dem dynamischen Bereich über eine `HashMap` verwaltet werden. Da der Aufbau des betroffenen Teils der Oberflächen wieder über eine `for`-Schleife, welche über die Anzahl der Attribute der ausgewählten Tabelle läuft, statt-

findet, haben alle Felder und `JComboBoxen` den gleichen Objektnamen. Aber mithilfe einer weiteren `HashMap` und dem Schleifenzähler kann ich für alle Komponenten einen eindeutigen Schlüssel erzeugen. In den späteren `for`-Schleifen kann das Programm so immer die Informationen über das richtige Attribut abfragen. Dies ist notwenig, da ich beim Generieren der Daten neben der Nutzerauswahl auch den Attributsnamen und den Datentyp benötige.

```
DataHaendler.fillGuiNames.put("attributeLabel"+i, attributeLabel);
DataHaendler.fillGuiNames.put("dataTypLabel"+i, dataTypLabel);
DataHaendler.fillGuiNames.put("functionBox"+i, functionBox);
```

In den späteren Schleifen erstelle ich mir einen neue `JComboBox` und setze sie mit der entsprechenden `JComboBox` aus der Oberfläche gleich, indem ich über den Schleifenzähler den Schlüssel in der `HashMap` nachbilde und so die richtige `JComboBox` aus der `HashMap` abfragen kann. Von dieser Hilfs-`JComboBox` vergleiche ich dann jeweils das `SelectedItem` mit den Namen aller meiner Funktionen, bis ich die Nutzerauswahl darunter finde.

```
String fillType ="functionBox"+j;
JComboBox helperBox =
        (JComboBox)DataHaendler.fillGuiNames.get(fillType);
if(commaCounter==1)
                statement +=",␣";
if (helperBox.getSelectedItem() == "foreign␣key"){
```

Ähnlich verfahre ich an den Stellen, wo ich noch einmal den SQL-Datentyp eines Attributs benötige um eventuelle Hochkommata einzufügen. Für das Erzeugen eines `insert`-Statements nutze ich diesen Weg ebenfalls, da man Spalten auslassen kann und deshalb die Tabellennamen der zu befüllenden Spalten aufgelistet werden müssen.
Das gesamte `insert`-Statement wird wie folgt aufgebaut. In der `for`-Schleife, die erst einmal alle Nutzereingaben auf die Notwendigkeit von Dialogen überprüft, wird schon eine Liste der wirklich zu befüllenden Spalten erzeugt. Dies ist nicht nur wegen der Funktion "leave column out" nötig, sondern auch, da ich eventuelle `auto_increment`-Spalten dem Nutzer nicht zum Befüllen anbiete. Dabei müssen vor allem die Kommata zwischen den Spaltennamen richtig gesetzt werden. Es wird also außerhalb der Schleife eine String `insertIntoColumn` leer initialisiert. Bei jedem Schleifendurchlauf wird dann als erstes abgefragt, ob die Spalte vielleicht gar nicht befüllt werden soll.

```
if (helperBox1.getSelectedItem() != "leave␣column␣out"){
        int helper=j-1;
        insertIntoColumn +=DataHaendler.listAttributes.get(helper);
        if (j!=DataHaendler.columnCount)
                insertIntoColumn +=",␣";
}
```

Für jede andere Funktion wird der aktuelle Attributsname erfragt und in das `insert`-Statement geschrieben. War dies noch nicht der letzte Schleifendurchlauf, so wird ein Komma dahinter geschrieben.
Anschließend wird in der `for`-Schleife, die über die Anzahl aller zu erzeugenden Tupel läuft, ein String `statement` initialisert, der schon den String `insertIntoColumn` verwendet, da dieser zu diesem Zeitpunkt schon fertig ist.

```
statement = "insert␣into␣"+DataHaendler.fillTablename.trim()
                +"("+insertIntoColumn+")" +"␣Values(";
```

Nun muss im Durchlauf der nächsten Schleife für jedes zu befüllende Attribut noch ein Wert eingetragen und wieder die Kommata an der richtigen Stelle gesetzt werden. Ist auch diese Schleife durchgelaufen, wird die Klammer hinter dem letzten Wert geschlossen und der String `statement` der Methode `stmt.executeUpdate(statement)` übergeben. Ein fertiges Statement sieht dann zum Beispiel so aus:

```
insert into PERSON(PERSONID, SURNAME, FRONTNAME, CITY, ZIPCODE, STATE,
COUNTRY, STREET, HOUSENUM) Values(10001, 'Coenen', 'Gisela',
'Stuttgart', 70180, 'BW', 'Deutschland', 'Konrad-Adenauer-Ring', 91)
```

3.9 Die verschiedenen Befüllungsfunktionen

Hier werden die verschiedenen Befüllungsfunktionen erklärt. Dabei geht es darum, welche SQL-Datentypen erzeugt werden können, und welche Realdaten damit abgedeckt werden.

3.9.1 Namen und Wörter

Eine der einfachsten Funktionen ist das Befüllen mithilfe von Wortlisten, die aus den XML-Dateien gewonnen wurden. Dies bietet sich erst einmal vor allem für "Varchar" oder "Char" Attribute an und ist beispielhaft für die geforderten Informationen wie Nachname, Vorname, Straßenname oder Ländername in den Funktionen surname, frontname, streetname und countryname umgesetzt. Dabei werden die Häufigkeiten bei den ersten drei Funktionen allein durch den Wert "Anzahl" in der XML-Datei simuliert. Hingegen geht es bei der letzten Funktion nicht um Häufigkeiten sondern um die Auswahl der Ländernamen nach Kontinenten. Deshalb ist hier das Anzahlfeld immer bei 1, dafür ist die Reihenfolge der Namen in der XML-Datei wichtig. Deutschland steht an erster Stelle und danach kommen alle europäischen Länder, danach folgen die Länder der anderen Kontinente jeweils sortiert. Dadurch bieten sich die Einfüllungsmöglichkeiten: ganze Welt, nur Deutschland oder jeweils ein Kontinent. Je nach Nutzerwahl wird mit Zufallszugriffen auf die gesamte `ArrayList` oder nur ein bestimmtes Intervall dieser Liste oder nur die Stelle 0 für Deutschland gearbeitet.
Dieser Weg wurde für die Funktion Currency noch verfeinert, indem zusätzlich zu dem Zugriff auf immer nur eine Teil der `ArrayList` der Wert "EUR" für Euro immer wieder eingefügt wurde, und zwar mit einer Anzahl von fünf. Dadurch kann dem Nutzer im fertigen Programm die Auswahl zwischen jeder Währung alleinstehend, aber auch jeder Währung in Kombination mit dem Euro, mit verschiedenen Häufigkeitsverteilungen zwischen Euro und der anderen Währung, angeboten werden. Der Nutzer sucht sich dabei wie bei der Countrynamefunktion bequem über ein Dialogfenster mit Dropdownlisten die von ihm benötigte Werteverteilung heraus.
Noch einen Schritt weiter gehe ich bei der Umsetzung der geforderten Abhängigkeit

zwischen Städtenamen, Postleitzahlen und Bundesländern, welche mit den Funktionen “city”, “state” und “zipcode” umgesetzt wurden. Um das einfache Schema der XML-Dateien und des XML-Parsers nicht vergrößern zu müssen, arbeite ich hier mit drei parallel befüllten XML-Dateien. Für diese drei Funktionen gibt es keinerlei Abfragedialog, und in der zweiten `for`-Schleife, welche die Tupel bildet, werden die drei Funktionen alle in der selben `if`-Bedingung abgehandelt.

```
else if(helperBox.getSelectedItem()=="city"
        ||helperBox.getSelectedItem()=="zipcode"
        ||helperBox.getSelectedItem()=="state"){
        if(cityInt==0){
                czsInt = rand.nextInt(DataHaendler.city.size());
        }
        if (helperBox.getSelectedItem()=="city")
                statement += "'"+DataHaendler.city.get(czsInt)+"'";
        if (helperBox.getSelectedItem()=="zipcode")
                if(DataHaendler.listDataTypes.get(j-1)=="VARCHAR"
                        ||DataHaendler.listDataTypes.get(j-1)=="CHAR")
                        statement += "'"
                                +DataHaendler.zipCode.get(czsInt)+"'";
                else
                statement += DataHaendler.zipCode.get(czsInt);
        if (helperBox.getSelectedItem()=="state")
                statement += "'"+DataHaendler.state.get(czsInt)+"'";
        if(cityInt==0)
                cityInt++;
}
```

Die Integervariable `cityInt` wird außerhalb der inneren `for`-Schleife und innerhalb der äußeren `for`-Schleife, also immer vor dem Erstellen eines neuen Tupels, mit dem Wert 0 initialisiert. Findet nun die innere `for`-Schleife eine der drei Funktionen als Nutzerauswahl, wird eine Zufallszahl erzeugt und in die Variable `czsInt` gespeichert. Mit dieser Zahl wird aus der entsprechenden Liste der Wert an dieser Stelle der Liste ausgelesen, danach wird `cityInt` auf 1 gesetzt. Hat der Nutzer nun weitere dieser drei Funktionen ausgewählt, steht für diesen Durchlauf `cityInt` schon auf 1 und für das Holen weiterer Werte wird die gleiche Zufallszahl genutzt wie in der ersten Funktion. Dadurch, dass die drei XML-Dateien parallel befüllt worden sind, bleibt hier die Beziehung zwischen den Werten erhalten. Vor dem nächsten Schleifendurchlauf wird `cityInt` dann wieder auf 0 gesetzt, so dass eine neue Zufallszahl erzeugt wird. Bei diesen Funktionen habe ich auf eine mehrmalige Verwendbarkeit verzichtet, weil ich für ein Data Warehouse keinen Sinn darin sehe. Es wäre aber jederzeit nach dem sonst genutzten Schema einfach zu implementieren. In der Funktion “zipcode” wird noch überprüft, ob sie vielleicht in ein Feld des Typs “Char” oder “Varchar” gefüllt werden soll, dann werden die Hochkommata erzeugt.

3.9.2 Zahlen

Für alle möglichen ganzzahligen SQL-Datentypen beziehungweise alle benötigten ganzzahligen Daten steht die Funktion "(house)number" zur Verfügung. Dieser Funktion kann man eine Start- und eine Endzahl für ein Intervall, das diese beiden Zahlen einschließt, übergeben. Dies geschieht in einem Dialog, nachdem man diese Funktion ausgesucht hat. Damit diese Funktion mehrmals mit verschiedenen Intervallen verwendbar ist, werden diese beiden Werte in eine `HashMap` gespeichert, zusammen mit einem reproduzierbaren Schlüssel auf Basis des Zählers der `for`-Schleife. Der zugehörige Code wurde im Abschnitt über die Klasse `DataHaendler` gezeigt. Denn die `for`-Schleife, die die Dialoge auslöst und die zweite `for`-Schleife, welche die Werte später erstellt, laufen beide über die Anzahl der zu befüllenden Attribute der Tabelle. Da sie beide bei jedem Schleifendurchlauf von oben nach unten eine Nutzerauswahl abarbeiten, sind sie im gleichen Durchlauf auch immer bei dem gleichen Attribut. Vor dem Befüllen in die `HashMap` wird überprüft, ob der Nutzer die beiden Werte in verkehrter Reihenfolge eingegeben hat, was in der späteren Funktion einen Fehler hervorrufen würde. Um leere Felder im Dialog abzufangen, werden die Defaultwerte 1 und 9 gesetzt.

```
if (endNumber<startNumber){
        int helper = startNumber;
        startNumber = endNumber;
        endNumber = helper;
```

Dazu muss man wissen, dass nach dem Drücken des "submits"-Buttons auf der `FillTablegui` nacheinander zwei `for`-Schleifen ablaufen, die den gleichen Zähler benutzen, nämlich die Anzahl der Attribute der zu befüllenden Tabelle. Die erste `for`-Schleife schaut sich für jedes Attribut die Auswahl des Nutzers an und startet wenn nötig einen Dialog um Werte, die verwendet werden sollen, abzufragen. Der Zähler dieser Schleife ist für jedes Attribut identisch mit dem Zähler der zweiten Schleife, die die eigentlichen Werte zum Einfügen erzeugt. Dadurch kann man dort den in der `HashMap` verwendeten Schlüssel wiederherstellen und so den Wert, der für diese Benutzung der Funktion vom Nutzer ausgesucht wurde, aus der `HashMap` abfragen.

```
String sNumber ="startNumber"+j;
int startNumber = (Integer)DataHaendler.numberMap.get(sNumber);
String eNumber ="endNumber"+j;
int endNumber = (Integer)DataHaendler.numberMap.get(eNumber);
```

Die Funktion "id" ist über ihren Namen selbsterklärend. Die Ids starten, falls schon Tupel in der Tabelle vorhanden sind, bei der Anzahl dieser Tupel plus 1. Auch dieses Vorgehen kann zu Dopplungen führen, aber doch zu erheblich weniger als ein Start bei 1. Auch das Abfragen des höchsten Wertes der schon vorhanden Ids würde zu keiner besseren Lösung führen, da zwischen den Tupeln gelöscht worden sein könnte. Dies ist nur interessant, wenn eine solche Id alleiniges Primattribut ist. Auch die Funktion "id" ist mehrfach verwendbar, aber durch den festen Startwert hat in jedem Tupel jede verwendete Id den gleichen Wert.

Hier ist auch nur zu erwähnen, dass man die Funktion "zipcode" natürlich auch zum Befüllen von zum Beispiel Integerfeldern benutzen kann kann.
Als letztes ist noch die Frage, wie das Programm Gleitkommazahlenfelder befüllen

kann. Hier gibt es die Möglichkeit mit der Funkton “own data” zu arbeiten. Zum Beispiel könnte der Nutzer die Werte 40, 40.5, 40.125, 40, 40.5, 40.5 eingeben, um somit verschiedene Discountgruppen von Geschäftskunden zu simulieren. Außerdem kann der Nutzer noch über die Funktion “foreign value” eine Spalte für Gleitkommazahlen befüllen, vorausgesetzt dass die Werte in der anderen Spalte valide sind. Auch die Funktion “(house)number” ist für die SQL-Datentypen “Double”, “Float” und “Real” verwendbar, solange man nicht die Grenzen für den höchsten möglichen Wert überstreitet, denn dieser ist durch die Nachkommastellen entsprechen niedriger als bei einem Integerwert.

3.9.3 Datum und Zeit

Für die drei SQL-Datentypen “Date”, “Time” und “Timestamp” gibt es einen gemeinsamen Funktionsablauf, bei dem mit Randomzahlen aus den jeweils gültigen Intervallen und einigen Zeichen ein Wert im gültigen Format entsteht. Einzig bei den Jahreszahlen kann der Nutzer ein eigenes Intervall festlegen.
Bei dieser Funktion war darauf zu achten, dass man entweder mehrere unabhängige einstellige Integerwerte in bestimmten zulässigen Kombinationen erzeugt, oder aber Werten, die kleiner als 10 sind, eine 0 vorsetzt, um das gültige Format des SQL-Datentyps erzeugen zu können. Zusätzlich musste man die gültige obere Grenze des Intervalls für einen Tag an der getroffenen Auswahl für den Monat orientieren. Das verhindert Datumsangaben, die es nicht geben kann, und welche auch von der DB2-Datenbank deshalb abgelehnt werden würden.

```
int dayInt = 0;
        String day ="";
        if("02".equals(month))
                dayInt = 1+rand.nextInt(28);
        else if("04".equals(month)||"06".equals(month)
                ||"09".equals(month)||"11".equals(month))
                dayInt = 1+rand.nextInt(30);
        else
                dayInt = 1+rand.nextInt(31);
        if(dayInt<=9)
                day = "0";
        day +=dayInt;
```

3.9.4 Eigene Daten

Um kleinere Mengen eigener Daten temporär für einen Befüllungsvorgang nutzen zu können, gibt es die Funktion “own data”. Wählt man sie aus, wird man solange nach eigenen Einzelwerten gefragt, bis man Abbruch oder bei leeren Feld “Ok” drückt. All diese Werte werden in eine `ArrayList` gespeichert. Dadurch hat man durch Mehrfachnennung eines Wertes wieder die Möglichkeit, Häufigkeitsverteilungen auf diesen Werten zu simulieren. Auch diese Funktion ist mittels Speicherung der `ArrayList` in einer `HashMap` mehrfach mit verschiedenen Wertelisten zu benutzen.
Weitere Möglichkeiten, eigene Daten ins Programm mit oder ohne Hilfe des Sourcecodes

einzubeziehen, werden im letzten Abschnitt dieses Kapitels besprochen.

3.9.5 Schon vorhandene Daten aus der Datenbank nutzen

Eine der Hauptaufgaben für dieses Programm ist es, Fremdschlüsselattribute valide befüllen zu können. Dafür gibt es eine eigene Funktion "foreign key". Diese wird überall angeboten, so dass man sie Verwenden kann und erst hinterher die Beziehung erstellt. Dabei kann man noch aus allen Tabellen wählen, bekommt aber nur die `Primary Key`-Attribute der fremden Tabellen zu sehen. Dies geschieht nacheinander über zwei Dialogfenster. Im ersten wählt man eine Tabelle und im zweiten dann die richtige Spalte. Hat man sich im ersten Fenster vertan, muss man abbrechen und den Dialogvorgang für alle Auswahlen nochmal bestreiten. Aber da man in der Informationsoberfläche alle schon vorhandenen Beziehungen sehen kann, kann man hier die richtige Auswahl des Nutzers erwarten. Will er hingegen später erst eine Beziehung erzeugen, so sollte er wissen, dass die Herkunftsspalte `Primary Key` sein muss. Ein Beschränken der `ForeignKey`-Spalten auf die alleinige Auswahlmöglichkeit dieser Funktion ist für mich die schon öfter erwähnte totale Einschränkung des versierten Nutzers. Denn dann ist er gezwungen, alle Tabellen erst einmal mit allen Beziehungen anzulegen und erst danach mit dem Befüllen zu beginnen. Dies sollte in einer Entwicklung sicher auch der richtige Weg sein, aber ich habe selber bei diesem Projekt gemerkt, wie oft sich eine vorher gewählte Struktur den Gegebenheiten anpassen muss. Deshalb hat man auch bei einem Attribut, welches mit Fremdschlüsseln befüllt werden muss, alle Möglichkeiten, die der SQL-Datentyp erlaubt. Kennt man die Werte des späteren Primattributs, auf das hier Bezug genommen werden soll, so kann man diese vorab mit der "own data"-Funktion simulieren. Natürlich muss dann jeder dieser Werte auch in der späteren `Primary Key` Spalte vorkommen.

3.9.6 Spezialfunktionen

Die Funktion "leave colum out" wurde schon in einem vorherigen Abschnitt, so wie beim Erstellen des Insert-Statements erwähnt, womit ihre Funktionalität erklärt sein dürfte. Außerdem gibt es die Funktion "null" die den Wert "null" erzeugt. Vorher sollte man wie schon erwähnt in der Informationsseite nachschauen, ob die Spalte vielleicht keine null-Werte erlaubt.
Zu erwähnen ist hier noch der Umgang mit `Primary Keys`. Diese werden von mir nicht vor dem Einfügeversuch speziell überprüft, da der Datenbankserver jedesmal automatisch überprüft, ob der `Primary Key` schon einmal benutzt wurde. Stattdessen fange ich nur die Exceptions, die dieser Fehler wirft, gesondert ab, ohne sie dem Nutzer zu zeigen. Ich merke mir stattdessen in einem Fehlerzähler die Gesamtanzahl der Fehler und erhöhe die geforderte Anzahl der Tupel entsprechend um 1. Dadurch wird bei genügend möglichen `Primary Keys` trotzdem der Nutzervorgabe über die Anzahl zu erstellender Tupel Folge geleistet. Für den Fall, dass es nicht genügend verschiedene `Primary Keys` gibt, sind zwei Abbruchbedingungen implementiert.

```
int helpFloat = errorCounter * 2;
if(helpFloat>rows)
        break;
if(errorCounter>500)
        break;
```

Zum einen bricht der Befüllungsvorgang ab, wenn die Anzahl der Fehler halb so groß ist wie die Anzahl der zu erzeugenden Tupel, wobei zu berücksichtigen ist, dass diese Anzahl der Tupel pro Fehler ebenfalls um 1 erhöht wird. Da es auch mit dieser Abbruchbedingung bei großen Mengen an Tupeln entsprechend lange bis zum Abbruch dauern kann, wird zum anderen nach 500 Fehlern insgesamt abgebrochen. Hierfür habe ich verschiedene Kombinationen der Abbruchbedingungen getestet, wobei es darum ging, zwischen möglichst vielen gefundenen `Primary Keys` und möglichst wenigen Fehlversuchen zu skalieren.

3.10 Erweiterbarkeit durch den Nutzer

Da es in dieser Arbeit nicht darum gehen konnte, für irgendwen alle benötigten Funktionen bereit zu stellen sondern nur Möglichkeiten zur Funktionalität aufzuzeigen, wurde versucht, die Struktur so einfach wie möglich zu halten. Dadurch wird es späteren Nutzern ermöglicht, Daten nach eigenen Bedürfnissen hinzuzufügen. Bei der Struktur der XML-Dateien ist dies sehr einfach, da diese Struktur sehr flach geblieben ist, wie im Abschnitt XML schon beschrieben. Will ein Nutzer nun eigene Funktionen nach dem selben Schema hinzufügen, so braucht er neben einer validen XML-Datei nur wenige Änderungen im Quellcode vorzunehmen.
Nachdem er die XML-Datei erstellt hat, fügt er der Klasse `DataHaendler` noch eine weitere `ArrayList` hinzu. Danach trägt er beides in die Methode `DOMParserStarter` des XML-Parsers nach folgenden Schema ein:

```
public void DOMParserStarter(){
DOMParser(new File("xmldata/surname.xml"),DataHaendler.surname);
.
.
}
```

Nun sollte sich der Nutzer überlegen, welchen programminternen Namen er der Funktion gibt und diesen ebenfalls in der Klasse `DataHaendler` eintragen. Dies geschieht in dort angelegten `Arrays` je nach SQL-Datentyp, der mit der Funktion später befüllt werden kann, wie im Abschnitt zur Klasse `DataHaendler` erwähnt.
Nun muss der Nutzer noch in der Klasse `FillTablesGui` die eigentliche Funktionalität implementieren, also die Aktionen, die ausgeführt werden sollen, wenn der Nutzer die neue Funktion auswählt.
Im einfachsten Fall wurde über das Anzahlattribut der Daten in der XML-Datei schon eine Häufigkeitsverteilung ausgebildet, und die Werte sind nur einem SQL-Datentyp zugeordnet. Dann braucht man keine Abfragedialoge mit dem Nutzer und hat so keine Probleme mit falschen Eingaben. Daraus würde ein einfaches Einfügen in den Ablauf der zweiten `for`-Schleife, die über die Anzahl der zu befüllenden Spalten nach diesem Schema läuft:

```
else if (helperBox.getSelectedItem() == "new function"){
        int newFunctionNames =
                rand.nextInt(DataHaendler.newFunction.size());
        statement += "'"+DataHaendler.surname.get(surname).trim()+"'";
}
```

Will man aus dem neu eingefügten Datensatz hingegen verschiedene Häufigkeitsverteilungen verfügbar machen, so muss man in der ersten `for`-Schleife, die über die Anzahl der zu befüllenden Attribute läuft, einen Abfragedialog mit dem Nutzer implementieren. Das Wichtigste ist hierbei, sich gute Defaultwerte zu setzen, um mögliche falsche Nutzereingaben konsequent abzufangen und nach Möglichkeit mit richtigen Werten zu ersetzen. Um die Funktion mehrmals innerhalb eines Befüllungsvorganges verwenden zu können, sollte man sich die Benutzerwahl mit einem eindeutigen Schlüssel in eine der schon vorhandenen `HashMaps` je nach Java-Datentyp der Nutzereingabe sichern. Diesen Schlüssel generiert man mithilfe des Schleifenzählers, wie schon vorher gezeigt wurde. All dies ist analog zu schon vorhandenen Funktionen im Programm.
Das eigentliche Generieren von Werten aus den vorgefertigten Listen findet dann in der zweiten `for`-Schleife, die über die Anzahl der zu befüllenden Spalten geht, statt. Hier muss man nun je nachdem die Nutzerauswahl wieder aufgreifen. Zusätzlich sollte man sich überlegen, ob die Funktion für SQL-Datentypen geeignet ist, welche beim Einfüllen Hochkommata vorsieht oder nicht. Braucht man die Funktion für verschiedene SQL-Datentypen mit und ohne Hochkommata, so muss man zwei verschiedene Statementfragmente vorbereiten. Auch dies geschieht alles analog zu schon implementierten Funktionen.
Soll das Programm weiterhin von verschiedenen Nutzern genutzt werden, so sollte die neue Funktion einen Button auf der Help-Oberfläche erhalten.

4 Programmablauf

Wie schon in Kapitel 3 erwähnt, gibt es im fertigen Programm sechs Oberflächen, die als `JFrames` verwirklicht sind. Daneben wird über `JOptionPanes` ein ständiger Dialog mit dem Nutzer geführt. In diesem Kapitel wird jede Oberfläche kurz vorgestellt und erklärt. Zusätzlich werden beispielhaft einige der Dialogfenster gezeigt. Außerdem wird das Ergebnis eines Befüllungsvorgangs besprochen.

4.1 Die Datenbankverbindung

Abbildung 6: Die Oberfläche zur Datenbankverbindung

Die hier abgebildete Oberfläche dient zur Eingabe der Verbindungsdaten. Diese werden solange vom Programm vorgehalten, bis man sich auf einen anderen Server verbindet. Ganz oben sind die normalen drei Buttons zum Minimieren, Maximieren und Beenden des Programms. Darunter sind fünf Eingabefelder für die Verbindungsdaten und ein "Submit"-Button. Wird beim Versuch der ersten Verbindung eine Exception abgefangen, so wird diese dem Nutzer als Nachricht angezeigt, während die Oberfläche die schon eingegebenen Daten weiter bereithält.

4.2 Die Tabellenübersicht

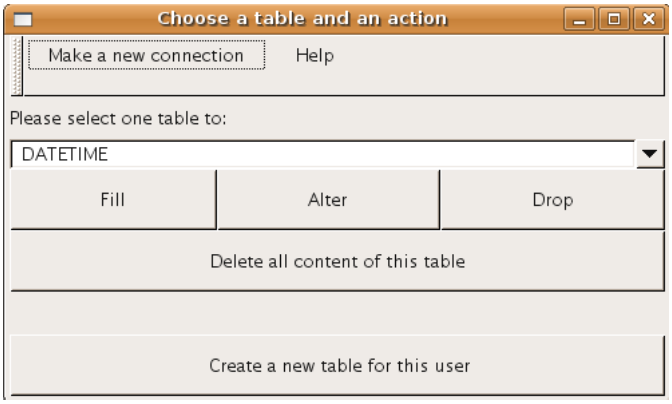

Abbildung 7: Die Tabellenübersicht des angemeldeten Nutzers

Die hier abgebildete Oberfläche "Tabellenübersicht" ist der Knotenpunkt des gesamten Programms. Von ihr kommt der Nutzer mit nur einem Klick überall hin, und er kommt, mit Ausnahme von der Verbindungsoberfläche, von jeder anderen Oberfläche mit einem Klick hierhin zurück. Am oberen Bildrand sieht man die Menübar des Programms. Diese enthält die Buttons: "Make a new connection" und "Help". Bei den anderen Oberflächen wird man noch einen dritten Button sehen, der "Choose another table" heißt. Ansonsten ist die Menübar immer gleich und wird hier ein einziges Mal erklärt. Der "Choose another table"-Button führt von den anderen Oberflächen zurück zur Tabellenübersicht. Somit kann man jeden begonnenen Vorgang abbrechen, wenn man es sich anders überlegt hat. Mit dem "Make a new connection"-Button kommt man von jeder Oberfläche wieder auf die Verbindungsoberfläche um die Datenbank oder den Datenbankuser zu wechseln. Der "Help"-Button ist selbst erklärend, und öffnet ein weiteres Fenster mit Informationen über alle Buttons und Funktionen des gesamten Programms. Dieses Fenster wird am Ende des Kapitels besprochen.
Nicht nur, wenn man eine andere Oberfläche verlässt, kommt man zur Tabellenübersicht zurück, sondern auch, wenn eine Aktion erfolgreich verlaufen ist, wird man auf die Tabellenübersicht zurückgeleitet. Dies hat den Grund, dass man nur von hier aus neue Aktionen starten kann, denn hier wird in einer "Dropdownliste" immer eine aktualisierte Tabellenübersicht für den angemeldeten Nutzer bereitgehalten. Man kann hier eine der Tabellen des angemeldeten Nutzers auswählen und dazu die Aktion, die man durchführen möchte.
Als Aktion stehen "Drop", "Alter", "Fill" und "Delete" zur Verfügung. Wählt man "Alter" oder "Fill", so verlässt man die Tabellenübersicht und öffnet eine neue Oberfläche. Wählt man hingegen "Drop" oder "Delete" so kommt ein Hinweisfenster und fragt nach, ob man diese Aktion wirklich durchführen will um Missgeschicke zu verhindern. Mit "Delete" hat man die Möglichkeit sämtliche Inhalte aus einer Tabelle zu löschen, mit "Drop" löscht man die Tabelle selbst.
Als letztes gibt es noch den "Create a new table"-Button, auch hier verlässt man die aktuelle Oberfläche.

4.3 Eine neue Tabelle erstellen

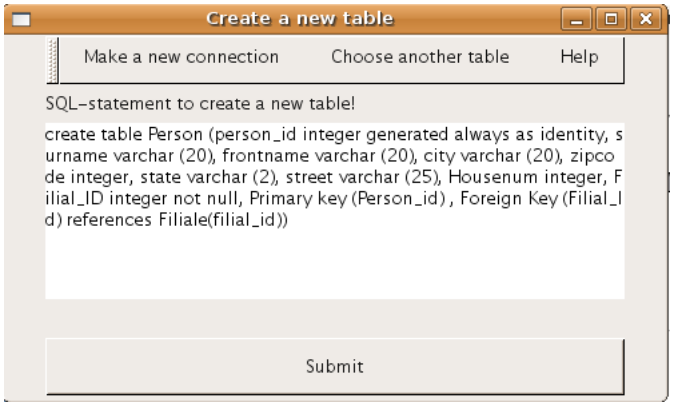

Abbildung 8: Die Oberfläche zum Anlegen neuer Tabellen

Neben der obligatorischen Menübar gibt es auf dieser Oberfläche noch ein Textfeld, in das man ein `create`-Statement schreiben kann und dieses über den “Submit”-Button der Klasse `ConnectionDB2` übergibt, welche es zum Datenbankserver weiterleitet. Gibt es keine Exception, gelangt man direkt zur um die neue Tabelle aktualisierten Tabellenübersicht zurück. Ansonsten wird dem Nutzer die Message der Exception gezeigt, und man hat die Möglichkeit, seine Eingabe zu bearbeiten und noch einmal abzuschicken oder über die Menübar zurück zur Tabellenübersicht zu gelangen.

4.4 Tabellenstrukturinformationen

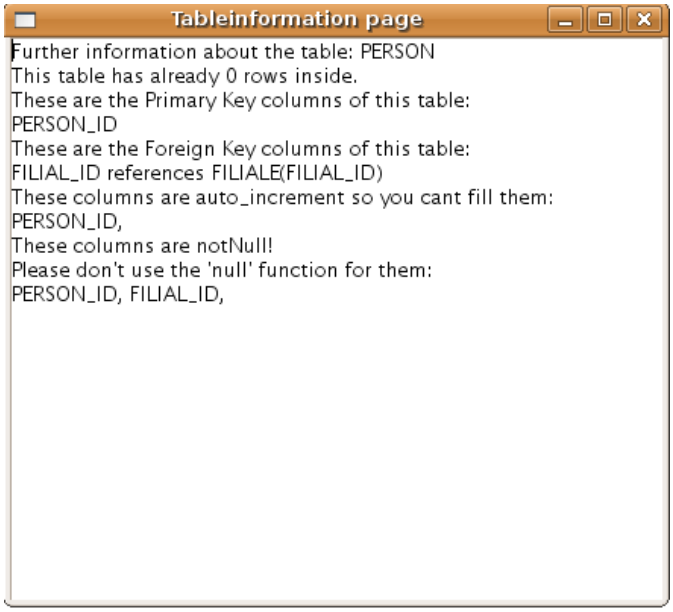

Abbildung 9: Die Oberfläche für Tabellenstrukturinformationen

Die Oberfläche mit den Tabellenstrukturinformationen enthält nur ein einziges Textfeld, welches bei Bedarf um eine Scrollbar erweitert wird. Diese Oberfläche ist nur aus zwei anderen Oberflächen über einen Button erreichbar und öffnet sich als zusätzliches Fenster. Anders als bei den anderen Oberflächen beendet die Benutzung des “x”-Buttons rechts oben nicht das ganze Programm, sondern schließt nur sich selber.
Verlässt man die Oberfläche, von der man die Informationsseite geöffnet hat, so bleibt diese wie später auch die Hilfe trotzdem weiterhin sichtbar. So kann man sich die Informationsseiten mehrerer Tabellen gleichzeitig anzeigen lassen und so alle Beziehungen zwischen den Tabellen nachvollziehen. Will der Nutzer die Tabelleninformationen allerdings während des Befüllungsvorganges benutzen, so muss er sie öffnen, bevor er den “Submit”-Button drückt. Ist der Nachfragedialog gestartet, hat man nicht mehr die Möglichkeit die Informationsseite oder die Hilfe zu öffnen.
Von oben nach unten werden, sofern es sie zu dieser Tabelle gibt, folgende Informationen bereitgestellt. Zuerst erfährt der Nutzer, ob schon Tupel, und wenn ja wie viele, in der Tabelle sind, so dass er sie nach Bedarf vorher löschen kann. Danach werden die Attribute gezeigt, die den `Primary Key` bilden, falls einer gesetzt wurde. Es folgen einzeln die Attribute, die mit Fremschlüsseln befüllt werden müssen, zusammen mit den Tabellennamen und den Attributsnamen des zugehörigen Primärschlüssels. Danach wird umgekehrt beschrieben, welche anderen Tabellen auf die Primattribute der aktuellen Tabelle referenzieren. Sind Attribute der Tabelle `auto_increment`, so werden sie gar nicht erst als zu befüllende Spalte angezeigt und deshalb hier erwähnt, damit der Nutzer sie nicht sucht. Als letztes werden noch sämtliche Spalten erwähnt, die nicht mit dem Wert “null” befüllt werden dürfen, damit der Nutzer dort keine Fehler hervorruft.

4.5 Eine bestehende Tabelle ändern

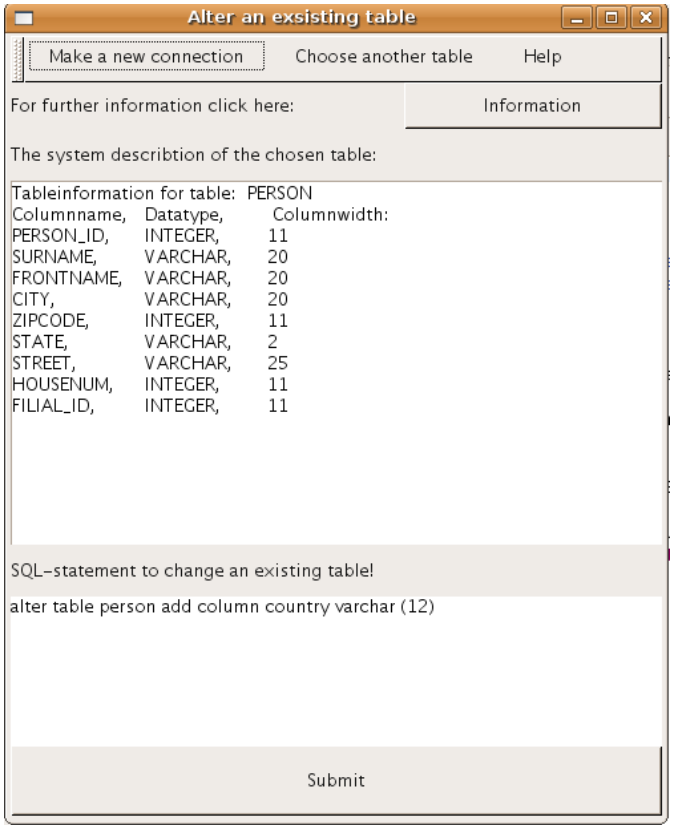

Abbildung 10: Die Oberfläche zum Ändern vorhandener Tabellen

Diese Oberfläche beinhaltet ein großes Textfeld, welches die Struktur der Tabelle zeigt, die man ändern möchte. Das Textfeld ist in einem `JScrollPane`, so dass auch bei Tabellen mit ungewöhnlich vielen Attributen alle Informationen verfügbar sind. Neben dem Tabellennamen erhält man eine Übersicht aller Attribute sowie ihrer Datentypen und Feldbreiten. Alle weiteren Informationen erhält man nur über die Tabellenstrukturinformationen. Um diese angezeigt zu bekommen muss man den "Information"-Button benutzen. Somit hat man alle Informationen, um im unteren Textfeld ein `alter`-Statement zu formulieren, welches man mit dem "Submit"-Button bestätigt. Ist das Statement SQL-konform, so kommt man automatisch zur Tabellenübersicht zurück. Ansonsten kann man die Message der Exception sehen, und erhält die Möglichkeit sein Statement erneut zu bearbeiten oder zur Tabellenübersicht zurückzukehren. Natürlich könnte man über die Menübar auch zur Verbindungsoberfläche zurückkehren.
Mit der Nutzereingabe im unteren Feld wird die Tabelle "PERSON", wie man im nächsten Bild sieht, um ein weiteres Attribut erweitert.

4.6 Eine bestehende Tabelle befüllen

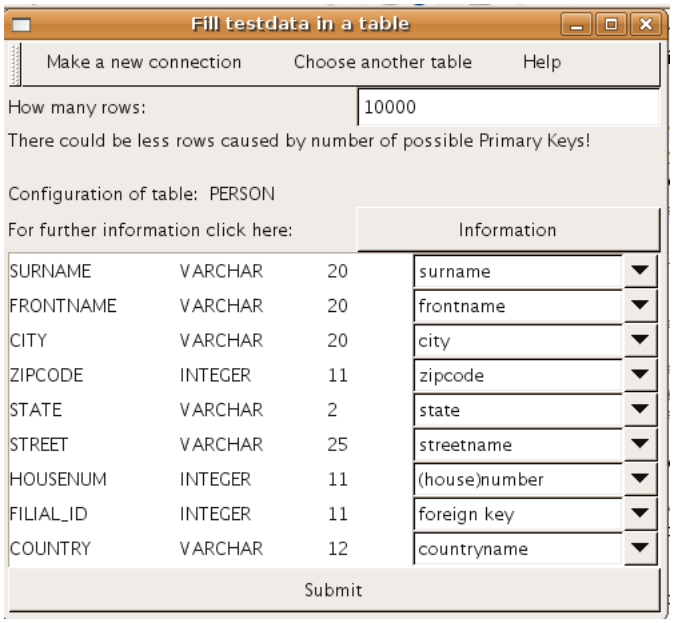

Abbildung 11: Die Oberfläche zum Befüllen vorhandener Tabellen

In das obere Textfeld muss der Nutzer die gewünschte Anzahl an Tupeln schreiben. Darunter erhält er eine Auflistung aller Attribute der Tabelle, mit den gleichen Informationen wie schon beim Ändern von Tabellen. Die Spalte “PERSON_ID” wird nicht angezeigt, da sie `auto_increment` ist. Ebenso wie bei der Oberfläche zum Verändern von Tabellen wird die Größe der Oberfläche dynamisch der ihrer Anzahl von Attributen angepasst. Dahinter steht jeweils eine Dropdownliste mit Befüllungsfunktionen vorsortiert nach dem Datentyp des jeweiligen Attributs. Dadurch soll ein falsche Auswahl durch den Nutzer verhindert werden. Nach dem für alle Attribute eine Funktion ausgewählt wurde, bestätigt man seine Auswahl mit dem “Submit”-Button. Nun fragt das Programm intern jede Auswahl einmal ab und entscheidet, ob es für diese Funktion Zusatzinformationen vom Nutzer benötigt.

4.6.1 Der Befüllungsvorgang

Drückt man mit den Einstellungen von Abbildung 8 den “Submit”-Button, so werden nach Bedarf Dialogfenster geöffnet, damit der Nutzer die benötigten Daten näher spezifizieren kann. Ob ein Dialog nötig ist, wird vom Programm von oben nach unten über die Auswahlen des Nutzers geprüft. Im folgenden zeige ich beispielhaft für alle diese möglichen Dialoge die beiden, die bei der getroffenen Auswahl in Abbildung 8 auftreten:
Die erste zusätzliche Nutzereingabe benötigt das Programm um das Intervall zu erfragen, aus dem die Hausnummern erstellt werden sollen. Die Funktion “(house)number” ist generell zum Erstellen aller Nummern/ Zahlen aus Intervallen verwendbar. Um Fehler durch den Nutzer zu verhindern, wird wie schon in Kapitel 3 erwähnt mit Defaultwerten gearbeitet. Bei dieser Funktion sind die Defaultwerte 1 und 9.

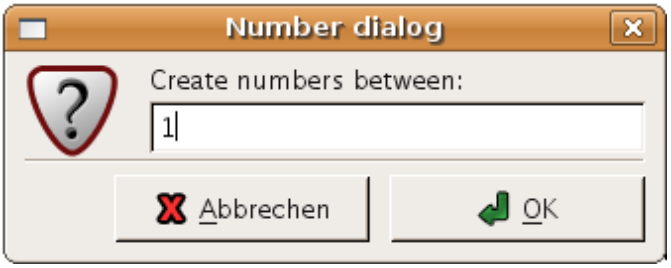

Abbildung 12: Untere Grenze des Intervalls abfragen

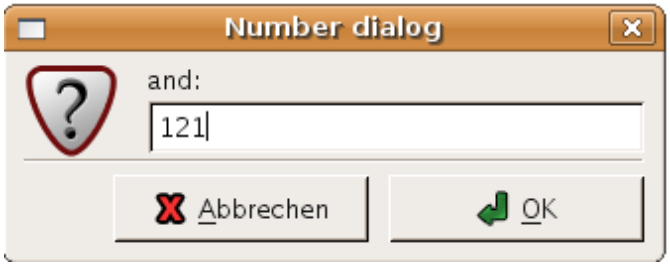

Abbildung 13: Obere Grenze des Intervalls abfragen

Abbildung 14: Die Tabelle aussuchen, die die Fremdschlüsselspalte enthält

Als nächstes soll die Spalte “FILIAL_ID” mit Fremdschlüsseln aus der Tabelle “FILIALE” befüllt werden. Dazu muss der Nutzer in einem ersten Dialogfenster die Tabelle und in einem zweiten Fenster dann aus den Primattributen die richtige Spalte auswählen.

Abbildung 15: Die Spalte auswählen, aus der die Fremdschlüssel entnommen werden

Abbildung 16: Ländernamendialog

Danach startet ein dritter Dialog um die Ländernamen in der Spalte “COUNTRY” zu erzeugen. Hierbei kann der Nutzer aus einer Dropdownliste wählen, und muss dies über “Ok” bestätigen. Ein Abbruch des Dialogs führt zum Verlassen der aktuellen Oberfläche zurück zur Tabellenübersicht.
Im Falle der Ländernamen habe ich einige Möglichkeiten für den Nutzer implementiert um die Möglichkeiten des Programms zu zeigen. Der Nutzer kann entweder mit Ländernamen der ganzen Welt oder aber eines Kontinents arbeiten. Zusätzlich gibt es noch die hier gewählte Option “nur Deutschland”.

Nun werden programmintern nacheinander die Insert-Statements erzeugt und eingefügt, solange bis ein Fehler auftritt. Basiert der Fehler auf doppelten `Primary Keys` oder Ids oder der Überschreitung der Feldlänge, so hat das Programm gewisse Routinen um die Befüllung mit neuen Werten weiter zu versuchen. Erst, wenn sich die Fehler häufen, bricht das Programm ab. Ist zum Beispiel der Wert von “CITY” für den Varchar auf 12 gesetzt und das vorhandene Mönchengladbach würde ausgewählt, führt das zu einer Überschreitung der Feldlänge. Da es aber trotzdem genügend mögliche Tupel gäbe, die keinen Fehler erzeugen, bricht das Programm hier nicht ab. Stattdessen wird die Anzahl der zu erzeugenden Tupel um eins erhöht, da die Schleife den Fehlversuch mitzählt. Ebenfalls wird der zugehörige Fehlerzähler inkrementiert. Erst, wenn dieser Fehlerzähler eine gewisse Höhe erreicht hat, wird der Befüllungsvorgang gestoppt. Dies muss sein, da die mit Städtenamen zu befüllende Spalte auch die Feldbreite 2 haben könnte, so dass nie eine valider Eintrag erzeugt werden würde. Bei `Primary Keys` verhält es sich genauso, da es vielleicht weniger mögliche gültige Schlüssel gibt, als die Anzahl der geforderten Tupel. Bei Ids hingegen wird nie abgebrochen, da es theoretisch immer noch einen größeren Integerwert gibt. Zwar hat auch der Datentyp Integer eine obere Grenze, doch die liegt bei über 99 Milliarden. Um bei bereits befüllten Tabellen nicht bei 1 zu starten, und den ersten gültigen Eintrag vielleicht erst bei 10000 zu finden, startet die Id-Funktion bei der Anzahl der schon vorhandenen Tupel +1. Dies ist fast immer ein Zeitgewinn, selbst wenn schon mehr Tupel in der Tabelle waren und sie durcheinander gelöscht wurden.
Das Ergebnis sind im besten Fall, in dem vom Nutzer keine Fehler gemacht wurden, welche das Programm nicht behebt oder nicht beheben kann, 10000 Tupel dieses Schematas:

```
insert into PERSON(SURNAME, FRONTNAME, CITY, ZIPCODE, STATE, STREET
, HOUSENUM, FILIAL_ID, COUNTRY) Values('Bayer', 'Kerstin', 'Hamburg'
, 22305, 'HH', 'Pfarrstraße', 85, 345, 'Deutschland')
insert into PERSON(SURNAME, FRONTNAME, CITY, ZIPCODE, STATE, STREET
, HOUSENUM, FILIAL_ID, COUNTRY) Values('Breuer', 'Hannah', 'München'
, 81249, 'BY', 'Poststraße', 43, 335, 'Deutschland')
insert into PERSON(SURNAME, FRONTNAME, CITY, ZIPCODE, STATE, STREET
, HOUSENUM, FILIAL_ID, COUNTRY) Values('Albrecht', 'Tim', 'Bremen'
, 28777, 'HB', 'Pfarrstraße', 41, 39, 'Deutschland')
insert into PERSON(SURNAME, FRONTNAME, CITY, ZIPCODE, STATE, STREET
, HOUSENUM, FILIAL_ID, COUNTRY) Values('Müller', 'Victoria', 'Leipzig'
, 04157, 'SN', 'Vereinsstraße', 74, 517, 'Deutschland')
insert into PERSON(SURNAME, FRONTNAME, CITY, ZIPCODE, STATE, STREET
, HOUSENUM, FILIAL_ID, COUNTRY) Values('Westermann', 'Markus', 'Bremen'
, 28777, 'HB', 'Harfnerstraße', 75, 75, 'Deutschland')
.
.
.
insert into PERSON(SURNAME, FRONTNAME, CITY, ZIPCODE, STATE, STREET
, HOUSENUM, FILIAL_ID, COUNTRY) Values('Kramer', 'Ruth', 'Berlin'
, 10963, 'BE', 'Kirchstraße', 12, 519, 'Deutschland')
insert into PERSON(SURNAME, FRONTNAME, CITY, ZIPCODE, STATE, STREET
, HOUSENUM, FILIAL_ID, COUNTRY) Values('Ackermann', 'Peter', 'Essen'
, 45136, 'NW', 'Albert-Dürer-Straße', 43, 577, 'Deutschland')
insert into PERSON(SURNAME, FRONTNAME, CITY, ZIPCODE, STATE, STREET
, HOUSENUM, FILIAL_ID, COUNTRY) Values('Baumann', 'Paula', 'Essen'
, 45147, 'NW', 'Nelly-Sachs-Straße', 74, 174, 'Deutschland')
.
.
.
insert into PERSON(SURNAME, FRONTNAME, CITY, ZIPCODE, STATE, STREET
, HOUSENUM, FILIAL_ID, COUNTRY) Values('Schmidt', 'Oskar', 'Bremen'
, 28777, 'HB', 'Poststraße', 19, 549, 'Deutschland')
insert into PERSON(SURNAME, FRONTNAME, CITY, ZIPCODE, STATE, STREET
, HOUSENUM, FILIAL_ID, COUNTRY) Values('Baumann', 'Ellen', 'Lüneburg'
, 21335, 'NI', 'Pfarrstraße', 75, 331, 'Deutschland')
insert into PERSON(SURNAME, FRONTNAME, CITY, ZIPCODE, STATE, STREET
, HOUSENUM, FILIAL_ID, COUNTRY) Values('Bernhardt', 'Hannes', 'Berlin'
, 10117, 'BE', 'Vereinsstraße', 70, 552, 'Deutschland')
```

Am Ende des Befüllungsvorgangs erhält der Nutzer folgende Nachricht und gelangt zur Tabellenübersicht zurück, nachdem er die Nachricht bestätigt hat.
In diesem Fall hat das reine Erstellen der Tupel ohne die Zeit des Dialogs mit dem Nut-

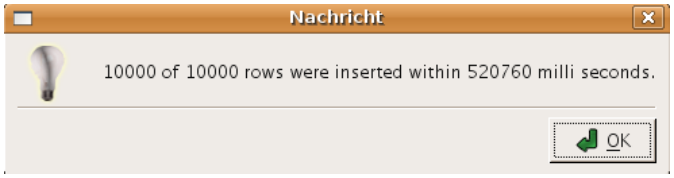

Abbildung 17: Benachrichtigung nach erfolgreichem Beenden des Befüllungsvorganges

zer weniger als 9 Minuten gedauert. Diese Zahl unterliegt so vielen Einflüssen, dass eine Vorhersage in Relation zur Anzahl der geforderten Tupel nicht möglich ist. Neben der Zahl der Tupel beeinflusst auch die Anzahl der verschiedenen Tabellenattribute die Performance. Außerdem verhält sich jede angebotene Befüllungsfunktion unterschiedlich hinsichtlich des Zeitbedarfs. Ebenfalls erhöhen die eventuell durch den Nutzer hervorgerufenen Fehler den Zeitverbrauch.
Trat einer der oben beschriebenen Abbruchgründe auf, sieht die Fehlermeldung so aus:

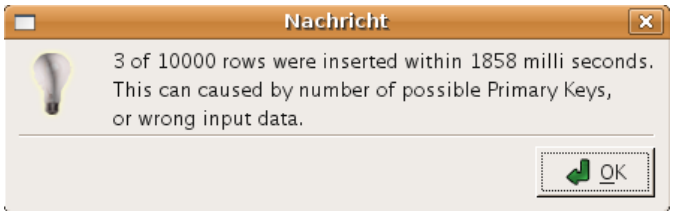

Abbildung 18: Benachrichtigung nach Abbruch des Befüllungsvorganges

4.7 Hilfe

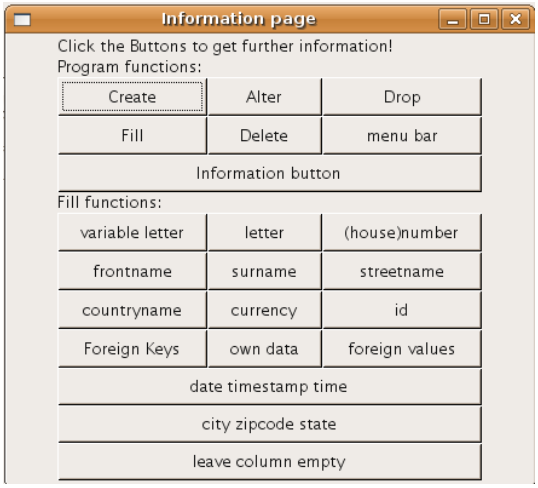

Abbildung 19: Hilfsinformationen zum Programm

Hier erhält man eine Übersicht und Hinweise zu allen Programmfunktionen, die man von überall aus erreichen kann. Auch hier, wie schon bei der Oberfläche für die Tabellenstrukturinformationen, beendet das Benutzen des “x”-Buttons nur das “Help”-Fenster.
Die einzelnen Texte erreicht man als Nachrichtenfenster über die Buttons. Dabei gibt es zwei Kategorien, einmal “Program functions:” und “Fill functions:”. In der ersten Kategorie findet man die Funktionen, die das Programm insgesamt bereithält, und in der zweiten Kategorie die Funktionen zum Befüllen von Tabellen.
Als Beispiel zeige ich hier die Informationen zur Befüllung mit Nachnamen:

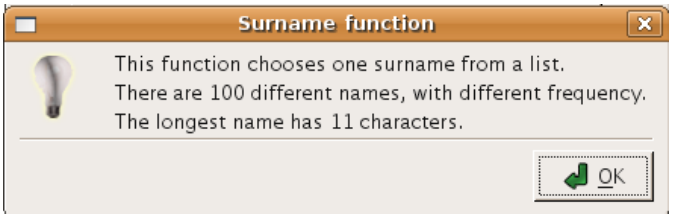

Abbildung 20: Hilfsinformationen zu “currency”-Funktion

Dabei wird einmal die Anzahl an verschiedenen Nachnamen in der Datenbasis erwähnt, so dass der Nutzer sich Gedanken über eine mögliche Anzahl von `Primary Keys` je nach Kombination mit anderen Attributen machen kann. Zusätzlich wird der längste Nachname erwähnt, so dass man beim Aussuchen der Befüllungsfunktionen und auch beim Anlegen neuer Tabellen auf die Feldbreite Rücksicht nehmen kann.

5 Erweiterungen und Ausblick

5.1 Erweiterungen

Für den oben vorgestellten Data Generator sind viele Erweiterungen denkbar. Da er durchaus auch in der Entwicklung anderer Software eine Rolle spielen könnte, werden die Anforderungen an ihn ständig variieren. Eine immer größere Erweiterung des Funktionsumfangs würde aber oft kontraproduktiv sein, da viele von dem einen dringend geforderte Erweiterungen von anderen nie genutzt würden. Die Auswahllisten mit Befüllungsfunktionen für Attribute von Typ "Varchar" oder "Char" sind zum jetzigen Zeitpunkt schon recht lang. Da aber bei einem Einsatz in der Forschung oder Entwicklung von einem kundigen Nutzer ausgegangen werden darf, könnte jeder Nutzer sich das Programm über den Sourcecode leicht selbst anpassen. Deshalb soll der hier implementierte Funktionsumfang auch nur einen Ausblick auf die Möglichkeiten eines Data Generators bieten.

5.1.1 Länder-Versionen

Im Moment wirkt es vielleicht etwas befremdend, dass ein englischsprachiges Programm Tupel mit deutschen Städten und Postleitzahlen erstellt. Allerdings sollte es kein Problem sein, das sehr einfach gehaltene verwendete XML-Schema um einen Wert "Language" zu erweitern und beim Programmstart vom Nutzer die gewünschte Sprache abzufragen. Die verschiedenen XML-Dateien werden nur dann in eine `ArrayList` geparst, wenn ihr Languagewert der Nutzerwahl entspricht. In der Klasse `DataHaendler` wird zusätzlich vermerkt, welche `ArrayList` mit Funktionsnamen für welchen SQL-Datentyp bei welcher Sprachwahl zu benutzen ist.
Hierbei könnte oder müsste man aber die verwendete XML-Struktur erweitern. Der Text für die Hilfe müsste mit in die XML-Datei und im Programm aus den aktuell verwendeten XML-Dateienen für die Hilfe zur Verfügung gestellt werden. Da es sein kann, dass bestimmte Funktionen nur in manchen Sprachversionen vorliegen, wäre es recht praktisch, wenn die Liste der Funktionen je nach SQL-Datentyp aus den XML-Datein erweitert werden würde. Allgemeine Funktionen wie "foreign key", "foreign value" und "own data", sowie alle anderen Funktionen, die nicht auf XML-Daten basieren, würden weiterhin vorher fest in diese Listen eingetragen werden.

5.1.2 XML-Dateien des Nutzers temporär einlesen

Zusätzlich zur leichten Erweiterbarkeit durch das Verändern der bereits vorhandenen XML-Dateien, wie in Kapitel 3.10 beschrieben, könnte eine weitere Funktion bereitgestellt werden. Für weniger versierte Nutzer oder auch einfach aus Zeitgründen ist es eine denkbare sinnvolle Erweiterung, das Benutzen eigener XML-Dateien zu ermöglichen. Strukturell wäre es am ehesten denkbar, dass diese neuen XML-Daten nur während eines Befüllungsvorganges gespeichert bleiben, und der jetzigen Progarmmstruktur folgend, könnte man mehrere verschiedene solcher Datensätze während eines Befül-

lungsvorganges verwenden. Voraussetzung ist natürlich, dass der Nutzer sich an die vorgegebene Struktur für XML-Dateien hält. Allerdings ist das nur eine vereinfachte Version der schon vorhandenen “own data”-Funktion und wie diese sehr anfällig für falsche Nutzereingaben.

5.2 Ausblick

Der entstandene Data Generator ist sicher nicht nur für Data Warehouse benutzbar, aber schon innerhalb des Feldes der möglichen Anwendungen bezüglich Data Warehouses zeigt sich, dass jeder Nutzer eigene Anforderungen an das Programm hat. Dementsprechend scheint es am sinnvollsten das Programm ausführlicher zu dokumentieren, so dass jeder Anwender es sich selbst so modifiziert, wie er es braucht. Dies ginge dann in die Richtung einer Open-Source-Software. Sicherlich ist der Bedarf allgemein vorhanden, wenn man sieht, wie die Anwendungsbereiche von Datenbanken wachsen, bzw. wie in immer mehr Anwendungsbereichen Datenbanken als Hilfsmittel eingesetzt werden. Gerade auch in der Software- oder Web-Entwicklung, wo später zu verarbeitende Daten noch nicht vorliegen, kann durch einen extra angepassten Data Generator das entwickelte und angelegte Datenbankschema vorab mit großen, so realitätsnah wie möglich gestalteten und dennoch in relativ kurzer Zeit verfügbaren Datenmengen getestet werden.

Literatur

[Bor01] BORN, Günter: *XML*. München : Markt + Technik Verlag, 2001

[Bor02] BORMANN, Carsten: *SQL*. Berlin : SPC TEIA Lehrbuch Verlag, 2002

[CL04] CADENHEAD, Rogers ; LEMAY, Laura: *Java 2*. München : Markt + Technik Verlag, 2004

[Con07] CONRAD, Stefan: *Skript zur Vorlesung Data Warehouse*. Düsseldorf : Heinrich Heine Universität Düsseldorf, 2007

[HS00] HEUER, Andreas ; SAAKE, Gunter: *Datenbanken*. Landsberg : mipt Verlag, 2000

[PR97] POE, Vidette ; REEVES, Laura: *Aufbau eines Data Warehouse*. München : Prentice Hall, 1997

Abbildungsverzeichnis