Systementwicklungsprojekt

Thema:
Implementierung einer Entwicklungsumgebung zur Generierung von Benutzungsoberflächen mit Datenbank-Backend

Bearbeiter:
Matthias Assel

Betreuer:
Riitta Höllerer, Dr. Alfons Brandl

Inhaltsverzeichnis

1. Einleitung

2. Planungsphase

   2.1 Ausgangslage
   2.2 Anforderungen/Ziele
   2.3 Handlungsalternativen
   2.4 Einführung Datenbanken
   2.5 Fazit

3. Realisierung

   3.1 Verwendte Hilfsmittel (Technologien)
   3.2 Gewähltes Konzept
   3.3 Konkrete Implementierung

      3.3.1 Veränderungen bzgl. der Vorgaben
      3.3.2 Erweiterungen der Visitoren
      3.3.3 Implementierung der Datenbank-Anbindung
      3.3.4 Ergänzungen/Erweiterungen des *emu_runtime* Paketes

4. Anwendung

   4.1 Möglichkeiten der Benutzung
   4.2 Umgang mit dem entwickelten Programm

5. Fazit

   5.1 Erreichte Ziele
   5.2 Konkrete Erweiterungsvorschläge
   5.3 Ausblick

1. Einleitung

Das Werkzeug *EMUGEN* dient zur Generierung von Java-Swing basierten Benutzungsoberflächen. Diese ermöglichen es, dem Benutzer Daten in die generierten Dialogmasken einzugeben, welche anschliessend bisher als XML-Dokument gespeichert werden können. Gerade bei sehr großen Projekten wäre es daher sinnvoll, die entstehenden Datenmengen in einer Datenbank zu speichern. Damit man unabhängig von der Eingabemaske und der Infrastruktur der Datenbank dies realisieren kann, soll in diesem Systementwicklungsprojekt eine Entwicklungsumgebung geschaffen werden, welche es ermöglicht, die *EMUGEN*-Eingabe interaktiv in eine GUI einzugeben, und nicht wie bisher bei *VISUAL EMUGEN* als Eingabesprache in einem Editor, und dabei auch eine Anbindung an eine Datenbank zu beschreiben. Wie man diesen Prozess realisieren und dann auch umsetzen kann wird im Folgenden genauer beschrieben.

2. Planungsphase

Bevor man an die eigentliche Umsetzung bzw. konkrete Implementierung überhaupt denken kann, muss man sich mit den gegebenen Rahmenbedingungen auseinandersetzen und verschiedene Szenarien der möglichen Umsetzung diskutieren und darauf aufbauend entsprechende Ziele sprich Mindestanforderungen definieren.

2.1 Ausgangslage

Die grundsätzliche Struktur der Eingabe und damit die GUI der Entwicklungsumgebung (im Folgenden als IDE bezeichnet) wurde in Teilen vorgegeben. Wie in Abbildung 1 und 2 zu erkennen, wurde die IDE ebenfalls mit Hilfe von *EMUGEN* generiert und dient lediglich zur Orientierung für die spätere Implementierung. Man erkennt sehr deutlich die Gliederung der späteren Eingabe in drei wesentliche Abschnitte.

- Eingabe des *EMUGEN*-Datenmodells
- Eingabe der Datenbank-Tabellenstrukturen
- Eingabe eines entsprechenden Mappingkonzeptes

Damit ergeben sich im Wesentlichen die folgenden Generierungsaufgaben. Zunächst muss eine Generierung der *EMUGEN*-Eingabe aus der IDE implementiert werden. Als Ausgangspunkt hierfür dient der vorgegebene Visitor (*EMUCodegenTopDownVisitor*), welcher in sehr groben Zügen den Ablauf der Generierung darstellt. Im Anschluss daran müssen die Datenbank-Tabellendefinitionen aus der IDE generiert oder entsprechend eingelesen werden. Um eine Verknüpfung zwischen Datenmodell und Datenbank-Tabellendefinitionen herstellen zu können, muss in einem weiteren Schritt ein Mappingkonzept entwickelt und implementiert werden. Den Abschluss bildet die Konfiguration von technischen Elementen, wie z.B. Datenbank-Treiber, Host, Benutzername und Passwort.

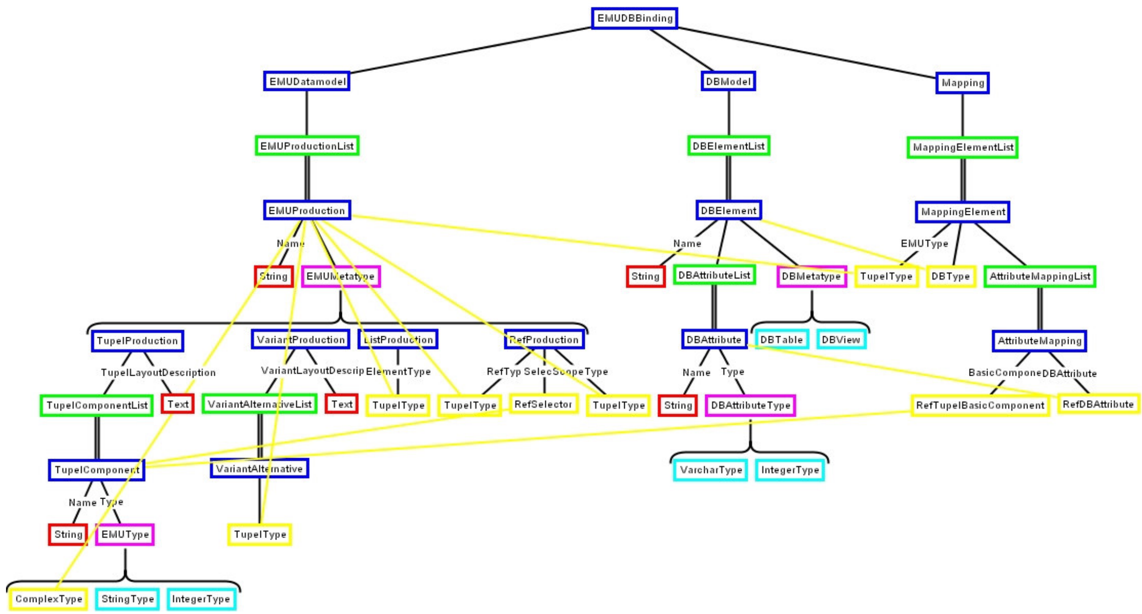

Abbildung 1: Vorgegebene Struktur der IDE

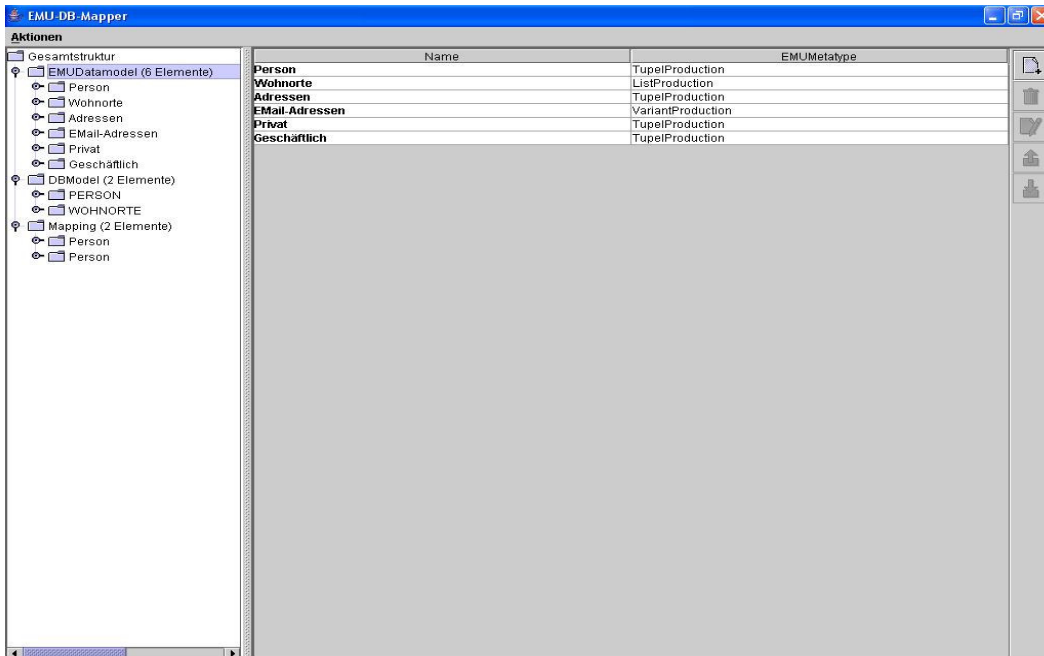

Abbildung 2: Resultierende GUI der *EMUGEN*-Eingabe

2.2 Anforderungen/Ziele

Das primäre Ziel des Systementwicklungsprojektes ist die Erstellung einer Entwicklungsumgebung (mögliche Variante siehe Abbildung 2), welche es dem Benutzer ermöglicht, eine *EMUGEN*-Eingabe interaktiv in die GUI einzugeben, und gleichzeitig dafür zu sorgen, dass der aus der Eingabe resultierende Dialog, eine Anbindung an eine entsprechende Datenbank erhält. Dabei sollte nach Möglichkeit unabhängig von der *EMUGEN*-Eingabe, sprich es ist egal was der Benutzer für einen Dialog generieren will, eine Anbindung an eine Datenbank geschaffen werden.
Abhängig von diesem Primärziel kann man weitere Anforderungen an die Entwicklungsumgebung stellen. Sie sollte, wie auch schon *VISUAL EMUGEN,* Plattform unabhängig sein und keinerlei besondere Ansprüche an Hard- und Software stellen. Einzige Vorraussetzung ist eine funktionsfähige Java-Runtime-Version (am bestem Version 1.4.2 oder höher).
Weiterhin muss die Möglichkeit gegeben sein, die Datenbank-Anbindung sehr variabel zu gestalten, sprich der verwendete Treiber für die Verbindung zur Datenbank sollte nach Möglichkeit relativ einfach zu ändern sein. Auch die Einstellungen der eigentlichen Datenbank-Parameter (Host, DB-Name, Benutzername, Passwort) sollten, wenn möglich, vom Benutzer einzugeben und veränderbar sein.
Der aus der Eingabe resultierende Dialog sollte, wie auch schon bein *VISUAL EMGUEN* nicht nur aus der Applikation heraus startbar sein, sondern ebenfalls als kompakte Form (kann als Jar-Archiv exportiert werden) vorliegen. Auch für ihn gelten die Anfordungen bzgl. der Anbindung an eine Datenbank.
Ein sehr wichtiger Punkt bei der Entwicklung von Softwareprodukten ist die Benutzerfreundlichkeit im Hinblick auf Bedienbarkeit und Übersicht. Auch hier sollte ein kompaktes und durchdachtes Konzept bzgl. der Eingabe, Konfiguration und Anwendbarkeit entwickelt werden. Dies kann natürlich nicht nach der ersten Version zur vollsten Zufriedenheit der Anwender geschehen, jedoch sollten von Anfang an gewisse Kriterien und Regeln der Softwareentwicklung eingehalten werden. Hierzu gehört auch die korrekte und ausführliche Dokumentation, sowohl als Anwender- als auch Entwicklerdokumentation.
Zu all diesen grundlegenden Zielen gibt es noch eine ganze Reihe weiterer „kleinerer“ Ziele. Hierzu zählt zum Beispiel, der Im- und Export von *.emu*-Dateien, oder das Einlesen von schon generierten Datenbank-Dateien. Inwieweit all diese „Wünsche“ zeitlich realisiert werden können, muss man im Laufe der Entwicklung abwägen.

2.3 Handlungsalternativen

Grundsätzlich kann man die in der Grundstruktur vorhandene IDE komplett eigenständig neu implemetieren. Dies hätte zum Vorteil, dass man eine saubere und unabhängige Konzeption bzgl. der Oberfläche und Dialogsteuerung bekäme. Hinsichtlich Zeitrahmen und, warum nicht bereits bestehende Möglichkeiten nutzen, sollte man die Schwerpunkte bei der Entwicklung der IDE auf andere Teile legen.
Ein großer Bereich, der gerade hinsichtlich der Datenbank-Anbindung wichtig ist, wären die verschiedensten Möglichkeiten des Datenbank-Designs und der automatischen Generierung von Datenbanken und Tabellen. Weitere Einzelheiten hierzu werden im nächsten Kapitel näher erläutert.
Ein weiterer Diskussionspunkt zur Implementierung der Funktionalitäten wäre, inwieweit man die vorhandenen Sourcen von *EMUGEN* hinsichtlich der neuen Aufgaben

erweitert bzw. verändert. Da man zur Generierungszeit bereits die Datenbank-Anbindung gewährleisten muss, gibt es nur zwei Alternativen um dieses Problem zu lösen. Entweder man generiert zusätzlich zu den bereits vorhandenen Funktionen eine weitere, welche es ermöglicht, Datenbank-Queries auszuführen (gleiches wird bei der Speicherung der Daten als XML-Datei angewandt) oder man macht sich das Prinzip von *EMUGEN* zu nutze, indem man während der Erzeugung der Source-Dateien, Aktionen für den entstehenden Dialog definiert und diese somit ebenfalls von *EMUGEN* mitgenerieren lässt. Einen großen Vorteil bietet natürlich das zweite Konzept, denn man muss nicht in die Sourcen von *EMUGEN* direkt eingreifen, was aber die Entwicklung aufgrund der vorgegebenen Funktionalität stark einschränken könnte.
Der wohl größte Spielraum bei der Implementierung liegt im Bereich des Mapping-konzeptes. Zwar wurde ein rudimentäres System bei der Erzeugung der IDE vorgegeben, jedoch stellt gerade dieser Part eine sehr wichtige und entscheidende Rolle bei der Anbindung der Datentypen an die entsprechenden Datenbank-Tabellen. Hier kommt es vor allem auf die Datenbank-Tabellenstruktur an, welche den Rahmen des Mappingkonzeptes darstellt. Fragen wie, welcher Schlüssel bezieht sich auf welche Tabelle (= Datentypklasse) und wie werden diese Schlüssel bei der Generierung behandelt (näheres im Anschluss).

2.4 Einführung Datenbanken

Die entscheidende Rolle bei der Entwicklung der IDE spielt der Teil der Anbindung von *EMUGEN* an eine Datenbank. Da das Thema Datenbanken ein sehr weit ge-fechertes Gebiet umfasst, kann an dieser Stelle lediglich auf gundlegende Über-legungen in Bezug auf eine Datenbank-Konzeption eingegangen werden.
Datenbanken dienen dazu, große Mengen an Daten übersichtlich, skalierbar und redundant in dafür vorgesehenen Tabellen zu speichern und dem Benutzer zu ermöglichen, auf diesen Tabellen spezielle Anfragen (= Queries) auszuführen. Hierfür gibt es eine Reihe von Ansätzen, dies performant und sicher zu designen. In unserem Fall, können wir von einem sehr einfachen Konzept, mit nicht allzu grossem Augenmerk auf Performance und Sicherheit ausgehen. Trotzdem sollte man einige Regeln bei dem Entwurf eines solchen Systemes beachten.
Die erste Überlegung in Bezug auf das Konzept ist, ob der Benutzer auf die Tabellen und Queries direkten Einfluss nehmen sollte oder es vom System her vorgegeben wird. Da die Entwicklungsumgebung ein sehr einfaches, sprich für „jederman" bedienbares Instrument sein sollte, ist daher die Überlegung, sämtliche Vorgaben bzgl. der Tabellen und Queries, vom System her festzulegen. Das einzige, was der Benutzer beeinflussen kann (besser muss), ist der Name der Datenbank und deren Verbindungsparameter (Host, Benutzer, Passwort). Damit wird zumindest im groben gewährleistet, dass auch Benutzer ohne jegliche Datenbankkenntnisse mit der IDE Arbeiten können.
Im groben stimmt die obige Aussage über den Einfluss eines Users auf die Daten-bankstruktur, jedoch muss bei der Eingabe des *DBModels* im Wesentlichen eine Art Tabelle angelegt werden. Es müssen zwar nur die Spaltennamen und deren Daten-typen, was sich in unserem Fall auf Integer- und String-Datentypen beschränkt, eingegeben werden (eine weitere Spalte, die Länge des Datentyps wird im Laufe der Implementierung hinzukommen, vergleiche schon jetzt Abbildung 3). Im Hinblick auf Performance und Skalierbarkeit sollte man schon jetzt wissen, dass es immer schlechter ist einen Stringwert (MySQL verwendet dafür den Datentyp *varchar* oder *char*) anzugeben als einen Integerwert, da Integerwerte im Allgemeinen weniger

Speicherplatz benötigen und damit besser zu verarbeiten sind. Gibt ein Benutzer wie in Abbildung 3 zu erkennen die entsprechenden Werte für das *DBModel* ein, so wird vom System ein Query generiert, welcher genau aus diesen Werten eine Tabelle erzeugt. Im Folgenden werden nur MySQL-Statements verwendet, da MySQL die häufigste und „einfachste“ frei erhältiche Datenbank im „semi-professionellen“ Bereich ist.

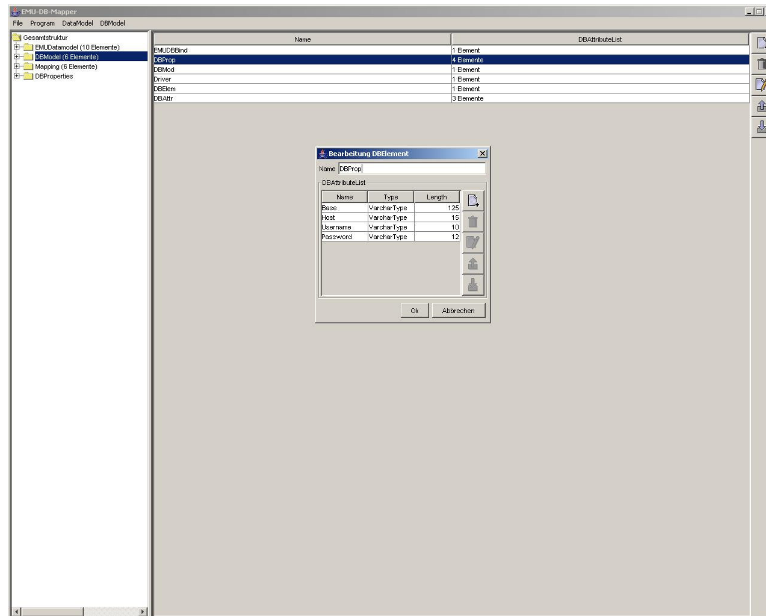

Abbildung 3: Eingabe der Werte für das *EMU-DBModel*

Wie schaut nun der Query zum Anlegen der oben eingegebenen Parameter aus? Neue Tabellen werden in MySQL mit dem *CREATE TABLE +Tabellenname+* – Statement erzeugt. Würde man nur *CREATE TABLE +Tabellenname+* als Query ausführen so bekämen wir eine Fehlermeldung des Datenbank-Servers, da noch keine Spalten angegeben sind (Hinweis: es muss **immer** mindestens eine Spalte angegeben werden). Somit müssen wir also zum Erzeugen von unserer Eingabe folgenden Query anwenden.

CREATE TABLE +Tabellenname+ (Base varchar(125), Host varchar(15), Username varchar(10), Password varchar (12))

Man erkennnt sofort, dass man nach dem Tabellennamen in den Rundenklammern die Spalten mit deren MySQL-Datentypen eingeben muss. Die Klammer hinter den Datentypen gibt die Länge in Byte an. Bei *varchar* muss immer eine Länge angegeben werden, hingegen bei Integer nicht. Maximale Längen sind bei *varchar* 255 Byte und bei Integer 11 Byte. Diese 11 Byte sind zugleich der Defaultwert eines Integerwertes, und er wird gesetzt sobald man keine Länge beim Anlegen angibt. Es gibt natürlich noch weitere Datentypen, aber hier sollen nur diese zwei Standardtypen betrachtet werden. Dies hängt unteranderem mit *EMUGEN* zusammen, da auch hier 90% aller eingegebenen Typen, mit *varchar* und Integer beschrieben werden können (weitere Informationen zu den Datentypen siehe http://dev.mysql.com/doc/mysql/en/Column_types.html).
So nun haben wir (die IDE) also die erste Tabelle in unserer Datenbank angelegt. Bevor man jedoch solche Tabellen anlegen kann, sollte man eine Datenbank entweder neu anlegen oder in eine spezielle Datenbank wechseln.

Bei der Eingabe der Datenbankparameter muss auch ein Datenbankname mit angegeben werden. Dieser wird dann mit dem MySQL-Statement *CREATE DATABASE +Datenbankname+* eine neu Datenbank anlegen. Existiert bereits eine Datenbank, kann man mit folgendem Statement auf dieser arbeiten.

USE +Datenbankname+

Jetzt haben wir die grundsätzlichen Operationen (DDL, **D**ata **D**escription **L**anguage = Beschreibung der Daten (-typen)) zum Arbeiten mit MySQL angesprochen. Kommen wir nun zum eigentlich wesentlichen Teil einer Datenbanksprache (= SQL, **S**tructured **Q**uery **L**anguage). Hier wird das sogenannte DML (= **D**ata **M**anipulating **L**anguage) verwendet. Es dient dazu, auf die vorhandenen Tabellen Anfragen zu generieren, und dabei den Datenbestand zu verändern. Man unterscheidet im Wesentlichen zwischen vier Hauptanfragen *SELECT, INSERT, UPDATE* und *DELETE*. Mit *SELECT* kann man sich sämtliche Daten (= Zeilen) anzeigen lassen, wohingegen *INSERT* und *UPDATE* zum Verändern des Datenbestandes führen. *DELETE,* wie leicht zu erkennen ist, löscht einen bestimmten Datensatz. Fügen wir nun einen bestimmten Datensatz in unsere vorhin erzeugte Tabelle ein. Dies geschieht mit folgendem Statement.

INSERT INTO +Tabellenname+ VALUES ('Testdatenbank', 'server', 'assel', 'passwort')

Dieser Query fügt nun eine neue Zeile in die Datenbank ein, und zwar werden jeweils den einzelnen Spalten die oben eingetragen Werte zugeordnet.
Man erkennt sofort, dass man *varchar*-Felder beim Einfügen durch Hochkommata angeben muss.
Wie können wir nun diesen Wert wieder auslesen um eventuell die Werte an eine Applikation weiterzugeben. Hierfür gibt es das *SELECT*-Statement. In unserem Beispiel können wir den eben eingefügten Datensatz mit folgender Query selektieren.

*SELECT * FROM +Tabellenname+*

Der * steht für das Auslesen aller Spaltenwerte. Man könnte sich hier genauso gut nur bestimmte Spalten ausgeben lassen, indem man die Spaltennamen durch Komma getrennt von einander angibt (*SELECT Host, Password FROM +Tabellenname+*).
Diese Anfragen haben leider einen großen Nachteil. Man bekommt in diesem Fall alle in der Tabelle enthaltenen Werte, was bei großen Tabellen sehr lange dauern könnte.
Deshalb kann man in einer *WHERE*-Klausel die Ausgabe auf bestimmte Weise einschränken, z.B.

*SELECT * FROM +Tabellenname+ WHERE Host = 'server'*

Dies kann aber in speziellen Fällen auch zu einem verfälschten Ergebnis führen, bzw. man möchte genau einen Datensatz mit einer Anfrage bekommen und erhält stattdessen mehrere, da es viele Datensätze mit, in unserem Beispiel einem *Host = 'server'* geben kann. Hierfür gibt es die Möglichkeit einen speziellen Schlüssel beim Anlegen der Tabelle (kann auch nachträglich geändert werden) mit anzugeben, welcher einen Datensatz **eindeutig** identifiziert. Es handelt sich hierbei um den sogenannten *Primary Key*, zu Deutsch Primärschlüssel. Dieser wird bei jedem Einfügen eines neuen Datensatzes immer als eindeutig angesehen, zwei Datensätze mit gleichem Primärschlüssel gibt es nicht. Daher sollte man sich schon beim Anlegen der Tabellen einen geeigneten Primärschlüssel überlegen. Meistens wird dieser eine

eindeutige ID (Integerwert) sein, welcher sich automatisch beim Einfügen eines neuen Datensatzes um eins erhöht. Somit könnte man unsere vorhin angelegte Tabelle auch besser auf diese Weise erstellen,

CREATE TABLE +Tabellenname+ (ID integer auto_increment primary key, Base varchar(125), Host varchar(15), Username varchar(10), Password varchar (12))

oder mit

ALTER TABLE +Tabellenname+ DROP PRIMARY KEY, ADD PRIMARY KEY(ID)

verändern. Im zweiten Fall muss jedoch zunächst eine weitere Spalte angelegt werden, was ebenfalls mit dem *ALTER*-Statement geschehen kann.

ALTER TABLE +Tabellenname+ ADD `ID` INT NOT NULL

Das Attribut *auto_increment* kann logischerweise auch weggelassen werden, erfüllt aber genau in diesem Fall seinen Zweck.
Mit dieser Erweiterung ändert sich auch unserer Query zum Einfügen von Datensätzen wie folgt.

INSERT INTO +Tabellenname+ VALUES (ID, 'Testdatenbank', 'server', 'assel', 'passwort')

Hier wird also die Spalte ID direkt, ohne Wert angegeben, wobei der Wert beim Einfügen, falls mit *auto_increment* angegeben, sich automatisch um eins erhöht.
Ist er nicht als *auto_increment* angegeben, muss man per Hand eine eindeutige ID angeben.
Aufbauend auf diese wichtige Eigenschaft, kann man nun die beiden weiteren Anfragen betrachten. *UPDATE* und *DELETE* basieren genau auf diesem Prinzip der Eindeutigkeit. Die folgenden zwei Statements sollen deren Gebrauch verdeutlichen.

UPDATE +Tabellenname+ SET Password = 'test' WHERE ID = 1

und

DELETE FROM +Tabellenname+ WHERE ID = 1

Es wird sofort ersichtlich, dass nur aufgrund unseres Primärschlüssels wir die beiden obigen Queries ausführen können (Hinweis: möchte man eine Tabelle komplett löschen, sollte man das *TRUNCATE +Tabellenname+* Statement benutzen und nicht das *DELETE FROM +Tabellenname+*, da *DELETE FROM* jede Zeile einzeln löscht und dabei die Tabelle für weitere Anfragen sperrt, was bei sehr großen Tabellen zu erheblichen Performance Problemen führen kann).

Haben wir nun die wichtigsten SQL-Statements kennengelernt, widmen wir uns jetzt einem weiteren wichtigen Punkt beim Entwurf einer Datenbank. Bisher haben wir lediglich eine einzige Tabelle betrachtet, was aber im Normalfall nicht die Regel sein wird, sondern man wird es immer mit mehreren Tabellen zu tun haben. Grundsätzlich gelten die gleichen Anfrage-Statements wie bisher, nur müssen wir daran denken, diese Tabellen gegebenenfalls zu verknüpfen.
Genau hier treffen wir auf ein Problem, geeignete Tabellen in einer Datenbank so anzulegen, das Daten redundant und zugleich performant mit einander verknüpft

werden, um bei sehr komplexen Queries den Datenbank-Server nicht unnötig zubelasten. Betrachten wir folgenden Szenario.
Es soll eine Tabellen-Struktur in MySQL erstellt werden, welche Personen mit ihren verschiedenen Wohnorten speichern kann. Die einzelnen Attribute von Personen und Wohnorten spielt hier nur eine nebensächliche Rolle, da man es beliebig ändern und erweitern könnte.
Die einfachste, aber unsauberste Lösung wäre alles in einer einzigen Tabelle zu speichern, was allein schon aus Übersichtsgründen bei mehreren Wohnorten zu Problemen führen könnte. Daher liegt es nahe, sowohl für Personen, als auch für die Wohnorte einer Person, jeweils eine separate Tabelle zu erstellen und diese dann zu verbinden. Hier gibt es genau zwei entsprechende Ansätze. Das Anlegen der beiden Tabellen liegt diesen Ansätzen natürlich zu Grunde. Generieren wir nun die beiden Tabellen mit den folgenden Statements.

```
create table Personen
        ( PersID integer primary key,
          Name varchar(25) not null,
          Vorname varchar(25),
          Telefonnummer varchar(25),
          EMail varchar(30) );

create table Wohnorte
        ( AdressID integer primary key,
          Strasse varchar(40),
          Hausnummer integer,
         PLZ integer,
          Ort varchar(30) );
```

Hat man nun die beiden Tabellen angelegt, kommt man zu dem Punkt diese miteinander zu verknüpfen.
Die erste Idee wäre eine dritte Tabelle zu erstellen, welche genau die ID einer Person auf einen Wohnort referenziert. Es wird aber sehr schnell ersichtlich, dass bei sehr vielen Tabellen immer weitere Tabellen hinzugefügt werden müssen, was unter Umständen wieder zu erheblichen Performanceproblemen führen könnte. Dennoch erhalten wir mit dieser Lösung genau den gewüschten Erfolg, und zwar eine Tabelle auf eine andere Tabelle zu referenzieren und damit eine eindeutige Zuordnung von Datensätzen zu erhalten. In unserem Beispiel sähe diese dritte Tabelle folgendermassen aus.

```
create table wohnen
        ( PersID integer references Personen on delete cascade,
          AdressID integer references Adressen on delete cascade,
          primary key(PersID, AdressID) );
```

Man erkennt sofort, dass man die ID´s der Tabelle als Primärschlüssel definieren muss, um genau diese Eindeutigkeit zu erhalten. Das *on delete cascade* Statement bezieht sich auf den Fall, wenn ein Datensatz gelöscht wurde, müssen auch sämtliche Referenzen mitgelöscht werden, um eine Verwechslung auszuschliessen.
Die zweite, vielleicht schönere und effektivere Variante wäre, dass man, bleiben wir bei unserem Beipspiel, der Tabelle Wohnorte einen sogenannten *Foreign Key* (Fremdschlüssel) hinzufügt, der genau wie im ersten Fall auch, auf die Tabelle Personen referenziert, jedoch man keine weitere Tabelle benötigt, welche diesen Schlüssel beinhaltet. Damit ergäbe sich die folgende neue Tabelle Wohnorte.

```
create table Wohnorte
        ( AdressID integer primary key,
          Strasse varchar(40),
          Hausnummer integer,
          PLZ integer,
          Ort varchar(30)
          Person integer not null,
          foreign key (Person) references Personen on delete set null );
```

Auch hier wird sofort die Referenz auf die Tabelle Personen deutlich. Der Zusatz *on delete set null* bewirkt in diesem Fall, eine NULL-Referenz auf einen Datensatz, d.h. er wird keiner Person zugeordnet werden können.
Diese beiden gezeigten Ansätze sollen verdeutlichen, dass man sich bei der Implementierung einer Datenbank schon vor dem eigentlichen Anlegen der Tabellen bzgl. der Struktur der späteren Daten Gedanken machen muss, um gewisse Zuordnungen und Eindeutigkeiten von Datensätzen gewährleisten zu können. Es gibt noch weitere Konzepte von Datenbank-Designs, welche aber für unseren Gebrauch eine nebensächliche Rolle spielen. Hat man nun die grundlegenden Konzepte und Begrifflichkeiten verstanden, sollte man in der Lage sein eine entsprechende Datenbank-Struktur für seine Applikation zu erstellen und deren Nutzen zu schätzen und auszuschöpfen.

2.5 Fazit

Die Implementierung der eigentlichen Applikation mit ihrer Anbindung an eine Datenbank stellt eine große Herausforderung an Konzept und Design. Aus diesem Grund ist gerade die Planungsphase einer solchen Aufgabenstellung sehr wichtig, da man von vorne herein merkt, welche Schwachpunkte bzw. Vorteile gewisse Ansätze bieten. Den Ansatz, eine komplett eigene GUI der Entwicklungsumgebung zu implementieren, kann man daher getrost bei Seite legen, da man mit der generellen Aufgabe, der „automatischen" Anbindung an eine Datenbank, genug beschäftigt sein wird. Ausserdem erscheint es unumgänglich, gewisse Teile der generierten Sourcen händisch verändern zu müssen, da man gerade den Teil der Datenbank-Anbindung mit *EMUGEN* noch nicht einfach so modellieren und anschliessend generieren kann. Hinsichlich des Mappingkonzeptes, wie bereits erwähnt, ist bzgl. der Vorraus-setzungen wahrscheinlich ein komplett überarbeitetes Konzept notwendig. Inwieweit alle angesprochenen Ziele und Wünsche erfüllt werden können, wird die eigentliche Implementierungsphase zeigen. Es könnte unter Umständen auch sein, dass eventuell neue Ziele und Alternativen betrachtet werden müssen, da die vorgestellten Ansätze im Moment nur theoretischer Natur sind, und in der Praxis zwar umsetzbar wären, aber aufgrund von diversen Schwierigkeiten hinsichtlich Vorgabe, Ver-änderungen der Sourcen und Zeitrahmen einfach nicht zu realisieren sind. Abschliessend kann man jedoch sagen, dass das Werkzeug *EMUGEN* durch diese Erweiterung wieder einen großen Schritt in Richtung einer professionellen Applikation machen wird, da gerade der Bereich Datenbanken in Bezug auf generierte Dialoge eine wichtige Rolle spielt. Deshalb wird jeder einzelne Schritt, welcher er im End-effekt auch sein mag, einen neuen Anreiz bieten, *EMUGEN* auch in Zukunft weiter zu entwickeln.

3. Realisierung

EMUGEN ist ein in Java entwickeltes Werkzeug, welches eine eigenständige Generierungssprache verwendet. Dieser Teil spielt auch bei der Implementierung der Entwicklungsumgebung eine erhebliche Rolle, da man sich gewisse Eigenschaften der bereits vorhandenen Funktionen zunutze machen kann. Welche dies im Wesentlichen sind, zeigt der folgende Abschnitt der Implementierung der Entwicklungsumgebung mit Datenbank-Anbindung.

3.1 Verwendte Hilfsmittel (Technologien)

Im Rahmen der Implementierung wurde die freie Entwicklungsumgebung *Eclipse Version 2.1.2* von IBM verwendet. Sie eignet sich hervorragend zum Entwickeln von Java-Applikationen, da sie die verschiedensten Möglichkeiten von Java-Projekten bietet. Man kann unter anderem seine Applikation mit Hilfe von *ANT* übersetzen lassen, oder mam lässt sie direkt aus *Eclipse* heraus starten und exportiert die fertige Applikation als sogenanntes *JAR*-Archiv, welches man dann auf jedem „java-fähigen“ Rechner direkt starten kann (mehr dazu in Punkt 4.1).
Als Basisarchitektur, sprich *Java-Runtime-Version*, diente die von Sun veröffentlichte Version 1.4.2_03. Da man sehr viele Java-Swing-Komponenten hinsichtlich der IDE benötigt, sei jedem der die Applikation benutzen möchte, empfohlen, mindestens die *Java-Runtime-Version 1.4* (oder natürlich höher) zu verwenden. Bei älteren Versionen, ist ein nicht reibungsloser Ablauf, nicht völlig auszuschliessen.
Hinsichtlich der Verwendung von *EMUGEN* sind weiterhin die folgenden Java-Pakete notwendig.

- **grace-1.0:** Wird zur Generierung der Graphendarstellung von *EMUGEN* benötigt.

- **JFlex-1.3.5:** Scanner, der die Generierungssprache von *EMUGEN* enthält

- **java-cup-10k-b2:** Ein Java-Parser zum Einlesen der *EMUGEN*-Grammatik

- **emu_runtime.jar:** Enthält die Klassen zum Ausführen der generierten Dialoge

Bezüglich der Datenbank-Anbindung ist ein bestimmtes Paket notwendig, welches einen Treiber zur Ansteuerung an eine Datenbank enthält. Hierfür wurde das Paket **mm.mysql-2.0.4-bin** verwendet, welches einen Treiber zur Anbindung an eine MySQL-Datenbank enthält. Es gibt verschiedenste Pakete, die eine Datenbank-Verbindung herstellen können. Würde man ein anderes Paket verwenden, müsste man lediglich den Pfad des Treibers ändern.
Bei der Entwicklung der IDE wurde speziell darauf geachtet, die neuen Funktionen unabhängig von den bisherigen Sourcen zu implementieren. Dies eröffnet die Möglichkeit, weitere Funktionalitäten hinsichtlich der Datenbank-Anbindung auch über dieses SEP hinaus, zu implementieren.

3.2 Gewähltes Konzept

Nach eingehender Prüfung der verschiedensten Alternativen/Konzepte wurde der folgende Ansatz zur Realisierung gewählt:
Grundsätzlich wird die vorgegebene Struktur, sprich die generierten Sourcen der Entwicklungsumgebung, beibehalten, d.h. der Ansatz eine eigene GUI für die IDE zu erstellen wurde als nicht sinnvoll angesehen und daher nicht angewandt.
Was das Design der Datenbank und deren Tabellen angeht, wurde die zweite vorgestellte Alternative als die bessere der beiden angesehen. Somit werden die Verküpfungen der einzelnen erstellten Tabellen mit Hilfe des Fremdschlüssel Prinzips umgesetzt (siehe Punkt 2.4). Zusätzlich dazu wird dem Benutzer jeglicher Eingriff auf die zu erzeugenden Queries von der Applikation abgenommen, sprich er muss zwar über die IDE die Tabellen definieren, aber das eigentliche Anlegen in MySQL übernimmt die Applikation direkt beim Erzeugen der Sourcen der zu erstellenden Dialoge. Ebenso verhält es sich mit den Queries bzgl. der Datenmanipulation. Sie werden ebenfalls direkt beim Generieren der Sourcen mit an die Applikation übergeben und sind somit für den Benutzer nicht veränderbar, sprich der Handlungsspielraum, was SQL-Statements angeht wurde auf ein Minimum reduziert.
Zunächst sollen für jegliche Arten von Tupel- und Listenproduktionen eine Anbindung an eine Datenbank umgesetzt werden. Varianten und Referenzen werden für den Moment nicht berücksichtigt, da sie aufgrund der Komplexität der Tabellenstruktur hinsichtlich der Tabellenverknüpfung noch mal eine Ebene schwieriger zu realisieren sind, als die beiden Erstgenannten. Da Listen nichts anderes sind als eine Tabelle von Tupeln, kann von folgendem einfachen Prinzip ausgegangen werden. Für jedes Tupel, sprich jede Liste von Tupeln wird eine einzelne eigene Tabelle verwendet. Damit wird sofort ersichtlich, dass jede zu generierende Applikation aus der IDE heraus, mit einer Listenproduktion starten muss (wird im Laufe des Kapitels noch genauer erklärt). Dies ist die zweite Einschränkung neben der Tatsache, dass im Moment nur Tupel und Listen betrachtet werden können.
Somit ergibt sich für das Mappingkonzept ein etwas neuer Ansatz als vorgegeben. Um eine entsprechende Verknüpfung zwischen den Datentypklassen (*EMUDataModel*) und den Datenbanktabellen (*DBModel*) herstellen zu können, werden die folgenden Informationen benötigt: Jeweils eine Referenz auf den *EMUDataModel*-Typ und *DBModel*-Typ (ist nichts anderes als der eingegebene Name der Produktion). Eine weitere Referenz auf den Parentknoten (Produktion, welche eine Ebene höher steht wie die gerade zu verknüpfende Produktion). Das entsprechend zu dem Parentknoten angegebene Attribut (z.B. getWohnorte() aus unserem obigen Beispiel) und den entsprechenden MetaTyp der Parentproduktion (in unseren Fällen, kann es nur eine Liste oder ein Tupel sein). Warum genau diese Informationen notwendig sind, wird im Laufe des Kapitels noch näher erläutert werden.
Die Einbindung der SQL-Statements könnte wie oben beschrieben genau auf zweierlei Arten geschehen. Damit man die Sourcen von *EMUGEN* nicht erweitern muss, wird nach Eingabe der Daten (*EMUDataModel, DBModel, Mapping*) mit Hilfe von Visitoren eine *.emu*-Datei erzeugt, welche dann mit dem Prinzip von *EMUGEN* die Sourcen der Applikation erstellt, und gleichzeitig die einzelnen Queries mitgenerieren und an zwei eigene neue Emitter (Source-File-Generator) übermitteln, welche dann Aktionen definieren, die genau die Queries ausführen können (vergleiche Abbildung 4).

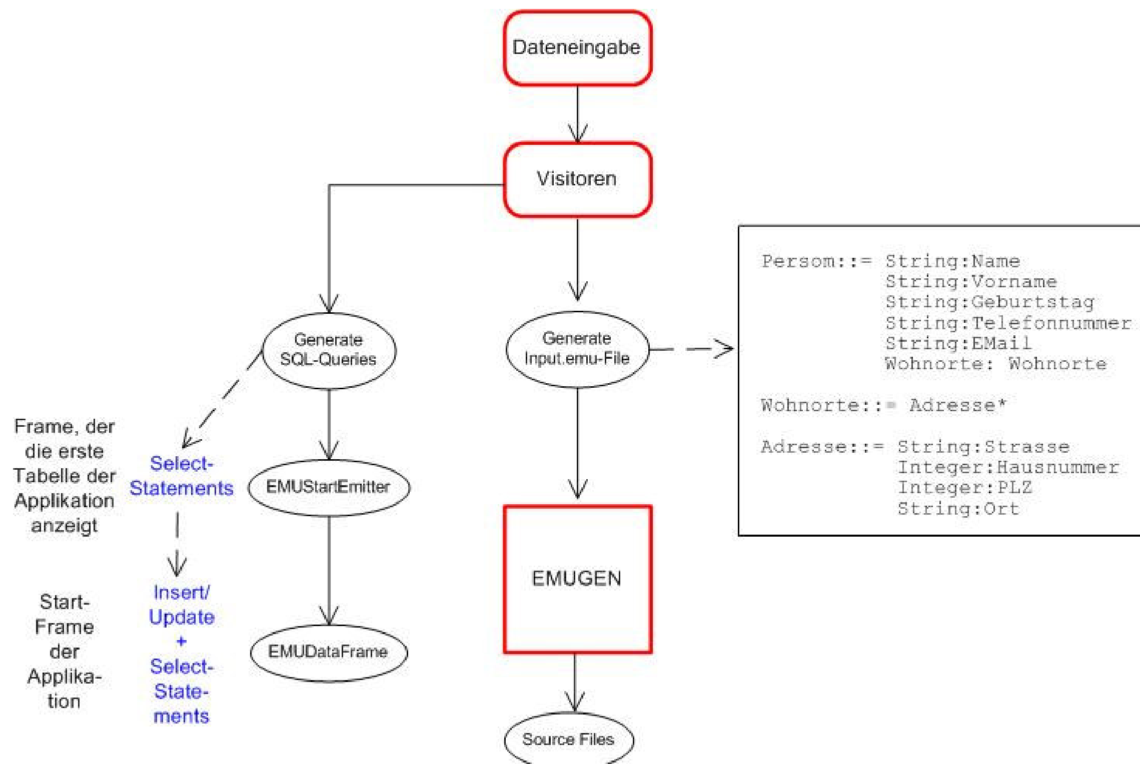

Abbildung 4: Grober Ablauf des Generierungsprozesses

Der *EMUStartEmitter* erzeugt aus der ersten Listenproduktion ein Frame, über welches man die Parameter zur Anbindung an eine Datenbank eingeben kann, und anschliessend die Daten aus der Datenbank laden kann, um diese weiter zu Bearbeiten oder neue Hinzuzufügen.
Der *EMUDataFrameEmitter* erzeugt ebenfalls ein Frame, welches als Rahmen der eigentlichen Applikation dient (ähnlich dem DataFrame von *EMUGEN*), und die beiden Aktionen zum Laden und Speichern von Datensätzen (können sowohl Insert´s, als auch Update´s sein) beinhaltet.

3.3 Konkrete Implementierung

Wie genau die einzelnen Schritte implementiert sind, und welche Änderungen und neuen Funktionen dafür notwendig sein werden, wird das folgende Kapitel genauer betrachten.

3.3.1 Veränderungen bzgl. der Vorgaben

Ausgehend von der IDE vom Dezember 2003 wurden die folgenden Änderungen vorgenommen (vergleiche Abstrakter Syntaxbaum Abbildung 5 und daraus resultierende GUI Abbildung 6 mit geladener Testeingabe).

1. Erstellung einer weiteren vierten Ebene *DBProperties* zur Eingabe der Datenbank-Parameter (Konfiguration der technischen Hilfsmittel)
2. Im Bereich des *DBModel* wurde ein drittes *DBAttribut* (*Length* – Länge des Datentyps in Bytes) hinzugefügt. (Weitere ließen sich ohne Weiteres integrieren)

3. Der Bereich *Mapping* wurde komplett überarbeitet und gliedert sich nun in folgende Bereiche:
    - *EMUType*: Referenz auf den entsprechenden *EMUDatamodel*-Typ
    - *DBType*: Referenz auf den entsprechenden *DBModel*-Typ
    - *RefParentType*: Gibt eine Referenz zu dem entsprechenden Elternknoten (wird nur bei *ComplexType* benötigt)
    - *RefParentAttribut*: Zeigt das entsprechende Parentattribut an, welches diesen *EMUTypen* referenziert
    - *RefEMUMetaType*: Referenziert den jeweiligen *MetaTyp* des Elternknoten (kann im Moment nur Liste oder Tupel sein)
4. Der Teil des EMUDatamodel wurde beibehalten.

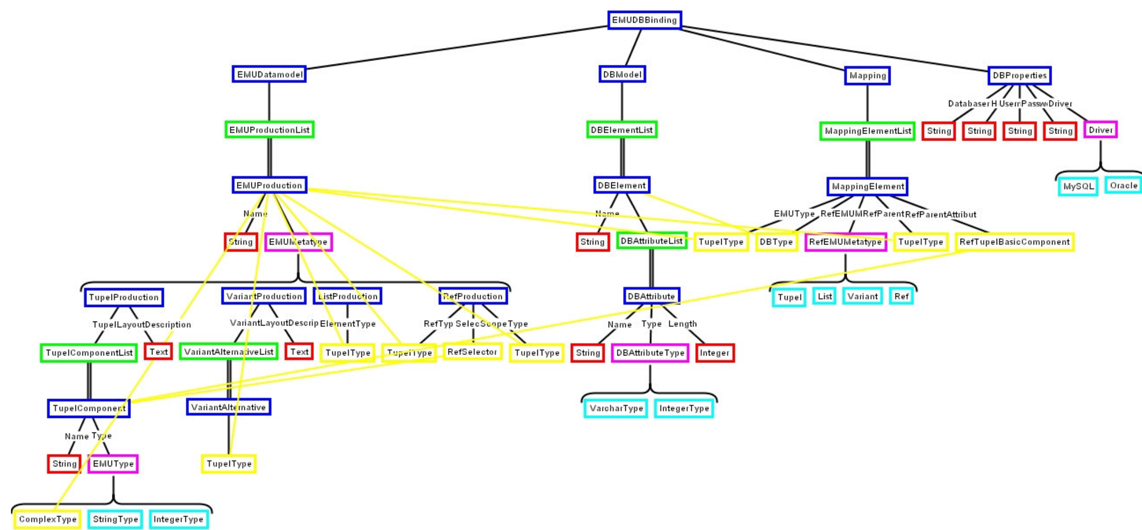

Abbildung 5: Abstrakter Syntaxbaum der veränderten Eingabe

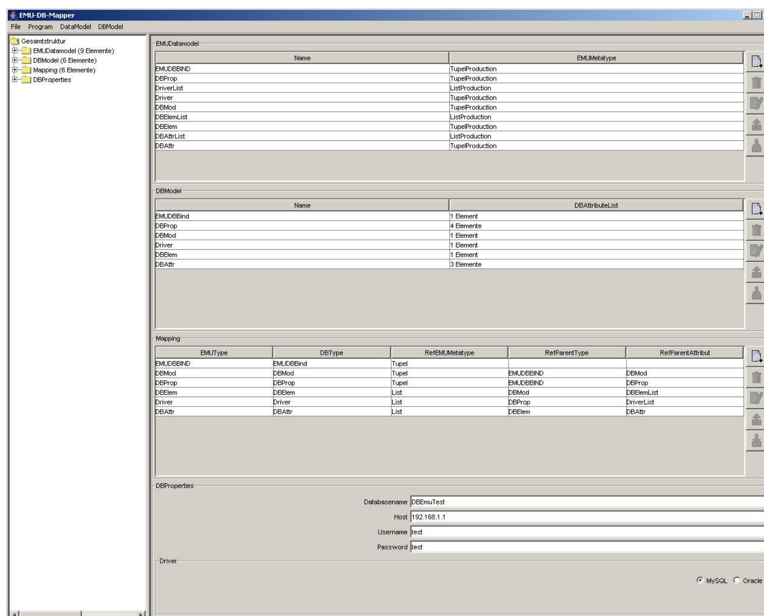

Abbildung 6: GUI der EMUGEN-Eingabe

Durch diese entsprechenden Erweiterungen lässt sich nun zu allen beliebigen Tupel- und Listenproduktionen eine GUI mit Datenbank-Anbindung generieren.

3.3.2 Erweiterungen der Visitoren

Um sich aus einer Eingabe die entsprechenden Files generieren zu lassen, benötigt man die sogenannten Visitoren. Sie dienen dazu, alle eingegebenen Parameter auszuwerten und entsprechend zu verarbeiten. Dies basiert auf dem Konzept von *EMUGEN*. *EMUGEN* ist nach dem Top-Down-Prinzip aufgebaut, sprich die eingegebenen Parameter werden von oben nach unten durchlaufen und entsprechend ihrer Herkunft zugeordnet. Verdeutlichen wir uns es an einem konkreten Beispiel. Ein Benutzer der Entwicklungsumgebung gibt folgendes Datenmodell ein.

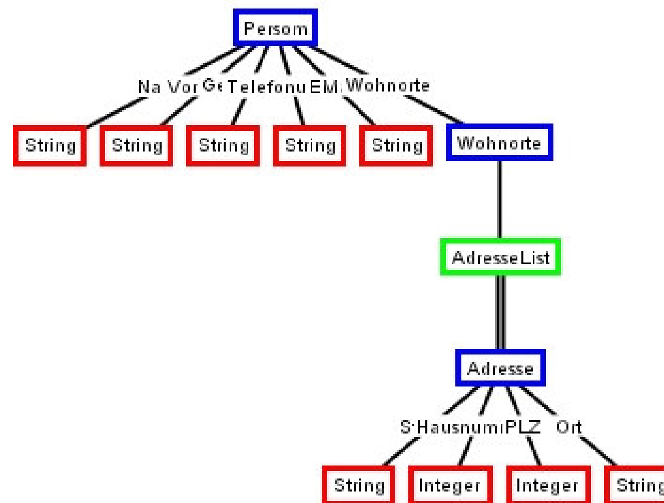

Abbildung 7: Person mit ihren zugehörigen Wohnorten

Dann bekommen wir mit Hilfe unseres einen Visitors ein *.emu*-File heraus, welches das folgende Aussehen hat.

```
Person::= String:Name
          String:Vorname
          String:Geburtstag
          String:Telefonnummer
          String:Email
          Wohnorte:Wohnorte

Wohnorte::= Adresse*

Adresse::= String:Strasse
           Integer:Hausnummer
           Integer:PLZ
           String:Ort
```

Der Visitor geht also den Syntaxbaum von oben nach unten durch und ordnet entsprechend den gefundenen Knoten deren Eigenschaften zu. Er weiss also, dass z.B. Person vom Typ *Tupelproduction* ist.

Somit erhält man also aus einer Eingabe der Entwicklungsumgebung ein *.emu*-File, welches dann widerum benutzt wird, die entsprechenden Sourcen zu generieren. (vergleiche Abbildung 4).
Um genau zu sein, gibt es in unserem Fall drei Visitoren, welche für die drei Teile *EMUDataModel, DBModel, Mapping* die einzelnen Informationen sammeln und entsprechend weitergeben.
Im ersten Schritt wird der sogenannte *EMUMappingTopDownVisitor* aufgerufen, welcher nichts weiter macht, als die Eingabe aus der GUI in Vektoren zu Speichern, damit die anderen beiden Visitoren über die Mappinginformationen verfügen können. So erhalten wir insgesamt 5 Vektoren, die die im Mapping-Schritt eingegebenen Werte (*EMUType, DBType, RefParentType, RefParentAttribut, RefEMUMetaType*) entsprechend ihrer Eingabe-Reihenfolge enthalten. Worauf beziehen sich die einzelnen Werte?
Als erstens wird nach dem entsprechenden *EMUType* gefragt, welcher die vorher angelegte DatenModelKlasse widerspiegelt. Ebenso verhält es sich mit dem *DBType*, welcher widerum die passende Klasse (hier besser Tabelle), aber diesmal aus dem *DBModel* angibt. Das dritte Attribut bezieht sich auf den entsprechenden Parent-knoten und ist wichtig für das spätere Erstellen der *SaveToDBAction*, da man wissen muss, auf welcher Ebene man im abstrakten Syntaxbaum sich befindet. Es dient zur Zuordnung der verschiedenen Tabellen, bzgl ihrer *foreign keys*. Die vierte Eingabe referenziert das entsprechende Attribut des Parentknoten, d.h. es ist der Name, des *ComplexTypes* der Parentklasse (vergleiche hierzu Abbildung 9, rot-markierter Text). Das letzte, was der Benutzer angeben muss, ist ob der Parentknoten eine Tupel-, Listen-, Varianten-, Referenzproduktion ist, da dies ebenfalls zum Erstellen der *SaveToDBAction* benötigt wird (der bisherige Stand erlaubt leider nur ein Selektieren von Tupel- oder Listenproduktionen).
Im zweiten Schritt werden im *EMUDBTopDownVisitor* alle in der GUI eingegebenen Datenbank-Tabellen erzeugt, indem bei jedem besuchten *DBElement* ein SQL-Query generiert wird und dieser an eine Klasse names *SQLConnector* übergeben wird, welche dann die Verbindung zur Datenbank aufbaut und die entsprechenden Tabellen generiert. Es werden zunächst für jede Tabelle ein Primary Key aus dem Tabellennamen und dem Wort ‚ID' gebildet. Im Moment wird dieser Primary Key mit dem Attribut Auto-Increment versehen, was ein automatisches Erhöhen des Keys beim Speichern in die Datenbank zur Folge hat. Anschliessend werden die einzelnen Attribute, welche vorher in der GUI spezifiziert wurden angelegt. Handelt es sich bei der gerade zu erzeugenden Tabelle nicht um die Anfangstabelle (Tabelle der ersten Tupelproduktion), so wird dieser ein Fremdschlüssel (*foreign key*) hinzugefügt, damit man später beim Laden weiss, zu welchen Einträgen in der Anfgangstabelle, die entprechenden Einträge in den weiteren Tabellen gehören. Dieser *foreign key* referenziert sich also auf die Parentknoten.
Diese Informationen kommen genau von den vorher gesammelten Mapping-informationen, welche den passenden *DBType* und dessen Referenz (den foreign key) auf den Parentknoten (*RefParentType*) enthalten (vergleiche die folgenden SQL-Statements zum Erzeugen der Tabellen).

```
CREATE TABLE IF NOT EXISTS Personen (PersonenID integer auto_increment
primary key, Name varchar(255), Vorname varchar(255), Geburtstag
varchar(255), Telefonnummer varchar(255), EMail varchar(255))

CREATE TABLE IF NOT EXISTS Adressen (AdressenID integer auto_increment
primary key, Strasse varchar(255), Hausnummer integer(11), PLZ integer(11),
Ort varchar(255), PersonID integer(11), foreign key (PersonID) references
Person on delete set null)
```

Der *EMUDataModelTopDownVisitor* besucht ebenfalls alle in der GUI eingegebenen Objekte und funktioniert ähnlich einem *EMUGEN*-Visitor. Allerdings gibt es zwei wesentliche Unterschiede. Zum einen erzeugt er ein *Input.emu* – File, (siehe oben) welches anschliessend zur Weiterverarbeitung benötigt wird. Als zweites erzeugt er für die erste Tupelproduktion (entspricht gleichzeitig der ersten Tabelle in der Datenbank) eine *SaveToDBAction* und *LoadFromDBAction* (vergleiche Abbildung 9), welche zum Speichern bzw. Laden der Daten benötigt werden. Diese werden einfach in einem StringBuffer gespeichert und an den entsprechenden Emitter übergeben. So spart man sich einen direkten Eingriff in die EMUGEN-Sourcen. Aber wie schaut diese Aktion genau aus?
Bei jeder besuchten Tupelproduktion, wird schon wie im *EMUDBTopDownVisitor* ein SQL-Query erzeugt, jedoch nicht ausgeführt, sondern nur als Aktionsaufruf mitübergeben, damit man dann aus der anschliessend erzeugten GUI die Daten speichern/laden kann. Um diesen Query zu erstellen, benötigt man wieder die Mappinginformationen. Man sucht sich zunächst den passenden *EMUType* der gerade besuchten Tupelproduktion heraus und speichert dessen Index. Dann wird überprüft, ob der *RefEMUMetaType* ein Tupel oder einen Liste war. Anschliessend überprüft man, ob die letzte besuchte Produktion ein *RefParentType* der aktuell Besuchten ist. Man benötigt jetzt nur noch das *RefParentAttribut,* um damit die Referenz vom Elternknoten auf den passenden Kinderknoten herstellen zu können (siehe rot-markierter Text im anschliessenden Codeauschnitt). Nun hat man alle Informationen beisammen, um einen SQL-Query zu erzeugen, und diesen anschliessend wieder mit dem *SQLConnector* zu verknüpfen, jedoch ohne ihn direkt auszuführen (vergleiche hierzu Abbildung 8).

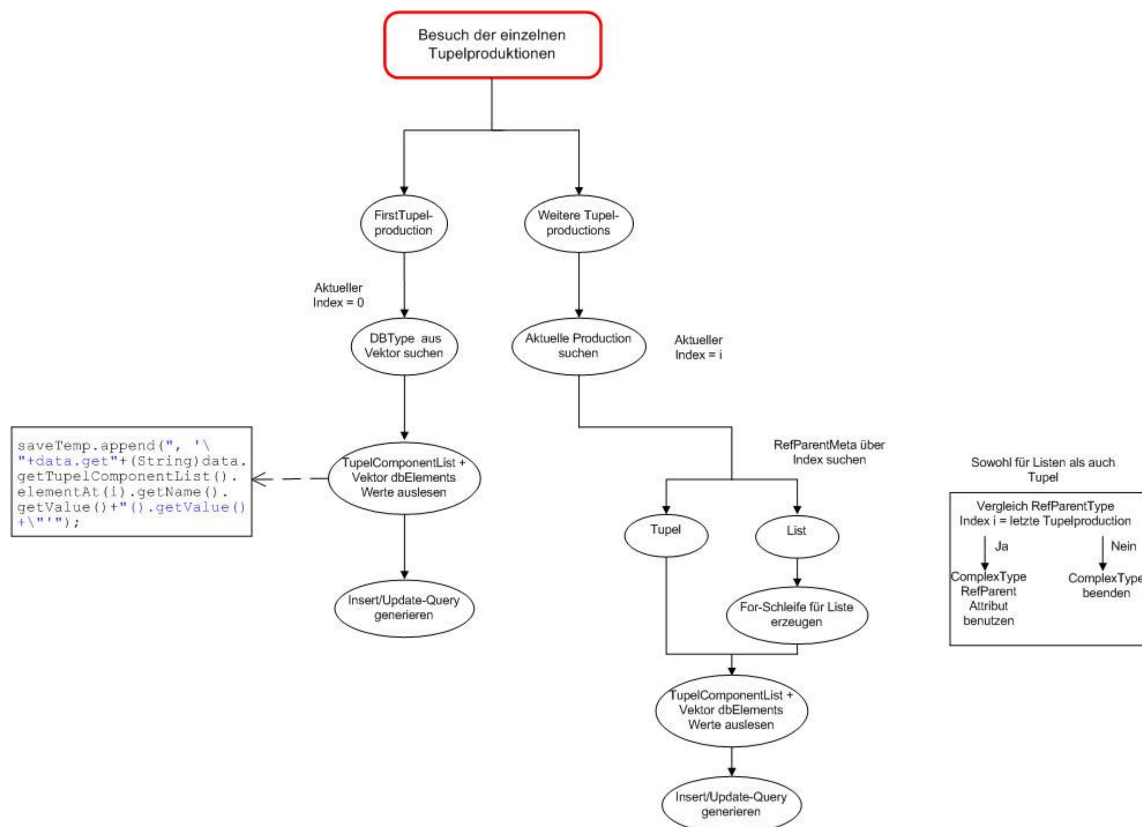

Abbildung 8: Grobe Darstellung der SQL-Query-Generierung mit Hilfe der Mappinginformationen

Nun hat man also alle Informationen aus der GUI der Entwicklungsumgebung verarbeit und an die entsprechenden Stellen weitergegeben.
Wie genau funktionieren aber die Anbindung an die Datenbank und das damit verbundene Ausführen der Queries?

SaveToDBAction:

```
int id = getPersonID().getValue();
if(emu_runtime.sql.Options.isNew == true)
{
emu_runtime.sql.SQLConnector.getInstance().insert("INSERT INTO Personen VALUES (PersonenID,
'"+data.getName().getValue()+"', '"+data.getVorname().getValue()+"',
'"+data.getGeburtstag().getValue()+"', '"+data.getTelefonnummer().getValue()+"',
'"+data.getEMail().getValue()+"')");
for(int a = 0; a < data.getWohnorte().size(); a++)
{
emu_runtime.sql.SQLConnector.getInstance().insert("INSERT INTO Adressen VALUES (AdressenID,
'"+data.getWohnorte().elementAt(a).getStrasse()+"',
"+data.getWohnorte().elementAt(a).getHausnummer()+",
"+data.getWohnorte().elementAt(a).getPLZ()+", '"+data.getWohnorte().elementAt(a).getOrt()+"',
"+id+")");
}
}
else
{
emu_runtime.sql.SQLConnector.getInstance().updateTupel("Person", "PersonID", "Name =
'"+data.getName().getValue()+"', Vorname = '"+data.getVorname().getValue()+"', Geburtstag =
'"+data.getGeburtstag().getValue()+"', Telefonnummer = '"+data.getTelefonnummer().getValue()+"',
EMail = '"+data.getEMail().getValue()+"'", id);
emu_runtime.sql.SQLConnector.getInstance().truncateTable("Adressen");
for(int a = 0; a < data.getWohnorte().size(); a++)
{
emu_runtime.sql.SQLConnector.getInstance().updateList("INSERT INTO Adressen VALUES
(AdressenID, '"+data.getWohnorte().elementAt(a).getStrasse()+"',
"+data.getWohnorte().elementAt(a).getHausnummer()+",
"+data.getWohnorte().elementAt(a).getPLZ()+", '"+data.getWohnorte().elementAt(a).getOrt()+"',
"+id+")");
}
}
```

LoadFromDBAction:

```
int parentID = data.getPersonID().getValue();
java.sql.ResultSet result1 = emu_runtime.sql.SQLConnector.getInstance().load("Adressen", "Person",
parentID);
try
{
int count = 0;
while(result1.next())
{
Adressen dataMod = new Adressen();
dataMod.getStrasse().setValue(result1.getString(2));
dataMod.getHausnummer().setValue(Integer.parseInt(result1.getString(3)));
dataMod.getPLZ().setValue(Integer.parseInt(result1.getString(4)));
dataMod.getOrt().setValue(result1.getString(5));
data.getWohnorte().append(dataMod);
count++;
}
result1.close();
}
catch(Exception ex) {}
```

Abbildung 9: Generierter Java-Code mit entsprechendem SQL-Query

3.3.3 Implementierung der Datenbank-Anbindung

Um die soeben erzeugten Queries ausführen zu können, bzw. um überhaupt eine Verbindung mit einer Datenbank herzustellen, benötigt man den schon mehrfach erwähnten *SQLConnector*. Er stellt alle Funktionen bereit, die man zum Arbeiten auf einer Datenbank benötigt. Neben der Funktion, eine Verbindung auf- und abbauen zu können, stellt er für jedes SQL-Statement eine eigene Methode bereit. Der folgende Codeausschnitt soll anhand von zwei Methoden, desen Funtkionsweise verdeutlichen.

```
public void insert(String query)
{
	Statement statement = null;

	System.out.println(query);

	try
	{
		if(!isOpen)
		{
			open();
		}

		statement = connection.createStatement();
		statement.executeQuery(query.toString());
		statement.close();

		if(autoclose)
		{
			close();
		}
	}
	catch(Exception ex)
	{
		Object[] object = new Object[] {
			"Cannot insert into Table! "+ex.getMessage()+". \n" +
			"Please check Database Properties!"
		};

		JOptionPane.showMessageDialog(new JFrame(), object, "Error",
JOptionPane.INFORMATION_MESSAGE);

		Options.GOON = false;
	}
}

public ResultSet load(String tableName, String columnName, int id)
{
	ResultSet result = null;
	Statement statement = null;
	StringBuffer query = null;

	try
	{
		query = new StringBuffer("SELECT * FROM "+tableName+" WHERE
"+columnName+"ID = "+id+"");

		System.out.println(query);

		if(!isOpen)
```

```
            {
                  open();
            }

            statement = connection.createStatement();
            result = statement.executeQuery(query.toString());

            statement.close();

            if(autoclose)
            {
                  close();
            }

            return result;
      }
      catch(Exception ex)
      {
            Object[] object = new Object[] {
                  "Cannot load from Table! "+ex.getMessage()+"." +
                  "Please check Database Properties!"
            };

            JOptionPane.showMessageDialog(new JFrame(), object, "Error",
JOptionPane.INFORMATION_MESSAGE);

            Options.GOON = false;

            return null;
      }
}
```

Die beiden Methoden dienen zum Speichern und Laden von Datensätzen in einer Datenbank. Das prinzipielle Vorgehen, bei all diesen Methoden, ist immer zunächst zu schauen, ob bereits eine Verbindung mit einer Datenbank besteht. Wenn nicht wird sie hergestellt, ansonsten wird einfach sofort mit dem Erzeugen und Ausführen des Statements begonnen. War es erfolgreich, wird die Verbindung geschlossen, ansonsten, wird eine Fehlermeldung bzgl. des Problems angezeigt. Im Falle eines *Select*-Statements wird zudem noch ein Java.sql.Resultset zurückgeliefert, welches die selektierten Werte enthält und entsprechend zum Weiterverabeiten der Werte verwendet werden kann (vergleiche *LoadFromDBAction*).
Zu diesen beiden Methoden gibt es, wie gesagt eine ganz Reihe weiterer Methoden, welche von ihrer Funktionsweise im Wesentlichen diesen beiden ähneln. Natürlich können jederzeit weitere Funktionsweisen von MySQL implementiert werden. Darunter können das *Delete* und *Alter*-Statement fallen.
Bei einem eventuellen Versuch sich auf eine Oracle Datenbank zu verbinden und Queries auszuführen, müsste man unter Umständen gewisse Funktionen ändern. Dies wird aber hauptsächlich in der Syntax der Queries sein, da ansonsten Java eine sehr allgemeine Klasse (Java.sql) zum Verarbeiten von SQL-Statements entwickelt hat. Da ich leider keine Oracle-Datenbank zur Verfügung hatte, kann ich also keine Allgemeingültigkeit der Klasse *SQLConnector* geben, dennoch wird der Hauptanteil der Anwender der Entwicklungsumgebung mit einer MySQL-Datenbank arbeiten, und dafürkann ich eine fast 100-prozentige Garantie geben, dass die implementierten Methoden reibungslos funktionieren werden.

3.3.4 Ergänzungen/Erweiterungen des *emu_runtime* Paketes

Um später mit der generierten Applikation arbeiten zu können, wurde ein neues Unterpaket dem *emu_runtime* Paket hinzugefügt. Es hat den Namen *emu_runtime.sql* und enthält die Klassen, welche die Applikation zur Anbindung an die Datenbank benötigt. Hierfür wird neben dem in Punkt 3.3.3 beschriebenen *SQLConnector* auch die Klasse *DBProperties* und *DBPropertiesFrame* benötigt, welche dem Benutzer über eine Eingabemaske ermöglichen, die Parameter der Datenbank (Datenbankname, Host, Benutzername und Passwort) einzugeben. Diese werden dann in der Modellklasse *DBProperties* gespeichert und können während der Arbeit mit der generierten Applikation verändert werden (vergleiche Abbildung 10).

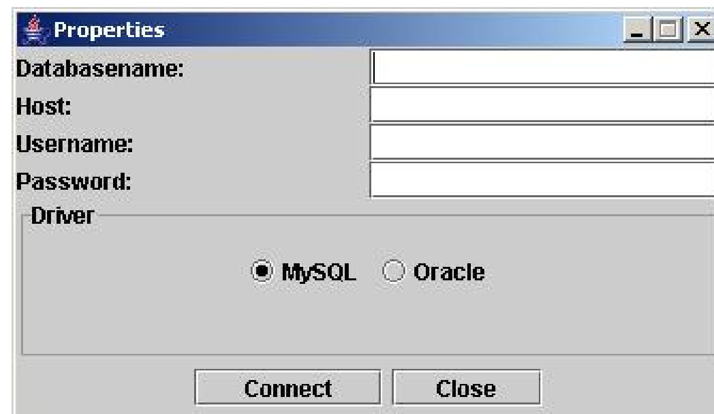

Abbildung 10: DBPropertiesFrame zum Eingaben der Datenbankparameter

Warum hat man die Möglichkeit in der generierten Applikation die Datenbank-parameter nochmals zu ändern?
Damit jeder Benutzer unabhängig von seinen Arbeitsplatz mit einer Datenbank arbeiten kann, wurde das *DBPropertiesFrame* mit in die generierte Applikation integriert. Es hat also den großen Vorteil, dass man jederzeit egal in welchem Netzwerk man sich befindet, eine Verbindung zu einer Datenbank herstellen kann. Somit kann eine relativ system-unabhängige Lösung generiert werden, welche direkt aus dem erzeugten JAR-Archiv gestartet werden kann.
Um der Entwicklungsumgebung gewisse Aktionen, sprich eine Dialogsteuerung integrieren zu können, musste die Klasse *TreeFrame* aus dem *emu_runtime* Paket verändert und erweitert werden. Ihr wurde zunächst die Eigenschaft eines Javax.swing.JMenu hinzufügt und mit den einzelnen Aktionen zum Erstellen eines fertigen, aus der Enwicklungsumgebung generierten Programm, versehen. Dafür waren die folgenden beiden Erweiterungen notwendig.

- Erstellen einer weiteren Klasse, names *Helper*, welche die Funktionalitäten zum Generieren der Sourcen, Compillieren der Sourcen, Erstellen eines JAR-Archives aus den compilierten Sourcen und schliesslich zum Ausführen des fertigen Programmes bereitstellt (ist ähnlich dem Compilerbau Paket von *VISUAL EMUGEN*)

- Integration der oben erwähnten Funktionen in die entsprechenden Aktionen der Menu-Struktur (vergleiche den folgenden Codeausschnitt)

```
if(e.getActionCommand().equals("Generate Sources"))
{
	try
	{
		tempDir = File.createTempFile("emu", null);
		tempDir.delete();
		tempDir.mkdir();
	}
	catch(Exception ex) { }

	File file = new File(tempDir, "src");
	file.mkdir();

	filePath = file.getAbsolutePath();

	GlobalOptions.overwrite = true;
	GlobalOptions.GEN_FORM_ONLY = true;
	GlobalOptions.GEN_TABLE = true;
	GlobalOptions.GENERATION_DIRECTORY = file;

	String emuFileName = EMUDBBindingActionDelegator.generate(data,
filePath);

	if(emuFileName != null && Options.GOON)
	{
		Main.generate(filePath + System.getProperty("file.separator") +
emuFileName);

		generate.setEnabled(false);
		compileClasses.setEnabled(true);

		EMUDataFrameEmitter frameEmitter = new
EMUDataFrameEmitter(filePath);
		EMUStartEmitter startEmitter = new
EMUStartEmitter(EMUDataModelTopDownVisitor.getFirstProduction(), filePath);

		Helper.deleteOnExit(file);

		File runtimeFile = new File(tempDir.getAbsolutePath() +
System.getProperty("file.separator") + "emu_runtime.jar");
		File driverFile = new File(tempDir.getAbsolutePath() +
System.getProperty("file.separator") + "mm.mysql-2.0.4-bin.jar");

		File emuRuntime = new File("librarys" +
System.getProperty("file.separator") + "emu_runtime.jar");
		File emuDriver = new File("librarys" +
System.getProperty("file.separator") + "mm.mysql-2.0.4-bin.jar");

		Helper.copyFile(emuRuntime, runtimeFile);
		Helper.copyFile(emuDriver, driverFile);
	}
	else
	{
		generate.setEnabled(false);
		compileClasses.setEnabled(false);
	}
}
```

Die oben dargestellte Methode generiert aus den eingegebenen Werten der GUI die Source-Files, welche zunächst mit den in Punkt 3.3.2 bechreibenen Visitoren ein *.emu*-File erzeugt, und anschliessend mit diesem Input, mit Hilfe von *EMUGEN* die

Sourcen der zu generierenden Applikation in einem Temp-Verzeichnis, erstellt. Anschliessend werden mit den beiden Emittern (*EMUStartEmitter, EMUDataFrameEmitter* – Details 3.2 Gewähltes Konzept) die zusätzlichen Files generiert und ebenfalls in dem Temp-Verzeichnis gespeichert. Weiterhin werden die beiden JAR-Archive (*emu_runtime.jar und mm.mysql-2.0.4-bin.jar*) auch in dieses Temp-Verzeichnis kopiert um anschliessend die gesamten Sourcen fehlerfrei zu compilieren (vergleiche hierzu Abbildung 17 in Kapitel 4).

4. Anwendung

Nach den Erläuterungen der Implementierung kommen wir nun zum direkten Umgang mit der fertigen Entwicklungsumgebung.

4.1 Möglichkeiten der Benutzung

Im Wesentlichen gibt es zwei grundsätzliche Arten die Enwicklungsumgebung zu benutzen.
Einerseits wird sie als kompaktes ZIP-Archiv bereitgestellt, welches alle notwendigen Sourcen enthält. Es besteht aus einem JAR-Archiv, welches das Programm in compilierter Form enthält und auf jedem java-fähigen Rechner gestartet werden kann. Darüberhinaus enthält es einen Order names *libarys*, welcher alle notwendigen zusätzlichen Pakete enthält (emu_runtime, grace, jflex, cup und den MySQL-Treiber). Wichtig ist, dass sich sowohl das JAR-Archiv des Programmes und der Ordner *libarys* in einem gemeinsamen Verzeichnis befinden, denn nur dann kann das Programm fehlerfrei gestartet werden.
Neben dieser Möglichkeit gibt es noch eine weitere, und zwar wird das Programm als ANT-Version bereitgestellt. Dies bietet neben der normalerweise garantierten Systemunabhängigkeit noch einen weiteren großen Vorteil. Mit diesem ANT-File lässt sich das Programm z.B. leicht in Eclipse integrieren und kann von dort aus direkt über das ANT-File gestartet werden. Somit steht dem Benutzer oder Entwickler die Möglichkeit offen, weitere Punkte in das Programm zu integrieren und es immer aus Eclipse heraus direkt zu compilieren und zu starten.

4.2 Umgang mit dem entwickelten Programm

Hat man das Programm, auf einen der beiden oben genannten Wege gestartet, beginnt nun die eigentliche Arbeit mit der Entwicklungsumgebung. Zunächst kommt der Benutzer auf ein „leeres“ Fenster, welches noch keine Daten enthält. Zu Beginn hat er genau zwischen vier Aktionen die Auswahl (vergleiche hierzu Abbildung 11).

- Eingaben der Daten zum Erstellen einer Applikation
- Laden einer alten Eingabe (*Menu – File - Open*)
- Datenbank-Parameter einstellen (*Menu – Database - Properties*)
- Programm beenden (*Menu – File - Exit*)

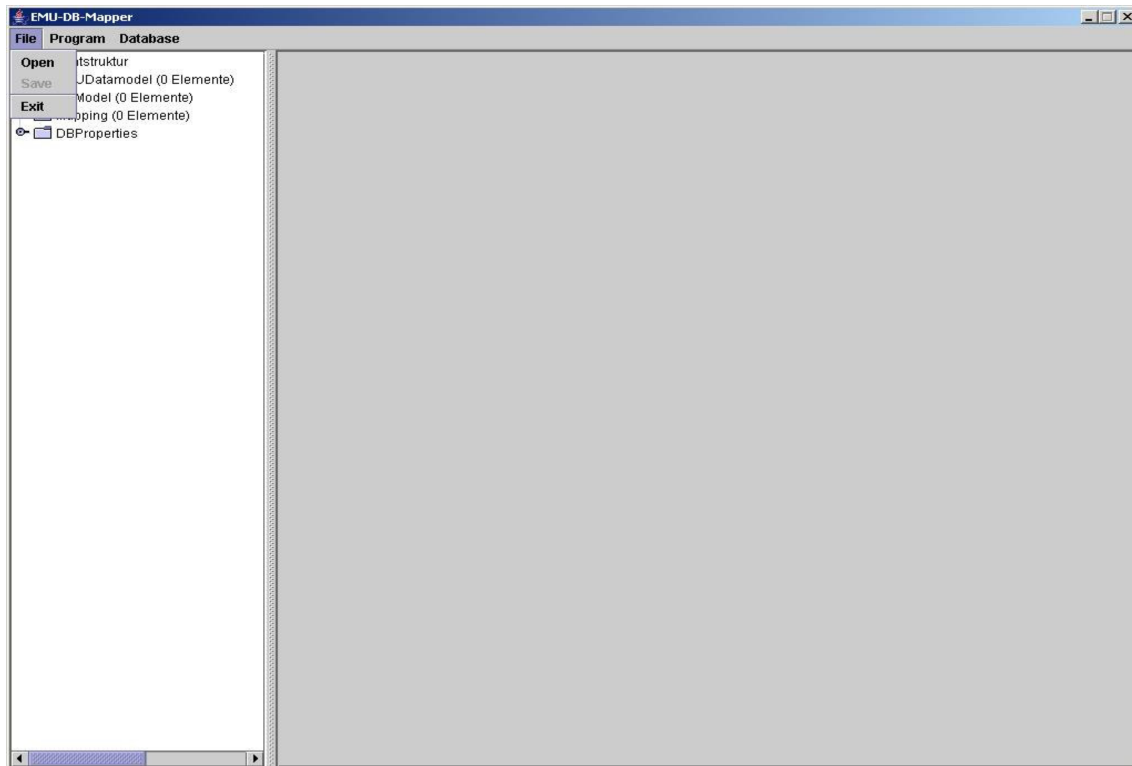

Abbildung 11: Startbildschirm der Entwicklungsumgebung

Enscheidet er sich zum Laden eines Files, dann würde sich der Bildschirm wie folgt ändern (wir bleiben bei dem Beispiel aus Kapitel 3 – Laden der Eingabe der Personendaten)

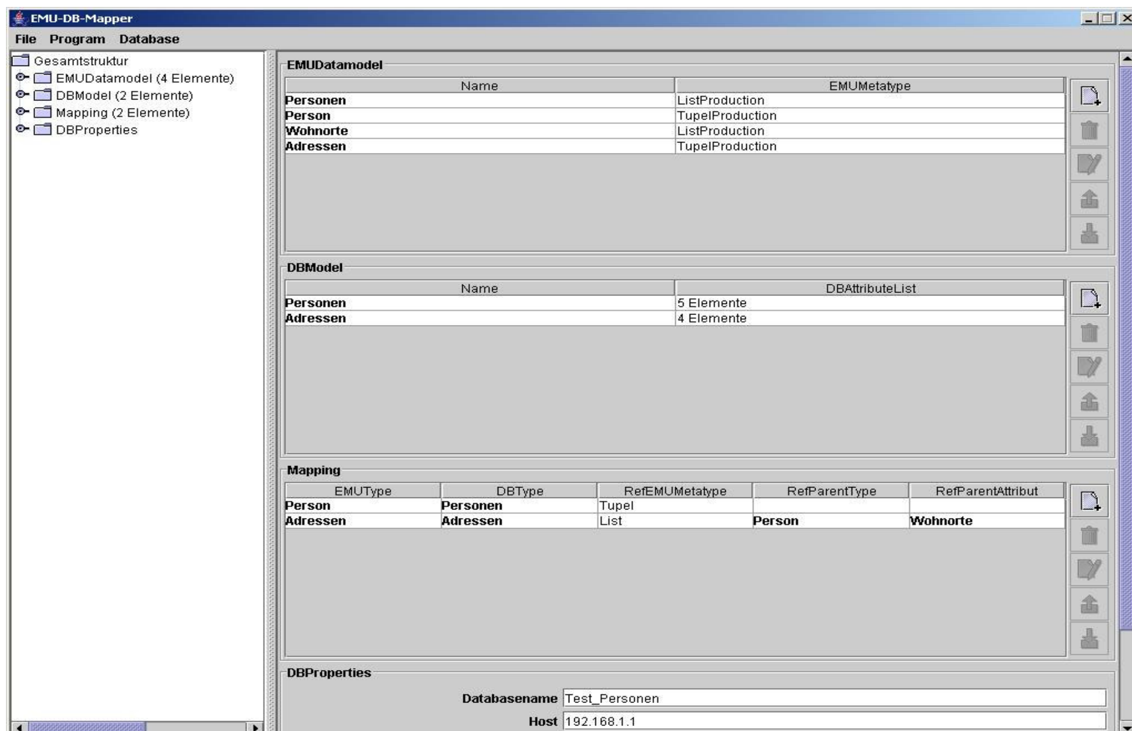

Abbildung 12: Laden eines Files mit bereits vorhandener Eingabe

Hat man nun die Daten geladen kann man mit ihnen eine Applikation erstellen. Zuerst muss man unter *Menu-Program-Generate Sources* die Source-Files generieren. Diese werden im Temp-Verzeichnis angelegt. Anschliessend müssen sie mit *Menu-Program-Compile* compiliert werden. Falls ein Fehler bei der Compilierung auftritt, bricht das Programm mit einer Fehlermeldung ab. Ist dies nicht der Fall, öffnet sich ein Dateimanager, über welchen man die compilierten Sourcen als JAR-Archiv speichern kann (siehe Abbildung 13).

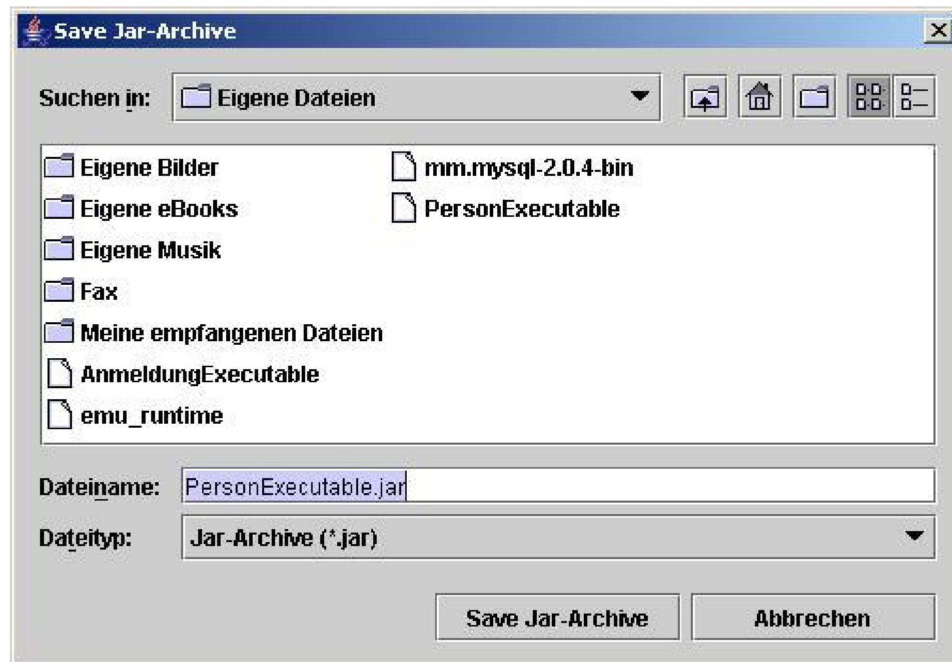

Abbildung 13: Filechooser zum Speichern des Jar-Archives

Hat man ein entsprechendes Verzeichnis ausgewählt, speichert man das File ab. Bei bereits vorhandenem Namen kommt eine Meldung mit passendem Hinweis zum Überschreiben.
So nun kann man das Programm mit *Menu-Program-Run Program* starten. In unserem Fall würde der Startbildschirm folgendermassen ausschauen (siehe Abbildung 14).
Jetzt kann man mit dem gerade erzeugten Programm arbeiten und seine Eingaben in der Datenbank speichern.
Hat man kein File zur Hand, welches eine Testeingabe enthält muss man bevor man die Sourcen generieren und das Programm starten kann, alle Informationen in die Entwicklungsumgebung eingeben. Zuerst muss man die Datenmodel-Klassen in die GUI eingeben. Hierbei erstellt man analog zu *Classgen* oder *EMUGEN* das Datenmodel, und zwar nicht direkt über die Eingabe der Grammatiksprache, sondern interaktiv über die GUI. Hierbei ist lediglich darauf zu achten, dass man keine Referenzen vergisst einzugeben (z.B. auf welchen Tupeltyp sich eine Listenproduktion bezieht). Ist dies jedoch der Fall, bricht das Programm automatisch bei der Generierung der Sourcen mit eine Fehlermeldung ab.
Die Eingabe des *DBModel* gestaltet sich sehr analog zu der des Datenmodels. Man sollte hier wissen, dass man nur Datenbanktabellen (*DBTable*) und keine Datenbankviews (*DBView*) erstellen kann (war in der alten Version noch mit inbegriffen, ist aber in der aktuellen nicht vorgesehen und kann bei Bedarf wieder eingebaut werden – man muss dann das Konzept der Datenbank-Tabellenerzeugung

wieder überarbeiten). Eine typische Eingabe zeigt Abbildung 15. Hier werden die jeweiligen Spaltenattribute mit den jeweiligen Datentypen und deren Länge eingegeben. Die Mappinginformationen können logischerweise erst zum Schluss eingetragen werden, da man sowohl die Informationen aus dem *EMUDataModel* als auch *DBModel* benötigt.

Abbildung 14: Startbildschirm der generierten Applikation

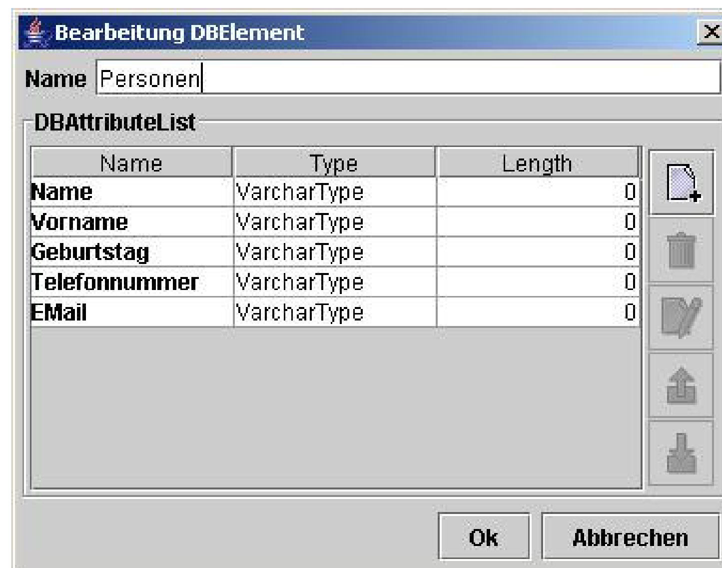

Abbildung 15: Eingabe des *DBModels*

Hat man nun alle Informationen eingetragen, muss man, bevor die Sourcen generiert werden können, unter *Menu – Database – Properties* die Datenbank-Parameter eintragen und eventuell eine Datenbank anlegen. Ist bereits eine Datenbank vorhanden, kann man sofort die Verbindung testen, indem man auf den *Connect*-Button

drückt. Es besteht ebenfalls noch die Möglichkeit, eine bereits existierende Datenbank zu löschen (*Drop DB*-Button). Hat man nun eine Verbindung hergestellt, bzw. eine Datenbank erstellt, kann man wie schon oben bei der geladenen Eingabe mit der Generierung der Sourcen beginnen (siehe Abbildung 16).

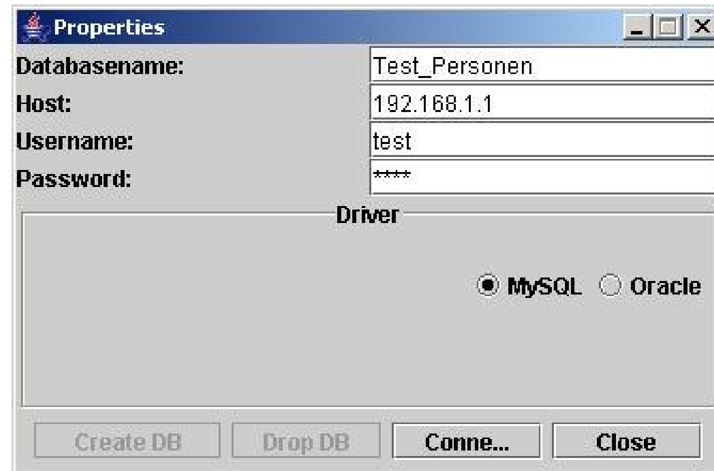

Abbildung 16: Eingabe der Datenbankparameter

Die folgende Graphik soll nochmal die grundlegenden Schritte und deren dazu gehörigen Abläufe verdeutlichen und dem Benutzer einen Leitfaden zur Bedienung des Programmes bieten.

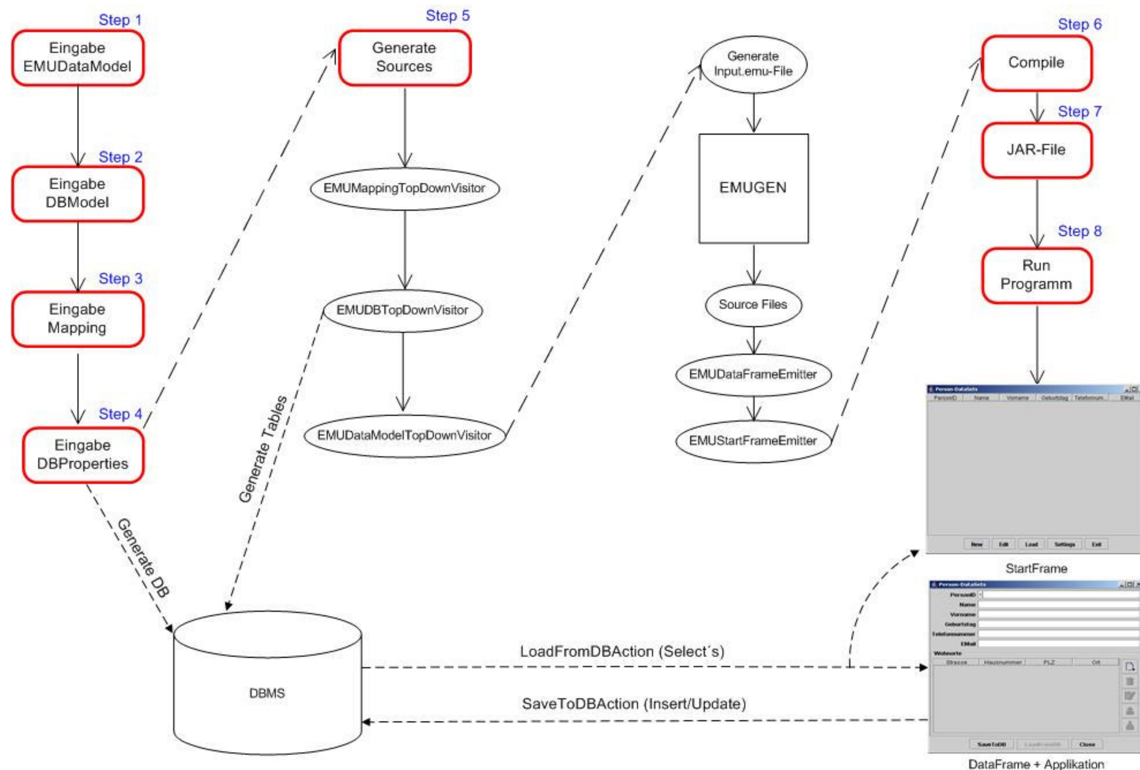

Abbildung 17: Ablaufszenario des EMU-DB-Mappers

Die in der Graphik rot-markierten Schritte sind vom Benutzer auszuführen, wohingegen alle anderen Abläufe automatisch von Programm übernommen werden.

5. Fazit

Anlässlich der Anforderungen einer automatischen Generierung von Benutzungsoberflächen mit gleichzeitiger Anbindung an eine Datenbank, bindet diese Aufgabe zwei sehr interessante Themen. Zum einen soll eine sehr einfache Applkation erstellt werden, welche aber gleichzeitig ein sehr komplexes Backend beinhaltet. Daher stellte die Verknüpfung der beiden Gebiete eine sehr interessante, aber auch sehr schwierige Aufgabe an den Bearbeiter. Dennoch muss abschliessend gesagt werden, dass gerade solche Themen in der Informatik mit Zukunft behaftet sind, und mir schon jetzt ein gelungener Einblick ermöglicht wurde.

5.1 Erreichte Ziele

Grundsätzlich wurde das Ziel, eine Anbindung von automatisch generierten Benutzungsoberflächen an eine Datenbank im Großen und Ganzen erreicht. Leider wie schon mehrfach erwähnt, musste man in gewissen Dingen Abstriche machen. So ist im Moment nur eine Anbindung von Tupel- und Listenproduktionen mit der Datenbank möglich. Selbst dieses Ziel war nicht ohne kleinere Schwierigkeiten erreichbar. Da man schon zur Generierungszeit sämtliche SQL-Queries parat haben muss, um diese später aus der Applikation heraus zu benutzen, war es von Nöten, bereits beim Anlegen des *.emu*-Files, diese aus den eingegebenen Werten zu erzeugen. Vielleicht hätte man hier doch den besseren Ansatz, den Eingriff in die Sourcen von *EMUGEN* wählen sollen, dies wäre zwar anfangs mehr Aufwand gewesen, aber vielleicht am Ende mit noch mehr Erfolg gekrönt. So muss man halt mit der Einschränkung auf Tupeln und Listen leben, aber selbst damit lassen sich viele Szenarien durch-probieren. Eine weitere Einschränkung, jedoch nicht wirklich von großer Bedeutung, ist die Tatsache, dass man sein *EMUDataModel* **immer** mit einer Listenproduktion beginnen muss. Nur so kann die spätere Funktionalität, wie sie im Moment implementiert ist, ausgeschöpft werden. Worauf man auch unbedingt bei der Eingabe der Daten achten muss, ist die Reihenfolge der einzelnen Attribute beim *EMUDataModel* und *DBModel*. Diese müssen absolut identisch in ihrer Anordnung sein, ansonsten kann keine Zuordnung der Werte in der Datenbank garantiert werden.
Was die Systemunabhängigkeit angeht, sollten keine großen Probleme auftreten, da Java sowieso als system-übergreifende Sprache entwickelt wurde.
Auch die Flexibilität in Bezug auf die Datenbank-Parameter sollte im Großen und Ganzen erreicht sein. Der Benutzer kann hier seine individuellen Einstellungen vornehmen und diese dann entsprechend benutzen. In Bezug auf den Datenbanktreiber ist es etwas schwierig hinsichtlich der Austauschbarkeit. Zwar müsste nur an einer Stelle im Quellcode der Treiber geändert werden, jedoch war es über die Oberfläche nut bedingt möglich, und daher wurde es nicht eingesetzt.
Die Möglichkeit, die generierte Applikation auch ausserhalb der eigentlichen Entwicklungsumgebung benutzen zu können, wurde ebenfalls integriert. Das Prinzip basiert hier, wie bei *VISUAL EMUGEN,* auf dem Export eines JAR-Archives, welches dann auf einem beliebigen anderen java-fähigen Rechner benutzt werden kann. Genau aus diesem Grund wurde die Funktionalität der Änderungen der Datenbank-Parameter mit in die generierte Applikation eingebunden.
Hinsichtlich der Übersicht und Benutzbarkeit des Programmes müssen erste Reaktionen von anderen Benutzern abgewartet werden. Meines Erachtens stellt die

GUI der Entwicklungsumgebung einen sauberen Rahmen dar, welcher natürlich noch um gewisse Funktionalitäten erweitert werden kann.
Die weiteren „kleineren“ Ziele waren aufgrund des Zeitmangels leider nicht zu realisieren.

5.2 Konkrete Erweiterungsvorschläge

Neben den kleineren Zielen (genaueres siehe Kapitel 2 Punkt 2.2) könnte man die Applikation mit folgenden Punkten ergänzen und damit noch Benutzerfreundlicher machen.

- Integration einer SQL-Eingabe, mit der man direkt SQL-Queries auf die Datenbank ausführen könnte
- Eine *DeleteFromDB*-Funktion, mit der man Datensätze aus der Datenbank löschen könnte
- Erweiterung des *DBModel* im Hinblick auf weitere SQL-Attribute wie, direkte Angabe von Primary Keys und Indexen, Angabe von Auto_Increment Spalten, weitere Datentypen (Float, Double, Enumerations...)
- Bessere Fehlerbehandlung im Falle von nicht zugeordneten Attributen
- Erweiterungen der Entwicklungsumgebung im Sinne von Dialogsteuerung und Eingabehilfen
- Hinweise beim Einfügen, Laden und Updaten aus der generierten Applikation, im Hinblick auf Erfolg oder Misserfolg der Datenbank-Anfragen
- Online Hilfe bzw. integriertes Hilfesystem
- ...

Es gibt sicherlich noch viele weitere Projekte, welche man im Anschluss an dieses ansätzen könnte, aber die oben genannten Punkte sollen erste Stützen für weitere Aufgaben und Ergänzungen sein.

5.3 Ausblick

Für die Zukunft würde ich mir persönlich wünschen, dass es noch viele Änderungen und Erweiterungen über dieses Projekt hinaus gibt, und man irgendwann in der Lage sein sollte, sämtliche Benutzungsoberflächen, mit welchem Backend auch immer, aus *EMUGEN*, oder wie immer es heissen wird, generieren kann.
Ich möchte mich sowohl bei Herrn Dr. Alfons Brandl, der immer ein offens Ohr für mich hatte, und mit dem ich heisse Diskussionen in den verschiedensten Punkten führen konnte, bedanken. Auch Frau Riitta Höllerer möchte ich meinen Dank aussprechen, denn ohne Sie wäre wahrscheinlich das Projekt erst gar nicht entstanden.