HOCHSCHULE KONSTANZ TECHNIK, WIRTSCHAFT UND GESTALTUNG
UNIVERSITY OF APPLIED SCIENCES

Modellierung und Generierung von Testdaten für Datenbank-basierte Anwendungen

Nikolaus Moll
287336
Konstanz, 25. Oktober 2013

Master-Arbeit

Master-Arbeit

zur Erlangung des akademischen Grades

Master of Science

an der

Hochschule Konstanz
Technik, Wirtschaft und Gestaltung

Fakultät Informatik
Studiengang Master Informatik

Thema: **Modellierung und Generierung von Testdaten für Datenbank-basierte Anwendungen**

Verfasser: Nikolaus Moll
TODO
TODO TODO

1. Prüfer: TODO TODO
TODO
TODO
TODO TODO

2. Prüfer: PRUEFERBTITLE PRUEFERB
TODO
TODO
TODO TODO

Abgabedatum: 25. Oktober 2013

Abstract

Thema:	Modellierung und Generierung von Testdaten für Datenbank-basierte Anwendungen
Verfasser:	Nikolaus Moll
Betreuer:	TODO TODO PRUEFERBTITLE PRUEFERB
Abgabedatum:	25. Oktober 2013

Das Abstract befindet sich in `formal/abstract.tex`.

To do (1)

Ehrenwörtliche Erklärung

Hiermit erkläre ich *Nikolaus Moll,* geboren am *22.12.1981* in *TODO,* dass ich

(1) meine Master-Arbeit mit dem Titel

Modellierung und Generierung von Testdaten für Datenbank-basierte Anwendungen

selbständig und ohne fremde Hilfe angefertigt und keine anderen als die angeführten Hilfen benutzt habe;

(2) die Übernahme wörtlicher Zitate, von Tabellen, Zeichnungen, Bildern und Programmen aus der Literatur oder anderen Quellen (Internet) sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen innerhalb der Arbeit gekennzeichnet habe.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben kann.

Konstanz, 25. Oktober 2013

Nikolaus Moll

Inhaltsverzeichnis

Abstract **v**

Ehrenwörtliche Erklärung **vii**

1 Einleitung **1**

1.1 Zielsetzung der Arbeit 1

1.2 Aufbau der Arbeit 1

2 Grundlegende Konzepte **3**

2.1 Modellgetriebene Software-Entwicklung 3

2.2 Software-Tests 3

2.3 Konventionen für Diagramme 5

2.4 STU (Simple Test Utils) 6

3 Anforderungsanalyse **9**

3.1 Allgemeine Anforderungen 9

3.2 Modellierungskonzepte für Beziehungen 10

3.3 Fortlaufendes Beispiel 11

3.4 Modellierungsvarianten der Testdaten für DbUnit 14

4 Entwurf einer Modellierungssprache für Testdaten **23**

4.1 Entwurf der DSL 23

4.2 Wahl der Technologie 27

4.3 Ergebnis: Definition der DSL für Testdaten 31

4.4 Beispiel-DataSet in Groovy 32

5 Realisierung der Sprache **35**

5.1 Änderungen am Generator-Modell 36

5.2 Neue DataSet-Builder-Klassen 40

5.3 Tabellenparser 41
5.4 Referenzen und Scopes 41
5.5 Nutzung des DataSets in Unit-Tests 43
5.6 Komposition von DataSets 44
5.7 Erweiterungen in generierter API 45
5.8 JavaDoc 47
5.9 Verhalten bei Fehlern in den Tabellendefinitionen 48
5.10 Nicht umgesetzt 49

6 Generieren von Testdaten **53**
6.1 Betrachtung existierender Werkzeuge 53
6.2 Generierung von Beziehungen 55
6.3 Komplexität bei der Generierung von Beziehungen 60
6.4 Algorithmus zur Generierung von Beziehungen 60
6.5 Praktischer Einsatz / Evaluation 71
6.6 Konkrete Implementierung und Integration in Toolset 72
6.7 Offene Punkte 73

7 Zusammenfassung und Ausblick **75**

Abbildungsverzeichnis **78**

Listings **80**

Literaturverzeichnis **82**

A Modell **83**

B Generierte DSL **85**

Kapitel 1

Einleitung

Softwaretests sind ein wichtiger Baustein für die Qualitätssicherung von modernen Softwareprojekten. Für Tests werden Testdaten spezifiziert. Auf deren Basis wird das Verhalten der Software geprüft. Bei Datenbank-basierten Anwendungen werden Testdaten in der Regel sehr umfangreich und komplex. In diesem Kontext spricht man auch von der Modellierung von Testdaten.

Die Komplexität ergibt sich aus der Beschreibung von Beziehungen zwischen den einzelnen Datensätzen. Besonders bei Systemen mit komplexen Datenbank-Schemata kann ein Testdaten-Modell schnell unübersichtlich werden.

Für den Tester sind unübersichtliche Testdaten aus verschiedenen Gründen ein Problem. Einerseits machen sie die Pflege fehleranfällig. Andererseits ist es schwer, die modellierten Daten zu erfassen und zu verstehen. Tester wünschen deshalb häufig Daten, die für mehrere Tests nutzbar sind. Dadurch reicht es aus, nur ein oder zumindest wenige Daten-Modelle zu verstehen und zu pflegen.

1.1 Zielsetzung der Arbeit

Die Beschreibung von Testdaten soll vereinfacht werden. Dazu soll eine Modellierungssprache entwickelt werden, die leicht zu verstehen und zu erlernen ist. Besonderer Fokus liegt auf der Modellierung von Beziehungen. Die Lösung soll einfach zu benutzen sein und sich in Entwicklungsumgebungen (wie Eclipse) integrieren lassen.

Darüber hinaus sollen Testdaten automatisch generiert werden können. Hier wird besondere Aufmerksamkeit auf die Generierung von Daten für Beziehungen zwischen Datensätzen gelegt. Wichtig ist vor allem, dass die erzeugten Daten zum Datenbank-Schema passen, d.h. dass gültige Beziehungen erzeugt werden.

1.2 Aufbau der Arbeit

Zunächst werden in Kapitel 2 einige grundlegende Konzepte bzw. Technologien und in der Arbeit verwendete Konventionen beschrieben, die für das weitere Verständnis notwendig sind. Danach werden in Kapitel 3 die Anforderungen an die Modellierungssprache erarbeitet und die Aufgabenstellung konkretisiert. Die Entwicklung und die Definition der Sprache

wird in Kapitel 4 beschrieben. Kapitel 5 beschreibt die Implementierung und schließt damit das Thema Modellierungssprache ab. Mit der Generierung von Testdaten befasst sich Kapitel 6. Abschließend wird in Kapitel 7 eine Zusammenfassung und ein Ausblick auf mögliche Folgearbeiten gegeben.

Kapitel 2

Grundlegende Konzepte

Dieses Kapitel geht auf einige grundlegende Themen ein, die für das Verständnis dieser Arbeit notwendig sind. Dabei werden einige Begriffe aus der modellgetriebenen Software-Entwicklung erklärt und Software-Tests im Rahmen von Datenbank-Anwendungen beschrieben. Abschließend werden die in der Arbeit verwendeten Konventionen für Diagramme erläuert und ein kurzer Überblick über die Bibliothek *STU* gegeben.

2.1 Modellgetriebene Software-Entwicklung

In der modellgetriebenen Software-Entwicklung hat sich eine Vier-Schichten-Meta-Architektur etabliert. Die vier Schichten werden als M0 bis M3 bezeichnet [Mof]:

- **M0**: Konkrete Information
- **M1**: Meta-Daten zum Beschreiben der Information. Auch als *Modell* bezeichnet.
- **M2**: Metamodell, das das Modell beschreibt.
- **M3**: Meta-Metamodell, das das Meta-Modell und sich selbst beschreibt.

To do (2)

2.2 Software-Tests

Eine zu prüfende Anwendung wird im Kontext von Software-Tests als *System Under Test* (abgekürzt SUT) bezeichnet. Dabei bezeichnet SUT Klassen, Objekte, Methoden oder vollständige Anwendungen. [Mes07, 810f]

Alle Voraussetzungen und Vorbedingungen für einen Test werden unter der Bezeichnung *Test Fixture* zusammengefasst. Ein Test Fixture repräsentiert den Zustand des SUT vor den Tests. [Mes07, S. 814] Es gibt verschiedene Arten von Test Fixtures. Folgende zwei Test Fixtures sind für diese Arbeit relevant:

- **Standard Fixture**: Ein Test Fixture wird als Standard Fixture bezeichnet, wenn es für alle bzw. fast alle Tests verwendet werden kann. Ein Standard Fixture reduziert nicht nur den Aufwand zum Entwerfen von Testdaten für die einzelnen Tests, sondern verhindert darüber hinaus, dass der Tester sich bei verschiedenen Tests immer wieder in unterschiedliche Test-Daten hineinversetzen muss. Nur in Ausnahmefällen sollten Tests modifizierte oder eigene Testdaten verwenden. [Mes07, S. 305]

- **Minimal Fixture**: Ein Fixture, das speziell für einen Test erstellt wurde und dessen Umfang auf die für diesen Test notwendigen Daten reduziert ist, wird als Minimal Fixture bezeichnet. Aufgrund ihres Umfangs lassen sich Minimal Fixtures von Testern im Allgemeinen leichter verstehen. Außerdem kann der Einsatz von Minimal Fixtures zu Leistungsvorteilen bei der Ausführung von Tests führen, da z.B. das Einspielen in die Datenbank schneller ablaufen kann als bei umfangreichen Daten. [Mes07, S. 302]

2.2.1 Datenbank-Tests

Ein übliches Muster, Systeme in Verbindung mit Datenbanken zu testen, ist *Back Door Manipulation*. Die Idee hinter diesem Muster ist es, auf die Datenbank direkt zu zu greifen am zu testenden System vorbei – quasi in Form einer Hintertür. Auf diese Weise wirken sich Fehler im zu testenden System nicht auf den Zugriff der Datenbank aus. Das Muster besteht aus vier Schritten. Im ersten Schritt, dem *Setup*, wird die Datenbank über direkten Zugriff in den Anfangszustand gebracht. Anschließend können im *Exercise*-Schritt die zu testenden Operationen am System durchgeführt werden. Die Überprüfung, ob sich das System richtig verhalten hat, findet im als *Verify* bezeichneten Schritt statt. Dabei wird der Zustand der Datenbank mit dem erwarteten Zustand verglichen, ebenfalls am zu testenden System vorbei. Abschließend kann der vierte Schritt, *Teardown*, noch optionale Aufräumarbeiten durchführen. Abbildung 2.1 stellt Back Door Manipulation grafisch dar. [Mes07, 327ff]

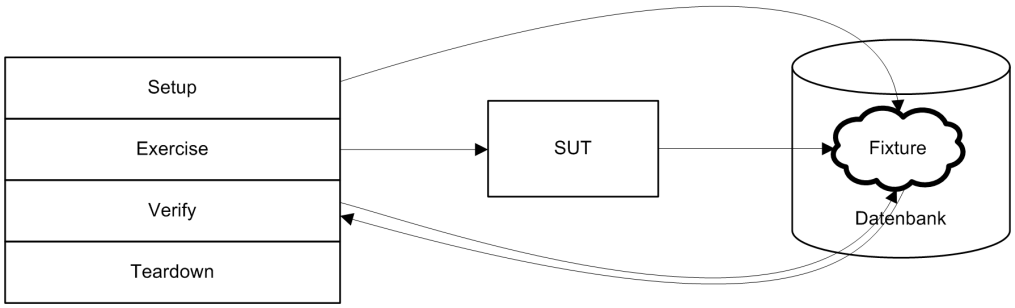

Abbildung 2.1: Back Door Manipulation

Es gibt mehrere Vorteile, die Datenbank nicht über das zu testende System in den Anfangszustand zu bringen. Einerseits können semantische Fehler im zu testenden System unter Umständen nur so gefunden werden. Ein semantischer Fehler stellt z.B. der Zugriff auf die falsche Spalte dar: Das SUT könnte die Zugriffe auf die Spalten Vor- und Nachname konsequent vertauschen. Das System würde normal funktionieren, die Datenbank würde sich allerdings in einem falschen Zustand befinden.

Andererseits kann der Zustand mitunter schneller in die Datenbank geschrieben werden, wenn nicht der Weg über das zu testende System gemacht wird. Außerdem bietet es in

Bezug auf die Zustände eine höhere Flexibilität: Die Datenbank kann auch in Zustände gebracht werden, die über das System nicht erreicht werden können. Dafür leidet die Flexibilität an einer anderen Stelle: Die Tests sind abhängig vom konkret verwendeten Datenbank-System. Wird die Datenbank von SQL auf NoSQL umgestellt, müssen die Tests angepasst werden. Außerdem setzt der direkte Zugriff auf die Datenbank voraus, dass die Semantik der zu testenden Anwendung berücksichtigt wird. Aus Sicht der Anwendung dürfen sich von der Anwendung eingespielte Daten in ihrer Form nicht von den manuell in die Datenbank geschriebenen Daten unterscheiden.

To do (3)

2.3 Konventionen für Diagramme

Die in dieser Arbeit abgebildeten Diagramme verwenden einen einheitlichen Modellierungsstil. Der Stil ist abhängig von der Art des Diagramms.

Als Grundlage für die Beispiel-Diagramme dient ein Modell aus zwei Entitätstypen (Tabellen) mit jeweils einem Attribut. Die beiden Entitätstypen heißen *Tabelle 1* und *Tabelle 2*. *Tabelle 1* enthält das Attribut *feld*, während *Tabelle 2* das Feld *anderes_feld* enthält. Eine Entität aus *Tabelle 1* kann mit beliebig vielen, aber mindestens einer Entität aus *Tabelle 2* in Beziehung stehen. Eine Entität aus *Tabelle 2* muss mit genau einer Entität aus *Tabelle 1* in Beziehung stehen.

2.3.1 ER-Diagramme

Das entsprechende ER-Diagramm ist in Abbildung 2.2 dargestellt. Ein Entitätstyp wird von einer Box repräsentiert, der Name des Typs und seine Attribute sind durch eine horizontale Linie voneinander getrennt. Eine Beziehung zwischen Entitätstypen wird durch eine Verbindungslinie dargestellt. Diese kann bezeichnet sein, der Bezeichner ist dann mittig an der Verbindungslinie platziert. Die Kardinalitäten befinden sich an den Enden der Verbindungsline.

Auf die Angabe der Stereotypen für die Entitäten und Beziehungen wird verzichtet.

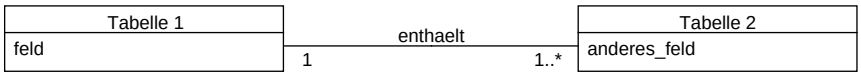

Abbildung 2.2: Stil von ER-Diagrammen

2.3.2 Relationale Datenbank-Diagramme

Für relationale Datenbank-Diagramme wird der Modellierungsstil verwendet, der auch von Ambler in [AS06] genutzt wird wird. Dieser Stil erweitert das dem UML-konform das Klassendiagramm.

Tabellen werden durch Boxen repräsentiert. Diese Boxen sind unterteilt in zwei Bereiche: den Tabellenbezeichner (oben) und die Attribute der Tabelle (unten). Attribute werden in

der Form *Bezeichner : typ* beschrieben. Es gibt die beiden Stereotypen *PK* für *Primary Key* und *FK* für *Foreign Key*, um die Attribute entsprechend zu klassifizieren.

Beziehungen zwischen Tabellen werden durch Verbindungslinien dargestellt. Die Kardinalitäten beschreiben die Art der Verbindung. Die Bedeutung der Kardinalitäten lassen sich am besten über ein Beispiel erklären (Abbildung 2.3).

Auf die Angabe der Stereotypen für die Tabellen und Beziehungen wird verzichtet.

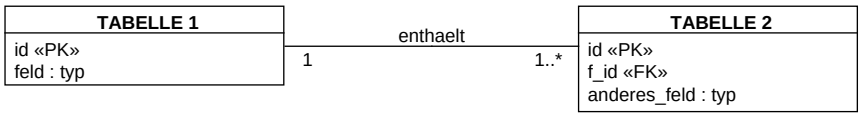

Abbildung 2.3: Stil relatianaler Datenbank-Diagramme nach Ambler

2.4 STU (Simple Test Utils)

STU ist eine Bibliothek zur Vereinfachung von Unit-Tests für Java-basierte Anwendungen. Es steht unter der *Apache License 2.0* [Fou04]. Das heißt, es ist Open-Source und kann in kommerziellen Softwareprojekten genutzt werden. *STU* wird maßgeblich von der Firma SEITENBAU entwickelt. Diese Arbeit thematisiert nur den Teil von *STU*, der Tests von Datenbank-Anwendungen vereinfachen soll.

Für Unit-Tests von Datenbank-Anwendungen nutzt *STU* die Bibliothek *DbUnit*. *DbUnit* stellt eine Erweiterung für JUnit dar und stellt Funktionen speziell für den Test von Datenbank-Anwendung zur Verfügung [Mes07, S. 748]. Mit Hilfe von *DbUnit* kann eine Datenbank zu Beginn eines Tests in einen definierten Zustand gebracht werden. Dazu kann *DbUnit* DataSets in die Datenbank einspielen. Ein DataSet fasst eine Menge an Daten zu einer Einheit zusammen. Außerdem kann DbUnit den Inhalt einer Datenbank mit einem DataSet vergleichen. So kann nach dem Test überprüft werden, ob das SUT die richtigen Modifikationen an den Daten vorgenommen hat.

Ein Ziel von *STU* ist, Datenbank-Tests weiter zu vereinfachen und bietet die Möglichkeit, DataSets über ein eigenes API zu modellieren. *STU* enthält einen Generator, der mit Hilfe eines Datenbank-Modells individuelles API erzeugen kann. Die generierten Klassen setzen das Builder-Pattern [Blo08, 11ff] mit einem Fluent API [Fow10, 68ff] um. In Java werden Fluent APIs mit Hilfe von Method Chaining verwirklicht. Dabei liefern Methoden zum Konfigurieren des Objekts das Objekt selbst zurück. Auf diese Weise können mehrere Modifikationen an einem Objekt mit nur einem Ausdruck durchgeführt werden [Fow10, 373f]. Die Nutzung könnte so aussehen: `student.name("Moll").vorname("Nikolaus").`

Abbildung 2.4 stellt grafisch dar, wie aus einem Datenbank-Modell das Fluent API erzeugt wird. Ausgangspunkt ist ein relationales Datenbankmodell. Dieses Modell muss in ein für den Generator interpretierbares Modell, das STU-Modell, transformiert werden. Dies kann automatisch (z.B. wenn das Modell in Form eines *Apache-Torque*-Modell vorliegt) oder manuell geschehen. Das STU-Modell enthält Informationen zu Tabellennamen und Angaben zu den Spalten, z.B. Namen und Datentypen.

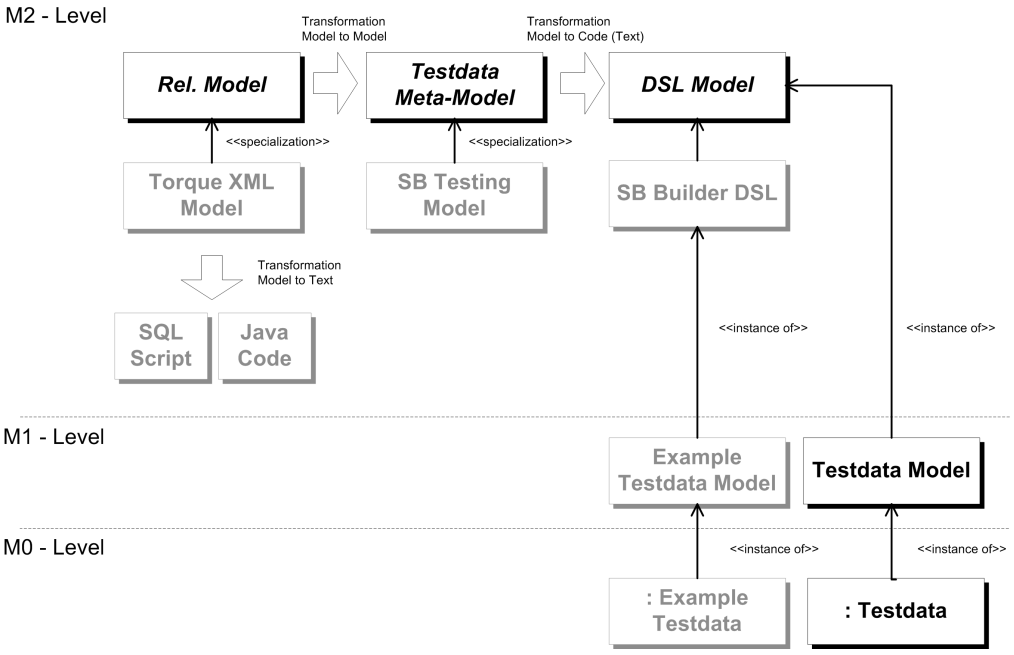

Abbildung 2.4: Modell-Beschreibung

Das STU-Modell enthält keine Datenbank-Constraints. Eine Abbildung dieser würde keine wesentliche Vorteile bringen. Das API bzw. die erzeugten DataSets sind ausschließlich für den Einsatz im Test-Umfeld gedacht. Sollte ein DataSet Daten enthalten, die gegen die in der Datenbank definierten Constraints verstoßen, scheitert das Einspielen des DataSets in die Datenbank und eine Fehlermeldung wird ausgelöst. Aus Sicht des Testers ist dieses Verhalten ausreichend, da die Exception zum Scheitern der Unit-Tests führen wird. Der Mehrwert, dass ungültige DataSets schon vor dem Einspielen als solches zu erkennen, ist gering im Vergleich zu dem Aufwand, Constraint-Mechanismen verschiedener Datenbanksysteme nachzubauen.

Der Generator nutzt für die Code-Generierung *Apache Velocity*. Velocity ist eine Template-Engine, die Dokumente mit Hilfe von Templates erzeugt. Diese Templates bestehen aus Text und besonderen Velocity-Anweisungen. Zu den Anweisungen gehören unter anderem Platzhalter, die während der Generierung durch konkrete Werte ausgetauscht werden. Velocity bietet mit Verzweigungen und Schleifen auch Anweisungen zur Steuerung der Generierung.

Die Namen der generierten Klassen ergeben sich aus den im Modell enthaltenen Informationen. Es werden Klassen der folgenden Kategorien erzeugt:

- **DataSet**: Ein DataSet repräsentiert eine Menge von Testdaten. Es umfasst alle Tabellen eines Datenbankmodells. Es wird eine abstrakte DataSet-Klasse generiert. Der Zugriff auf die Tabellen erfolgt über öffentliche Felder. Die Klasse enthält die Methode `createDBUnitDataSet`, um die für die Unit-Tests benötigten DbUnit-DataSets zu erzeugen. Dabei werden Template-Methoden [Gam+94, S. 325] definiert, die genutzt werden können, um in den Erzeugungsprozess von DataSets einzugreifen. Die Klasse enthält darüber hinaus Methoden zum Hinzufügen von Zeilen in die entsprechende Tabellen.

- **Table**: Die `Table`-Klasse fasst alle Testdaten in Form von Zeilen einer Tabelle zusammen. Für jede Tabelle wird eine individuelle Klasse generiert. Der Klassenname

setzt sich aus dem Namen der Tabelle und dem Suffix „Table“ zusammen. Die Klasse stellt Methoden zur Modellierung und für den Zugriff auf die Tabellendaten bereit. Zur Integration in DbUnit implementiert sie das DbUnit-Interface `ITable`.

- **RowBuilder**: Eine Zeile einer Tabelle wird von einem `RowBuilder` repräsentiert. Zu jeder Tabelle wird eine individuelle `RowBuilder`-Klasse erzeugt. Sie beinhaltet für jede Spalte mehrere Methoden zum Setzen und Abfragen des jeweiligen Wertes. Die Methodennamen setzen sich zusammen aus der Aufgabe (`get` bzw. `set`) und dem Spaltennamen.
- **FindWhere**: Für einfache Suchanfragen gibt es für jede Tabelle die innere Klasse `FindWhere`. Sie ermöglicht die Suche nach einem Wert in einer Spalte und liefert eine Liste von Tabellenzeilen. Die Methode ist zum Auffinden von Tabellenzeilen vorgesehen, von denen erwartet wird, dass es sie gibt.

Abbildung 2.5 stellt das logische Klassendiagramm der DataSet-Klassen dar. Der Unterschied zum tatsächlichen Klassendiagramm besteht darin, dass im logischen Diagramm alle `Table`-Klassen zusammengefasst werden, obwohl diese unterschiedlichen Typs sind. Außerdem entsprechen die Klassennamen, bis auf die Klasse `FindWhere`, nicht den tatsächlichen Bezeichnern.

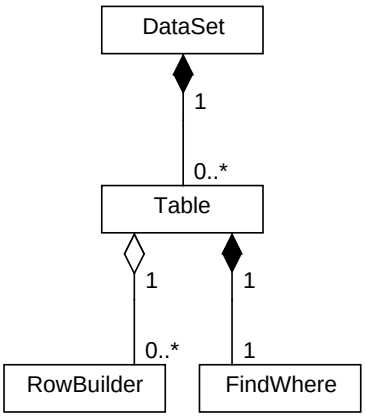

Abbildung 2.5: Logisches Klassendiagramm der STU-DataSet-Klassen

Kapitel 3

Anforderungsanalyse

In diesem Kapitel werden die Anforderungen an die zu entwickelnde Lösung konkretisiert. Es beinhaltet Bewertungen von bestehenden Modellierungssprachen. Für diesen Vergleich werden Kriterien für die Bewertung der Sprache definiert und ein Datenbank-Modell vorgestellt, das als fortlaufendes Beispiel dient und auch in den folgenden Kapiteln verwendet wird.

3.1 Allgemeine Anforderungen

Die Hauptziele dieser Arbeit stellen sich wie folgt dar:

1. Vereinfachen der Beschreibung von Beziehungen
2. Modellierte Testdaten übersichtlicher machen
3. Automatisches Generieren von Testdaten

Für die Modellierung gelten diese allgemeinen Anforderungen:

- **Integration in bestehende Werkzeugkette**: Die Lösung sollte sich nach Möglichkeit in die bestehende Werkzeugkette von SEITENBAU integrieren lassen.
- **IDE-Integration**: Bedienbarkeit für den Tester stellt eine der wichtigsten Anforderungen dar. Daten sollen komfortabel modelliert werden können. Die Integration in Entwicklungsumgebungen wie Eclipse oder IntelliJ IDEA muss gegeben sein.
- **Beziehungen**: Beziehungen sollen einfach modellieren werden können.
- **Gültigkeitsbereiche**: **To do** (4)
- **Veränderbarkeit von DataSets**: DataSets sollen sich während der Modellierung beliebig verändern lassen.
- **Komposition**: DataSets sollen sich aus anderen DataSets zusammensetzen lassen.
- **Typ-Sicherheit**: Die Beschreibung der Daten sollte typsicher erfolgen. Idealerweise sollten falsche Typen schon während des Compilierns erkannt werden.

- **Funktionen als Werte**: Es soll möglich sein, Hilfsfunktionen zur Berechnung von Werten zu verwenden, z.B. zum Einlesen von Binary Large Objects (BLOBs) aus Dateien.
- **Zielgruppe**: Die Zielgruppe für die DSL sind Software-Entwickler und Tester. Der Code zur Modellierung der Daten sollte auch für andere Projekt-Mitglieder lesbar und verständlich sein.
- **Ungültige Daten**: Es sollen sich auch aus Sicht der Datenbank oder des SUT ungültige Daten modellieren lassen.

Für die Generierung der Testdaten lassen sich die Anforderungen folgendermaßen zusammenfassen:

- **Gültige Daten**: Die erzeugten Daten müssen gültig sein und zum Datenbank-Schema passen.
- **Mehrfach verwendbare Daten**: Die generierten Daten sollen für verschiedene Tests verwendet werden können, also ein Standard-Fixture darstellen. Die Daten müssen deshalb viele Test-Fälle abdecken können.
- **Fokus auf Beziehungen**: Das Generieren von sinnvollen Beziehungen stellt eines der zentralen Ziele für den Daten-Generator dar.
- **Datenmenge selbst bestimmen**: Der Generator soll ohne Konfigurationsaufwand eine geeignete Menge an Test-Daten erzeugen.
- **Deterministische Generierung**: Auch wenn die Test-Daten aus Zufallsdaten bestehen, sollen sie deterministisch generiert werden können. Das heißt, dass die Generierung des Modells mit den selben Einstellungen auch zum selben Ergebnis führt.
- **Kompatibilität**: Die Generierung der Testdaten soll in unterschiedliche Ausgabe-Formen erfolgen können, z.B. in einer DSL, in XML oder auch in SQL-Statements.

3.2 Modellierungskonzepte für Beziehungen

Je nach Beziehungsart gibt es unterschiedliche Ansätze, wie diese in einem ER-Diagramm umgesetzt werden können. Dabei können neben den Entitäten auch die Beziehungen selbst Attribute haben. Die folgenden drei grundsätzlichen binären Beziehungsarten (also zwischen zwei Entitätstypen) werden dabei unterschieden:

3.2.1 1:1-Beziehungen

Eine binäre Beziehung zwischen zwei Entitätstypen, wobei jede Entität innerhalb dieser Beziehung maximal einer anderen Entität zugeordnet sein kann. Eine solche Beziehung kann realisiert werden, indem eine Tabelle um einen Fremdschlüssel auf die andere erweitert wird. Dabei sollte der Fremdschlüssel und auch die beziehungsbeschreibenden Attribute immer der Tabelle hinzugefügt werden, deren Entitäten eine Beziehung voraussetzt.

Wenn viele Beziehungsattribute vorhanden sind oder die Beziehung auf beiden Seiten optional ist, kann es auch sinnvoll sein, eine 1:1-Beziehung wie eine n:m-Beziehung zu modellieren.

3.2.2 1:n-Beziehungen

Eine binäre Beziehung zwischen zwei Entitätstypen, wobei jede Entität des einen Typs in Beziehung mit mehreren Entitäten des anderen Typs stehen kann. Diese Entitäten können auch nur mit maximal einer Entität in Beziehung stehen. Es ist möglich festzulegen, wie viele Beziehungen eine Entität mindestens und höchstens haben darf.

Die Tabelle der Entitäten, die maximal einer andere Entität zugeordnet sind, wird um einen Fremdschlüssel und um für jede Beziehung individueller Attribute erweitert. Die Beziehungsattribute, die für alle Beziehungen der beteiligten Entität gelten, werden ihrer Tabelle hinzugefügt.

3.2.3 n:m-Beziehungen

Eine binäre Beziehung zwischen zwei Entitätstypen, wobei jede Entität des einen Typs mit mehreren Entitäten des anderen Typs in Beziehung stehen kann – und umgekehrt. Es ist möglich, untere und obere Grenzwerte für die Anzahl der Beziehungen auf beiden Seiten festzulegen. Solche als assoziativ bezeichneten Beziehungen werden über eine Hilfstabelle modelliert, die assoziative Tabelle genannt wird. Diese besteht aus den beiden Fremdschlüsseln auf die beteiligten Tabellen und den beziehungsbeschreibenden Attributen.

Grundsätzlich können assoziative Tabellen für alle binären Beziehungen verwendet werden. Vor allem wenn die Beziehung viele Attribute enthält, kann eine assoziative Tabelle für übersichtlichere Tabellenstrukturen sorgen.

3.2.4 Andere Beziehungen

In der aktuellen *STU*-Implementierung müssen andere Beziehungen manuell umgesetzt werden. Dies gilt für zirkuläre, für reflexive und für alle nicht-binären Beziehungen.

3.3 Fortlaufendes Beispiel

Ein einheitliches fortlaufendes Beispiel soll der Arbeit als Grundlage dienen. Die Problemstellung besteht aus einem Modell und einer Menge von Testdaten. Diese Testdaten dienen als Grundlage für die Diskussion der unterschiedlichen Modellierungsvarianten.

3.3.1 Anforderungen an das Beispiel

Der Schwerpunkt der Modellierung liegt bei der Darstellung von Beziehungstypen zwischen Entitätstypen. Dabei soll die Problemstellung einerseits nicht zu komplex sein, damit sie überschaubar bleibt. Andererseits soll sie komplex genug sein, um möglichst alle Beziehungsarten zwischen Entitäten abzudecken. Die Testdaten sollten gleichzeitig ein *Standard Fixture* und ein *Minimal Fixture* darstellen (siehe Abschnitt 2.2).

3.3.2 Gewählte Problemstellung

Das gewählte Beispiel stellt eine starke Vereinfachung des Prüfungswesens an Hochschulen dar. Auf eine praxisnahe Umsetzung wird zugunsten der Komplexität verzichtet. Personenbezogene Begriffe werden in der maskulinen Form verwendet, ohne dabei Aussagen über

das Geschlecht der repräsentierter Personen zu machen. Es beinhaltet die folgenden vier Entitätstypen:

- **Professor**: Ein Professor leitet Lehrveranstaltungen.
- **Lehrveranstaltung**: Eine Lehrveranstaltung wird von einem Professor geleitet. Es kann zu jeder Lehrveranstaltung eine Prüfung geben.
- **Prüfung**: Eine Prüfung ist einer Lehrveranstaltung zugeordnet. Außerdem hat mindestens ein Professor Aufsicht.
- **Student**: Studenten können an Lehrveranstaltungen und an Prüfungen teilnehmen. Studenten haben außerdem die Möglichkeit, Tutoren von Lehrveranstaltungen zu sein.
- **Raum**: Ein Professor kann einen Raum als Büro zugewiesen bekommen.

Die Beziehungen der Entitätstypen stellen sich wie folgt dar:

- **leitet**: Eine Lehrveranstaltung muss von genau einem Professor geleitet werden, ein Professor kann beliebig viele oder keine Lehrveranstaltungen leiten.
- **geprüft**: Eine Prüfung ist genau einer Lehrveranstaltung zugeordnet. Eine Lehrveranstaltung kann mehrere Prüfungen haben (z.B. Nachschreibprüfung).
- **beaufsichtigt**: Eine Prüfung muss mindestens von einem Professor beaufsichtigt werden, ein Professor kann in beliebig vielen Prüfungen Aufsicht haben.
- **besucht**: Jeder Student kann beliebig vielen Lehrveranstaltungen besuchen. Lehrveranstaltungen benötigen jedoch mindestens drei Besucher um stattzufinden und sind aus Kapazitätsgründen auf 100 Teilnehmer begrenzt.
- **ist Tutor**: Jeder Student kann bei beliebig vielen Lehrveranstaltungen Tutor sein und jede Lehrveranstaltung kann beliebig viele Tutoren haben.
- **schreibt**: Jeder Student kann an beliebig vielen Prüfungen teilnehmen und umgekehrt eine Prüfung von einer beliebigen Anzahl von Studenten geschrieben werden.
- **hat Büro**: Jeder Professor hat ein Büro. Ein Raum kann einem oder keinem Professor zugeordnet sein.

Abbildung 3.1 zeigt das Beispiel grafisch in Form eines ER-Diagramms. Den verschiedenen Entitätstypen werden dabei Attribute zugeordnet.

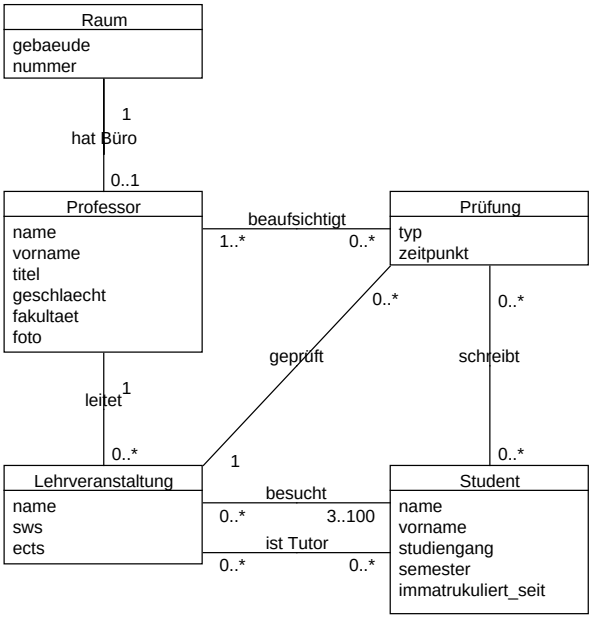

Abbildung 3.1: ER-Diagramm des fortlaufenden Beispiels

Das entsprechende relationale Datenbank-Schema wird in Abbildung 3.2 dargestellt. Assoziative Tabellen realisieren die n:m-Beziehungen.

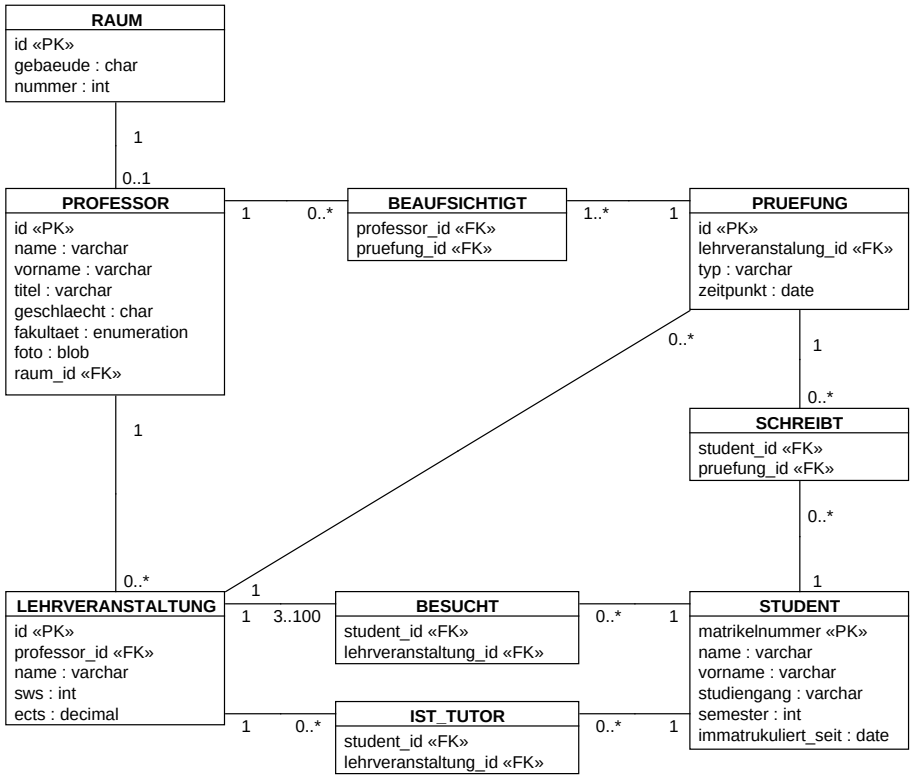

Abbildung 3.2: Relationales Datenbank-Schema des fortlaufenden Beispiels

Das Attribut „fakultaet“ in der Tabelle Professor soll als Aufzählungstyp (enumeration) realisiert werden. Mögliche Werte sind: Architektur, Bauingenieurwesen, Elektrotechnik, Informatik, Maschinenbau und Wirtschaftswesen. Das Foto des Professors wird als BLOB dargestellt.

Im folgenden Abschnitt werden verschiedene Modellierungsvarianten für DbUnit auf Grundlage dieses Beispiels betrachtet und bewertet.

3.4 Modellierungsvarianten der Testdaten für DbUnit

In *DbUnit* werden die Datenbankzustände durch DataSets repräsentiert. Für einen Test werden gewöhnlich zwei DataSets benötigt: das erste für den Anfangszustand, das zweite für den erwarteten Zustand. Allerdings bieten DbUnit-DataSets nur begrenzte Möglichkeiten (z.B. kein Löschen von Zeilen aus einem DataSet möglich), das DataSet mit dem erwarteten Zustand aus dem DataSet mit dem Anfangszustand zu erzeugen.

Im Folgenden werden verschiedene Modellierungsarten für DbUnit-DataSets diskutiert. Diese soll anhand der im nächsten Abschnitt beschriebenen Kriterien erfolgen. Die Ergebnisse stellen die Grundlage für die konkretere Zielsetzung dar.

3.4.1 Kriterien für Bewertung

Für die Bewertung von Modellierungssprachen werden die folgenden Kriterieren verwendet. Einige Punkte sind messbar, andere wiederum relativ subjektiv. Als Vorbild für die beiden letzten Punkte dient die Norm ISO IEC 9126.

- **Zeilen**: Die Anzahl der Zeilen, die für ein DataSet benötigt werden.
- **Zeichen pro Zeile**: Ist die Sprache für die Darstellung auf Bildschirmen geeignet?
- **Typsicherheit**: Wann und wie werden falsche Datentypen bei der Modellierung erkannt?
- **Redundanz**: Müssen Daten oder Sprachelemente redundant verwendet werden?
- **Benutzbarkeit (Verständlichkeit und Erlernbarkeit)**: Wie gut drückt die Sprache aus, welche Daten und Beziehungen modelliert werden? Wie leicht ist die Sprache zu lernen?
- **Modifizierbarkeit**: Wie leicht lassen sich Daten ändern? Wie leicht können können bestehende Daten an ein neues Datenbankschema angepasst werden?

3.4.2 XML-DataSet

Eine Möglichkeit, ein DataSet für DbUnit zu modellieren, stellt XML dar. DbUnit selbst bietet zwei Varianten an, DataSets über XML zu modellieren.

Die erste Variante stellt das `XmlDataSet` dar. Diese Klasse liest eine XML-Datei nach einem von DbUnit vorgegebenen Dokumententyp ein. Das Listing 3.1 zeigt einen Ausschnitt einer solchen XML-Datei, in dem die beiden Tabellen *Professor* und *Lehrveranstaltung* definiert werden.

```
1  <!DOCTYPE dataset SYSTEM "dataset.dtd">
2  <dataset>
3      <table name="PROFESSOR">
4          <column>id</column>
5          <column>name</column>
6          <column>vorname</column>
7          <column>titel</column>
8          <column>fakultaet</column>
9          <row>
10             <value>1</value>
11             <value>Wäsch</value>
12             <value>Jürgen</value>
13             <value>Prof. Dr.-Ing.</value>
14             <value>Informatik</value>
15         </row>
16         <row>
17             <value>2</value>
18             <value>Haase</value>
19             <value>Oliver</value>
20             <value>Prof. Dr.</value>
21             <value>Informatik</value>
22         </row>
23     </table>
24     <table name="LEHRVERANSTALTUNG">
25         <column>id</column>
26         <column>professor_id</column>
27         <column>name</column>
28         <column>sws</column>
29         <column>ects</column>
30         <row>
31             <value>1</value>
32             <value>2</value>
33             <value>Verteilte Systeme</value>
34             <value>4</value>
35             <value>5</value>
36         </row>
37         <row>
38             <value>2</value>
39             <value>2</value>
40             <value>Design Patterns</value>
41             <value>4</value>
42             <value>3</value>
43         </row>
44     </table>
45   ...
46 </dataset>
```

Listing 3.1: XML-DataSet

Die Bewertung anhand der Kriterien stellt sich für das `XmlDataSet` wie folgt dar:

- **Zeilen**: Die XML-Datei mit den Beispiel-Daten umfasst 127 Zeilen. Für jede Entität werden jeweils eine Zeile pro Attribut und weitere zwei Zeilen für die umschließenden XML-Tags.

  DbUnit-konforme XML-Dateien wachsen schnell in vertikaler Richtung und enthalten unter Umständen auch viel syntaktischen Overhead. Von den etwas über 40 gezeigten Zeilen enthalten nur 20 Zeilen wirkliche Daten oder drücken Beziehungen aus.

- **Zeichen pro Zeile**: Mit maximal 47 Zeichen pro Zeile ist diese Modellierungsvariante in Bezug auf die Breite gut für die Bildschirmdarstellung geeignet.

- **Typsicherheit**: Zur Modellierung müssen Meta-Informationen zu den Daten hinterlegt werden. Diese beschränken sich allerdings auf die Bezeichnungen der Spalten (Zeilen 4-8 und 25-29). Da weitere Meta-Informationen fehlen, können fehlerhafte Datentypen oder Verstöße gegen Datenbank-Constraints erst zur Laufzeit beim Einspielen des DataSets erkannt werden.

- **Redundanz**: Das Modellieren von Beziehungen führt zu Werte-Redundanz. Die konkreten Werte von Primärschlüsseln müssen an anderer Stelle als Fremdschlüssel verwendet werden.

- **Benutzbarkeit (Verständlichkeit und Erlernbarkeit)**: Die positiven Eigenschaften bei der Modellierung mit XML sind unter anderem, dass für XML ein breites Angebot an Werkzeugen zur Verfügung steht. Diese können über den Dokumententyp prüfen, ob die Datei den Regeln entspricht. Der Umgang mit XML-Dateien kann als bekannt angenommen werden für die Zielgruppe.

  Die manuelle Pflege von Primär- und Fremdschlüsseln ist unübersichtlich und damit fehleranfällig. Die Value-Tags selbst lassen keinen Rückschluss auf die Spalte zu, die sie repräsentieren. Das erschwert die Lesbarkeit.

  In Bezug auf Verständlichkeit zeigt die XML-Datei Schwächen: Ohne zusätzliche Kommentare ist eine solche XML-Datei weder leicht zu lesen noch leicht zu pflegen. Auch Beziehungen sollten über Kommentare verdeutlicht werden.

  DbUnit unterstützt BLOBs in XML in Form Base64-codierter Daten. Bei größeren Datenmengen leidet die Übersicht unter dem Einbetten von BLOBs, nicht nur wegen dem zusätzlichen Platzbedarf aufgrund der Codierung. Spezielle Mechanismen, BLOBs aus anderen Dateien einzulesen, bringt DbUnit nicht mit. Solche Funktionen müssen manuell implementiert werden.

- **Modifizierbarkeit**: Daten lassen sich relativ leicht ändern – sofern man die richtige Stelle gefunden hat, was ohne Kommentare nicht immer so leicht ist. Eine solche XML-Datei an ein neues Datenbank-Schema anzupassen kann einfach aber auch mühsam sein. Das Umbenennen von Spalten ist sehr einfach, das Entfernen oder Hinzufügen von Spalten bei umfangreichen Daten ohne den geschickten Umgang mit Text-Editoren sehr umständlich.

Ein großer Nachteil bei der Nutzung von `XmlDataSet` ist, dass der erwartete Datenbankzustand selbst wieder den kompletten Datenbankbestand umfassen muss. DbUnit erlaubt zwar mehrere DataSets zu einem zusammenzufassen, das Entfernen von Datensätzen ist darüber aber nicht möglich. Mehrere XML-Dateien mit ähnlichen, überwiegend sogar gleichen Daten, sorgen für ein hohes Maß an Redundanz. Darüber hinaus sieht DbUnit keinen Mechanismus für die Komposition von XML-DataSets auf Modellierungsebene vor, d.h. es geht aus einer solchen XML-Datei nicht hervor, dass sie auf anderen DataSets aufbaut und diese erweitert.

Das `FlatXmlDataSet` stellt die zweite Variante dar. Hierbei gibt es keine von DbUnit vorgegebene DTD, da die Tags den Tabellen-Namen entsprechen[1]. Eine solche XML-Datei kommt ohne explizite Meta-Informationen zu den Tabellen aus. Stattdessen stellen sie eine Art Sprachelement dar und werden für die Zuweisung der Werte verwendet. In Bezug auf die Meta-Informationen ist das `FlatXmlDataSet` übersichtlicher als das XmlDataSet (siehe Listing 3.2).

```
1 <?xml version='1.0' encoding='UTF-8'?>
2 <dataset>
3     <PROFESSOR id="1"
4         name="Wäsch"
5         vorname="Jürgen"
6         titel="Prof. Dr.-Ing."
7         fakultaet="Informatik" />
```

[1] Es ist möglich, eine eigene DTD zu definieren.

```
8       <PROFESSOR id="2"
9           name="Haase"
10          vorname="Oliver"
11          titel="Prof. Dr."
12          fakultaet="Informatik" />
13      <LEHRVERANSTALTUNG id="1"
14          professor_id="2"
15          name="Verteilte Systeme"
16          sws="4"
17          ects="5" />
18      <LEHRVERANSTALTUNG id="2"
19          professor_id="2"
20          name="Design Patterns"
21          sws="4"
22          ects="3" />
23  ...
24  </dataset>
```

Listing 3.2: Flat-XML-DataSet

Das `FlatXmlDataSet` hat große Ähnlichkeit zum `XmlDataSet`, das zeigt sich auch in der Bewertung. Einige vorher genannte Punkte gelten hier weiterhin.

- **Zeilen**: Die selben Beispieldaten lassen sich hier mit 63 Zeilen ausdrücken. Die Datei kommt mit weniger Meta-Informationen und etwas weniger syntaktischen Overhead aus. Allerdings sollte auch hier jedes Attribut in eine Zeile geschrieben werden.
- **Zeichen pro Zeile**: Mit maximal 40 Zeichen pro Zeile für die gewählten Testdaten ist das `FlatXmlDataSet` für die Bildschirmdarstellung gut geeignet.
- **Typsicherheit**: Wie auch beim `XmlDataSet` können die Typen erst beim Einspielen in die Datenbank überprüft werden.
- **Redundanz**: Das Beschreiben von Beziehungen erfordert die selbe Daten-Redundanz wie beim `XmlDataSet`.
- **Benutzbarkeit (Verständlichkeit und Erlernbarkeit)**: Durch die fehlende Hierarchie wirkt das `FlatXmlDataSet` etwas unübersichtlich. Die Spalten-Bezeichner stellen eine Art Sprachelement dar, d.h. sie werden als XML-Attribut-Bezeichner bei der Datenzuweisung verwendet. Das ist übersichtlich und verständlich. Außerdem müssen die Attribute nicht zwingend in der selben Reihenfolge angegeben werden.
- **Modifizierbarkeit**: Das Ändern der Daten wird dadurch erleichtert, dass Spaltennamen und Wert direkt beieinander stehen. Ansonsten gelten die bereits für das `XmlDataSet` genannten Punkte.

3.4.3 Default-DataSet

DbUnit erlaubt auch die programmatische Modellierung von DataSets. Dazu stellt es die Klasse `DefaultDataSet` bereit. Mit den Mitteln, die eine Programmiersprache wie Java bietet, lassen sich einige der Nachteile in Verbindung mit den XML-basierten DataSets direkt umgehen.

So können Beziehungen mit Hilfe symbolischer Konstanten ausdrucksstärker modelliert werden. Auch wenn die Beziehungen immer noch etwas umständlich modelliert werden müssen, können symbolische Konstanten dabei helfen, Redundanz zu vermeiden und damit das Risiko für Fehler zu senken, die bei der Änderung solcher Werte auftreten können.

```
1  DefaultTable professor = new DefaultTable(
2      "professor",
3      new Column[] {
4        new Column("id", DataType.INTEGER),
5        new Column("name", DataType.VARCHAR),
6        new Column("vorname", DataType.VARCHAR),
7        new Column("titel", DataType.VARCHAR),
8        new Column("fakultaet", DataType.VARCHAR),
9      }
10   );
11 professor.addRow(new Object[] {
12       Parameters.Professor.WAESCH_ID,
13       "Wäsch",
14       "Jürgen",
15       "Prof. Dr.-Ing.",
16       "Informatik",
17     });
18 professor.addRow(new Object[] {
19       Parameters.Professor.HAASE_ID,
20       "Haase",
21       "Oliver",
22       "Prof. Dr.",
23       "Informatik",
24     });
25 dataSet.addTable(professor);
26
27 DefaultTable lehrveranstaltung = new DefaultTable(
28     "lehrveranstaltung",
29     new Column[] {
30       new Column("id", DataType.INTEGER),
31       new Column("professor_id", DataType.INTEGER),
32       new Column("name", DataType.VARCHAR),
33       new Column("sws", DataType.INTEGER),
34       new Column("ects", DataType.INTEGER),
35     }
36   );
37 lehrveranstaltung.addRow(new Object[] {
38       Parameters.Lehrveranstaltung.VSYSTEME_ID,
39       Parameters.Professor.HAASE_ID,
40       "Verteilte Systeme",
41       4,
42       5,
43     });
44 lehrveranstaltung.addRow(new Object[] {
45       Parameters.Lehrveranstaltung.DESIGN_PATTERNS_ID,
46       Parameters.Professor.HAASE_ID,
47       "Design Patterns",
48       4,
49       3,
50     });
51 dataSet.addTable(lehrveranstaltung);
```

Listing 3.3: Default-DataSet

Diese Variante löst allerdings nicht alle Probleme. So müssen immer noch Meta-Informationen zu den Tabellen modelliert werden. Die Bewertung stellt sich wie folgt dar:

- **Zeilen**: Ähnlich wie für die Modellierung über XML-Dateien sind für eine übersichtliche Formatierung viele Zeilen notwendig und umfangreiche Datensets werden daher unübersichtlich. Für die Modellierung der Beispiel-Daten werden 152 Zeilen benötigt, wovon ein beträchtlicher Teil nur für die Meta-Informationen zu den Tabellen beansprucht wird (siehe Zeilen 3-9 und 29-36).

- **Zeichen pro Zeile**: Mit maximal 74 Zeichen pro Zeile gibt es keine Probleme mit der Bildschirmdarstellung für die Beispieldaten.

- **Typsicherheit**: Obwohl die Meta-Informationen zu den Tabellen sogar Typinformationen beinhalten, werden Typ-Fehler erst zur Laufzeit beim Einspielen in die Datenbank erkannt.
- **Redundanz**: Redundante Werte können durch den Einsatz symbolischer Konstanten vermieden werden.
- **Benutzbarkeit (Verständlichkeit und Erlernbarkeit)**: Der Code ist zwar nicht unbedingt intuitiv, aber überschaubar und für einen Java-Programmierer auch leicht erlernbar. Die Verständlichkeit leidet unter den selben Problemen wie die `XmlDataSets`: Daten werden einfach in Form geordneter Listen übergeben, ohne dass direkt in der Umgebung ersichtlich ist, welcher Wert sich auf welche Spalte bezieht. Der Einsatz von Konstanten kann hier helfen. Davon kann auch die Modellierung von Beziehungen profitieren.

  Allerdings können solche Konstanten auch falsch verwendet werden: Eine doppelte Belegung ist genauso möglich wie das Zuweisen eines Fremdschlüssels auf den Primärschlüssels einer falschen Tabelle.
- **Modifizierbarkeit**: Das Anpassen des DataSets an neue Datenbank-Schemen kann relativ umständlich werden, wenn Spalten hinzukommen oder wegfallen. Das Umbenennen von Spalten ist relativ leicht möglich. Einige Änderungen können durch IDE-Rafactoring-Unterstützung einfach durchgeführt werden, z.B. das Ändern von Bezeichnern für Konstanten.

Insgesamt bietet die Nutzung dieser Java-DataSets wenig Vorteile gegenüber den XML-DataSets.

3.4.4 STU-DataSet

Die Bibliothek *STU* ermöglicht die Modellierung von DbUnit-DataSets mit Hilfe eines Datenbank-Modell-spezifischen API. Dieses API wird über einen Generator erzeugt (siehe auch 2.4).

STU führt eine eigene DataSet-Klasse ein, über die die Daten modelliert werden. Diese DataSet-Klasse kann bei Bedarf von den aktuell in den *STU*-Klassen gehaltenen Daten ein neues und unabhängiges DbUnit-DataSet erzeugen. Auf diese Weise können DataSets aus *STU* einfacher und umfangreicher als DbUnit-DataSets modifiziert werden. So unterstützt *STU* auch das Löschen von Zeilen aus einem DataSet.

Eine Stärke von *STU* ist es, dass verhältnismäßig einfach Varianten eines DbUnit-DataSets erzeugt werden können, z.B. ein DataSet mit dem Ausgangszustand und ein DataSet mit dem erwarten Zustand am Ende des Tests.

Die Java-DSL sorgt für statische Typsicherheit, so dass Java-IDEs fehlerhafte Typen bereits während der Entwicklung kenntlich machen. Verglichen mit den DbUnit-Xml-DataSets und dem Default-DataSet ist die Syntax ist etwas kompakter und ausdrucksstärker. Spaltennamen und Werte stehen beieinander und nicht über die Datei verteilt.

```
1 table_Professor
2   .insertRow()
3     .setId(Parameters.Professor.HAASE_ID)
4     .setName("Haase")
5     .setVorname("Oliver")
6     .setTitel("Prof.␣Dr.")
```

```
7       .setFakultaet("Informatik")
8     .insertRow()
9       .setId(Parameters.Professor.WAESCH_ID)
10      .setName("Wäsch")
11      .setVorname("Jürgen")
12      .setTitel("Prof. Dr.-Ing.")
13      .setFakultaet("Informatik");
14
15  table_Lehrveranstaltung
16    .insertRow()
17      .setId(Parameters.Lehrveranstaltung.VSYSTEME_ID)
18      .setProfessorId(Parameters.Professor.HAASE_ID)
19      .setName("Verteilte Systeme")
20      .setSws(4)
21      .setEcts(5)
22    .insertRow()
23      .setId(Parameters.Lehrveranstaltung.DESIGN_PATTERNS_ID)
24      .setProfessorId(Parameters.Professor.HAASE_ID)
25      .setName("Design Patterns")
26      .setSws(4)
27      .setEcts(3);
```

Listing 3.4: STU DataSet (1)

Für *STU* ergibt sich folgende Bewertung:

- **Zeilen**: Auch in *STU* dient es der Übersicht, für jeden Spaltenwert eine Zeile zu verwenden. Da die Meta-Informationen in den Builder-Klassen enthalten sind, kommt das Beispiel-DataSet auf 88 Zeilen.
- **Zeichen pro Zeile**: Mit maximal 67 Zeichen pro Zeile für das Beispiel-DataSet gibt es bei der Bildschirm-Darstellung keine Probleme.
- **Typsicherheit**: Die Modellierung mit *STU* ist typsicher, da statische Typ-Prüfungen durchgeführt werden. Diese werden von gängigen IDEs bereits während der Eingabe des Codes durchgeführt.
- **Redundanz**: Da konkrete Werte durch benannte Konstanten ersetzt werden können, stellt Redundanz für *STU* kein Problem dar.
- **Benutzbarkeit (Verständlichkeit und Erlernbarkeit)**: Der Code ist verständlich, aber nicht unbedingt übersichtlich. Die Nutzung gestaltet sich einfach, da die Builder-Klassen für jede Tabelle angepasste Methoden bieten und damit auch die Auto-Vervollständigung von IDEs zur Verfügung steht.

  In Bezug auf die Modellierung von Beziehungen gelten die Nachteile des Default-DataSets (siehe Abschnitt 3.4.3).
- **Modifizierbarkeit**: Durch Refactoring-Mechanismen von IDEs lassen sich viele Änderungen Werkzeug-unterstützt durchführen. Die statische Code-Analyse hilft beim Auffinden übrig gebliebener Daten nach dem Entfernen von Spalten aus dem Datenbank-Schema. Das Hinzufügen von Spalten erfordert mehr Handarbeit, ist aber nicht aufwändiger als in den bisher bewerteten Varianten.

Eine Erweiterung des Datenbank-Modells und des Generators kann die Modellierung von Beziehungen bereits etwas verbessern. Diese Erweiterung erlaubt es, anstelle eines Fremdschlüssels eine vorher eingefügte Zeile anzugeben (siehe Listing 3.5, Zeilen 20 und 27). Hier können referenzierte Primärschlüssel auch automatisch vergeben werden.

```
RowBuilder_Professor haase =
  table_Professor
    .insertRow()
      .setName("Haase")
      .setVorname("Oliver")
      .setTitel("Prof.␣Dr.")
      .setFakultaet("Informatik");
RowBuilder_Professor waesch =
  table_Professor
    .insertRow()
      .setName("Wäsch")
      .setVorname("Jürgen")
      .setTitel("Prof.␣Dr.-Ing.")
      .setFakultaet("Informatik");

RowBuilder_Lehrveranstaltung vsys =
  table_Lehrveranstaltung
    .insertRow()
      .setName("Verteilte␣Systeme")
      .refProfessorId(haase)
      .setSws(4)
      .setEcts(5);
RowBuilder_Lehrveranstaltung design_patterns =
  table_Lehrveranstaltung
    .insertRow()
      .setName("Design␣Patterns")
      .refProfessorId(haase)
      .setSws(4)
      .setEcts(3);
```

Listing 3.5: STU DataSet (2)

Kapitel 4

Entwurf einer Modellierungssprache für Testdaten

Ein Ziel dieser Arbeit ist es, die Modellierung von Testdaten zu vereinfachen. DbUnit ist eine bewährte Bibliothek zur Unterstützung von Unit-Tests datenbankbasierter Anwendungen. Welche Schwächen DbUnit-DataSets in Bezug auf die Modellierung haben, wurde bereits in Abschnitt 3.4 thematisiert.

Ein üblicher Weg, bestehende Schnittstellen zu vereinfachen, stellen Fassaden dar. Eine Fassade stellt eine Schnittstelle auf höherer Abstraktionsebene dar, um das System einfach zu verwenden [Gam+94, S. 185]. Eine solche Fassade kann auf unterschiedliche Arten realisiert werden.

Eine Möglichkeit stellt die Definition einer speziellen Sprache dar. Eine solche, für einen Anwendungszeck definierte Sprache wird als *Domänenspezifische Sprache* (engl. Domain-Specific Language, abgekürzt DSL) bezeichnet.

Das von *STU* bereitgestellte Fluent Interface stellt bereits eine Form einer DSL dar. Eine solche DSL, die vollständig in eine andere Sprache eingebettet ist und im Wesentlichen die Sprachelemente dieser Sprache nutzt, wird als *interne DSL* bezeichnet [Fow10, S. 68].

In diesem Abschnitt soll eine Modellierungssprache für Testdaten entwickelt werden. Die Vor- und Nachteile der in Kapitel 3 diskutierten Modellierungsvarianten sollen in die Sprache einfließen. Die Sprache muss nicht zwingend auf Java basieren, sie kann auch als *externe DSL* realisiert werden. Der Vorteil von externen gegenüber internen DSLs besteht in der größeren Freiheit in Bezug auf die Syntax [Fow10, S. 89].

4.1 Entwurf der DSL

Als Grundlage für die Defintion der Modellierungsprache sollen verschiedene Beispiel-Entwürfe dienen. Dabei wird das fortlaufende Beispiel aus Kapitel 3 mit unterschiedlichen Entwürfen modelliert und die Vor- und Nachteile analysiert. Bei den Entwürfen wird versucht, die Nachteile der bisher bewerteten Modellierungssprachen auszugleichen.

Mit Hilfe dieser Analyse wird die finale Modellierungssprache (DSL) definiert. Bei den Bewertungskriterien wird auf den implementierungsabhängige Aspekte verzichtet. So wird der Punkt Typsicherheit nicht betrachtet und bei der Modifizierbarkeit keine Aussage zu IDE-unterstützten Refaktorisierungsmöglichkeiten gemacht.

4.1.1 Beispiel-Entwurf 1

Eine DSL, die sich stark an *STU* orientiert, könnte wie folgt aussehen:

```
1  HAASE = professor {
2    name       "Haase"
3    vorname    "Oliver"
4    titel      "Prof. Dr."
5    fakultaet  "Informatik"
6  }
7
8  WAESCH = professor {
9    name       "Wäsch"
10   vorname    "Jürgen"
11   titel      "Prof. Dr.-Ing."
12   fakultaet  "Informatik"
13 }
14
15 VSYS = lehrveranstaltung {
16   name       "Verteilte Systeme"
17   sws        4
18   ects       5
19 }
20
21 DPATTERNS = lehrveranstaltung {
22   name       "Design Patterns"
23   sws        4
24   ects       3
25 }
26
27 ...
28
29 HAASE leitet VSYS
30 HAASE leitet DPATTERNS
31 HAASE beaufsichtigt  P_DPATTERNS
32 WAESCH beaufsichtigt P_VSYS
33 ...
```

Listing 4.1: Beispiel zum Entwurf 1

Die in Listing 4.1 gezeigte DSL kommt ohne manuell vergebene Schlüsselwerte (Primary-Keys) aus und verwendet Variablennamen für die Modellierung von Beziehungen. Für diesen Entwurf sieht die Bewertung folgendermaßen aus:

- **Zeilen**: Für die Modellierung der Beispieldaten werden 64 Zeilen benötigt. Für jeden Spaltenwert und jede Beziehung wird jeweils eine eigene Zeile benötigt. Umfangreiche Daten können in Bezug auf den Umfang deshalb schnell unübersichtlich werden.
- **Zeichen pro Zeile**: Der Entwurf ist für die Bildschirmdarstellung gut geeignet, die Beispieldaten erreichten höchstens 50 Zeichen pro Zeile.
- **Redundanz**: Die Datenredundanz wird durch das Verwenden von Bezeichnern für Entitäten (Zeilen 1, 8, 15 und 21) vermieden, die für die Beschreibung von Beziehungen verwendet werden.
- **Benutzbarkeit (Verständlichkeit und Erlernbarkeit)**: Der Entwurf ist verständlich und leicht zu erlernen, da sich die meisten Sprachelemente auf das Datenbank-Schema beziehen.

  Die Beschreibung der Beziehungen entfernt von der Definition der Daten kann den Umgang mit den Daten erschweren und die Übersicht einschränken.
- **Modifizierbarkeit**: Auch ohne Refaktorisierungsmöglichkeiten sollte sich ein Data-Set leicht an neue Datenbank-Schemata anpassen lassen, z.B. über Suchen/Ersetzen oder Makro-Funktionen in Text-Editoren.

4.1.2 Beispiel-Entwurf 2

Ein leicht abgewandelter Entwurf (siehe Listing 4.2) zeigt, wie sich die Beziehungen näher an den eigentlichen Daten beschreiben lassen könnten. Das Problem, dass die Daten relativ viele Zeilen benötigen, besteht weiterhin.

```
1  HAASE = professor {
2    name       "Haase"
3    vorname    "Oliver"
4    titel      "Prof. Dr."
5    fakultaet  "Informatik"
6    leitet     VSYS, DPATTERNS
7    beaufsichtigt  P_DPATTERNS
8  }
9
10 WAESCH = professor {
11   name       "Wäsch"
12   vorname    "Jürgen"
13   titel      "Prof. Dr.-Ing."
14   fakultaet  "Informatik"
15   beaufsichtigt  P_VSYS
16 }
17
18 VSYS = lehrveranstaltung {
19   name       "Verteilte Systeme"
20   sws        4
21   ects       5
22 }
23
24 DPATTERNS = lehrveranstaltung {
25   name       "Design Patterns"
26   sws        4
27   ects       3
28 }
29
30 ...
```

Listing 4.2: Beispiel zum Entwurf 2

Die Änderungen im Vergleich zum ersten Entwurf sind überschaubar. Dies spiegelt sich auch in der Bewertung wider:

- **Zeilen**: Die Möglichkeit, mehrere Beziehungen zusammenzufassen (Zeile 6), kann einige Zeilen einsparen. Das Beispiel-DataSet kommt hier allerdings noch auf 62 Zeilen, ein Wert benötigt nach wie vor eine Zeile.
- **Zeichen pro Zeile**: Hier hat sich nichts geändert, der Entwurf ist ebenfalls für die Bildschirmdarstellung geeignet.
- **Redundanz**: Wie beim ersten Entwurf keine erkennbare Werte-Redundanz.
- **Benutzbarkeit (Verständlichkeit und Erlernbarkeit)**: Die Möglichkeit, Beziehungen einer Entität direkt bei ihrer Definition beschreiben zu können, kann die Verständlichkeit verbessern. Ansonsten gelten die für Entwurf 1 genannten Punkte.
- **Modifizierbarkeit**: Im Vergleich zum ersten Entwurf keine Änderungen.

4.1.3 Beispiel-Entwurf 3

Der dritte Entwurf ist inspiriert von dem Test-Framework *Spock*, das es erlaubt, Testdaten in Unit-Tests tabellarisch zu definieren [Nie+12]. Der Entwurf ist in Listing 4.3 dargestellt. Es wird versucht die Daten durch eine tabellarische Struktur übersichtlich zu gestalten. Die

Syntax der Sprache ist sehr schlicht und ausdrucksstark. Ein Label vor einer Tabelle drückt aus, welche Daten folgen (Zeilen 1 und 6). Die Tabelle selbst beginnt mit einer Kopfzeile, die die Spaltenreihenfolge beschreibt (Zeilen 2 und 7). Einzelne Spalten werden vom Oder-Operator (|) getrennt. Die erste Spalte nimmt Zeilen-Identifikatoren auf und ist von den Daten mit Hilfe des Double-Pipe-Operators (||) abgrenzt.

```
1  professor:
2  REF    || name    | vorname  | titel           | fakultaet    | leitet          | beaufsichtigt
3  HAASE  || "Haase" | "Oliver" | "Prof. Dr."     | "Informatik" | VSYS, DPATTERNS | P_DPATTERNS
4  WAESCH || "Wäsch" | "Jürgen" | "Prof. Dr.-Ing." | "Informatik" |                 | P_VSYS
5
6  lehrveranstaltung:
7  REF       || name               | sws | ects
8  VSYS      || "Verteilte Systeme" | 4   | 5
9  DPATTERNS || "Design Patterns"   | 4   | 3
10
11 ...
```

Listing 4.3: Beispiel zum Entwurf 3

Der Entwurf sieht vor, dass Beziehungen innerhalb beider Entitätstypen ausgedrückt werden können. So kann eine Tabelle um Spalten für Beziehungen ergänzt werden, die in dieser Form nicht Teil des relationalen Modells (siehe Abb. 3.2) sind. Dazu gehören die Spalten „leitet“ und „beaufsichtigt“ der Professor-Tabelle. Erstere drückt die 1:n-Beziehung zu einer Lehrveranstaltung aus, letztere die m:n-Beziehung zu Prüfungen.

- **Zeilen**: Der größte Vorteil dieser Darstellung ist, dass nicht jeder Wert, sondern dass nur jede Entität zu einer Zeile führt. Die Beispieldaten kommen mit dieser Sprache auf gerade mal 19 Zeilen.
- **Zeichen pro Zeile**: Die Beispieldaten erreichen eine maximale Breite von bis zu 144 Zeichen bei einer Tabelle mit 10 Spalten. Für die Bildschirmdarstellung kann dies problematisch sein und horizontales Scrollen notwendig machen.

  Probleme bzw. Nachteile in der Darstellung können auftreten, wenn die Länge der Werte in einer Spalte stark variiert. Die Spaltenbreite wird vom längsten Element bestimmt. Der Entwickler ist selbst dafür verantwortlich, die übersichtliche Darstellung einzuhalten. Bei vielen Spalten wächst diese Darstellung horizontal. Optionale bzw. kaum genutzte Spalten erhalten bei dieser Darstellung unter Umständen viel Platz.
- **Redundanz**: Wie die beiden bisherigen Entwürfe tritt hier keine erkennbare Daten-Redundanz auf. Dieser Entwurf vermeidet außerdem die Redundanz der Spaltenbezeichner.
- **Benutzbarkeit (Verständlichkeit und Erlernbarkeit)**: Die tabellarische Darstellung von Daten ist verständlich und auch nicht schwer zu erlernen. Einige Entwicklungsumgebungen wie Eclipse bieten spezielle Block-Bearbeitungsfunktionen an, die beim Arbeiten an einer Tabellen-DSL hilfreich sein können. So können beispielsweise in einer Spalte über mehrere Zeilen hinweg Leerzeichen eingefügt oder entfernt werden.

  Bei umfangreichen Tabellen, die möglicherweise nicht mehr auf eine Bildschirmseite passen, kann es sinnvoll sein, den Tabellenkopf zu wiederholen. Dies sollte von der Implementierung genauso unterstützt werden wie die Definition neuer Tabellenköpfe mit unterschiedlichen Spalten. Dies erlaubt auch, für Spalten mit optionalen Werten kleinere Teiltabellen zu definieren.
- **Modifizierbarkeit**: Mit Hilfe geeigneter Editoren ist ein DataSet leicht an neue Datenbank-Schemata anzupassen. Die Daten selbst lassen sich auch relativ einfach ändern.

4.1.4 Finaler DSL-Entwurf

Der dritte Entwurf zeigt, dass eine tabellarische Schreibweise viele Schwächen der anderen Varianten vermeidet. Die Darstellung wirkt übersichtlich, da sie wenig syntaktischen Ballast hat. Es können schnell viele Daten überblickt werden. Die tabellarische Schreibweise wird für die Zielgruppe vertraut wirken und intuitiv sein.

Darüber hinaus soll es möglich sein, Beziehungen auch außerhalb der Tabellen zu beschreiben. Dafür wäre eine Syntax denkbar, die sich an die Modellierung der Beziehungen aus Entwurf 1 orientiert (siehe Listing 4.1).

Der finale Entwurf stellt eine Kombination aus der tabellarischen Syntax von Entwurf 3 und der Modellierung der Beziehungen aus Entwurf 1 dar.

4.2 Wahl der Technologie

Die DSL soll sich in die bisherige Werkzeugkette von SEITENBAU integrieren lassen (siehe Abschnitt 3.1). In [Gho10, S. 148] empfiehlt Ghosh die Programmiersprache Groovy als Host für DSLs in Verbindung mit Java-Anwendungen.

Groovy ist eine dynamisch typisierte Sprache[1], die direkt in vollständig komplatbilen Java-Bytecode übersetzt wird und damit auch in einer Java Virtual Machine (JVM) ausgeführt wird. Dies ermöglicht die Nutzung von Groovy-Klassen und Groovy-Objekten in Java und umgekehrt.

Java-Code ist bis auf wenige Ausnahmen gültiger Groovy-Code. Allerdings bietet Groovy Möglichkeiten, Code von syntaktischem Ballast zu befreien. Beispielsweise können Semikolons am Ende einer Anweisung meistens weggelassen werden. Der Punkt zwischen einer Variable und der Methode ist unter gewissen Bedingungen ebenfalls optional. Häufig kann auch auf die Klammerung von Methodenparametern verzichtet werden. werden. Auf diese Weise ermöglicht Groovy eine Syntax, die den Code mehr wie eine natürlichen Sprache aussehen lässt.

Listing 4.4 zeigt die selben Anweisungen einmal in typischer Java-Syntax (Zeile 1) und einmal mit den Syntax-Vereinfachungen von Groovy (Zeile 2). Das Beispiel stammt aus [LK12, S. 65]. Die zweite Zeile liest sich wie ein natürlichsprachiger, englischer Satz:

```
1 take(coffee).with(sugar, milk).and(liquor);
2 take coffee with sugar, milk and liquor
```

Listing 4.4: Vereinfachung von Ausdrücken in Groovy

Groovy hebt sich ferner durch die Möglichkeit Operatoren zu überladen und durch Closures von Java ab. Ein Closure (Funktionsabschluss) ist ein Codeblock, der wie eine Funktion aufgerufen und genutzt werden kann. In Java lassen sich Closures mit syntaktisch umfangreicheren Methoden-Objekten nachbilden. Ein Methoden-Objekt stellt eine Instanz einer (möglicherweise anonymen) Klasse dar, die nur eine Methode implementiert. [Kön07, S. 40] [Bec96, S. 34]

Die Unterstützung zur Meta-Programmierung stellt sich beim Implementieren einer DSL ebenfalls als nützlich heraus. Dadurch ist es möglich, zur Laufzeit Klassen innerhalb von Groovy um Methoden zu erweitern oder auf den Zugriff von nicht definierten Klassenelementen zu reagieren.

[1] Im Gegensatz zu statisch typisierten Sprachen finden bei dynamisch typisierten Typ-Überprüfungen überwiegend zur Laufzeit statt.

4.2.1 Implementierungsvarianten mit Groovy

Eine DSL kann auf unterschiedliche Arten implementiert werden. Groovy bietet dafür zwei Möglichkeiten der Meta-Programmierung an: Laufzeit-Meta-Programmierung und Compiler-Zeit-Meta-Programmierung. Für die Compilier-Zeit-Meta-Programmierung bietet Groovy AST-Transformationen. an. Beide Ansätze bieten individuelle Vorteile, die im Folgenden betrachtet werden.

Laufzeit-Meta-Programmierung

Eine Möglichkeit zur Realisierung der DSL stellt eine Klasse zum Parsen von Closures dar. Die Closures enthalten die Daten. Diese Klasse, `TableParser`, enthält dafür die Methode `parseTableClosure`. Die Methode soll als Ergebnis eine Liste von Tabellenzeilen zurückliefern. Eine Tabellenzeile selbst stellt zu diesem Zeitpunkt eine Liste von Spaltenwerten dar. Die Analyse und Interpretation der Daten findet erst später statt.

Der Ansatz ist, Operator-Überladen für das Parsen zu verwenden. Der Operator `or` ist ein binärer Operator. Üblicherweise ruft Groovy einen binären Operator auf der vom Operator linksseitigen Instanz mit dem rechtsseitigen Operand als Parameter. [Kön07, S. 58]).

Auch wenn sich dank der Möglichkeiten der Meta-Programmierung Klassen in Groovy zur Laufzeit um Methoden ergänzen lassen, ist dieses Vorgehen nicht empfehlenswert, um eine Tabelle zu parsen. Dieser wenig generische Ansatz müsste jeden in den Tabellen mögliche Datentyp berücksichtigen – kommen neue Datentypen hinzu, müsste der Code erweitert werden. Gravierender ist jedoch, dass diese Anpassungen global für die entsprechenden Klassen gelten. Das könnte ungewollte Seiteneffekte nach sich ziehen, wenn Oder-Operatoren auch an anderer Stelle verwendet werden, wie z.B. zum Berechnen eines Spalten-Wertes.

Groovy bietet eine zweite Möglichkeit für das Operator-Überladen an. Anstatt den Operator als Methode dem linken Operand (bzw. der Klasse) hinzuzufügen, wird der Operator als statische Methode realisiert. Die statische Methode kann in einer beliebigen Klasse definiert sein. Da eine statische Methode ohne Kontext ausgeführt wird, benötigt sie alle beteiligten Operanden als Parameter. Eine solche Methode wird als Kategoriemethode bezeichnet. Über das Schlüsselwort `use`[2] können Kategoriemethoden in einem Closure verwendet werden. [Kön07, S. 192]

```
1  class TableParser {
2
3    static or(Object self, Object arg) {
4      ...
5    }
6
7    def parseTableClosure(Closure tableData){
8      use(TableParser) {
9        tableData()
10     }
11   }
12
13 }
```

Listing 4.5: Tabellen-Parser Grundgerüst mit Operator-Überladen

Listing 4.5 zeigt das Grundgerüst des Tabellenparsers. Die Methode `or` erwartet zwei Parameter vom Typ `Object`. Obwohl in Groovy alle Typen von `Object` abgeleitet sind, gibt

[2] `use` wird in der Literatur meistens als Schlüsselwort bezeichnet, tatsächlich handelt es sich jedoch um eine Groovy-Methode in `java.lang.Object`

es Oder-Ausdrücke, bei denen diese Methode nicht aufgerufen wird. Ein in der Klasse definierter Operator mit passenden Datentypen wird dieser allgemeinen Methode bevorzugt. Die Klasse Integer definiert einen `or`-Operator für ein Oder für zwei Integer-Werte, die von Groovy vorrangig aufgerufen wird. Doch auch in solchen Fällen können die Operatoren überschreiben werden, wenn für die Datentypen passende Kategoriemethoden definiert werden. Diese sind in Listing 4.6 für die Typen Integer und Boolean dargestellt.

Für die Definition der Daten sollen Bezeichner für die Entitäten genutzt werden können. Diese Bezeichner werden nicht im Closure selbst, sondern in einer anderen Klasse definiert. Zu diesen Variablen gehören die Bezeichner der Spalten und die der Zeilen. Groovy erlaubt es Closures im Kontext eines Delegaten auszuführen. In diesem Beispiel wird die aktuelle Instanz der Klasse `TableParser` verwendet (Zeile 22). Dieser Delegat wird bei der Auflösung von Variablen- und Methoden-Bezeichern verwendet, d.h. es können innerhalb des Closures alle Elemente des Delegaten genau so verwendet werden, als wären sie im Closure selbst definiert. In Zeile 23 wird festgelegt, dass bei der Auflösung zuerst im Delegaten gesucht wird.

In diesem Prototyp sind keine Variablen für Spalten oder Zeilen definiert. Folglich schlägt das Auflösen fehl. Groovy ruft Groovy dann die Methode `propertyMissing` in der jeweiligen Klasse auf. Eine Property ist eine Variable oder eine parameterlose Get-Methode. Eine solche Get-Methode kann in Groovy wie eine Variable genutzt werden, wenn auf das Präfix `get` und die Klammern verzichtet wird. Durch Überschreiben dieser Methode kann auf nicht auflösbare Bezeichner reagiert werden.

```
1  class TableParser {
2
3    static or(Object self, Object arg) {
4      ...
5    }
6
7    static or(Integer self, Integer arg) {
8      ...
9    }
10
11   static or(Boolean self, Boolean arg) {
12     ...
13   }
14
15   def propertyMissing(String property) {
16     ...
17   }
18
19
20   def parseTableClosure(Closure tableData){
21     use(TableParser) {
22       tableData.delegate = this    // Change closure's context
23       tableData.resolveStrategy = Closure.DELEGATE_FIRST
24       tableData()
25     }
26   }
27
28 }
```

Listing 4.6: Tabellen-Parser Grundgerüst mit Operator-Überladen

Die statischen `or`-Methoden haben keinen Zugriff auf Instanz-Variablen der Klasse `TableParser`. Ihre Zwischenergebnisse, also die geparsten Spaltenwerte, können nur in statische Elementen abgelegt werden. Um die Klasse Thread-sicher zu machen, d.h. das Erlauben von gleichzeitigem Parsen von Tabellen aus verschiedenen Threads heraus, wird für die Ergebnisse eine threadlokale Variable verwendet. [Goe09, S. 45]

Die Laufzeit-Meta-Programmierung kann die Syntax der Sprache nicht beliebig erweitern. Groovy kennt keinen Double-Pipe-Operator. Deshalb kann dieser weder überladen noch

über Laufzeit-Meta-Programmierung eingeführt werden. Folglich ist es nicht möglich, den dritten Entwurf über reine Laufzeit-Meta-Programmierung zu realisieren. Allerdings kann eine Syntax erreicht werden, die dem Entwurf sehr nahe kommt (siehe Listing 4.7). Das DataSet wird als Map definiert, mit den Tabellennamen als Schlüsseln und Closures als Werte. Ein Platzhalter (Unterstrich) verhindert Syntax-Fehler, wenn in einer Spalte kein Wert vorkommt (siehe Zeile 4, Spalte „leitet“). Der Platzhalter könnte auch verwendet werden, um einem Datensatz keinen Bezeichner für Referenzen zu zu weisen.

```
1  def dataset = [
2    professor: {
3      REF    | name    | vorname  | titel            | fakultaet    | leitet           | beaufsichtigt
4      WAESCH | "Wäsch" | "Jürgen" | "Prof. Dr.-Ing." | "Informatik" | _                | P_VSYS
5      HAASE  | "Haase" | "Oliver" | "Prof. Dr."      | "Informatik" | VSYS & DPATTERNS | P_DPATTERNS
6    },
7
8    lehrveranstaltung: {
9      REF       | name               | sws | ects
10     VSYS      | "Verteilte Systeme" | 4   | 5
11     DPATTERNS | "Design Patterns"   | 4   | 3
12   },
13
14   ...
15 ]
```

Listing 4.7: DSL-Entwurf 3 für Laufzeit-Meta-Programmierung angepasst

AST-Transformation

Die AST-Transformationen stellen ein mächtiges Werkzeug zur Erweiterung der Syntax der Sprache dar. Mit Hilfe der Transformationen ist es möglich, Änderungen am AST durchzuführen, bevor er in Java-Bytecode übersetzt wird. Das Test-Framework Spock nutzt AST-Transformationen zum Parsen.

Dass AST-Transformationen mehr syntaktische Möglichkeiten bieten, zeigt sich auch daran, dass hier der Double-Pipe-Operator verwendet werden kann. Außerdem können Labels erkannt werden und Daten einer Tabelle müssen nicht zwangsläufig in einem eigenen Block definiert werden.

Allerdings muss zum Auswerten einer Tabelle bei AST-Transformationen ein relativ großer Aufwand betrieben werden. Ein AST-Transformationsklasse, erhält über das Visitor-Pattern Zugriff auf die abstrakten Syntaxbäume einzelner Module [Gam+94, 331ff]. Groovy Module beinhalten Klassen, aber auch die modulspezifischen Import-Anweisungen. Auf die einzelnen Klassen kann erneut über das Visitor-Pattern auf die einzelnen Methoden zugegriffen werden. Diese lassen sich dann Statement für Statement untersuchen.

Für das Parsen relevant sind Statements vom Typ `ExpressionStatement`. Es kann abgefragt werden, ob ein Label Teil des Statements ist. Über ein solches Label können die Daten den einzelnen Tabellen zugeordnet werden. Das eigentliche `ExpressionStatement` kann danach analysiert werden. Die folgenden drei Arten von Ausdrücken sind relevant für das Parsen:

- **`BinaryExpression`**: Ein binärer Ausdruck besteht aus zwei Operanden und einem Operator. Wenn es sich beim Operator um einen Pipe oder Double-Pipe-Operator handelt, werden die beiden Operanden behandelt. Dies führt zu einer rekursiven Auswertung, da die Operanden vom Typ `ExpressionStatement` sind.
- **`ConstantExpression`**: Konstante Ausdrücke sind Literale, die als Spaltenwert verwendet werden.

- **`VariableExpression`**: Ein Bezeichner einer Variablen. Dazu gehören die Spalten-Bezeichner und die Bezeichner für die einzelnen Zeilen.

Ein einfacher Prototyp zum Parsen von Tabellen zeigt, dass viel Aufwand betrieben werden, um den AST zu analysieren. Aus diesem Grund wird die weitere Implementierung eines Tabellen-Parsers mit Hilfe der AST-Transformationen nicht betrachtet. Möglicherweise kann durch die Nutzung von sogenannten DSL-Descriptoren für die Groovy-Plugins gängiger IDEs auch eine IDE-Unterstützung für Labels und Spalten erreicht werden.

4.2.2 Implementierungsentscheidung

Der Vergleich zwischen Laufzeit-Meta-Programmierung und AST-Transformation zeigt, dass sich Groovy als Host-Sprache für die DSL eignet. Grundsätzlich kann das Parsen der Tabelle über beide Varianten durchgeführt werden und beide Varianten erfüllen die Anforderungen.

Die Entscheidung fällt auf die Laufzeit-Meta-Programmierung. Der Grund dafür ist, dass sie einfacher zu verwenden ist. Der Code für AST-Transformationen stellt sich als komplexer und wartungsunfreundlicher dar. Es ist fraglich, ob AST-Transformationen einen Mehrwert in Bezug auf die Anforderungen zeigen würden.

4.3 Ergebnis: Definition der DSL für Testdaten

Die Entscheidung für die Laufzeit-Meta-Programmierung führt zu einigen Änderungen an der Sprache aus dem dritten Entwurf (siehe Abschnitt 4.1.3). Wie beschrieben, wird auf den Double-Pipe-Operator verzichtet. Anstelle der Labels bietet die DataSet-Klasse für jede Tabelle eine vordefinierte Variable, deren Bezeichnung sich aus dem Tabellennamen ergibt. Auf diesen Variablen kann zum Definieren von Tabellendaten die Methode `rows` aufgerufen werden. Die Daten werden in Form eines Closures übergeben.

Zur Übersicht sollen einzelne DataSets als eigene Klassen definiert werden, die auf einer Datenbank-Modell-spezifischen abstrakten Klasse basieren. Für die Definition der Tabellen-Daten ist die Methode `tables` vorgesehen, über die DSL ausgedrückte Beziehungen sollen innerhalb der Methode `relations` modelliert werden.

Die Syntax für die Definition der Daten einer Tabelle wird mit Hilfe der Erweiterten Backus-Naur-Form in Listing 4.8 definiert.

```
1  Table       = TableName, "Table.rows ", TableData;
2  TableName   = ? Name einer Tabelle im Modell ?;
3  TableData   = "{", NewLine, { HeadRow, { DataRow } }, NewLine, "}";
4  HeadRow     = ColumnName, { Separator, ColumnName }, NewLine;
5  DataRow     = ColumnData, { Separator, ColumnData }, NewLine;
6  ColumnName  = ? vorgegeben durch Tabelle ?;
7  ColumnData  = ? numerische Literale, symbolische Konstanten,
8                Methodenaufrufe ?;
9  Separator   = "|";
10 NewLine     = "\n";
```

Listing 4.8: EBNF für Tabellen

Abbildung 4.1 stellt die EBNF für Tabellen grafisch dar.

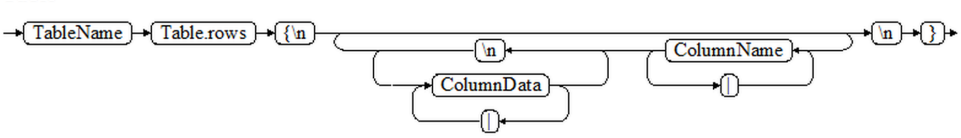

Abbildung 4.1: Grafische Darstellung der EBNF für Tabellen

Die Syntax für die Definition von Beziehungen veranschaulichen Listing 4.9 und die Abbildung 4.2.

```
Relations   = { Relation, NewLine };
Relation    = Ref, ".", RelationName, "(", RefList, ")", [ Attributes ];
RefList     = Ref, [ { ", ", Ref } ];
Ref         = ? Bezeichner einer Ref-Variable ?;
Attributes  = { ".", Attribute, "(", Value, ")" };
Attribute   = ? Von Beziehung abhängiger Attribut-Bezeichner ?;
Value       = ? numerische Literale, symbolische Konstanten,
              Methodenaufrufe ?;
NewLine     = "\n";
```

Listing 4.9: EBNF für Beziehungen

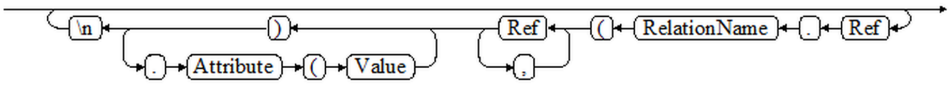

Abbildung 4.2: Grafische Darstellung der EBNF für Beziehungen

4.4 Beispiel-DataSet in Groovy

In Kapitel 3 wurden Beispiel-Daten in unterschiedlichen Verfahren modelliert. Aufgrund der Übersicht wurden dort nur jeweils zwei Tabellen dargestellt. Listing 4.10 zeigt, wie sich dieselben Daten mit der neuen DSL modellieren lassen – diesmal allerdings in größerem Umfang.

```
class HochschuleDataSet extends HochschuleBuilder
{

  def tables() {

    professorTable.rows {
      REF    | name    | vorname  | titel            | fakultaet
      WAESCH | "Wäsch" | "Jürgen" | "Prof. Dr.-Ing." | "Informatik"
      HAASE  | "Haase" | "Oliver" | "Prof. Dr."      | "Informatik"
    }

    lehrveranstaltungTable.rows {
      REF        | name               | sws | ects
      VSYS       | "Verteilte Systeme" | 4   | 5
      DPATTERNS  | "Design Patterns"   | 4   | 3
    }

    pruefungTable.rows {
      REF          | typ   | zeitpunkt
      P_VSYS       | "K90" | DateUtil.getDate(2013, 4, 1, 14, 0, 0)
      P_DPATTERNS  | "M30" | DateUtil.getDate(2013, 1, 6, 12, 0, 0)
    }

    studentTable.rows {
      REF        | matrikelnummer | name         | vorname    | studiengang
      MUSTERMANN | 123456         | "Mustermann" | "Max"      | "BIT"
```

```
27        MOLL        | 287336          | "Moll"        | "Nikolaus" | "MSI"
28
29        REF         | semester | immatrikuliert_seit
30        MUSTERMANN  | 3        | DateUtil.getDate(2012, 3, 1)
31        MOLL        | 4        | DateUtil.getDate(2011, 9, 1)
32      }
33
34    }
35
36    def relations() {
37      WAESCH.beaufsichtigt(P_VSYS)
38      HAASE.leitet(VSYS, DPATTERNS)
39      HAASE.beaufsichtigt(P_DPATTERNS)
40      P_VSYS.stoffVon(VSYS)
41      DPATTERNS.hatPruefung(P_DPATTERNS)
42      MOLL.schreibt(P_VSYS)
43      MOLL.besucht(VSYS)
44      VSYS.hatTutor(MOLL)
45      MUSTERMANN.besucht(DPATTERNS)
46    }
47
48  }
```

Listing 4.10: DataSet modelliert mit dem Table Builder API

Insgesamt ist die Darstellung sehr übersichtlich. Innerhalb der Methode `tables` werden die Daten der einzelnen Tabellen auf Datenbank-Ebene modelliert. Die Methode `relations` erlaubt die Modellierung von Beziehungen auf der höheren ER-Ebene. Sogar Beziehungen, die über assoziative Tabellen realisiert werden, können so modelliert werden. Entsprechend der Kriterien stellt sich die Bewertung für die Sprache wie folgt dar:

- **Zeilen**: Die komplette Klasse, die das DataSet repräsentiert, erreicht 48 Zeilen. Normalerweise benötigt jede Entität eine Zeile, Ausnahmen sind aber möglich.
- **Zeichen pro Zeile**: Die Anzahl der Zeichen pro Zeile kann zu groß für eine Bildschirmdarstellung werden. Dies kann auf Kosten zusätzlicher Zeilen allerdings vermieden werden (siehe Zeilen 25 bis 31). Dort wurde der Code aufgrund der eingeschränkten Seitenbreite angepasst und die Tabellendefinition in zwei Teiltabellen aufgeteilt.
- **Redundanz**: Wie bei den Entwürfen zuvor ist keine Daten-Redundanz zu erkennen.
- **Benutzbarkeit (Verständlichkeit und Erlernbarkeit)**: Die Sprache ist sehr schlicht und die Sprachelemente werden überwiegend vom Datenbank-Modell vorgegeben. Dadurch ist sie verständlich und auch leicht erlernbar.

  Die Sprache kann flexibel genutzt werden: Es ist möglich, in einem Tabellen-Block verschiedene Teiltabellen zu definieren. Auf diese Weise kann die Zeilenbreite verringert werden, der Tabellenkopf für die Übersicht wiederholt oder auch Spalten mit optionalen Werten in eine Teiltabelle verschoben werden.

  Implementierungsspezifische Maßnahmen, die die Benutzbarkeit verbessern sollen, werden in Kapitel 5 beschrieben.
- **Modifizierbarkeit**: Die Daten lassen sich relativ leicht an neue Datenbank-Schemata anpassen und die Daten selbst sich leicht ändern.

Die Flexibilität der Sprache zeigt sich auch darin, dass Beziehungen auch auf der Datenbank-Ebene modelliert werden können. Listing 4.11 veranschaulicht diese Variante.

```
class HochschuleDataSet extends HochschuleBuilder
{

  def tables() {

    ...

    lehrveranstaltungTable.rows {
      REF       | professor
      VSYS      | HAASE
      DPATTERNS | HAASE
    }

    beaufsichtigtTable.rows {
      professor | pruefung
      WAESCH    | P_VSYS
      HAASE     | DPATTERNS
    }

  }

  ...

}
```

Listing 4.11: Beziehungen innerhalb von Tabellen

Kapitel 5

Realisierung der Sprache

Das folgende Kapitel beschreibt die Implementierung der in Kapitel 4 entwickelten Sprache. *STU* stellt die Basis für Erweiterungen und Verbesserung dar. Es verfolgt bereits das Ziel, Tests von Datenbank-basierten Anwendungen zu erleichtern und vereinfacht die Modellierung von DbUnit-DataSets. Auf Abwärtskompatibilität wird wenn notwendig zu Gunsten der Benutzbarkeit und Wartbarkeit verzichtet.

Aus einem domänenspezifischen Datenbank-Modell erzeugt *STU* ein individuelles API zur Modellierung von DataSets. Um die Modellierung zu vereinfachen, vor allem in Bezug auf die Beziehungen, sollen die generierten Klassen um eine Fassade ergänzt werden.

Eine Möglichkeit, eine solche Fassade zu realisieren, stellen domänenspezifische Sprachen dar. Eine domänenspezifische Sprache zeichnet sich dadurch aus, dass sie für ein spezielles Problemfeld entworfen wurde. Martin Fowler erklärt in [Fow10, S. xix], dass die meisten domänenspezifischen Sprachen lediglich eine dünne Fassade über einer Bibliothek oder einem Framework sind.

Der Code-Generator aus *STU* erzeugt zwei APIs für die Modellierung von DataSets:

- Das **Fluent Builder API** ist ein **Java**-basiertes API. Der Name spiegelt wieder, dass es ein Java Fluent API bereit stellt (siehe auch Abschnitt 2.4). Ein solches API wird auch als interne DSL bezeichnet.
- Das **Table Builder API** ist das **Groovy**-basierte API bzw. die neue DSL. Über diese DSL können die Testdaten tabellarisch modelliert werden.

Abbildung 5.1 stellt die Architektur eines Tests grafisch dar. Der Test basiert auf einem Test-Framework wie *JUnit*, der Testbibliothek *STU* und der Bibliothek *DbUnit*. *STU* setzt sich aus den beiden oben genannten Schichten zusammen.

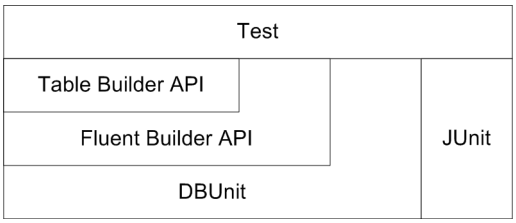

Abbildung 5.1: Architektur

Das neue Table Builder API stellt eine Schicht über dem bisherigen Fluent Builder API dar. Neue Funktionen müssen jedoch nicht zwangsläufig im Table Builder API hinzugefügt werden, unter Umständen kann es vorteilhaft sein, sie direkt in das Fluent Builder API zu integrieren. Gründe dafür sind unter anderem:

- **Code-Qualität**: Die neuen Funktionen können direkt in bestehende Klassen integriert werden, anstatt neue Typen einzuführen. Auf Adapterklassen und Delegation kann auf diese Weise verzichtet. Das erleichtert die Pflege der Implementierung und vermeidet redundanten Code.
- **Mehrwert auch für bisheriges API**: Auch wenn auf das neue Table Builder API verzichtet wird, kann das Builder API so einen Mehrwert gegenüber der bisherigen *STU*-Implementierung bieten. Verbesserungen im Fluent Builder API sind auch in reinen Java-basierten Tests nutzbar.
- **Einheitlicher Funktionsumfang**: Das Table Builder API und das Fluent Builder API sollen – so weit möglich und sinnvoll – den selben Funktionsumfang bieten. Erweiterungen für das Fluent Builder API stehen automatisch auch im Table Builder API zur Verfügung.

5.1 Änderungen am Generator-Modell

Die hinzugekommenen Funktionen erfordern Erweiterungen in den Klassen zur Modellierung der zu Grunde liegenden Datenbank. Das bisherige API in *STU* nutzt überladene Methoden mit vielen optionalen Parametern zur Beschreibung von Spalten. Dies ist weder wartungs- noch anwenderfreundlich. Jeder neue optionale Parameter würde die Anzahl der Methoden unter Umständen verdoppeln. Für Anwender ist es nicht immer einfach, sich die Reihenfolge langer Parameterlisten zu merken, v.a. wenn es mehrere Varianten der selben Methode gibt.

Die Klassen zur Beschreibung des Generator-Modells werden deshalb auf das Builder-Pattern umgestellt. Dieses löst beide Probleme. Jeder zusätzliche optionale Parameter führt zu einer neuen Methode. Die Reihenfolge der Methodenaufrufe ist im Gegensatz zu Parameterlisten beliebig. Der Code liest und schreibt sich einfacher, da Parameter durch die Methodenaufrude benannt sind.

Die neuen Builder-Klassen decken den Funktionsumfang des alten API ab. Dabei werden Eigenschaften für Spalten nicht mehr über ein `EnumSet` festgelegt, sondern über Methoden für die vordefinierten Flags. In Abschnitt 5.1.1 wird weiter auf das Thema Flags eingegangen. Darüber hinaus bieten die neuen Klassen die Möglichkeit, Beschreibungen zu Tabellen und Spalten hinzu zu fügen. Diese werden bei der Code-Generierung für die Erstellung von JavaDoc-Kommentaren verwendet (siehe Abschnitt 5.8).

Erweiterungen am Generator-Modell betreffen vor allem die Modellierung von Beziehungen zwischen Tabellen. Der folgende Abschnitt beschreibt die Modellierungskonzepte für Beziehungen in Datenbanken.

5.1.1 Spalten-Eigenschaften

Meta-Informationen von Spalten beinhalten neben dem Namen der Spalte und dem Typ weitere Eigenschaften. Diese Eigenschaften werden in *STU* mit Hilfe sogenannter Flags

beschrieben. Die Flags sind bislang in einem *Enum* zusammengefasst. Alle für eine Spalte gesetzten Flags müssen beim Hinzufügen einer Spalte über ein *EnumSet* übergeben werden. Bei dem neuen Builder-API werden die Flags über spezielle Methoden gesetzt.

Zu den in *STU* enthaltenen Standard-Spalten-Flags gehören:

- **Identifier**: Dieses Flag gibt an, dass die Werte einer Spalte die Zeile eindeutig identifizieren. Sollen Werte in einer Zeile abgefragt oder verändert werden, kann die Zeile mit Hilfe einer solchen Spalte identifiziert werden.

  Da die Werte zur Identifikation verwendet werden, ist ein nachträgliches Ändern nicht erlaubt. Dies soll anhand eines kurzen Beispiels begründet werden (siehe Listing 5.1). Es zeigt einen Ausschnitt einer Studenten-Tabelle. Die Spalten `id` und `matrikelnummer` sind mit dem Flag `Identifier` versehen. In Zeile 2 werden Daten mit der ID 1 und der Matrikelnummer 123456 definiert. Zeile 3 steht für beliebige Anweisungen, in Zeile 4 und 5 wird die Studententabelle erweitert, z.B. innerhalb eines Unit-Tests. Zeile 5 definiert Daten mit der ID 2 und der vorherigen Matrikelnummer. Beziehen sich beide Zeilen auf den selben Studenten und die ID soll verändert werden? Oder wurde die Matrikelnummer irrtümlich falsch angegeben? Die ID des Studenten Mustermann auf 2 zu ändern könnte zu Problemen führen, da nicht bekannt ist, an welchen Stellen bereits auf ID 1 Bezug genommen wird.

  ```
  1 id | matrikelnummer | Name
  2 1  | 123456         | "Mustermann"
  3 ...
  4 id | matrikelnummer | vorname
  5 2  | 123456         | "Nikolaus"
  ```

  Listing 5.1: Beispiel für unveränderliche Identifikatoren

  Das Flag wird über die Methode `identifier()` gesetzt, dabei wird das Flag `Immutable` implizit aktiviert.

- **Default Identifier**: Dieses Flag stellt eine Art Erweiterung für die das Flag `Identifier` dar. Die mit diesem Flag markierte Spalte wird für Foreign-Key-Beziehungen auf die Tabelle verwendet, sofern nicht explizit eine andere Spalte angegeben wird. Die Methode zum setzen des Flags ist `defaultIdentifier()`, die Flags `Identifier` und `Immuatable` werden automatisch aktiviert.

- **Add Next Method**: *STU* bietet die Möglichkeit, Werte-Generatoren zu verwenden um einen Spaltenwert manuell oder auch automatisch mit einem generierten Wert zu belegen. Aufgerufen wird der Generator über eine sogenannte Next-Value-Methode auf dem RowBuilder. Ihr Name setzt sich aus dem Präfix `next` und dem Spaltennamen zusammen. Der Generator erzeugt für die jeweilige Spalte allerdings nur dann eine Next-Value-Methode, wenn das entsprechende Flag über `addNextMethod()` aus dem Builder-API gesetzt wurde. Standardmäßig muss die Next-Value-Methode manuell aufgerufen werden, über ein Flag kann dies auch automatisch erfolgen.

- **Auto Invoke Next**: Ist dieses Flag aktiviert, wird die Next-Value-Methode beim Anlegen einer neuen Tabellenzeile automatisch aufgerufen. Beim Setzen des Flags über die Builder-Methode `autoInvokeNext()` wird automatisch auch das Flag zum Generieren der Next-Value-Methode gesetzt.

- **Immutable**: Ist dieses Flag gesetzt, kann ein Wert in einer Spalte nur ein Mal gesetzt werden, und danach nicht mehr verändert werden. Wenn das Flag zum automatischen Aufruf der Next-Value-Methode aktiviert ist, kann der automatisch erzeugte Wert allerdings überschrieben werden. Die Methode zum Aktivieren des Flags heißt `immutable()`.

5.1.2 Modellierung von Relationen über Builder-Klassen

Das API zur Modellierung des Datenbankschemas als Grundlage für den Generator stellt eine der größten Veränderungen in der Implementierung dar. Die größten Änderungen betreffen die Definition von Beziehungen zwischen Tabellen. Über `reference` kann der Builder zur Beschreibung der Relation aufgerufen werden. Die Beschreibung findet in zwei Bereichen statt:

- **local**: Der als `local` bezeichnete Teil beschreibt die Beziehung aus Sicht der Tabelle, in der sich die Spalte befindet.
- **foreign**: Der `foreign`-Teil dient der Beschreibung der Beziehung aus Sicht der Tabelle, mit der die Beziehung hergestellt wird.

In beiden Bereichen kann ein Bezeichner angegeben werden. Dieser Bezeichner drückt die Beziehung in die jeweilige Richtung aus, in *local* wird die Beziehung in Richtung *foreign* bezeichnet. Diese Bezeichner werden für die Methoden zur Modellierung der Beziehungen verwendet. Daneben können auch noch Beschreibungstexte angegeben werden, die für die JavaDoc genutzt werden.

In Abschnitt 5.1.3 befindet sich ein Beispiel für die Modellierung von Relationen.

Durch die Nutzung des Builder-Patterns lassen sich weitere Attribute verhältnismäßig einfach hinzufügen, z.B. für die Generierung der Testdaten (siehe Kapitel 6).

5.1.3 Alte und neue Builder-Klassen im Vergleich

Die Vorteile der Umstellung auf das Builder-Pattern sollen die beiden folgenden Listings zeigen. Sie zeigen die Modellierung der Datenbank für den Generator. Der Übersicht halber wurde der Code auf die Anweisungen im Konstruktor der Modell-Klasse und die Definition von zwei Tabellen reduziert. Listing 5.2 zeigt die Modellierung im ursprünglichen *STU*, während Listing 5.3 die neuen Builder veranschaulicht.

```
1  database("Hochschule");
2  packageName("com.seitenbau.testing.dbunit.hochschule");
3
4  Table professoren = addTable("professor")
5      .addColumn("id", DataType.BIGINT, Flags.AutoInvokeNextIdMethod)
6      .addColumn("name", DataType.VARCHAR)
7      .addColumn("vorname", DataType.VARCHAR)
8      .addColumn("titel", DataType.VARCHAR)
9      .addColumn("fakultaet", DataType.VARCHAR);
10
11 Table lehrveranstaltungen = addTable("lehrveranstaltung")
12     .addColumn("id", DataType.BIGINT, Flags.AutoInvokeNextIdMethod)
13     .addColumn("professor_id", DataType.BIGINT, professoren.ref("id"))
14     .addColumn("name", DataType.VARCHAR)
15     .addColumn("sws", DataType.INTEGER)
16     .addColumn("ects", DataType.DOUBLE);
```

Listing 5.2: Beispiel alte *STU*-Builder

In diesem Beispiel – inkl. der nicht dargestellten Tabellen-Definitionen – werden lediglich drei der insgesamt neun `addColumn`-Methoden verwendet.

Die Codes zur Modellierung mit der alten und der neuen API ähnelt sich, die Unterschiede liegen abgesehen von den Flags und Relationen eher im Detail. Listing 5.3 zeigt die Modellierung der selben Tabellen mit dem neuen API. Die kürzeren Parameterlisten und die zusätzlichen Funktionen führen dazu, dass die selben Modelle in *STU* einige Zeilen länger werden. Dieser Nachteil wird durch die gewonnene Ausdrucksstäke mehr als ausgeglichen.

```
database("Hochschule");
packageName("com.seitenbau.testing.dbunit.hochschule");

Table professoren = table("professor")
    .description("Die Tabelle mit den Professoren der Hochschule")
    .column("id", DataType.BIGINT)
      .identifierColumn()
      .autoInvokeNext()
    .column("name", DataType.VARCHAR)
    .column("vorname", DataType.VARCHAR)
    .column("titel", DataType.VARCHAR)
    .column("fakultaet", DataType.VARCHAR)
  .build();

Table lehrveranstaltungen = table("lehrveranstaltung")
    .description("Die Tabelle mit den Lehrveranstaltungen der Hochschule")
    .column("id", DataType.BIGINT)
      .identifierColumn()
      .autoInvokeNext()
    .column("professor_id", DataType.BIGINT)
      .reference
        .local
          .name("geleitetVon")
          .description("Gibt an, von welchem Professor eine
              Lehrveranstaltung geleitet wird.")
        .foreign(professoren)
          .name("leitet")
          .description("Gibt an, welche Lehrveranstaltungen ein Professor
              leitet.")
    .column("name", DataType.VARCHAR)
    .column("sws", DataType.INTEGER)
    .column("ects", DataType.DOUBLE)
  .build();
```

Listing 5.3: Beispiel neue *STU*-Builder

Bei der Modellierung von n:m-Beziehungen kann auf den `local`-Teil der Beziehung verzichtet werden. *STU* verwendet automatisch den `foreign`-Teil der assoziierten Spalte. Anstelle der Methode `table` wird eine assoziative Tabelle mit der Methode `associativeTable` beschrieben. Listing 5.4 zeigt ein Beispiel für die Modellierung einer assoziativen Tabelle:

```
associativeTable("besucht")
  .column("student_id", DataType.BIGINT)
    .reference
      .foreign(studenten)
        .name("besucht")
        .description("Die Lehrveranstaltungen, die ein Student besucht.")
  .column("lehrveranstaltung_id", DataType.BIGINT)
    .reference
      .foreign(lehrveranstaltungen)
        .name("besuchtVon")
        .description("Die Studenten, die eine Lehrveranstaltung besuchen.")
 .build();
```

Listing 5.4: Beispiel für assoziative Tabelle

5.2 Neue DataSet-Builder-Klassen

Für die tabellarisch definierten DataSets wird eine neue Builder-Klasse generiert, die über Komposition und Delegation die bisherige, auf dem Fluent-Builder-API-basierende DataSet-Klasse nutzt.

Der Großteil des IDE-Supports wird über Adapter-Klassen für die bisherigen Tabellen realisiert. Zu jeder Tabellen-Klasse wird eine zusätzliche Adapter-Klasse generiert. Dort sind die Tabellen-spezifischen Spaltenbezeichner für die tabellarische DSL definiert. Die Methode `rows` startet das Parsen der Tabellenzeilen, die wie im Entwurf als Closure übergeben werden. Innerhalb dieses Closures sind die in der Tabelle definierten Bezeichner nutzbar. Neben den Spaltenbezeichnern wird auch ein Spaltenbezeichner `REF` und auch der Platzhalter (Unterstrich) generiert. Die Adapter nutzen intern eine aggregierte Tabellen-Klasse. Dabei bildet der Adapter die Schnittstelle der Tabellen-Klasse nach und delegiert die Aufrufe.

Jeder Spaltenbezeichner stellt eine anonyme Klasse dar, die die abstrakte Klasse `ColumnBinding` erweitert. Diese enthält unter anderem Meta-Informationen zu der zugehörigen Spalte auch Methoden, die das Parsen der Tabellen erleichert (siehe Abschnitt 5.3).

Für jede Tabelle gibt es in der Builder-Klasse eine öffentliche Instanz der Adapter-Klasse. Auf diese Weise wird der IDE-Support bzgl. der Tabellennamen sichergestellt. Das Klassendiagramm ist Abbildung 5.2 dargestellt.

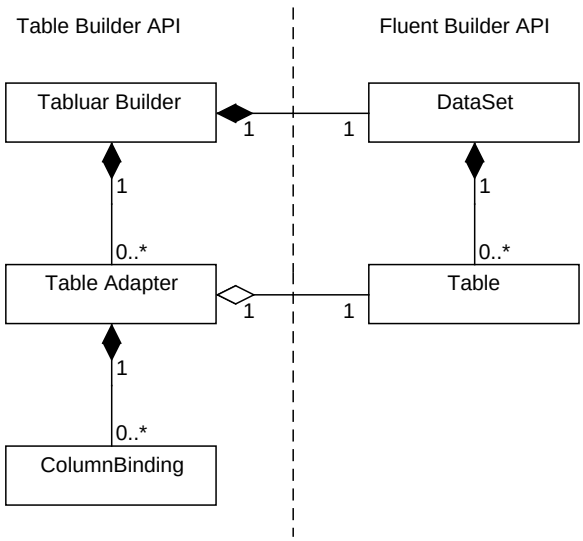

Abbildung 5.2: Klassendiagramm der DataSet-Builder

5.3 Tabellenparser

Der Code zum Parsen der Tabellen-Closures basiert auf dem dem in Abschnitt 4.2.1 gezeigten Entwurf. Die Logik an sich ist relativ generisch, je nach konkreter Tabelle muss allerdings mit unterschiedliche Datentypen gearbeitet werden.

Folgende zwei Möglichkeiten bieten sich an, den Parser zu realisieren:

1. Für jede Tabellen-Adapter-Klasse wird individueller Parser-Code generiert.
2. Die Tabellen-Adapter nutzen eine generische Parser-Klasse.

Der Nachteil, redundanten Code zu generieren, mag gering erscheinen. Gerade bei generiertem Code wird redundanter Code weniger kritisch gesehen. Allerdings erreicht der Code zum Parsen einer Tabelle eine gewisse Komplexität, die die Pflege des Codes auf Template-Ebene erschwert. Die generische Klasse muss zwar einige Hürden überwinden, hat aber einige Vorteile: Die Wartung erfolgt IDE-unterstützt und Änderungen erfordern in der Regel keine Neu-Generierung der Tabellen-Klassen. Aus Architektur-Sicht ist der größte Vorteil jedoch, dass keine der zu generierenden Klassen spezielle Groovy-Features nutzen muss und es damit ausreicht, Java-Klassen zu verwenden.

Die Schwierigkeiten, die mit der Entscheidung zugunsten des generischen Parsers gelöst werden müssen, betreffen Operationen, die der Parser auf der Tabelle durchführen muss: Anlegen neuer Zeilen, Suchen nach Zeilen und Setzen von Werten auf den Zeilen. Dies wird unter anderem mit einem weiteren Adapter zwischen den bereits bekannten Table-Adapter-Klassen und dem generischen Table-Parser erreicht. Dieser Adapter implementiert das generische Interface `TableParserAdapter`. Die generischen Typ-Parameter enthalten Informationen zu den konkret verwendeten Klassen wie dem RowBuilder. Darüber hinaus bietet es die benötigten Methoden zum Erstellen und Suchen von Tabellen-Zeilen.

Das Setzen der Werte auf den RowBuildern ist deshalb ein Problem, weil die Bezeichner der Set-Methoden die Spaltennamen enthalten. Eine Lösung sind die bereits im letzten Abschnitt angesprochenen `ColumnBinding`-Klasse. Sie definiert die abstrakte generische Methode `set(R row, Object value)`, wobei `R` der Typ-Parameter für den RowBuilder ist. In die Implementierungen der `set`-Methoden kann der korrekte Bezeichner für den jeweiligen Setter auf dem RowBuilder generiert werden.

Mit automatischen Typumwandlungen bietet `STU` eine Komfortfunktion, die die Lesbarkeit weiter verbessert. So versucht `STU`, Werte dynamisch beim Parsen in den vom Modell erwarteten Datentyp zu bringen. Im Beispiel ist der Spalte `ects` in der Lehrveranstaltungstabelle der Datentyp `Double` zugeordnet. Die in der Spalte auftauchenden `Integer`-Werte werden automatisch in `Double`-Werte umgewandelt. Da der Parser nicht mit primitiven Datentypen sondern nur auf Objekten arbeitet, geht die Umwandlung über einen einfachen Type-Cast hinaus.

5.4 Referenzen und Scopes

Neben der Möglichkeit, Daten tabellarisch zu modellieren, gehören die neuen Referenz-Datentypen zu der wichtigsten Erweiterung. In *STU* ist eine Referenz eine Art Stellvertreter für eine Entität (Tabellenzeile). Die Referenz kann bei der Modellierung oder auch

bei Such-Anfragen anstelle konkreter Werte (wie Primärschlüssel) verwendet werden. Die Such-Anfrage-Möglichkeiten werden in Abschnitt 5.7 erläutert.

Referenzen müssen an ihre Datensätze gebunden werden. Im Table Builder API ist dafür die Spalte `REF` vorgesehen, die in jeder Tabelle genutzt werden kann, das Fluent Builder API bietet auf den RowBuilder-Klassen die Methode `bind()`. Die Listings 5.5 und 5.6 zeigen die Modellierung der selben Zeile einmal mit dem neuen Table Builder API und einmal mit dem erweiterteren Fluent Builder API.

```
professorTable.rows {
  REF    | name    | vorname  | titel            | fakultaet
  WAESCH | "Wäsch" | "Jürgen" | "Prof. Dr.-Ing." | "Informatik"
  ...
}
```

Listing 5.5: Binden von Referenzen (Table Builder API)

```
table_Professor.insertRow()
  .bind(WAESCH)
  .setName("Wäsch")
  .setVorname("Jürgen")
  .setTitle("Prof. Dr.-Ing.")
  .setFakultaet("Informatik")
...
```

Listing 5.6: Binden von Referenzen (Fluent Builder API)

Da Referenzen die zugehörigen RowBuilder kennen, können ihre Werte auch direkt auf der Referenz abgefragt werden (siehe Listing 5.7).

```
WAESCH.getName()     // Java style
WAESCH.name          // Groovy style
```

Listing 5.7: Zugriff auf Werte über Referenzen

Darüber hinaus können über Referenzen Beziehungen modelliert werden. Sie enthalten Methoden zum Ausdrücken von Beziehungen. Die Methodennamen entsprechen den im Generator-Modell angegebenen Relationsnamen. Listing 5.8 zeigt ein Beispiel,wie die Relation zwischen einem Professor und einer Prüfung modellieren lässt.

```
WAESCH.beaufsichtigt(P_VSYS)
```

Listing 5.8: Definition von Beziehungen über Referenzen

Die Referenzen müssen vor ihrer Nutzung definiert (also deklariert und instantiiert) werden. Zwar könnten in Groovy auch nicht explizit definierte Referenzen verwendet werden, allerdings würde Tool-Unterstützung verloren gehen (z.B. beim Umbenennen von Referenzen, Erkennen von Tippfehlern bei Bezeichnern). Außerdem könnten sie auch nicht im normalen Java-Code verwendet werden. Es bietet sich an, sie als globale Variablen zu definieren. Verschiedene DataSets (mit dem selben Datenbank-Modell) können die selben Referenzen nutzen, auch wenn sie unterschiedliche Werte repräsentieren.

Damit die selben Referenzen in unterschiedlichen DataSets genutzt werden können, werden die RowBuilder immer im Kontext des gerade aktiven DataSets gebunden. Das aktive DataSet wird über die `DataSetRegistry` festgelegt (und abgefragt). Pro Datenbank-Modell ist immer ein (oder kein) DataSet aktiv. Das heißt, dass wenn verschiedene Datenbank-Modelle genutzt werden, aus jedem Modell jeweils ein DataSet gleichzeitig aktiv sein kann.

5.5 Nutzung des DataSets in Unit-Tests

Wie das Beispiel-DataSet aus Listing 4.10 in einem JUnit-Test verwendet werden kann, zeigt Listing 5.9. Das System Under Test (siehe Abschnitt 2.2) ist ein Spring-Service, der von der Variable `sut` (Zeile 20) repräsentiert wird.

Das in einem Test verwendete DataSet kann als Klasse über die Annotation `DatabaseSetup` konfiguriert werden (Zeile 26). Sie sorgt dafür, dass die angegebene DataSet-Klasse instantiiert und der Variable zugewiesen wird, die mit der Annotation `InjectDataSet` markiert wurrde (Zeilen 22 und 23). Außerdem wird dieses DataSet auch bei der `DataSetRegistry` als aktives DataSet registriert und die Daten in die Datenbank eingespielt. Dadurch kommen die Test-Methoden ohne Verwaltungsaufgaben aus. Der Test `removeStudent` testet, ob das System die richtigen Änderungen in der Datenbank vornimmt, wenn der Student `MUSTERMANN` entfernt wird. Da dem Service die zu löschende Entität übergeben werden muss (Zeile 38), wird in den Zeilen 29 bis 35 eine entsprechende Instanz erstellt und konfiguriert.

Der Test verwendet eine `DatabaseTesterRule` (Zeile 9), die unter anderem für die Vergleiche der Datenbank mit den DataSets verantwortlich ist. Dazu muss ihr die Datenbank bekannt sein, die in Form einer `DataSource` vorliegt (Zeile 6). Da dieses Feld durch *Depenency Injection* ([HM05, S. 4]) erst nach der Instantiierung der Klasse belegt wird, kann bei der Erzeugung von `dbTester` der Wert noch nicht verwendet werden. Dies wird durch die Verwendung eines Future-Objekts gelöst, das die `DataSource` erst dann zurückliefert, wenn sie gebraucht wird (Zeilen 10 bis 15).

```
1  @RunWith(SpringJUnit4ClassRunner.class)
2  @ContextConfiguration(classes=HochschuleContext.class)
3  public class HochschuleDataSetDatabaseTest {
4
5    @Autowired
6    DataSource dataSource;
7
8    @Rule
9    public DatabaseTesterRule dbTester =
10     new DatabaseTesterRule(new Future<DataSource>(){
11       @Override
12       public DataSource getFuture()
13       {
14         return dataSource;
15       }
16     }).addCleanAction(new ApacheDerbySequenceReset()
17       .autoDerivateFromTablename("_SEQ"));
18
19   @Autowired
20   HochschuleService sut;
21
22   @InjectDataSet
23   HochschuleBuilder dataSet;
24
25   @Test
26   @DatabaseSetup(prepare = HochschuleDataSet.class)
27   public void removeStudent() throws Exception {
28     // prepare
29     Student student = new Student();
30     student.setMatrikelnummer(MUSTERMANN.getMatrikelnummer());
31     student.setVorname(MUSTERMANN.getVorname());
32     student.setName(MUSTERMANN.getName());
33     student.setStudiengang(MUSTERMANN.getStudiengang());
34     student.setSemester(MUSTERMANN.getSemester());
35     student.setImmatrikuliertSeit(MUSTERMANN.getImmatrikuliertSeit());
36
37     // execute
38     sut.removeStudent(student);
39
40     // verify
41     dataSet.studentTable.deleteRow(MUSTERMANN);
```

```
42      dataSet.besuchtTable.deleteAllAssociations(MUSTERMANN);
43
44      dbTester.assertDataBase(dataSet);
45    }
46
47    ...
48
49  }
```

Listing 5.9: JUnit-Tests (reiner Java-Code)

In den Zeilen 41 und 42 werden die erwarteten Änderungen im DataSet ebenfalls durchgeführt, um in Zeile 44 die Datenbank gegen das DataSet zu vergleichen.

Die neue DSL kann in Groovy-basierten Tests verwendet werden. Listing 5.10 zeigt beispielhaft eine entsprechende Test-Methode. In diesem Test wird eine neue Lehrveranstaltung erstellt und einem Professor zugeordnet.

```
1   @Test
2   @DatabaseSetup(prepare = HochschuleDataSet)
3   def addLehrveranstaltung() {
4     // prepare
5     Lehrveranstaltung lv = new Lehrveranstaltung()
6     lv.setName("Programmieren")
7     lv.setProfessor(HAASE.id)
8     lv.setSws(4)
9     lv.setEcts(6.0)
10
11    // execute
12    def addedLv = sut.addLehrveranstaltung(lv)
13
14    // verify
15    dataSet.lehrveranstaltungTable.rows {
16      id          | professor | name             | sws | ects
17      addedLv.id | HAASE     | "Programmieren" | 4   | 6.0
18    }
19
20    dbTester.assertDataBase(dataSet)
21  }
```

Listing 5.10: Test-Methode in Groovy

Sicherheitshalber wird die vom Spring-Service erzeugte ID verwendet, um die Änderungen am Test-DataSet durchzuführen. Auf diese Weise bleibt der Test stabil, auch wenn sich das Verhalten des Services bezüglich der ID-Generierung ändern sollte.

5.6 Komposition von DataSets

DataSets lassen sich für Tests auch aus anderen zusammensetzen. Dieses Feature setzt nicht auf Konzepte der Objektorientierung wie Vererbung. Vererbung würde zu mehr syntaktischem Ballast führen, da die Methoden `tables` und `relations` explizit die Methoden aus der Super-Klasse aufrufen müssten.

Der realisierte Mechanismus sieht vor, dass DataSets andere DataSet-Klassen als Basis verwenden können. Gibt es ein Basis-Datenset, kann die Methode `extendsDataSet` so überschrieben werden, dass sie die Klasse des Basis-DataSets zurückliefert. Analog dazu gibt es die Methode `extendsDataSets`, falls es mehrere Basis-DataSets gibt. Diese muss eine Liste von DataSet-Klassen zurückliefern. Listing 5.11 zeigt, wie ein DataSet ein anderes als Basis verwendet.

```
1   class ExtendedHochschuleDataSet extends HochschuleBuilder {
2
3     def extendsDataSet() { HochschuleDataSet }
```

```
4
5   def tables() {
6
7    lehrveranstaltungTable.rows {
8       REF        | id   | name                  | sws | ects
9       PROGR      | 3    | "Programmieren"       | 4   | 6.0
10    }
11
12  }
13
14  def relations() {
15    HAASE.leitet(PROGR)
16  }
17
18 }
```

Listing 5.11: Erweitertes DataSet

Die Syntax für die Komposition aus den drei DataSet-Klassen `DataSet1`, `DataSet2` und `DataSet3` ist in Listing 5.12 dargestellt:

```
1   def extendsDataSets() { [ DataSet1, DataSet2, DataSet3 ] }
```

Listing 5.12: Erweitertes DataSet

Das erweiterte DataSet kann in denselben Unit-Tests verwendet werden. Dabei reicht es aus, die Annotation `DatabaseSetup` entsprechend anzupassen (siehe Listing 5.13).

```
1   @Test
2   @DatabaseSetup(prepare = ExtendedHochschuleDataSet)
3   public void assignedLehrveranstaltungen() throws Exception {
4     // prepare
5     Professor haase = new Professor();
6     haase.setId(HAASE.id);
7
8     // execute
9     List<Lehrveranstaltung> items = sut.findLehrveranstaltungen(haase);
10
11    // verify
12    def findWhere = dataSet.lehrveranstaltungTable.findWhere
13    int count = findWhere.professorId(HAASE).rowCount
14    assertThat(items).hasSize(count);
15  }
```

Listing 5.13: Test auf erweiterem DataSet

5.7 Erweiterungen in generierter API

Die meisten Erweiterungen an der Fluent-Builder-API-Schicht betreffen die Möglichkeit, Ref-Typen statt konkreter Werte zu verwenden. Dazu gehören unter anderem:

- **RowBuilder**: Die Erweiterungen der RowBuilder betreffen vor allem die verbesserten Möglichkeiten Relationen auszudrücken. So gibt es für Spalten, die eine Relation zu einer anderen Spalte enthalten, nun neben einem Setter für den konkreten Wert (z.B. des Fremdschlüssels) einen Setter zum Setzen des entsprechenden Ref-Typs.

  Anstelle des von der Ref repräsentierten Wertes wird die Ref selbst im RowBuilder abgespeichert. Das hat zwei Vorteile:

  1. **Reihenfolge**: Die Modellierung der Daten ist in diesem Fall keiner strengen Reihenfolge unterworfen. Es ist egal, ob die Zeile, auf die Bezug genommen wird, überhaupt schon initialisiert wurde.

2. **Konsistenz**: Die Werte werden nicht redundant gespeichert. Wird der Wert an einer Stelle geändert, ist dieser Wert unmittelbar im gesamten DataSet so sichtbar.

- **Future Values**: Eine der wenigen Erweiterungen, die nicht auf die Einführung der Ref-Typen zurückzuführen sind, sind Future Values. Dabei handelt es sich um Werte, die erst beim Abfragen ausgewertet werden. Dies kann nützlich sein, wenn sich Werte abhängig von anderen Daten ändern. Listing 5.14 zeigt ein Beispiel, in der die Lehrveranstaltungstabelle um eine Spalte erweitert wurde. Diese Spalte soll die Anzahl der Tutoren aufnehmen, die die Lehrveranstaltung betreuen.

```
class HochschuleDataSet extends HochschuleBuilder
{

  def tables() {

    lehrveranstaltungTable.rows {
      REF        | name                | sws | ects | tutoren
      VSYS       | "Verteilte Systeme" | 4   | 5    | tutors(VSYS)
      DPATTERNS  | "Design Patterns"   | 4   | 3    | tutors(DPATTERNS)
    }

    ...
  }

  ...

  // returns a Closure which is treated as future value
  def tutors(LehrveranstaltungRef ref) {
    return {
      def rows = isttutorTable.quietFindWhere.lehrveranstaltungId(ref)
      return rows.rowCount
    }
  }
}
```

Listing 5.14: Beispiel Lazy Valunes

Durch die Nutzung von Future Values enthält die Tabelle immer die korrekte Anzahl, ohne dass beim Modellieren der Tutoren-Beziehungen Anpassungen notwendig wurden. Da die Verwendung von Future Values den Code etwas aufbläht, interpretiert `STU` Groovy Closures in Tabellen automatisch als Future Values. Die Methode `tutors()` liefert ein solches Closure zurück.

- **findWhere**: Das bisherige API sah Suchen von Zeilen in einer Tabelle ausschließlich über konkrete Werte vor. Die Erweiterung ermöglicht es, dass Ref-Typen statt konkreter Werte verwendet werden können. Werden beispielsweise in der Professor-Tabelle alle Professoren mit einem bestimmten Vornamen gesucht und als Such-Wert eine Professor-Referenz übergeben, werden alle Professoren mit diesem Vornamen gesucht. Listing 5.15 zeigt zwei Such-Anfragen, die beide auf den Beispieldaten das selbe Ergebnis liefern.

```
dataSet.table_Professor.findWhere.vorname("Oliver");
dataSet.table_Professor.findWhere.vorname(HAASE);
```

Listing 5.15: Such-Beispiele

- **quietFindWhere**: In manchen Fällen kann es sinnvoll sein, bei einer Suche ohne Treffer keine Ausnahme auszulösen. Ein Beispiel dafür ist das Closure in Listing 5.14. Eine Lehrveranstaltung ohne Tutoren kann in diesem Beispiel normal sein.

- **getWhere**: Wenn davon auszugehen ist, dass eine Such-Anfrage genau eine Zeile als Ergebnis liefert, kann `getWhere` verwendet werden. Im Gegensatz zu `findWhere`

liefert es das Ergebnis nicht in Form einer Liste, sondern als `Optional`-Wert zurück [Com13]. Gibt es auf eine Suchanfrage mehr als einen Treffer, wird eine Exception ausgelöst.

- **find**: Sind die einfachen Such-Anfragen über `findWhere` bzw. `getWhere` nicht mächtig genug, können mit Hilfe von `find` Filter-basierte Suchen durchgeführt werden. In Listing 5.16 wird ein Filter gezeigt, der alle Professoren findet, deren Vorname die Länge sechs hat.

```
Filter<RowBuilder_Professor> FILTER =
  new Filter<RowBuilder_Professor>()
    {
      @Override
      public boolean accept(RowBuilder_Professor value)
      {
        return value.getVorname().length() == 6;
      }
    };

RowCollection_Professor profs = dataSet.professorTable.find(FILTER);
```

Listing 5.16: Beispiel für find

In Groovy können auch direkt Closures übergeben werden, die als Argument einen entsprechenden RowBuilder übergeben bekommen.

- **foreach**: Ein Zugriff auf die einzelnen Zeilen in einer Tabelle kann innerhalb eines Tests sinnvoll bzw. notwendig sein. Neben dem Zugriff auf eine Liste von RowBuildern ist es auch möglich, mit Hilfe der Methode `foreach` über die Zeilen zu iterieren. Listing 5.17 zeigt ein kurzes Java-Beispiel. In Groovy kann der Methode auch ein Closure übergeben werden, das den entsprechenden RowBuilder als Parameter übergeben bekommt.

```
Action<RowBuilder_Professor> ACTION =
  new Action<RowBuilder_Professor>()
    {
      @Override
      public void call(RowBuilder_Professor value)
      {
        System.out.println("Professor: " + value.getName());
      }
    };

dataSet.professorTable.foreach(ACTION);
```

Listing 5.17: Beispiel für foreach

5.8 JavaDoc

Zum guten IDE-Support gehört auch, dass der Tester beim Erstellen der Tests durch aussagekräftige JavaDoc unterstützt wird. Der Generator erzeugt für das DataSet, für die Tabellen und für die Referenz-Typen JavaDoc, das neben einer reinen Beschreibung auch Beispiel-Quellcodes für die Nutzung enthält.

Die in der JavaDoc enthaltenen Beispiel-Daten werden auf sehr einfache Art generiert, für jeden Java-Datentyp gibt es einen Beispielwert. Sie sollen mit Hilfe der Erkenntnisse bezüglich der Generierung von Testdaten verbessert werden.

Einige der Beispiel-Quellcodes werden über Unit-Tests überprüft. Dazu gehören die Builder-Klassen zur Beschreibung des Datenbank-Modells. Auf diese Weise soll sichergestellt werden, dass Änderungen am API auch auf die JavaDoc übertragen werden.

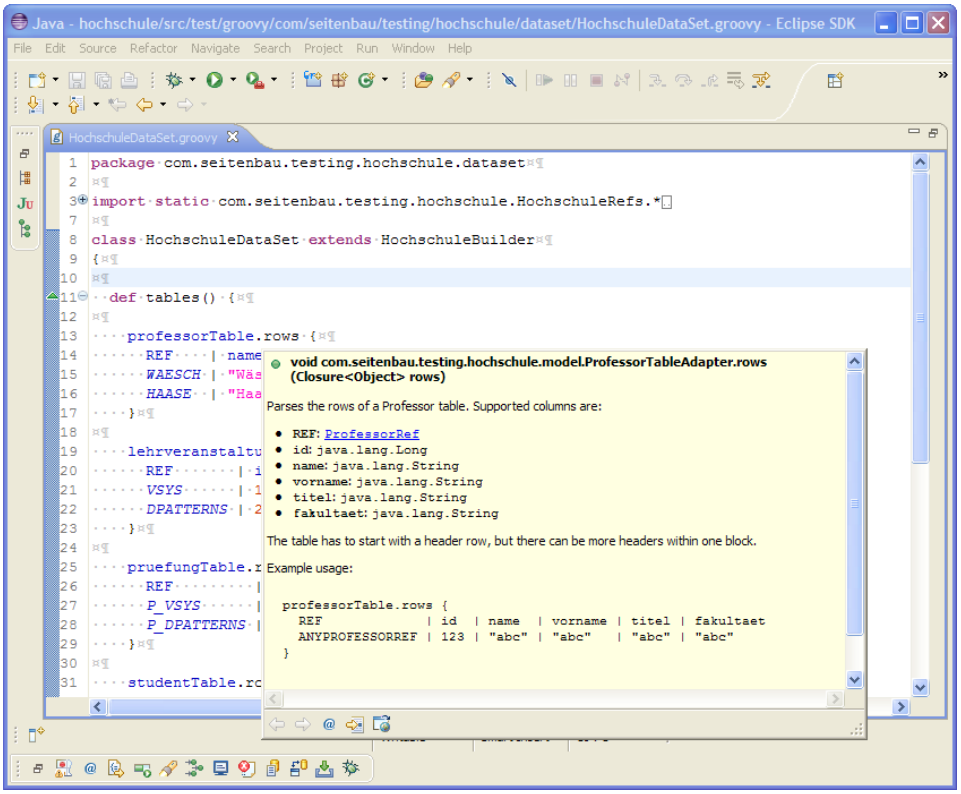

Abbildung 5.3: Beispiel JavaDoc-Tooltip

5.9 Verhalten bei Fehlern in den Tabellendefinitionen

Selbst eine übersichtliche Darstellung von Tabellendaten schützt nicht vor Fehleingaben. Viele Fehler lassen sich mit Hilfe statischer Analysen erkennen. So werden ungültige Tabellen- und Spaltennamen vom Compiler entdeckt.

Fehler in der eigentlichen Tabellenstruktur, z.B. eine abweichende Anzahl von Spalten, kann der Standard-Compiler nicht erkennen, genauso wie ungültige Werte bzw. ungültige Typen. Solche Fehler werden in der gegenwärtigen Implementierung zur Laufzeit erkannt und führen zum Scheitern der Tests. Dazu wirft der Tabellen-Parser eine Exception der Klasse `TableParserException`. Wenn ein falscher Typ verwendet wird, könnte die Meldung der Exception so aussehen: *Cannot set value <5> of type java.lang.Integer, expected class java.lang.String in [TableRowModel: <JobsRef> | 5 | "Creating software"]*

Um die Lokalisation der fehlerhaften Stelle zu erleichtern, wird der Stack-Trace der geworfenen Exception angepasst. Die Ursache dafür liegt in der Arbeitsweise des Tabellen-Parsers: Der Parser arbeitet zeilenweise, d.h. er liest immer eine Zeile vollständig ein und interpretiert die Daten erst im Anschluss - wenn die Ausführung der Zeile abgeschlossen ist. Kommt es zu einem Fehler, befindet sich das Programm aber nicht mehr in der Fehler-verursachenden Zeile. Deshalb wird beim Parsen bei jedem Tabellen-Element der Stack-Trace analysiert und das Stack-Trace-Element bestimmt, das zu der Tabellenzeile gehört. Sollte es beim Setzen der Werte einen Fehler geben, wird dieses Element als erstes Element des Stack-Traces hinzugefügt.

5.10 Nicht umgesetzt

Der folgende Abschnitt soll einen kurzen Überblick über nicht umgesetzte Funktionen geben. Außerdem wird begründet, warum diese Funktion nicht in *STU* implementiert ist.

5.10.1 Zusammengesetzte Schlüssel

Zusammengesetzte Schlüssel werden in *STU* nicht direkt unterstützt und müssen nach wie vor komplett manuell realisiert werden. Dazu muss für jeden Teilschlüssel eine Spalte im Datenbank-Modell angelegt werden. So lange zum Zugriff auf Tabellenzeilen die Referenz-Typen verwendet werden, stellt dies kein spürbarer Nachteil dar. Sollte die Zeile in Abhängigkeit ihres Schlüssels dynamisch gesucht werden, kann auf die neue `find`-Methode zurückgegriffen werden.

5.10.2 Unterstützung für weitere Beziehungstypen

Folgende Beziehungstypen müssen manuell umgesetzt werden.

- **Reflexive Beziehungen** Eine reflexive Beziehung kann in *STU* nur manuell ausgedrückt werden. Die Definition einer einfachen Tabelle für einen Baum, der aus einzelnen Konten besteht, könnte wie folgt aussehen (siehe Listing 5.18). Ein Knoten-Element kennt den zugehörigen Eltern-Knoten (Zeile 5). Eine Referenz auf die Tabelle ist an dieser Stelle nicht möglich, die Relation muss manuell ohne besondere Tool-Unterstützung durch *STU* realisiert werden.

```
1 Table knoten = table("knoten")
2     .column("id", DataType.BIGINT)
3       .defaultIdentifier()
4     .column("name", DataType.VARCHAR)
5     .column("parent", DataType.BIGINT)
6   .build();
```

Listing 5.18: Reflexive Beziehungen manuell

Die Konsequenz ist, dass die Beziehungen nicht typsicher über die Referenz-Klasse modelliert werden können, sondern die Primär- und Fremdschlüssel manuell im DataSet gepflegt werden müssen.

Ein anderer Ansatz stellt das Refaktorisieren der Datenbank und der beteiligten Systeme dar. Dies ist leider nicht immer möglich. Die Refaktorisierung sieht eine assoziative Tabelle für die Modellierung der Beziehung vor (siehe Listing 5.19).

```
1  Table knoten = table("knoten")
2      .column("id", DataType.BIGINT)
3        .defaultIdentifier()
4      .column("name", DataType.VARCHAR)
5    .build();
6
7  associativeTable("parents")
8      .column("parent", DataType.BIGINT)
9        .reference
10         .foreign(knoten)
11     .column("child", DataType.BIGINT)
12       .reference
13         .foreign(knoten)
14   .build();
```

Listing 5.19: Reflexive Beziehungen mit Hilfe assoziativer Tabelle

- **Zirkuläre Beziehungen** Reflexive Beziehungen stellen eine besondere Form von zirkulären Beziehungen dar. Die Probleme sind relativ ähnlich. Als kleines Modell dient eine Veranstaltungsplanung, bei der es Organisatoren und Teilnehmer geben kann. Eine Veranstaltung wird von einer Person organisiert, eine Person kann an einer Veranstaltung teilnehmen. Listing 5.20 zeigt eine Realisierung dieses Modells.

```
Table event = table("event")
    .column("id", DataType.BIGINT)
      .defaultIdentifier()
    .column("name", DataType.VARCHAR)
    .column("organizer", DataType.BIGINT)  // a person
  .build();

Table person = table("person")
    .column("id", DataType.BIGINT)
      .defaultIdentifier()
    .column("name", DataType.VARCHAR)
    .column("participates", DataType.BIGINT)
      .reference
        .foreign(event)
  .build();
```

Listing 5.20: Zirkuläre Beziehungen manuell

Das selbe Modell lässt sich – aus Sicht von *STU* – etwas typsicherer umsetzen. Für die Teilnahme wird eine assoziative Tabelle verwendet. Darüber hinaus kann hier auch eine Person an mehreren Veranstaltungen teilnehmen. Listing 5.21 zeigt die Realisierung einer zirkulären Beziehung mit Hilfe einer assoziativen Tabelle.

```
Table person = table("person")
    .column("id", DataType.BIGINT)
      .defaultIdentifier()
    .column("name", DataType.VARCHAR)
  .build();

Table event = table("event")
    .column("id", DataType.BIGINT)
      .defaultIdentifier()
    .column("name", DataType.VARCHAR)
    .column("organizer", DataType.BIGINT)
  .build();

associativeTable("parcipations")
    .column("event", DataType.BIGINT)
      .reference
        .foreign(event)
    .column("participant", DataType.BIGINT)
      .reference
        .foreign(person)
  .build();
```

Listing 5.21: Zirkuläre Beziehungen mit assoziativer Tabelle

- **Ternäre und andere höhergradigen Beziehungen** *STU* sieht keine spezielle Unterstützung für ternäre oder andere höhergradigen Beziehungen vor. Es bietet sich an, solche Beziehungen über eine zusätzliche Tabelle zu realisieren (siehe Listing 5.22, vergleichbar mit assoziativen Beziehungen für die Modellierung von n:m-Beziehungen.

```
Table student = table("student")
    .column("id", DataType.BIGINT)
      .defaultIdentifier()
    .column("name", DataType.VARCHAR)
  .build();

Table professor = table("professor")
    .column("id", DataType.BIGINT)
      .defaultIdentifier()
    .column("name", DataType.VARCHAR)
```

```
11    .build();
12
13  Table pruefung = table("pruefung")
14      .column("id", DataType.BIGINT)
15        .defaultIdentifier()
16      .column("name", DataType.VARCHAR)
17    .build();
18
19  table("relation")
20      .column("student_id", DataType.BIGINT)
21        .reference
22          .foreign(student)
23      .column("professor_id", DataType.BIGINT)
24        .reference
25          .foreign(professor)
26      .column("pruefung_id", DataType.BIGINT)
27        .reference
28          .foreign(pruefung)
29    .build();
```

Listing 5.22: Höhergradige Beziehungen mit zusätzlicer Tabelle

5.10.3 Komfortfunktionen

Die Realisierung könnte an manchen Stellen dem Test-Ingenieur mehr manuelle Arbeit abnehmen. So wird darauf verzichtet, beim Löschen einer Zeile aus einer Tabelle auch alle beteiligten Beziehungen zu entfernen. Listing 5.23 zeigt, wie ein Professor aus der Professoren-Tabelle entfernt wird. Die erste Zeile entfernt keine Einträge in anderen Tabellen wie z.B. der Beaufsichtigt-Tabelle. Folglich müssen die Relationen (mehr oder weniger) manuell aus anderen Tabellen entfernt werden.

```
1  dataSet.professorTable.deleteRow(HAASE);
2  dataSet.beaufsichtigtTable.deleteAllAssociations(HAASE);
```

Listing 5.23: Löschen von Zeilen

Diese Entscheidung hat unterschiedliche Gründe:

- **Einsatzgebiet**: Die Bibliothek soll Unit-Tests in Verbindung mit Datenbanken vereinfachen. Es handelt sich hier nicht um ein API, das in einer Anwendung ausgeliefert wird. Während es in einem API für produktive Anwendungen durchaus wünschenswert sein kann, dass das System beim Löschen von Entitäten gewisse Aufgaben automatisch erledigt, ist so ein Verhalten innerhalb einer Test-Bibliothek zweifelhaft. Explizites Löschen von Zeilen auf allen beteiligten Tabellen verbessert die Ausdrucksstärke des Tests.

- **Code-Qualität**: Eine Funktion (bzw. Methode) sollte genau eine Aufgabe erledigen. Wenn `deleteRow` zusätzlich beteiligte Relationen auflöst, erledigt diese Funktion mehr als nur eine Aufgabe [Mar09, 65f]. Außerdem würde es sich um einen unerwarteten Nebeneffekt handeln [Mar09, 75f].

- **Klarheit**: Es ist nicht eindeutig, wie beim Entfernen von Zeilen vorgegangen werden soll, wenn sie Teil einer Relation sind. Bei einer n:m-Relation könnte sich die Regel ableiten lassen, dass beim Löschen einer Zeile auch alle assoziierten n:m-Relationen entfernt werden können. Aber was ist bei einer 1:n-Relation? Wenn ein Professor entfernt wird, was soll mit Lehrveranstaltungen passieren, die ihm zugeordnet sind?

Kapitel 6

Generieren von Testdaten

Es gibt verschiedene Ansätze zur Generierung von Testdaten für Datenbank-basierte Anwendungen:

1. **Modell-basierte, Abfrage-unabhängige Generierung**: Anhand eines Datenbank-Modells werden Entitäten mit Zufallswerten für die Attribute (Spalten) erzeugt. Es gibt einige kommerzielle Werkzeuge aber auch frei nutzbare Internet-Seiten für die Generierung.

2. **Modell-basierte, Abfrage-basierte Generierung**: Ausgehend von konkreten Abfragen (z.B. in SQL) werden für die Abfrage passende Daten erzeugt. Binnig beschreibt in [BKL07] einen Ansatz, bei dem SQL-Abfragen als Grundlage für die Daten-Generierung verwendet werden. Mit AGENDA wird in [Cha+04] ein Toolset vorgestellt, das neben dem Datenbank-Schema den Anwendungsquellcode betrachtet.

3. **Anonymisierung realer Daten**: Bei diesem Ansatz findet keine echte Generierung statt. Stattdessen werden Daten einer realen Anwendung anonymisiert und für Tests verwendet.

Für die Generierung eines Standard Fixtures scheint nur die erste Variante sinnvoll zu sein: Es liegt bereits ein Modell vor und die generierten Daten sollen idealerweise für alle Tests verwendet werden können. Konkrete Anfragen als Grundlage für die Generierung sind nicht sinnvoll. Einerseits können sie vom SUT verborgen werden können, andererseits eigenen sie eher für Fixtures, die für einen einzelnen Unit-Test geeignet sind. Die Anonymisierung realer Daten stellt keine Daten-Generierung im eigentlichen Sinn dar und setzt bestehende Daten voraus.

Eine Auswahl existierender Modell-basierter, Abfrage-unabhängiger Datengeneratoren wird im folgenden Abschnitt auf die Anwendbarkeit hin untersucht.

6.1 Betrachtung existierender Werkzeuge

Es gibt bereits eine Reihe von Werkzeugen zur Generierung von Zufallsdaten für Datenbanken. In wie weit sich diese für die Aufgabenstellung nutzen lassen, soll im folgenden kurz analysiert werden.

6.1.1 Kommerzielle Werkzeuge

Zu den betrachteten kommerziellen Anwendungen zählen:

- **Datanamic Data Generator MultiDB**:
  http://www.datanamic.com/datagenerator/
- **DTM Data Generator**:
  http://www.sqledit.com/dg/
- **forSQL Data Generator**:
  http://www.forsql.com/
- **Red Gate SQL Data Generator**:
  http://www.red-gate.com/products/sql-development/sql-data-generator/

Insgesamt sind die Möglichkeiten der Anwendungen relativ ähnlich. Die größten Unterschiede aus Nutzer-Sicht liegen in der Bedienung. Die Werkzeuge arbeiten zufallsbasiert aber deterministisch, d.h. sie erzeugen bei gleichem Modell die gleichen Daten. Das sogenannte *Seed*, mit dem der Zufallszahlengenerator für eine einzelne Spalte initialisiert wird, lässt sich z.B. beim Red Gate SQL Data Generator komfortabel festlegen.

Die Werkzeuge sind vor allem für die Generierung von großen Datenmengen (Massen-Daten) vorgesehen. Dies zeigt sich auch darin, dass sie Beziehungen auch nur zufällig modellieren. Über eine entsprechend große Menge an Testdaten soll dann auch jeder notwendige Fall abgedeckt sein. Die Menge der zu erzeugenden Testdaten lässt sich für jede einzeln Tabelle konfigurieren. Eigene Vorschläge, wie viele Daten generiert werden sollten, machen die Werkzeuge nicht.

6.1.2 Andere Ansätze

In [HTW06] wird ein Algorithmus zur Generierung von Test-Daten vorgestellt, der das Datenbank-Modell als Graphen betrachtet. Tabellen stellen Knoten und ihre Beziehungen stellen gerichtete Kanten dar. Die Anzahl der generierten Entitäten wird über Verhältnisse vom Tester konfiguriert. Auch dieser Algorithmus ist eher für die Erzeugung von Massen-Daten geeignet.

6.1.3 Fazit

Die von dem zu entwickelnden Generator erzeugten Testdaten sollen allerdings überschaubar und wartbar sein. Dies steht in Widerspruch mit einer Massen-Daten-Generierung, wie sie die kommerziellen Werkzeuge bieten. Zum selben Schluss kommt Raza in [RC12, S. 126]. Massen-Daten eignen sich eher für Stabilitäts-, Performance- und Regressionstests.

Keines der kommerziellen Werkzeuge ist in der Lage, die Anzahl der zu generierenden Entitäten selbst zu bestimmen. Auch der Algorithmus aus [HTW06] ist dazu nicht in der Lage.

Aus diesem Grund soll ein Algorithmus entwickelt werden, der Beziehungen nicht nur zufällig generiert, sondern möglichst alle Grenzfälle erzeugt. Äquivalenzklassenbildung und

Grenzwertanalyse sind ein bewährtes Vorgehen, um die Menge von Test-Daten zu reduzieren. Eine Äquivalenzklasse fasst Eingangsdaten zusammen, für die das Verhalten des SUT das selbe ist. Tests werden statt mit vielen Werten aus einer Klasse nur mit wenigen oder einem Wert durchgeführt [SL10, S. 114]. Die Grenzwertanalyse stellt eine Ergänzung dar: Fehler treten häufig an Grenzfällen auf [SL10, S. 125]. Im Falle zu generierender Beziehungen sollten die Grenzfälle die minimale und die maximale Beziehungsmöglichkeiten abdecken.

6.2 Generierung von Beziehungen

Der zu entwickelnde Algorithmus übernimmt das Konzept der Äquivalenzklassenbildung und Grenzwertanalyse, um möglichst alle notwendigen Beziehungskombinationen zwischen zwei Entitätstypen zu modellieren. Die Menge der zu generierenden Entitäten soll dabei möglichst gering gehalten werden.

Unterschiedliche Beziehungstypen stellen unterschiedliche Anforderungen an den Daten-Generator. Binäre Beziehungstypen lassen sich in die drei Hauptkategorien 1:1, 1:n und n:m einordnen. Die folgenden Abbildungen stellen die zu generierenden Entitäten der beiden Entitätstypen A und B dar. Eine Entität wird von einem kleinen Kreis repräsentiert, ihr Entitätstyp über die Spalte festgelegt. Eine Beziehung zwischen zwei Entitäten wird über eine Verbindungsgerade beschrieben. Grundsätzlich können die beiden Typen A und B auch den selben Typen darstellen.

Ganz allgemein lassen sich alle binären Beziehungen als n..N:m..M-Beziehung ansehen. n und m stellen jeweils untere Grenzen darf, N und M die oberen. Die grundlegende Generierungsstrategie sieht die Generierung der folgenden vier Kombinationen vor:

- n:m
- n:M
- N:m
- N:M

Verschiedene Fälle können redundant sein, falls untere und obere Grenze identisch sind. Auf die Generierung dieser redundanten Beziehungen kann verzichtet werden.

Die Abbildungen stellen dar, welche Entitäten und Beziehungen *mindestens* generiert werden sollten, um die von den Grenzen der Multiplizitäten bestimmten Äquivalenzklassen abzudecken.

6.2.1 Kategorie der 1:1-Beziehungen

Unter die Kategorie 1:1-Beziehung fallen alle Beziehungstypen, bei denen eine oder keine Entität mit genau einer oder keiner Entität in Beziehung stehen kann.

1..1:1..1

Eine Entität des Typs A steht mit genau einer Entität des Typs B in Beziehung. Die Anzahl der generierten Entitäten muss übereinstimmen, es muss mindestens eine Entität pro Typ erzeugt werden (siehe Abbildung 6.1).

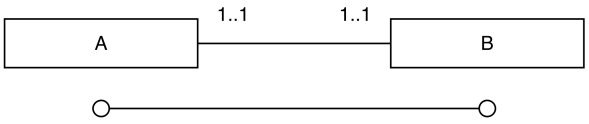

Abbildung 6.1: Beziehungen nach dem Schema 1..1:1..1

0..1:1..1

Im Gegensatz zu demr vorherigen Beziehungstyp muss bei dieser eine Entität nicht zwingend in Beziehung mit einer anderen stehen. Abbildung 6.2 zeigt die zu generierenden Entitäten der Typen A und B, wobei jede Entität von A mit einer Entität von B in Beziehung stehen muss, eine Entität von B jedoch nicht zwingend mit einer Entität von A. Daraus folgt, dass es von B mindestens eine Entität mehr geben muss als von A.

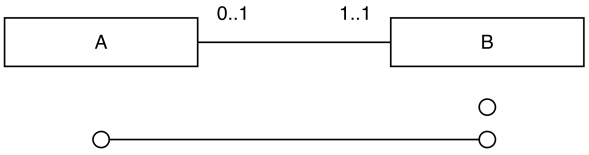

Abbildung 6.2: Beziehungen nach dem Schema 0..1:1..1

Der Generator muss mindestens zwei Entitäten des Typs B erzeugen, und eine des Typs A, um sicherzustellen, dass alle Fälle für diese Beziehung abgedeckt sind.

Die Beziehung 1..1:0..1 ist symmetrisch zu dieser.

0..1:0..1

Wenn für beide Entitätstypen die Beziehung optional ist, muss der Generator jeweils mindestens 2 Entitäten erzeugen. Jeweils eine Entität ohne Beziehung und jeweils eine Entität mit einer Beziehung (siehe Abbildung 6.3).

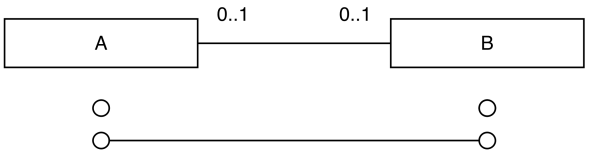

Abbildung 6.3: Beziehungen nach dem Schema 0..1:0..1

6.2.2 Kategorie der 1:n-Beziehungen

Eine Entität steht in Beziehung mit keiner oder mehreren anderen Entitäten. Dabei kann die Anzahl begrenzt sein (konkreter Wert für n) oder unbegrenzt. Der Generator kann nur eine begrenzte Anzahl von Entitäten erzeugen, die Grenze sollte konfigurierbar sein.

1..1:1..n

Die einfachste Form der 1:n-Beziehungen. Eine Entität des Typs A ist in einer Beziehung mit einer oder mehreren Entitäten des Typs B. Eine Entität des Typs B ist mit genau einer

Entität des Typs A in Beziehung. Die zu generierenden Entitäten sind in Abbildung 6.4 dargestellt.

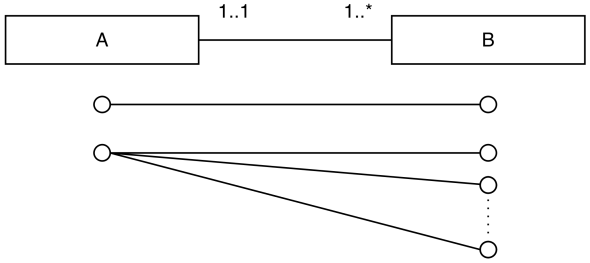

Abbildung 6.4: Beziehungen nach dem Schema 1..1:1..n

0..1:1..n

Eine Entität des Typs A steht in Beziehung mit einer oder mehreren Entitäten des Typs B. Eine Entität des Typs B steht entweder mit genau einer oder mit keiner Entität des Typs A in Beziehung. Abbildung 6.5 stellt die zu generierenden Entitäten dar.

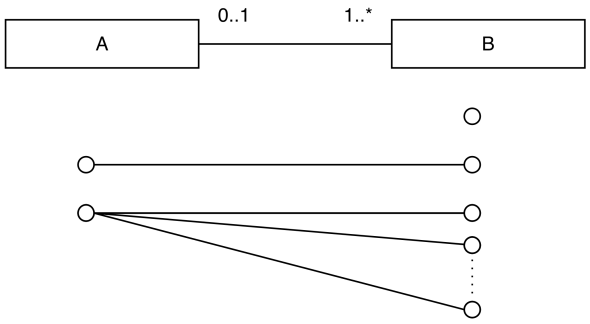

Abbildung 6.5: Beziehungen nach dem Schema 0..1:1..n

1..1:0..n

Eine Entität des Typs A steht in Beziehung mit keiner, einer oder mehreren Entitäten des Typs B. Eine Entität des Typs B muss mit genau einer Entität des Typs A in Beziehung stehen.

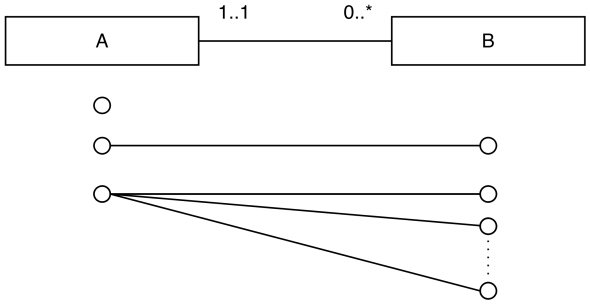

Abbildung 6.6: Beziehungen nach dem Schema 1..1:0..n

0..1:0..n

Eine Entität des Typs A steht mit keiner, einer oder mehreren Entitäten des Typs B in Beziehung. Eine Entität des Typs B kann mit keiner oder genau einer Entität des Typs A in Beziehung stehen.

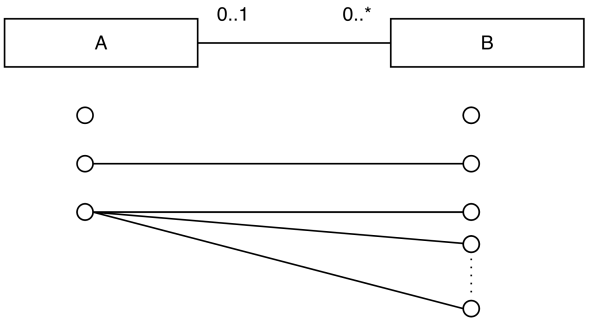

Abbildung 6.7: Beziehungen nach dem Schema 0..1:0..n

6.2.3 Kategorie der n:m-Beziehungen

Die allgemeinste Form einer Beziehung zwischen zwei Eintitätstypen stellt eine n..N:m..M-Beziehung dar. Dabei handelt es sich bei n und m jeweils um untere und bei N und M jeweils um obere Schranken. Untere und obere Schranken können identisch sein.

Jede der in diesem Abschnitt beschriebenen Beziehungsformen stellen Spezialfälle von n:m-Beziehungen dar, bei denen eine oder beide oberen Grenzen 1 sind. Sind beide obere Grenzen größer als 1, dann wird eine solche Beziehung üblicherweise über assoziative Tabellen realisiert.

Sollte eine der unteren Grenzen 0 sein, wird sie als 0..1 interpretiert. D.h es wird eine entsprechende Entität ohne Beziehung erzeugt, aber auch eine einzelne Entität, die in Beziehung mit anderen Entitäten steht (siehe dazu Abbildungen 6.8 und 6.10).

Um die unterschiedlichen Kombinationen aus n, N, m und M in den Grafiken zu verdeutlichen, wird die Generierung einer 1..3:0..4-Beziehung dargestellt.

1. Fall (n:m): 1:0..1

Im ersten Fall werden beide unteren Grenzen betrachtet. Da eine der Grenzen 0 ist, wird eine Entität ohne Beziehung generiert. Abbildung 6.8 visualisiert die zu erzeugenden Entitäten für die Kombination n:m.

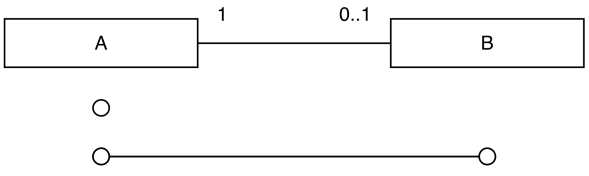

Abbildung 6.8: Beziehungen nach dem Schema 1:0..1 (n:m)

2. Fall (n:M): 1:4

Der zweite Fall betrachtet eine untere mit einer oberen Grenze. Die in diesem Beispiel erzeugten Entitäten zeigt Abbildung 6.9.

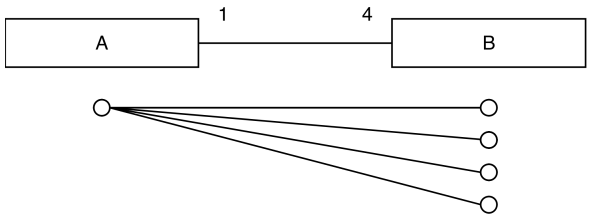

Abbildung 6.9: Beziehungen nach dem Schema 1:4 (n:M)

3. Fall (N:m): 3:0..1

Wie im ersten Fall ist hier eine der beiden Grenzen 0. Die in der Abbildung 6.10 dargestellte Entität ohne Beziehung ist der Vollständigkeit halber aufgezeigt, muss aber nicht generiert werden, da sie bereits generiert wurde (siehe Abbildung 6.8).

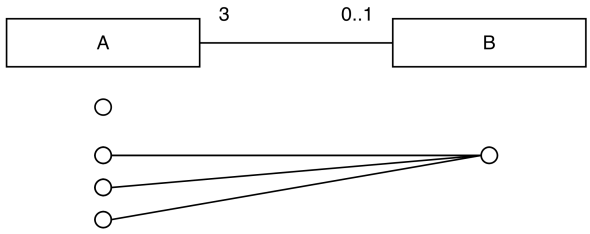

Abbildung 6.10: Beziehungen nach dem Schema 3:0..1 (N:m)

4. Fall (N:M): 3:4

Der vierte Fall behandelt schließlich die beiden oberen Grenzen. Abbildung 6.11 zeigt die Vollvermaschung der zu erzeugenden Entitäten.

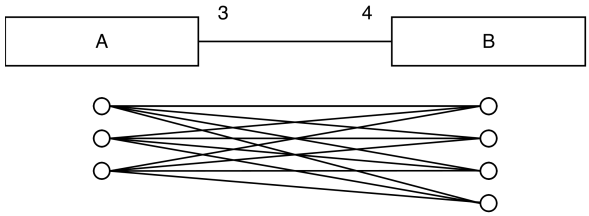

Abbildung 6.11: Beziehungen nach dem Schema 3:4 (N:M)

Gesamte Generierung (n..N:m..M): 1..3:0..4

Für eine Beziehung nach dem Schema 1..3:0..4 würde der Algorithmus die in Abbildung 6.12 dargestellte Entitäten und Beziehungen vorsehen.

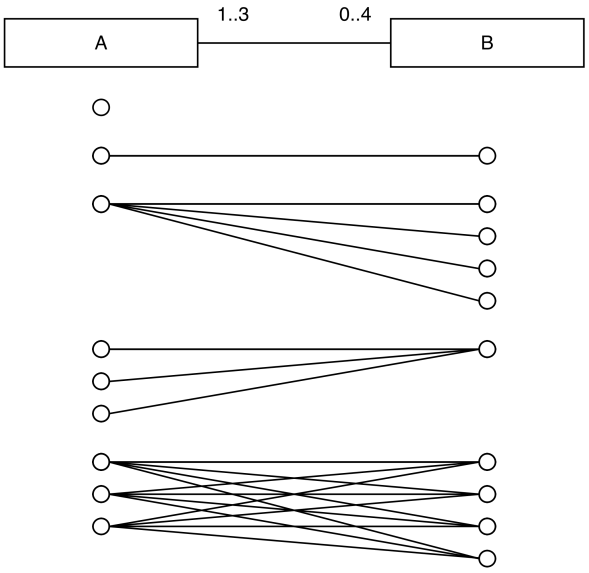

Abbildung 6.12: Beziehungen nach dem Schema 1..3:0..4 (n..N:m..M)

6.3 Komplexität bei der Generierung von Beziehungen

Im vorausgegangenen Abschnitt wurden nur Beziehungen zwischen zwei Entitätstypen betrachtet. In realen Anwendungen können Entitätstypen mit mehr als nur einem anderen Entitätstyp in Beziehung stehen und auch mit dem selben Entitätstyp mehr als nur ein Mal.

Bezüglich der Datengenerierung lassen sich hierbei zwei generelle Vorgehensweisen unterscheiden:

- **Beziehungen unabhängig betrachten**: Es wird angenommen, dass unterschiedliche Beziehungen voneinander unabhängig sind. Die Frage, ob ein Professor eine Lehrveranstaltung leitet, lässt keine Rückschlüsse zu, ob und welche Prüfungen er beaufsichtigt.
- **Beziehungen abhängig voneinander betrachten**: In der Praxis beeinflussen sich viele Beziehungen. Ein Student, der die Prüfung einer Lehrveranstaltung schreibt, darf wohl nicht gleichzeitig Tutor dieser Veranstaltung sein.

Alle Beziehungen abhängig voneinander zu betrachten kann schnell zu exponentiell zunehmenden Testdaten führen. Einen riesigen Bestand an Daten um für Tests unnötige Beziehungen und schließlich auch Entitäten zu verringern scheint aufwändiger, als einen kleinen Datenbestand um fehlende Beziehungen punktuell zu erweitern. Aus diesem Grund berücksichtigt der Algorithmus keine Beziehungen in Abhängigkeit von anderen – mit Ausnahme von assoziativen Tabellen.

6.4 Algorithmus zur Generierung von Beziehungen

Der entwickelte Algorithmus übernimmt aus [HTW06] die Idee, das Modell als Graphen zu betrachten und zu traversieren. Bevor der Pseudocode des Algorithmus beschrieben wird, werden einige verwendete Begriffe beschrieben und das Klassen-Diagramm des Modells vorgestellt, das der Daten-Generator verwendet.

6.4.1 Begriffserklärungen

Die Beschreibung des Algorithmus verwendet einige Begriffe und Konventionen, die im Folgenden beschrieben werden.

Knoten und Kanten

Die Idee, das Datenbank-Modell als Graphen zu betrachten, stammt aus [HTW06]. Tabellen stellen die Knoten des Graphs dar. Die Foreign-Key-Beziehungen zwischen zwei Tabellen stellen die Kanten des Graphs dar. Eine Kante ist gerichtet, von der Tabelle mit der Foreign-Key-Spalte zur referenzierten Tabelle hin. Aufgrund der Richtung hat jede Kante eine Start- und eine Zieltabelle.

Auch wenn es sich um gerichtete Kanten handelt, ist eine Traversierung in beide Richtungen möglich.

Assoziative Tabellen

Assoziative Tabellen verbinden zwei Tabellen. Die beiden verbundenen Tabellen werden im Folgenden als linke und rechte Tabelle bezeichnet, analog wird von linker und rechter Kante gesprochen.

Die Reihenfolge, in der die Spalten der Tabelle definiert sind, bestimmt die Einteilung in links und rechts. Der erste Fremdschlüssel referenziert die linke Tabelle. Das Ergebnis der Generierung hängt jedoch nicht von dieser Einteilung ab.

6.4.2 Klassendiagramm

Für die Datengenerierung wird das bisherig verwendete Klassen-Modell um eine Klasse `Kante` erweitert (siehe Abbildung 6.13). Diese Klasse repräsentiert eine Kante des Graphen. Zu einer Kante gehört eine Start- und eine Zieltabelle. Eine Tabelle selbst kann zu beliebig vielen Kanten gehören. Alle Tabellen eines Datenbankmodells werden zu einer Tabellenliste zusammengefasst.

6.4.3 Pseudocode

Der Algorithmus ist in mehrere Teilfunktionen unterteilt. Einige Funktionen werden rekursiv aufgerufen, um den Graphen entlang der Kanten zu traversieren. Der Einstiegspunkt stellt die Funktion `Generiere_Test_Daten` dar, die im folgenden Abschnitt beschrieben wird.

Generiere Test-Daten

Die Funktion `Generiere_Test_Daten` (siehe Listing 6.1) ist der Einstiegspunkt für den Algorithmus zur Generierung von Test-Daten. Im ersten Schritt wird die Reihenfolge der Tabellen festgelegt, die als Startpunkte in Frage kommen. Die Liste stellt sicher, dass auch Datenbank-Modelle, die aus mehreren unabhängigen Graphen bestehen, vollständig

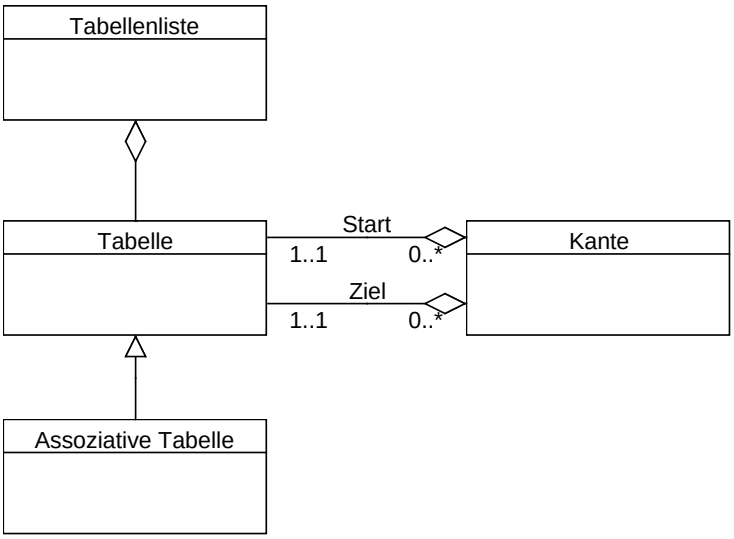

Abbildung 6.13: Diagramm der Generator-Modell-Klassen

generiert werden. In einem Datenbank-Modell, in dem alle Tabellen direkt oder indirekt in Beziehung stehen, würde die Festlegung einer Starttabelle ausreichen.

Zur Sortierung der Tabellen wird die Anzahl eingehender Kanten verwendet. Der Grund dafür und die Bedeutung der Reihenfolge der Tabellen wird in Abschnitt 6.5.1 beschrieben.

Anschließend wird über diese Tabellenliste iteriert. Wurde eine Tabelle noch nicht behandelt, wird die entsprechende Funktion zur Generierung der Daten aufgerufen. Dabei werden nicht-assoziative und assoziative Tabellen unterschiedlich behandelt. Der Grund dafür ist, dass assoziative Tabellen ein Hilfskonstrukt darstellen, das selbst eine Beziehung auf ER-Ebene darstellt.

Am Ende stellt der Algorithmus sicher, dass jede Entität gültig generiert wurde, also dass die Beziehungen die Constraints erfüllen. Gegebenenfalls werden hier Entitäten nach generiert.

```
1  GeneriereTestDaten
2  ------------------
3  L := Tabellenliste, nach Anzahl eingehender Kanten aufsteigend geordnet
4  FOR EACH (noch nicht besuchte Tabelle T IN der geordneten Tabellenliste L)
5  DO
6    Markiere Tabelle T als besucht;
7    IF (Tabelle T ist keine assoziative Tabelle)
8    THEN CALL Generiere_Daten_Fuer_Nichtassoziative_Tabelle(T);
9    ELSE CALL Generiere_Daten_Fuer_Assoziative_Tabelle(T);
10   END IF;
11 END FOR;
12 CALL Erweitere_Generierte_Daten_Zu_Konsistenten_Daten()
```

Listing 6.1: Generiere Test-Daten

Generiere Daten für nichtassoziative Tabelle

Der Generator betrachtet zur Generierung von Entitäten die Beziehungen der Tabellen. Aus diesem Grund erzeugt die Funktion `Generiere_Daten_Fuer_Nichtassoziative_Tabelle` (siehe Listing 6.2)

selbst keine Daten. Stattdessen werden die (noch unbehandelten) Kanten der Tabelle betrachtet.

Handelt es sich bei der an der Beziehung beteiligten Tabelle um eine assoziative Tabelle, wird mit der Generierung der assoziativen Beziehungen fortgesetzt. Ansonsten wird die Funktion zur Generierung der Daten für eine Kante aufgerufen.

Der Algorithmus aus [HTW06] bevorzugt ausgehende Kanten bei der Erzeugung von Daten, da sich diese mit weniger Aufwand abarbeiten lassen. Diese Bevorzugung wird für den hier entwickelten Algorithmus übernommen, auch wenn sich der Aufwand für ein- und ausgehende Kanten nicht unterscheidet.

```
1  Generiere_Daten_Fuer_Nichtassoziative_Tabelle(Tabelle T)
2  ------------------------------------------------
3  FOR EACH (ausgehende Kante K, die noch nicht besucht wurde)
4  DO
5    IF (Zieltabelle Z von Kante K ist keine assoziative Tabelle)
6    THEN Markiere Kante K als besucht;
7         CALL Generiere_Daten_Fuer_Kante(K);
8         IF (Zieltabelle Z noch nicht besucht)
9         THEN Markiere Tabelle Z als besucht;
10             CALL Generiere_Daten_Fuer_Nicht_Assoziative_Tabelle(Z);
11        END IF;
12   ELSE CALL Generiere_Daten_Fuer_Assoziative_Tabelle(Z);
13   END IF;
14 END FOR;
15 FOR EACH (eingehende Kante K, die noch nicht besucht wurde)
16 DO
17   IF (Starttabelle S von Kante K ist keine assoziative Tabelle)
18   THEN Markiere Kante K als besucht;
19        CALL Generiere_Daten_Fuer_Kante(K);
20        IF (Starttabelle S noch nicht besucht)
21        THEN Markiere Tabelle S als besucht;
22             CALL Generiere_Daten_Fuer_Nicht_Assoziative_Tabelle(S);
23        END IF;
24   ELSE CALL Generiere_Daten_Fuer_Assoziative_Tabelle(S);
25   END IF;
26 END FOR;
```

Listing 6.2: GeneriereDatenFuerNichtassoziativeTabelle

Generiere Daten für Kante

Die Funktion zum Generieren von Daten für eine Kante setzt die 6.2.3 beschriebenen Schritte um (siehe Listing 6.3). Es werden alle vier Kombinationen aus Minimum und Maximum behandelt (Zeilen 6-9). Die eigentliche Generierung ist in eine Hilfsfunktion ausgelagert, die mit allen Min-Max-Kombinationen aufgerufen wird.

```
1  Generiere_Daten_Fuer_Kante(Kante K)
2  -----------------------------
3  S := Starttabelle von Kante K;
4  Z := Zieltabelle von Kante K;
5  // Generierung der Daten entsprechend Abschnitt 6.2.3
6  CALL Generiere_Entitaeten_Und_Beziehungen(K, min(S), min(Z));
7  CALL Generiere_Entitaeten_Und_Beziehungen(K, min(S), max(Z));
8  CALL Generiere_Entitaeten_Und_Beziehungen(K, max(S), min(Z));
9  CALL Generiere_Entitaeten_Und_Beziehungen(K, max(S), max(Z));
```

Listing 6.3: Generiere Daten Fuer Kante

Generiere Entitäten und Beziehungen

Es soll vermieden werden, unnötige Entitäten und Beziehungen zu generieren. Unnötig sind vor allem redundante Beziehungen. Diese können beispielsweise entstehen, wenn die untere

und die obere Grenze einer Multiplizität identisch sind. Deshalb wird für jede Kombination aus Kante, Anzahl der beteiligten Entitäten der Starttabelle und Anzahl der beteiligten Entitäten der Zieltabelle die Generierung nur ein einziges Mal durchgeführt.

Der Algorithmus (siehe Listing 6.4) prüft, ob es sich um eine optionale Beziehung handelt (Zeile 7 und Zeile 10). Eine der Grenzen s bzw. z ist in einem solchen Fall gleich 0. Handelt es sich um eine optionale Beziehung, wird eine entsprechende Entität berechnet, die in Bezug auf die Kante in keiner Beziehung ist. Anschließend wird die Funktion rekursiv aufgerufen, diesmal mit dem Wert 1 als Grenze anstelle der 0 (Zeile 9 und Zeile 12).

Sofern s und z beide ungleich 0 sind, wird eine Entität in der Zieltabelle berechnet und s Entitäten in der Starttabelle. Zwischen diesen Entitäten wird die von der Kante K repräsentierte Beziehung hergestellt (Zeilen 14 bis 19).

Der Eingangswert z kann aufgrund der Tatsache, dass es sich um eine nicht-assoziative Tabelle handelt, nur 0 oder 1 sein.

```
1  Generiere_Entitaeten_Und_Beziehungen(K, s, z)
2  ----------------------------------------
3  // Sicherstellen, dass nicht mehr Entitäten als notwendig erzeugt werden
4  IF (Für die Kombination (K, s, z) wurden bereits Entitäten erzeugt)
5  THEN RETURN;
6  END IF
7  IF (Wert der Grenze s ist 0)
8  THEN Berechne Entität e in Zieltabelle, die in keiner Beziehung zur
       Starttabelle stehen darf;
9       CALL Generiere_Entitaeten_Und_Beziehungen(k, 1, z);
10 ELSE IF (Wert der Grenze z ist 0)
11 THEN Berechne Entität e in Starttabelle, die in keiner Beziehung zur
       Zieltabelle stehen darf;
12      CALL Generiere_Entitaeten_Und_Beziehungen(k, s, 1);
13 ELSE // z ist hier immer 1
14      Berechne Entität e in Zieltabelle, die in keiner Beziehung zur
            Starttabelle stehen darf;
15      FOR i = 1 TO s
16      DO
17        Berechne Entität se in Starttabelle, die in keiner Beziehung zur
              Zieltabelle stehen darf;
18        Stelle Beziehung zwischen Entität se und Entität e her;
19      END FOR;
20 END IF;
```

Listing 6.4: Generiere Entitäten Und Beziehungen

Generiere Daten für assoziative Tabelle

Assoziative Tabellen werden typischerweise zur Modellierung von n..N:m..M-Beziehungen verwendet. Diese Beziehungen werden entsprechend der in Abschnitt 6.2.3 beschriebenen Strategie generiert. D.h. es werden alle vier Kombinationen aus den jeweiligen Minimal- und Maximalwerten betrachtet.

Nach der Erzeugung der Daten für die assoziative Tabelle versucht der Algorithmus (siehe Listing 6.5), die Generierung bei den beiden assoziierten Tabellen fortzusetzen – falls diese noch nicht behandelt wurden.

```
1  Generiere_Daten_Fuer_Assoziative_Tabelle(Tabelle T)
2  ----------------------------------------------
3  LK := linke Kante der assoziativen Tabelle T;
4  RK := rechte Kante der assoziativen Tabelle T;
5  markiere Kanten LK und RK als besucht;
6  LM := Multiplizität der linken Beziehung, ausgehende Seite
7  RM := Multiplizität der rechten Beziehung, ausgehende Seite
8  CALL Generiere_Assoziative_Entitaeten_Und_Beziehungen(T, min(LM), min(RM));
9  CALL Generiere_Assoziative_Entitaeten_Und_Beziehungen(T, min(LM), max(RM));
```

```
10 CALL Generiere_Assoziative_Entitaeten_Und_Beziehungen(T, max(LM), min(RM));
11 CALL Generiere_Assoziative_Entitaeten_Und_Beziehungen(T, max(LM), max(RM));
12 LT := Zieltabelle von Kante LK;
13 RT := Zieltabelle von Kante RK;
14 IF (LT wurde noch nicht besucht)
15 THEN Markiere Tabelle LT als besucht;
16      IF (Tabelle LT ist keine assoziative Tabelle)
17      THEN CALL Generiere_Daten_Fuer_Nichtassoziative_Tabelle(LT);
18      ELSE CALL Generiere_Daten_Fuer_Assoziative_Tabelle(LT);
19      END IF;
20 END IF;
21 IF (RT wurde noch nicht besucht)
22 THEN Markiere Tabelle RT als besucht;
23      IF (Tabelle RT ist keine assoziative Tabelle)
24      THEN CALL Generiere_Daten_Fuer_Nichtassoziative_Tabelle(RT);
25      ELSE CALL Generiere_Daten_Fuer_Assoziative_Tabelle(RT);
26      END IF;
27 END IF;
```

Listing 6.5: Generiere Daten für assoziative Tabelle

Generiere assoziative Entitäten und Beziehungen

Die Funktion `Generiere_Assoziative_Entitaeten_Und_Beziehungen` erzeugt Entitäten und Beziehungen in Verbindung mit assoziativen Tabellen (siehe Listing 6.6). D.h. sie erzeugt Entitäten in der assoziativen Tabelle und bei Bedarf in den beiden an der Assoziation beteiligten Tabellen (linke und rechte Tabelle der assoziativen Tabelle).

Die Funktion erwartet drei Parameter:

- **Tabelle `AT`**: Die assoziative Tabelle, die behandelt wird.
- **Grenze `l`**: Der momentane Grenzwert der linksseitigen Multiplizität.
- **Grenze `r`**: Der momentane Grenzwert der rechtsseitigen Multiplizität.

Abbildung 6.14 veranschaulicht die Parameter und zeigt, dass die angegebenen Grenzen bestimmen, wie viele Entitäten der *anderen* Tabelle benötigt werden. D.h. `l` bestimmt, wie viele Entitäten der rechten Tabelle, und `r` bestimmt, wie viele Entitäten der linken Tabelle generiert werden müssen.

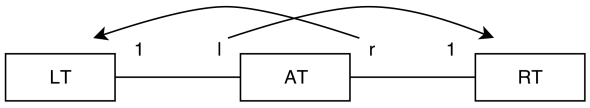

Abbildung 6.14: Parameter für Daten-Generierung bei assoziativer Tabelle

Für den Fall dass `l` oder `r` den Wert 0 enthalten, wird eine entsprechende Entität in der rechten bzw. linken Tabelle berechnet, die keine Beziehung zur assoziativen Tabelle `AT` hat. In den Zeilen 6 und 10 wird die Anzahl der zu berechnenden Entitäten der linken Tabelle (Variable `LA`) und der rechten Tabelle (Variable `RA`) berechnet. Als Minimalwert wird wie schon vorher bei optionalen Beziehungen der Wert 1 verwendet.

Sollten für die Kombination `LA` und `RA` für die assoziative Tabelle `AT` bereits Daten generiert worden sein, bricht die Funktion an dieser Stelle ab (Zeilen 14 und 15). Ein solcher Fall kann eintreten, wenn untere und obere Grenze einer Multiplizität identisch sind, oder es eine Multiplizität der Form 0..1 ist.

Der Algorithmus berechnet zuerst alle Entitäten in der linken (Zeilen 17 bis 20) und dann in der rechten Tabelle (Zeilen 21 bis 24). Danach werden die Entitäten in der assoziativen Tabelle erzeugt und die Beziehungen hergestellt (Zeilen 25 bis 33).

```
1  Generiere_Assoziative_Entitaeten_Und_Beziehungen(AT, l, r)
2  ----------------------------------------------------
3  LK := linke Kante der assoziativen Tabelle AT;
4  RK := rechte Kante der assoziativen Tabelle AT;
5  // Grenzen "tauschen" und Mindestanzahl auf 1
6  LA := Max(r, 1);  // r als Grenze für linke Tabelle, mindestens 1
7  IF (r ist gleich 0)
8  THEN Berechne Entität in linker Tabelle, die in keiner Beziehung zur
       assoziativen Tabelle AT stehen darf;
9  ENDIF;
10 RA := Max(l, 1);  // l als Grenze für rechte Tabelle, mindestens 1
11 IF (l ist gleich 0)
12 THEN Berechne Entität in rechter Tabelle, die in keiner Beziehung zur
       assoziativen Tabelle AT stehen darf;
13 ENDIF;
14 IF (Für die Kombination (AT, LA, RA) wurden bereits Entitäten erzeugt)
15 THEN RETURN;
16 END IF
17 FOR i = 1 TO RA
18 DO
19   Berechne Entität le[i] in linker Tabelle, die in keiner Beziehung zur
         assoziativen Tabelle AT stehen darf;
20 END FOR;
21 FOR j = 1 TO LA
22 DO
23   Berechne Entität re[j] in rechter Tabelle, die in keiner Beziehung zur
         assoziativen Tabelle AT stehen darf;
24 END FOR;
25 FOR i = 1 TO RA
26 DO
27   FOR j = 1 TO LA
28   DO
29     Erzeuge Entität e in assoziativer Tabelle AT;
30     Stelle Beziehung zwischen Entität e und Entität le[i] her;
31     Stelle Beziehung zwischen Entität e und Entität re[j] her;
32   END FOR;
33 END FOR;
```

Listing 6.6: Generiere Assoziative Entitaeten Und Beziehungen

Erweitere Generierte Daten zu konsistenten Daten

Es kann passieren, dass am Ende der Traversierung des Graphs ungültige Entitäten vorhanden sind. Ungültig bedeutet, dass sie bei einer oder mehrere Kanten nicht in ausreichend vielen Beziehungen steht. Dies kann passieren, wenn für eine bereits besuchte Tabelle zusätzliche Entitäten erzeugt werden müssen – beispielsweise wenn das Datenbank-Modell zirkuläre Abhängigkeiten enthält.

Der Algorithmus sieht nicht vor, bei zusätzlicher Erzeugung von Entitäten den Graphen rückwärts zu traversieren. Stattdessen werden am Ende alle Entitäten validiert und gegebenenfalls wird nachgeneriert.

Die Funktion zur Validierung iteriert über die Entitäten aller Tabellen. Es wird überprüft, ob die jeweilige Entität gültig generiert wurde auf Hinblick der aus- und eingehenden Kanten. Sollte dies für eine Entität nicht der Fall sein, wird eine entsprechende Beziehung hergestellt und die Validierung erneut von Anfang an durchgeführt.

Unter Umständen würde dieser Algorithmus nie terminieren, weshalb eine Abbruch-Bedingung sinnvoll ist. Diese wird nicht im Pseudo-Code (Listing 6.7) dargestellt. Die Implementierung bricht nach Erreichen eines Grenzwertes für die

nachträgliche Generierung von Entitäten ab. Das Problem wird in Abschnitt 6.7.3 thematisiert.

```
1  Erweitere_Generierte_Daten_Zu_Konsistenten_Daten()
2  ------------------------------------------------
3  FOR EACH (Tabelle T in Tabellenliste)
4  DO
5    FOR EACH (Entität e aus generierten Entitäten der Tabelle T)
6    DO
7      FOR EACH (Kante K aus ausgehenden Kanten der Tabelle T)
8      DO
9        IF (e erfüllt Constraints für Kante K nicht)
10       THEN Berechne Entität f in Zieltabelle der Kante K, f darf keiner
             Beziehung zur Tabelle T stehen;
11            Stelle Beziehung zwischen Entität e und Entität f her;
12            CALL Erweitere_Generierte_Daten_Zu_Konsistenten_Daten;
13            RETURN;
14       END IF;
15     END FOR;
16     FOR EACH (Kante K aus eingehenden Kanten der Tabelle T)
17     DO
18       IF (e erfüllt Constraints für Kante K nicht)
19       THEN Berechne Entität f in Starttabelle der Kante K, f darf in keiner
              Beziehung zur Tabelle T stehen;
20            Stelle Beziehung zwischen Entität e und Entität f her;
21            CALL Erweitere_Generierte_Daten_Zu_Konsistenten_Daten;
22            RETURN;
23       END IF;
24     END FOR;
25   END FOR;
26 END FOR;
```

Listing 6.7: ErweitereGenerierteDatenZuKonsistentenDaten

6.4.4 Beispiel

Der Algorithmus soll an einem kleinen Beispiel veranschaulicht werden. Dieses stellt einen Teil des Modells des fortlaufenden Beispiels dar. Das reduzierte Datenbank-Modell besteht aus den vier Tabellen RAUM, PROFESSOR, BEAUFSICHTIGT und PRUEFUNG (siehe Abbildung 6.15). Die Tabelle BEAUFSICHTIGT ist assoziativ. Die Beziehungen zwischen den Tabellen haben im Vergleich zum fortlaufenden Beispiel veränderte Multiplizitäten.

Das Modell lässt sich auf ER-Ebene folgendermaßen beschreiben: Ein Raum kann beliebig vielen Professoren zugeordnet sein. Umgekehrt kann ein Professor genau einem oder keinem Raum zugeordnet sein. Eine Professor kann beliebig viele Prüfungen beaufsichtigen. Eine Prüfung benötigt mindestens drei und höchstens fünf Professoren zur Aufsicht.

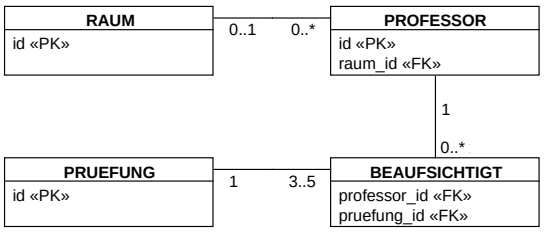

Abbildung 6.15: Einfaches Beispiel-Modell für den Algorithmus

Als konkreter Wert für die offene Grenze $\star$ wird der Wert 5 verwendet.

Die generierten Daten für das vollständige fortlaufende Beispiel befinden sich in Anhang B.

1. Schritt: Sortieren der Tabellen

Im ersten Schritt werden die Tabellen nach den eingehenden Kanten aufsteigend sortiert. Abbildung 6.16 zeigt die Kanten des Graphen. Die Zahl der eingehenden Kanten ist unter der jeweiligen Tabelle vermerkt.

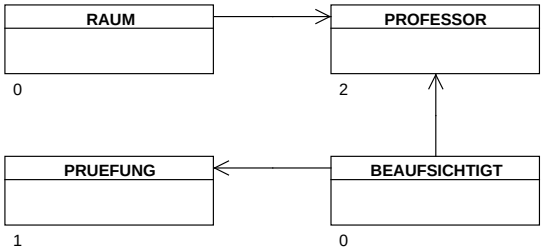

Abbildung 6.16: Kanten des einfachen Beispiel-Modells

Da zwei Knoten die selbe Anzahl eingehender Kanten haben, wird als Sekundärkriterium die Reihenfolge der Tabellendefinition verwendet. RAUM wurde vor der assoziativen Tabelle BEAUFSICHTIGT definiert und ist deshalb in der Ordnung weiter oben.

Die sortierte Tabellenliste stellt sich wie folgt dar:

1. **RAUM** (0 eingehende Kanten)
2. **BEAUFSICHTIGT** (0)
3. **PRUEFUNG** (1)
4. **PROFESSOR** (2)

Der Algorithmus iteriert über diese Liste und erzeugt ausgehend von der jeweiligen Tabelle Daten, sofern die Tabelle nicht bereits behandelt wurde. Die erste Tabelle ist RAUM.

2. Schritt: Generierung für RAUM

Der Algorithmus befindet sich bei der Generierung bei RAUM (siehe Abbildung 6.17).

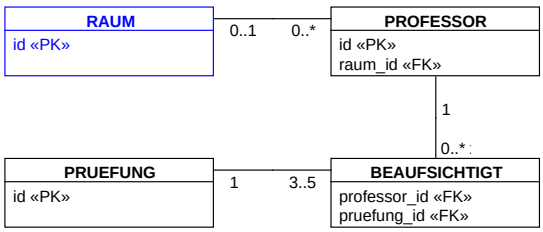

Abbildung 6.17: Schritt: RAUM

Es wird über die Kanten von RAUM iteriert und sofern die Kante noch nicht behandelt wurde, werden Daten für die jeweilige Kante erzeugt. RAUM hat nur eine Kante, diese führt zu PROFESSOR und wurde noch nicht besucht.

3. Schritt: Generierung für Kante zwischen RAUM und PROFESSOR

Die Kante repräsentiert eine 0..1:0..*-Beziehung, die Generierung für eine solche Kante ist in Abschnitt 6.2.2 beschrieben. Jeweils eine Entität aus RAUM und PROFESSOR steht in keiner Beziehung, zwei Entitäten befinden sich in einer 1:1-Beziehung und eine Entität aus RAUM steht mit fünf Entitäten aus PROFESSOR in Beziehung. In der Summe werden damit 3 Entitäten für RAUM und 7 Entitäten für PROFESSOR erzeugt. Die Zuordnung der Entitäten ist in Tabelle 6.1 dargestellt. RAUM_1 und PROF_1 haben keine Beziehung, RAUM_3 hat eine Beziehung mit fünf Professoren.

Tabelle 6.1: Zuordnung der Entitäten von RAUM und PROFESSOR

RAUM	PROFESSOR
RAUM_1	
	PROF_1
RAUM_2	PROF_2
RAUM_3	PROF_3
RAUM_3	PROF_4
RAUM_3	PROF_5
RAUM_3	PROF_6
RAUM_3	PROF_7
3 Entitäten	7 Entitäten

Nachdem die Generierung der Entitäten und Beziehungen für die Kante abgeschlossen ist, setzt der Algorithmus die Arbeit bei der Tabelle fort, die mit dieser Kante verbunden ist: PROFESSOR.

4. Schritt: Generierung für PROFESSOR

Die Generierung der Daten für die Kante zwischen RAUM und PROFESSOR ist abgeschlossen, RAUM ist als bereits besuchte Tabelle markiert. Abbildung 6.18 stellt den Generierungsstand grafisch dar.

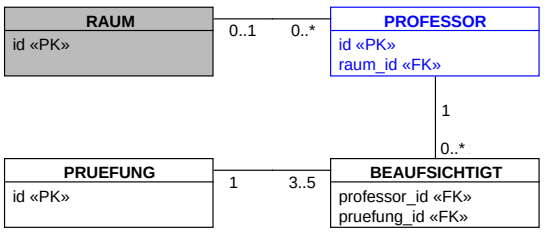

Abbildung 6.18: Schritt: PROFESSOR

Der Algorithmus iteriert über die Kanten von PROFESSOR. Dabei würden zuerst ausgehende Kanten betrachtet, dann die eingehenden. Die Tabelle hat Tabelle hat nur zwei eingehende, von denen eine bereits besucht wurde. Die Traversierung des Graphs wird mit der Kante zu Tabelle BEAUFSICHTIGT fortgesetzt. Bei BEAUFSICHTIGT handelt es sich um eine assoziative Tabelle, weshalb der nächste Schritt die Generierung von Daten für BEAUFSICHTIGT darstellt.

5. Schritt: Generierung für BEAUFSICHTIGT

Der Daten-Generator befindet sich bei der assoziativen Tabelle BEAUFSICHTIGT, die Tabellen RAUM und PROFESSOR wurden bereits besucht (siehe Abbildung 6.19).

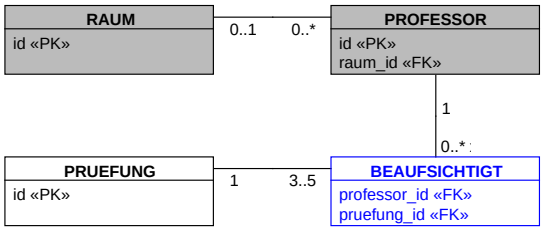

Abbildung 6.19: Schritt: BEAUFSICHTIGT

Die assoziative Tabelle BEAUFSICHTIGT dient der Modellierung einer 0..*:3..5-Beziehung zwischen den beiden Tabellen PROFESSOR und PRUEFUNG. Das Generierungsschema für assoziative Tabellen ist in Abschnitt 6.2.3 beschrieben.

Tabelle 6.2: Zuordnung der Entitäten von BEAUFSICHTIGT, PROFESSOR und PRUEFUNG

BEAUF.	PROFESSOR	PRUEFUNG	BEAUF.	PROFESSOR	PRUEFUNG
	PROF_1		B_24	PROF_13	PRUEF_8
B_1	PROF_2	PRUEF_1	B_25	PROF_13	PRUEF_9
B_2	PROF_3	PRUEF_1	B_26	PROF_13	PRUEF_10
B_3	PROF_4	PRUEF_1	B_27	PROF_13	PRUEF_11
B_4	PROF_5	PRUEF_2	B_28	PROF_13	PRUEF_12
B_5	PROF_6	PRUEF_2	B_29	PROF_14	PRUEF_8
B_6	PROF_7	PRUEF_2	B_30	PROF_14	PRUEF_9
B_7	PROF_8	PRUEF_2	B_31	PROF_14	PRUEF_10
B_8	PROF_9	PRUEF_2	B_32	PROF_14	PRUEF_11
B_9	PROF_10	PRUEF_3	B_33	PROF_14	PRUEF_12
B_10	PROF_10	PRUEF_4	B_34	PROF_15	PRUEF_8
B_11	PROF_10	PRUEF_5	B_35	PROF_15	PRUEF_9
B_12	PROF_10	PRUEF_6	B_36	PROF_15	PRUEF_10
B_13	PROF_10	PRUEF_7	B_37	PROF_15	PRUEF_11
B_14	PROF_11	PRUEF_3	B_38	PROF_15	PRUEF_12
B_15	PROF_11	PRUEF_4	B_39	PROF_16	PRUEF_8
B_16	PROF_11	PRUEF_5	B_40	PROF_16	PRUEF_9
B_17	PROF_11	PRUEF_6	B_41	PROF_16	PRUEF_10
B_18	PROF_11	PRUEF_7	B_42	PROF_16	PRUEF_11
B_19	PROF_12	PRUEF_3	B_43	PROF_16	PRUEF_12
B_20	PROF_12	PRUEF_4	B_44	PROF_17	PRUEF_8
B_21	PROF_12	PRUEF_5	B_45	PROF_17	PRUEF_9
B_22	PROF_12	PRUEF_6	B_46	PROF_17	PRUEF_10
B_23	PROF_12	PRUEF_7	B_47	PROF_17	PRUEF_11
			B_48	PROF_17	PRUEF_12
			48 Ent.	17 Entitäten	12 Entitäten

Tabelle 6.2 zeigt die Zuordnung von Entitäten aus PROFESSOR und PRUEFUNG. Jedes Paar führt zu einer Entität in der Tabelle BEAUFSICHTIGT.

Eine Entität aus PROFESSOR steht in keiner Beziehung mit Entitäten aus PRUEFUNG. Jeweils drei und fünf Entitäten aus PROFESSOR stehen mit genau einer Entität aus PRUEFUNG in Beziehung (1. und 2. Fall). Fünf Entitäten aus PROFESSOR stehen mit drei Entitäten aus PRUEFUNG (3. Fall) und weitere fünf Entitäten aus PROFESSOR schließlich mit fünf Entitäten aus PRUEFUNG (4. Fall) in Beziehung.

Insgesamt werden für die generierten Beziehungen 17 Entitäten aus PROFESSOR, 12 Entitäten aus PRUEFUNG und 48 Entitäten in der assoziativen Tabelle BEAUFSICHTIGT benötigt. 7 Entitäten aus PROFESSOR wurden bereits in Schritt 3 erzeugt, die anderen 10 Entitäten werden nachgeneriert.

Der Algorithmus hat die Generierung für BEAUFSICHTIGT abgeschlossen und setzt die Generierung bei PRUEFUNG fort, da PROFESSOR bereits besucht wurde.

6. Schritt: Generierung für PRUEFUNG

Abbildung 6.20 stellt die Ausgangssituation für den Algorithmus für die Generierung von Daten für PRUEFUNG dar. Die Tabellen RAUM, PROFESSOR und BEAUFSICHTIGT wurden bereits besucht, ebenso die Kanten von PROFESSOR nach RAUM, BEAUFSICHTIGT nach PROFESSOR und BEAUFSICHTIGT nach PRUEFUNG.

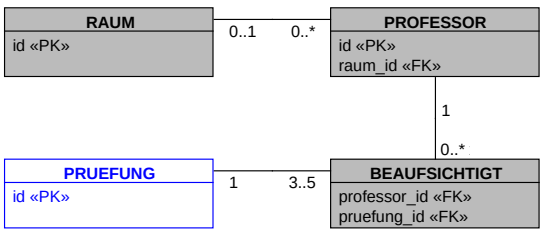

Abbildung 6.20: Schritt: PRUEFUNG

Es gibt keine unbesuchten Kanten in PRUEFUNG. Der Algorithmus kehrt zu PROFESSOR und von dort zu RAUM zurück, wo ebenfalls keine unbesuchten Kanten mehr sind. In der Liste der Tabellen befindet sich auch keine unbesuchten Tabellen mehr, so dass nun der letzte Schritt folgt: Die Erweiterung der Daten sofern notwendig.

7. Schritt: Erweitern der Daten sofern notwendig

In diesem Beispiel wurden einige Entitäten für PROFESSOR in Schritt 5 nachgeneriert. Da diese Entitäten nur eine optionale Beziehung zu einer Entität aus RAUM hat, sind die nachgenerierten Entitäten gültig. Eine Erweiterung der Daten ist somit nicht notwendig und die Generierung abgeschlossen.

6.5 Praktischer Einsatz / Evaluation

To do (5)

6.5.1 Einfluss der Tabellenreihenfolge

Die Art und Weise, wie Beziehungen generiert werden, hängen von der Reihenfolge ab, in der die Tabellen und Kanten behandelt werden. Die gewählte Reihenfolge basiert auf den Kriterien aus [HTW06]. Die Sortierung wird nicht aus Qualitätsgründen, sondern für ein deterministisches Verhalten des Algorithmus beibehalten. Da verschiedene Tabellen gleich viele eingehende Kanten haben können, muss das deterministische Verhalten durch weitere Sortierkritieren sichergestellt werden. Dies kann z.B. der Tabellen-Name sein, oder auch die Reihenfolge, in der die Tabellen definiert worden sind.

Empirische Versuche führten zu der Vermutung, dass die Reihenfolge keinen Einfluss auf die Anzahl der generierten Entitäten hat. Auf einen Beweis dafür wird an dieser Stelle verzichtet, da es weniger wichtig ist, die tatsächlich minimale Anzahl an Entitäten zu generieren. Viel wichtiger ist, dass gültige DataSets generiert werden.

6.6 Konkrete Implementierung und Integration in Toolset

Die Integration des Daten-Generators führt zu generierungsspezifischen Erweiterungen im Datenbank-Modell. So kann für jede Spalte individuell ein Werte-Generator festgelegt werden – ansonsten wird ein Standard-Werte-Generator für den jeweiligen Datentyp verwendet. Listing 6.8 zeigt den Ausschnitt der Modell-Definition des fortlaufenden Beispiels, ergänzt um Daten-generierungsspezifische Anweisungen. Ein vollständiges Modell ist in Anhang A aufgelistet. Im Listing wird lediglich für eine Spalte ein individueller Werte-Generator festgelegt (Zeile 17).

Die Zufallszahlengeneratoren der Werte-Generatoren werden über Seeds initialisiert. Damit der selbe Werte-Generator für verschiedene Spalten standardmäßig unterschiedliche Werte erzeugt, wird der Standard-Spalten-Seed mit Hilfe des Tabellen- und des Spaltennamen berechnet. Es ist außerdem möglich, tabellenspezifische Seeds und ein modellspezifisches Seed festzulegen, die jeweils den Standardwert 0 haben. Das tatsächliche Seed, mit dem die Zufallsgeneratoren initialisiert werden, stellt die Summe aus dem Spalten-Seed, dem Tabellen-Seed und dem Modell-Seed dar.

Auf diese Weise kann über Seeds verhältnismäßig einfach gesteuert werden, dass nur eine Spalte mit neuen Zufallswerten generiert wird (Zeile 19), alle Spalten einer Tabelle (Zeile 11) oder alle Spalten aller Tabellen (Zeile 6). Es ist darüber hinaus auch möglich, zwei Spalten mit selben Zufallswerten zu erzwingen.

Da der Daten-Generator die Anzahl zu erzeugender Entitäten über die Beziehungen bestimmt, würde eine Tabelle ohne eine Beziehung zu anderen Tabellen leer bleiben. Das Modell wird deshalb für jede Tabelle um einen Wert erweitert, der Mindestanzahl der zu generierenden Entitäten enthält (Zeile 12). Der Standardwert dafür ist 1.

Außerdem lässt sich für das Modell festlegen, welcher konkrete Wert anstelle der offenen Grenze $\star$ verwendet werden soll (Zeile 7).

```
1  public HochschuleModel()
2  {
3    database("Hochschule");
4    packageName("com.seitenbau.testing.hochschule.model");
5    enableTableModelClassesGeneration();
6    seed(1);
7    infinite(2);
8
9    Table raum = table("raum")
```

```
10        .description("Die Tabelle mit den Räumen der Hochschule")
11        .seed(3)
12        .minEntities(20)
13        .column("id", DataType.BIGINT)
14          .defaultIdentifier()
15          .autoInvokeNext()
16        .column("gebaeude", DataType.VARCHAR)
17          .generator(new GebaeudeGenerator())
18        .column("nummer", DataType.VARCHAR)
19          .seed(5)
20      .build();
21
22    ...
23  }
```

Listing 6.8: Ausschnitt des für die Daten-Generierung erweiterten Modells

6.7 Offene Punkte

Auch wenn der Algorithmus seine Tauglichkeit gezeigt hat, bietet er einige Möglichkeiten zur Verbesserung. Ein paar Probleme und Herausforderungen werden im Folgenden aufgezeigt.

6.7.1 Abhängigkeiten von Beziehungen

Bereits in Abschnitt 6.3 wurde das Problem angesprochen, dass Beziehungen nicht immer unabhängig voneinander sind. Der Algorithmus in dieser Form trägt diesem Problem nur bei assoziativen Tabellen Sorge.

6.7.2 Abhängigkeiten von Spaltendaten

Die Datengeneratoren für die Spalten arbeiten unabhängig voneinander. Allerdings können Werte in verschiedenen Spalten voneinander abhängen:

- **Vorname** und **Geschlecht**
- **PLZ** und **Ort**
- **Start-** und **Endwerte**, z.B. bei Datumsbereichen, Zeitspannen, Grenzwerten, ...
- Werte, die sich aus Werten anderen Spalten **zusammensetzen**

Außerdem können Spaltenwerte sich auch auf Beziehungen zu anderen Entitäten auswirken:

- **Geschlecht** und **Eltern-Eigenschaft**: männlich/weiblich muss zur Rolle Vater/Mutter passen
- **Fakultät** und **Raum-Nummer**: Das Büro eines Professors hängt von seiner Fakultät ab.
- **Überschneidungen von Lehrveranstaltungen** eines Professors: Ein Professor kann nicht zwei Lehrveranstaltungen lehren, die sich zeitlich überschneiden.

6.7.3 Unerfüllbare Datenbankschemata

Nicht für jedes formal gültige Datenbankschema lassen sich alle Beziehungsfälle generieren. Abbildung 6.21 zeigt ein zyklisches Datenbankschema mit den drei Tabellen A, B und C. Jedes Entität aus A steht genau mit einer Entität aus B und einer Entität aus C in Beziehung. Jede Entität aus B steht mit genau einer Entität aus A und einer Entität aus C in Beziehung. Jede Entität aus C steht mit genau einer Entität aus A in Beziehung und mit einer oder keiner Entität aus B.

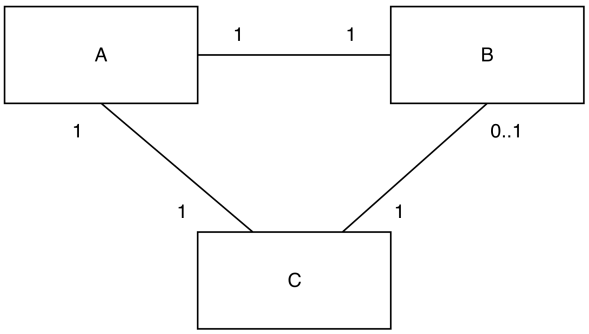

Abbildung 6.21: Formal korrektes aber „unerfüllbares“ Datenbankschema

Aus der Beschreibung geht schon hervor, dass es kein C geben kann, das nicht in Beziehung mit einer Entität aus B steht. Denn eine Entität in C muss mit einer Entität in A in Beziehung stehen, und eine Entität in A schließlich mit einer Entität in B. Und diese Entität in B benötigt eine Entität C für eine Beziehung. Es kommt entweder die Entität aus C in Frage, die nicht mit einer Entität aus B in Beziehung steht, oder es muss eine neue Entität in C generiert werden – dann beginnt der Zyklus allerdings von vorne.

Der Algorithmus würde sich hierbei immer für die zusätzliche Generierung einer weiteren Entität in C entscheiden und schließlich nicht terminieren. Die momentane Implementierung bricht die Generierung allerdings nach Überschreiten eines Grenzwerts für nachträglich generierte Entitäten ab.

Kapitel 7

Zusammenfassung und Ausblick

Das wesentliche Ziel dieser Arbeit war es, eine einfache Sprache zur Modellierung von Testdaten für Datenbank-basierte Anwendungen zu entwickeln. Die entwickelte Modellierungssprache hat dieses Ziel mehr als erreicht. Die Sprache zeichnet sich durch leichte Erlernbarkeit und eine übersichtliche Beschreibung von Beziehungen von Datensätzen aus.

Die Benutzbarkeit der Sprache und der Implementierung war ein Kernmerkmal, das durchgehend betrachtet wurde. Die Sprache wurde als DSL in Groovy realisiert und lässt sich somit im Java-Umfeld nutzen. Vor allem kann sie auch in Unit-Tests verwendet werden. Dadurch sind die Tests und die Testdaten nicht voneinander getrennt, wie es bei Testdaten der Fall wäre, die extern definiert wurden – also in XML-, CSV- oder anderen Dateien.

Die Modellierungssprache wird passend für ein Modell der Datenbank generiert. Dieses Modell beschreibt die Tabellen, deren Spalten und die Beziehungen. Mit diesen Informationen erzeugt ein Generator ein API als Grundgerüst für die Sprache.

Um den Testern die Arbeit mit der DSL zu erleichtern, wurden folgende Features implementiert. So können unterschiedliche DataSets definiert werden, die dieselben Bezeichner für Entitäten verwenden. Diese Bezeichner repräsentieren je nach aktivem DataSet eine andere Entität und damit möglicherweise auch unterschiedliche Daten. Die Lösung unterstützt die Komposition und das Erweitern von DataSets. Bei Fehlern in DataSet-Definitionen erhält der Tester sinnvolle Fehlermeldungen und kann schnell die Ursache finden. Die generierte JavaDoc zu der DSL erleichtert den Umgang mit Hilfe von Code-Beispiele und Informationen zum Datenbank-Modell.

Ein wichtiges Ziel der Arbeit stellte die Generierung von Testdaten dar. Es wurde gezeigt, dass die Generierung zwar kein neues Thema ist, die meisten Werkzeuge aber auf die Erzeugung von Massen-Daten ausgelegt sind. Es gibt Ansätze, die sich mit der Generierung überschaubarer Daten befassen. Diese arbeiten meistens Abfrage-orientiert, d.h. sie erzeugen die Daten auf Basis einer konkreten Datenbankabfrage. Es wurde ein Algorithmus entwickelt, der die Testdaten anhand der definierten Beziehungstypen im Datenbank-Modell generiert. Die Anzahl der zu generierenden Entitäten leitet sich aus den Beziehungen ab, an denen der Entitätstyp beteiligt ist.

Der Algorithmus wurde implementiert und die Nutzbarkeit anhand praktischer Anwendungen überprüft. Die erzeugten Daten entsprachen den Erwartungen und passten zum Datenbank-Schema.

Die Lösung unterstützt einige besondere Beziehungstypen nicht direkt. So fehlt die Unterstützung für reflexive Beziehungen und teilweise auch für zirkuläre Beziehungen. Die Ursache dafür liegt in den Klassen zum Beschreiben des Datenbank-Modells.

Das Thema Generierung bietet Potential für weiterführende Arbeiten. So betrachtet der Algorithmus keine Abhängigkeiten zwischen unterschiedlichen Beziehungen, mit Ausnahme von assoziativen Tabellen. Außerdem arbeiten die Datengeneratoren für die Spaltenwerte noch komplett unabhängig voneinander bzw. von der Entität. Es wäre sinnvoll, den Generator dahingehen zu erweitern, dass er Abhängigkeiten von Beziehungen und auch von Daten berücksichtigt.

To do (6)

Abbildungsverzeichnis

2.1 Back Door Manipulation 4
2.2 Stil von ER-Diagrammen 5
2.3 Stil relatianaler Datenbank-Diagramme nach Ambler 6
2.4 Modell-Beschreibung 7
2.5 Logisches Klassendiagramm der STU-DataSet-Klassen 8

3.1 ER-Diagramm des fortlaufenden Beispiels 13
3.2 Relationales Datenbank-Schema des fortlaufenden Beispiels 13

4.1 Grafische Darstellung der EBNF für Tabellen 32
4.2 Grafische Darstellung der EBNF für Beziehungen 32

5.1 Architektur 35
5.2 Klassendiagramm der DataSet-Builder 40
5.3 Beispiel JavaDoc-Tooltip 48

6.1 Beziehungen nach dem Schema 1..1:1..1 56
6.2 Beziehungen nach dem Schema 0..1:1..1 56
6.3 Beziehungen nach dem Schema 0..1:0..1 56
6.4 Beziehungen nach dem Schema 1..1:1..n 57
6.5 Beziehungen nach dem Schema 0..1:1..n 57
6.6 Beziehungen nach dem Schema 1..1:0..n 57
6.7 Beziehungen nach dem Schema 0..1:0..n 58
6.8 Beziehungen nach dem Schema 1:0..1 (n:m) 58
6.9 Beziehungen nach dem Schema 1:4 (n:M) 59
6.10 Beziehungen nach dem Schema 3:0..1 (N:m) 59
6.11 Beziehungen nach dem Schema 3:4 (N:M) 59
6.12 Beziehungen nach dem Schema 1..3:0..4 (n..N:m..M) 60
6.13 Diagramm der Generator-Modell-Klassen 62

6.14 Parameter für Daten-Generierung bei assoziativer Tabelle 65
6.15 Einfaches Beispiel-Modell für den Algorithmus 67
6.16 Kanten des einfachen Beispiel-Modells 68
6.17 Schritt: RAUM 68
6.18 Schritt: PROFESSOR 69
6.19 Schritt: BEAUFSICHTIGT 70
6.20 Schritt: PRUEFUNG 71
6.21 Formal korrektes aber „unerfüllbares“ Datenbankschema 74

Listings

3.1 XML-DataSet 15
3.2 Flat-XML-DataSet 16
3.3 Default-DataSet 18
3.4 STU DataSet (1) 19
3.5 STU DataSet (2) 21
4.1 Beispiel zum Entwurf 1 24
4.2 Beispiel zum Entwurf 2 25
4.3 Beispiel zum Entwurf 3 26
4.4 Vereinfachung von Ausdrücken in Groovy 27
4.5 Tabellen-Parser Grundgerüst mit Operator-Überladen 28
4.6 Tabellen-Parser Grundgerüst mit Operator-Überladen 29
4.7 DSL-Entwurf 3 für Laufzeit-Meta-Programmierung angepasst 30
4.8 EBNF für Tabellen 31
4.9 EBNF für Beziehungen 32
4.10 DataSet modelliert mit dem Table Builder API 32
4.11 Beziehungen innerhalb von Tabellen 34
5.1 Beispiel für unveränderliche Identifikatoren 37
5.2 Beispiel alte *STU*-Builder 38
5.3 Beispiel neue *STU*-Builder 39
5.4 Beispiel für assoziative Tabelle 39
5.5 Binden von Referenzen (Table Builder API) 42
5.6 Binden von Referenzen (Fluent Builder API) 42
5.7 Zugriff auf Werte über Referenzen 42
5.8 Definition von Beziehungen über Referenzen 42
5.9 JUnit-Tests (reiner Java-Code) 43
5.10 Test-Methode in Groovy 44
5.11 Erweitertes DataSet 44

5.12 Erweitertes DataSet 45
5.13 Test auf erweiterem DataSet 45
5.14 Beispiel Lazy Valunes 46
5.15 Such-Beispiele 46
5.16 Beispiel für find 47
5.17 Beispiel für foreach 47
5.18 Reflexive Beziehungen manuell 49
5.19 Reflexive Beziehungen mit Hilfe assoziativer Tabelle 49
5.20 Zirkuläre Beziehungen manuell 50
5.21 Zirkuläre Beziehungen mit assoziativer Tabelle 50
5.22 Höhergradige Beziehungen mit zusätzlicer Tabelle 50
5.23 Löschen von Zeilen 51
6.1 Generiere Test-Daten 62
6.2 GeneriereDatenFuerNichtassoziativeTabelle 63
6.3 Generiere Daten Fuer Kante 63
6.4 Generiere Entitäten Und Beziehungen 64
6.5 Generiere Daten für assoziative Tabelle 64
6.6 Generiere Assoziative Entitaeten Und Beziehungen 66
6.7 ErweitereGenerierteDatenZuKonsistentenDaten 67
6.8 Ausschnitt des für die Daten-Generierung erweiterten Modells 72
A.1 Das verwendete Datenbank-Modell 83
B.1 Generierte DSL 85

Literatur

[AS06] Scott W. Ambler und Pramod J. Sadalage. *Refactoring Databases, Evolutionary Database Design*. The Addison-Wesley Signature Series. Addison-Wesley, 2006. ISBN: 978-0-3212-9353-4. URL: http://www.addison-wesley.de/main/main.asp?page=aktionen/bookdetails&ProductID=108888.

[BKL07] Carsten Binnig, Donald Kossmann und Eric Lo. „Reverse Query Processing“. In: *ICDE*. Hrsg. von Rada Chirkova u. a. IEEE, 2007, S. 506–515.

[Bec96] Kent Beck. *Smalltalk Best Practice Patterns*. Prentice Hall, 1996. ISBN: 9780134769042. URL: http://www.pearson.ch/1471/9780134769042/Smalltalk-Best-Practice-Patterns.aspx.

[Blo08] Joshua Bloch. *Effective Java Second Edition*. The Java Series. Addison-Wesley, 2008. ISBN: 9780321356680. URL: http://books.google.com/books?id=ka2VUBqHiWkC&pg.

[Cha+04] David Chays u. a. „An AGENDA for testing relational database applications“. In: *Software Testing, Verification and Reliability*. 2004, S. 2004.

[Com13] Google Guava Community. *Optional (Guava: Google Core Libraries for Java 15.0-SNAPSHOT API)*. Google Guava Community. 2013. URL: http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/base/Optional.html (besucht am 19. 07. 2013).

[Fou04] The Apache Software Foundation. *Apache License, Version 2.0*. The Apache Software Foundation. 2004. URL: http://www.apache.org/licenses/LICENSE-2.0.html (besucht am 07. 05. 2013).

[Fow10] Martin Fowler. *Domain-Specific Languages*. The Addison-Wesley Signature Series. Addison-Wesley, 2010. ISBN: 978-0321712943. URL: http://martinfowler.com/books/dsl.html.

[Gam+94] Erich Gamma u. a. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. ISBN: 978-0-2016-3361-0. URL: http://www.addison-wesley.de/9780201633610.html.

[Gho10] Debasish Ghosh. *DSLs in Action*. Manning, 2010. ISBN: 978-1-935182-45-0. URL: http://www.manning.com/ghosh/.

[Goe09] Brian Goetz. *Java concurrency in practice*. 7. print. Addison-Wesley, 2009. ISBN: 978-0-321-34960-6. URL: http://www.gbv.de/dms/ilmenau/toc/601225643.PDF.

[HM05] Rob Harrop und Jan Machacek. *Pro Spring*. Apress, 2005. ISBN: 978-1-59059-461-2. URL: http://books.google.de/books/about/Pro_Spring.html?id=q2PIjAJoXsAC&redir_esc=y.

[HTW06] Kenneth Houkjaer, Kristian Torp und Rico Wind. „Simple and realistic data generation“. In: *Proceedings of the 32nd international conference on Very large data bases*. VLDB '06. Seoul, Korea: VLDB Endowment, 2006, S. 1243–1246. URL: http://dl.acm.org/citation.cfm?id=1182635.1164254.

[Kön07] Dierk König. *Groovy im Einsatz*. Fachbuchverl. Leipzig im Carl-Hanser-Verl., 2007. ISBN: 978-3-446-41238-5. URL: http://deposit.d-nb.de/cgi-bin/dokserv?id=2948820&prov=M&dok_var=1&dok_ext=htm.

[LK12] Guillaume Laforge und Paul King. *Groovy DSLs, from Beginner to Expert*. SpringSource. 2012. URL: http://de.slideshare.net/glaforge/groovy-dsls-from-beginner-to-expert-guillaume-laforge-and-paul-king-springone2gx-2011 (besucht am 21. 05. 2013).

[Mar09] Robert C. Martin. *Clean Code: Refactoring, Patterns, Testen und Techniken für sauberen Code*. mitp-Verlag, 2009. ISBN: 978-3-8266-5548-7. URL: http://www.it-fachportal.de/shop/buch/Clean%20Code%20-%20Refactoring,%20Patterns,%20Testen%20und%20Techniken%20f%C3%BCr%20sauberen%20Code/detail.html,b164659.

[Mes07] Gerard Meszaros. *XUnit Test Patterns: Refactoring Test Code*. The Addison-Wesley Signature Series. Addison-Wesley, 2007. ISBN: 978-0-13-149505-0. URL: http://xunitpatterns.com/index.html.

[Mof] *Meta Object Facility (MOF) Specification 1.4.1*. 2005. URL: http://www.omg.org/spec/MOF/ISO/19502/PDF/ (besucht am 17. 07. 2013).

[Nie+12] Peter Niederwieser u. a. *spock - the enterprise ready specification framework*. Peter Niederwieser u. a. 2012. URL: http://spockframework.org/ (besucht am 07. 05. 2013).

[RC12] Ali Raza und Stephen Clyde. *Creating Datasets for Testing Relational Databases*. LAP Lambert Academic Publishing, 2012. ISBN: 9783847337638.

[SL10] Andreas Spillner und Tilo Linz. *Basiswissen Softwaretest*. dpunkt.verlag, 2010. ISBN: 978-3-89864-642-0. URL: http://www.dpunkt.de/buecher/4075.html.

Anhang A

Modell

```
package model;

import com.seitenbau.testing.dbunit.generator.DataType;
import com.seitenbau.testing.dbunit.generator.DatabaseModel;
import com.seitenbau.testing.dbunit.generator.Table;
import com.seitenbau.testing.dbunit.generator.values.DateGenerator;
import com.seitenbau.testing.dbunit.generator.values.IntegerGenerator;
import com.seitenbau.testing.dbunit.generator.values.NachnameGenerator;
import com.seitenbau.testing.dbunit.generator.values.VornameGenerator;

public class HochschuleModel extends DatabaseModel
{
  public HochschuleModel()
  {
    database("Hochschule");
    packageName("com.seitenbau.testing.hochschule.model");
    enableTableModelClassesGeneration();
    seed(0);
    infinite(2);

    Table raum = table("raum")
        .description("Die Tabelle mit den Räumen der Hochschule")
        .seed(0)
        .minEntities(20)
        .column("id", DataType.BIGINT)
          .defaultIdentifier()
          .autoInvokeNext()
        .column("gebaude", DataType.VARCHAR)
        .column("nummer", DataType.VARCHAR)
      .build();

    Table professoren = table("professor")
        .description("Die Tabelle mit den Professoren der Hochschule")
        .column("id", DataType.BIGINT)
          .defaultIdentifier()
          .autoInvokeNext()
        .column("name", DataType.VARCHAR)
          .generator(new NachnameGenerator())
        .column("vorname", DataType.VARCHAR)
          .generator(new VornameGenerator())
        .column("titel", DataType.VARCHAR)
        .column("fakultaet", DataType.VARCHAR)
        .column("raum_id", DataType.BIGINT)
          .reference
            .local
              .name("hatRaum")
            .foreign(raum)
              .name("gehoert")
              .multiplicity("0..1")
      .build();

    Table lehrveranstaltungen = table("lehrveranstaltung")
        .description("Die Tabelle mit den Lehrveranstaltungen der Hochschule")
        .column("id", DataType.BIGINT)
          .defaultIdentifier()
          .autoInvokeNext()
        .column("professor_id", DataType.BIGINT)
          .reference
            .local
              .name("geleitetVon")
              .description("Gibt an, von welchem Professor eine Lehrveranstaltung geleitet wird.")
            .foreign(professoren)
              .name("leitet")
              .description("Gibt an, welche Lehrveranstaltungen ein Professor leitet.")
              .multiplicity("0..*")
        .column("name", DataType.VARCHAR)
        .column("sws", DataType.INTEGER)
          .generator(new IntegerGenerator(2, 6))
        .column("ects", DataType.DOUBLE)
      .build();

    Table pruefungen = table("pruefung")
        .description("Die Tabelle mit den Prüfungen der Hochschule")
        .column("id", DataType.BIGINT)
          .defaultIdentifier()
```

```
76            .autoInvokeNext()
77          .column("lehrveranstaltung_id", DataType.BIGINT)
78            .reference
79              .local
80                .name("stoffVon")
81                .multiplicity("1")
82                .description("Gibt␣an,␣zu␣welcher␣Lehrvanstaltung␣eine␣Prüfung␣gehört.")
83              .foreign(lehrveranstaltungen)
84                .name("hatPruefung")
85                .multiplicity("0..*")
86                .description("Ordnet␣Prüfungen␣einer␣Lehrveranstaltung␣zu.")
87          .column("typ", DataType.VARCHAR)
88          .column("zeitpunkt", DataType.DATE)
89            .generator(new DateGenerator())
90        .build();
91
92      Table studenten = table("student")
93          .description("Die␣Tabelle␣mit␣den␣immatrikulierten␣Studenten␣der␣Hochschule")
94          .column("matrikelnummer", DataType.BIGINT)
95            .defaultIdentifier()
96            .autoInvokeNext()
97          .column("name", DataType.VARCHAR)
98            .generator(new NachnameGenerator())
99          .column("vorname", DataType.VARCHAR)
100           .generator(new VornameGenerator())
101         .column("studiengang", DataType.VARCHAR)
102         .column("semester", DataType.INTEGER)
103           .generator(new IntegerGenerator(1, 8))
104         .column("immatrikuliert_seit", DataType.DATE)
105           .generator(new DateGenerator())
106       .build();
107
108     associativeTable("beaufsichtigt")
109         .column("professor_id", DataType.BIGINT)
110           .reference
111             .foreign(professoren)
112               .name("beaufsichtigt")
113               .description("Gibt␣an,␣welche␣Prüfungen␣ein␣Professor␣beaufsichtigt.")
114               .multiplicity("0..*")
115         .column("pruefung_id", DataType.BIGINT)
116           .reference
117             .foreign(pruefungen)
118               .name("beaufsichtigtVon")
119               .description("Gibt␣an,␣welche␣Professoren␣eine␣Prüfung␣beaufsichtigen.")
120               .multiplicity("0..*")
121       .build();
122
123     associativeTable("besucht")
124         .column("student_id", DataType.BIGINT)
125           .reference
126             .foreign(studenten)
127               .name("besucht")
128               .multiplicity("0..*")
129               .description("Gibt␣an,␣welche␣Lehrveranstaltungen␣ein␣Student␣besucht.")
130         .column("lehrveranstaltung_id", DataType.BIGINT)
131           .reference
132             .foreign(lehrveranstaltungen)
133               .name("besuchtVon")
134               .multiplicity("3..10")
135               .description("Gibt␣an,␣welche␣Studenten␣eine␣Lehrveranstaltung␣besuchen.")
136       .build();
137
138     associativeTable("isttutor")
139         .column("student_id", DataType.BIGINT)
140           .reference
141             .foreign(studenten)
142               .name("istTutor")
143               .multiplicity("0..*")
144               .description("Gibt␣an,␣bei␣welchen␣Lehrveranstaltungen␣ein␣Student␣Tutor␣ist.")
145         .column("lehrveranstaltung_id", DataType.BIGINT)
146           .reference
147             .foreign(lehrveranstaltungen)
148               .name("hatTutor")
149               .multiplicity("0..*")
150               .description("Gibt␣an,␣welche␣Tutoren␣eine␣Lehrveranstaltung␣hat.")
151       .build();
152
153     associativeTable("schreibt")
154         .column("student_id", DataType.BIGINT)
155           .reference
156             .foreign(studenten)
157               .name("schreibt")
158               .multiplicity("0..*")
159               .description("Gibt␣an,␣welche␣Prüfungen␣ein␣Student␣schreibt.")
160         .column("pruefung_id", DataType.BIGINT)
161           .reference
162             .foreign(pruefungen)
163               .name("geschriebenVon")
164               .multiplicity("0..*")
165               .description("Gibt␣an,␣welche␣Studenten␣eine␣Prüfung␣schreiben.")
166         .column("versuch", DataType.INTEGER)
167           .generator(new IntegerGenerator(1, 3))
168       .build();
169   }
170
171 }
```

Listing A.1: Das verwendete Datenbank-Modell

Anhang B

Generierte DSL

```
package dataset

import com.seitenbau.testing.hochschule.model.*

class DataSet extends HochschuleBuilder
{

  def tables() {

    raumTable.rows() {
      REF     | gebaude | nummer
      RAUM_1  | "Delta" | "dolor"
      RAUM_2  | "Maus"  | "Lorem"
      RAUM_3  | "amet"  | "amet"
      RAUM_4  | "Delta" | "Lorem"
      RAUM_5  | "ipsum" | "Beta"
      RAUM_6  | "Katze" | "dolor"
      RAUM_7  | "Beta"  | "Alpha"
      RAUM_8  | "Hund"  | "Alpha"
      RAUM_9  | "Gamma" | "Lorem"
      RAUM_10 | "ipsum" | "Hund"
      RAUM_11 | "Maus"  | "amet"
      RAUM_12 | "amet"  | "Alpha"
      RAUM_13 | "amet"  | "amet"
    }

    professorTable.rows() {
      REF          | name         | vorname    | titel   | fakultaet | raum_id
      PROFESSOR_1  | "Fischer"    | "Thomas"   | "Katze" | "Maus"    | RAUM_2
      PROFESSOR_2  | "Fischer"    | "Stefan"   | "Delta" | "Beta"    | RAUM_3
      PROFESSOR_3  | "Fischer"    | "Claudia"  | "ipsum" | "amet"    | RAUM_4
      PROFESSOR_4  | "Schneider"  | "Angelika" | "sit"   | "Lorem"   | RAUM_5
      PROFESSOR_5  | "Schmidt"    | "Simone"   | "Lorem" | "Gamma"   | RAUM_6
      PROFESSOR_6  | "Meyer"      | "Brigitte" | "sit"   | "Maus"    | RAUM_7
      PROFESSOR_7  | "Fischer"    | "Bettina"  | "Gamma" | "amet"    | RAUM_8
      PROFESSOR_8  | "Schneider"  | "Thorsten" | "Lorem" | "ipsum"   | RAUM_9
      PROFESSOR_9  | "Mustermann" | "Markus"   | "ipsum" | "sit"     | RAUM_10
      PROFESSOR_10 | "Müller"     | "Elke"     | "Lorem" | "Lorem"   | RAUM_11
      PROFESSOR_11 | "Mustermann" | "Heike"    | "ipsum" | "dolor"   | RAUM_12
      PROFESSOR_12 | "Müller"     | "Markus"   | "sit"   | "amet"    | RAUM_13
    }

    raum2Table.rows() {
      REF      | professor_id
      RAUM2_1  | _
      RAUM2_2  | PROFESSOR_1
      RAUM2_3  | PROFESSOR_2
      RAUM2_4  | PROFESSOR_3
      RAUM2_5  | PROFESSOR_4
      RAUM2_6  | PROFESSOR_5
      RAUM2_7  | PROFESSOR_6
      RAUM2_8  | PROFESSOR_7
      RAUM2_9  | PROFESSOR_8
      RAUM2_10 | PROFESSOR_9
      RAUM2_11 | PROFESSOR_10
      RAUM2_12 | PROFESSOR_11
      RAUM2_13 | PROFESSOR_12
    }

    lehrveranstaltungTable.rows() {
      REF                  | professor_id | name    | sws | ects
      LEHRVERANSTALTUNG_1  | PROFESSOR_2  | "Alpha" | 6   | "amet"
      LEHRVERANSTALTUNG_2  | PROFESSOR_3  | "Hund"  | 2   | "Gamma"
      LEHRVERANSTALTUNG_3  | PROFESSOR_3  | "Katze" | 4   | "ipsum"
      LEHRVERANSTALTUNG_4  | PROFESSOR_3  | "dolor" | 6   | "Delta"
      LEHRVERANSTALTUNG_5  | PROFESSOR_3  | "Lorem" | 3   | "Beta"
      LEHRVERANSTALTUNG_6  | PROFESSOR_3  | "Gamma" | 6   | "Gamma"
      LEHRVERANSTALTUNG_7  | PROFESSOR_4  | "Gamma" | 5   | "Gamma"
      LEHRVERANSTALTUNG_8  | PROFESSOR_4  | "Alpha" | 5   | "dolor"
      LEHRVERANSTALTUNG_9  | PROFESSOR_4  | "sit"   | 4   | "sit"
      LEHRVERANSTALTUNG_10 | PROFESSOR_4  | "Delta" | 4   | "Katze"
      LEHRVERANSTALTUNG_11 | PROFESSOR_4  | "amet"  | 5   | "ipsum"
      LEHRVERANSTALTUNG_12 | PROFESSOR_4  | "sit"   | 4   | "Delta"
    }

```

```
pruefungTable.rows() {
  REF          | lehrveranstaltung_id | typ     | zeitpunkt
  PRUEFUNG_1   | LEHRVERANSTALTUNG_2  | "sit"   | "12.01.1964"
  PRUEFUNG_2   | LEHRVERANSTALTUNG_3  | "Hund"  | "16.05.1911"
  PRUEFUNG_3   | LEHRVERANSTALTUNG_3  | "Delta" | "24.01.1971"
  PRUEFUNG_4   | LEHRVERANSTALTUNG_3  | "Lorem" | "29.06.1969"
  PRUEFUNG_5   | LEHRVERANSTALTUNG_3  | "Lorem" | "09.01.1958"
  PRUEFUNG_6   | LEHRVERANSTALTUNG_3  | "Alpha" | "12.01.1946"
  PRUEFUNG_7   | LEHRVERANSTALTUNG_4  | "Alpha" | "17.10.1944"
  PRUEFUNG_8   | LEHRVERANSTALTUNG_4  | "Alpha" | "02.06.2005"
  PRUEFUNG_9   | LEHRVERANSTALTUNG_4  | "Lorem" | "02.04.1986"
  PRUEFUNG_10  | LEHRVERANSTALTUNG_4  | "dolor" | "13.11.1921"
  PRUEFUNG_11  | LEHRVERANSTALTUNG_4  | "Alpha" | "15.12.2010"
  PRUEFUNG_12  | LEHRVERANSTALTUNG_4  | "ipsum" | "22.05.1939"
}

studentTable.rows() {
  REF         | name          | vorname    | studiengang | semester | immatrikuliert_seit
  STUDENT_1   | "Meyer"       | "Manuela"  | "Katze"     | 8        | "11.04.1952"
  STUDENT_2   | "Mustermann"  | "Manuela"  | "Alpha"     | 3        | "09.09.1954"
  STUDENT_3   | "Fischer"     | "Brigitte" | "Lorem"     | 4        | "21.01.1901"
  STUDENT_4   | "Meyer"       | "Silke"    | "ipsum"     | 8        | "21.05.1945"
  STUDENT_5   | "Müller"      | "Elke"     | "Alpha"     | 2        | "28.06.1939"
  STUDENT_6   | "Fischer"     | "Angelika" | "ipsum"     | 2        | "13.07.1955"
  STUDENT_7   | "Meyer"       | "Claudia"  | "Lorem"     | 1        | "01.04.1972"
  STUDENT_8   | "Müller"      | "Stefan"   | "Beta"      | 3        | "06.06.1925"
  STUDENT_9   | "Meyer"       | "Angelika" | "ipsum"     | 2        | "20.03.2008"
  STUDENT_10  | "Meyer"       | "Silke"    | "Lorem"     | 3        | "25.05.1907"
  STUDENT_11  | "Fischer"     | "Stefan"   | "sit"       | 7        | "02.12.1984"
  STUDENT_12  | "Schneider"   | "Simone"   | "ipsum"     | 6        | "13.11.1972"
  STUDENT_13  | "Schneider"   | "Thomas"   | "Beta"      | 5        | "10.05.1945"
  STUDENT_14  | "Mustermann"  | "Stefan"   | "Lorem"     | 4        | "19.02.1983"
  STUDENT_15  | "Fischer"     | "Brigitte" | "Alpha"     | 3        | "19.04.1953"
  STUDENT_16  | "Schneider"   | "Thorsten" | "Delta"     | 4        | "28.05.1989"
  STUDENT_17  | "Mustermann"  | "Claudia"  | "Katze"     | 6        | "19.05.1915"
  STUDENT_18  | "Mustermann"  | "Stefan"   | "Beta"      | 2        | "30.06.1934"
  STUDENT_19  | "Schneider"   | "Angelika" | "Delta"     | 4        | "09.08.1948"
  STUDENT_20  | "Müller"      | "Bettina"  | "Hund"      | 4        | "16.11.1929"
  STUDENT_21  | "Schmidt"     | "Simone"   | "dolor"     | 8        | "03.08.1962"
  STUDENT_22  | "Fischer"     | "Thomas"   | "Lorem"     | 6        | "13.04.1939"
  STUDENT_23  | "Meyer"       | "Andreas"  | "Alpha"     | 7        | "17.02.1966"
  STUDENT_24  | "Fischer"     | "Brigitte" | "dolor"     | 2        | "06.12.1924"
  STUDENT_25  | "Schneider"   | "Markus"   | "sit"       | 8        | "28.09.1977"
  STUDENT_26  | "Schneider"   | "Claudia"  | "Lorem"     | 4        | "13.10.1927"
  STUDENT_27  | "Mustermann"  | "Simone"   | "Maus"      | 8        | "27.02.1959"
  STUDENT_28  | "Müller"      | "Bettina"  | "Alpha"     | 8        | "27.09.1952"
  STUDENT_29  | "Mustermann"  | "Stefan"   | "Beta"      | 6        | "03.10.1987"
  STUDENT_30  | "Schmidt"     | "Thomas"   | "Katze"     | 2        | "19.05.1915"
  STUDENT_31  | "Meyer"       | "Stefan"   | "amet"      | 8        | "02.10.1937"
  STUDENT_32  | "Schmidt"     | "Claudia"  | "Delta"     | 7        | "25.10.1998"
  STUDENT_33  | "Schneider"   | "Heike"    | "amet"      | 7        | "22.12.1901"
  STUDENT_34  | "Mustermann"  | "Elke"     | "Hund"      | 2        | "02.12.1984"
  STUDENT_35  | "Mustermann"  | "Thomas"   | "Hund"      | 8        | "27.07.1950"
  STUDENT_36  | "Meyer"       | "Stefan"   | "amet"      | 7        | "19.03.1949"
  STUDENT_37  | "Meyer"       | "Heike"    | "Gamma"     | 1        | "18.09.1905"
  STUDENT_38  | "Müller"      | "Heike"    | "sit"       | 6        | "23.08.1941"
  STUDENT_39  | "Mustermann"  | "Brigitte" | "Delta"     | 2        | "08.09.1981"
  STUDENT_40  | "Meyer"       | "Bettina"  | "ipsum"     | 1        | "23.10.1915"
  STUDENT_41  | "Meyer"       | "Simone"   | "Hund"      | 6        | "07.03.1929"
  STUDENT_42  | "Meyer"       | "Thomas"   | "ipsum"     | 6        | "23.03.1973"
  STUDENT_43  | "Mustermann"  | "Brigitte" | "Gamma"     | 4        | "15.10.1988"
  STUDENT_44  | "Schmidt"     | "Silke"    | "Katze"     | 2        | "22.11.1975"
  STUDENT_45  | "Schmidt"     | "Thomas"   | "Gamma"     | 4        | "17.07.1971"
  STUDENT_46  | "Fischer"     | "Manuela"  | "dolor"     | 4        | "15.06.1956"
  STUDENT_47  | "Schmidt"     | "Markus"   | "Beta"      | 6        | "27.05.1919"
  STUDENT_48  | "Schmidt"     | "Elke"     | "Alpha"     | 7        | "25.10.1952"
  STUDENT_49  | "Müller"      | "Heike"    | "Beta"      | 6        | "07.01.1956"
  STUDENT_50  | "Schneider"   | "Thorsten" | "Lorem"     | 5        | "11.04.2002"
  STUDENT_51  | "Meyer"       | "Simone"   | "Beta"      | 2        | "31.12.1965"
  STUDENT_52  | "Mustermann"  | "Bettina"  | "Hund"      | 4        | "15.06.1981"
}

beaufsichtigtTable.rows() {
  professor_id | pruefung_id
  PROFESSOR_1  | PRUEFUNG_1
  PROFESSOR_2  | PRUEFUNG_2
  PROFESSOR_2  | PRUEFUNG_3
  PROFESSOR_3  | PRUEFUNG_4
  PROFESSOR_4  | PRUEFUNG_4
  PROFESSOR_5  | PRUEFUNG_5
  PROFESSOR_5  | PRUEFUNG_6
  PROFESSOR_6  | PRUEFUNG_5
  PROFESSOR_6  | PRUEFUNG_6
  PROFESSOR_7  | PRUEFUNG_7
  PROFESSOR_8  | PRUEFUNG_8
  PROFESSOR_8  | PRUEFUNG_9
  PROFESSOR_9  | PRUEFUNG_10
  PROFESSOR_10 | PRUEFUNG_10
  PROFESSOR_11 | PRUEFUNG_11
  PROFESSOR_11 | PRUEFUNG_12
  PROFESSOR_12 | PRUEFUNG_11
  PROFESSOR_12 | PRUEFUNG_12
}

besuchtTable.rows() {
  student_id | lehrveranstaltung_id
  STUDENT_1  | LEHRVERANSTALTUNG_1
  STUDENT_2  | LEHRVERANSTALTUNG_1
  STUDENT_3  | LEHRVERANSTALTUNG_1
  STUDENT_4  | LEHRVERANSTALTUNG_2
  STUDENT_4  | LEHRVERANSTALTUNG_3
  STUDENT_5  | LEHRVERANSTALTUNG_2
  STUDENT_5  | LEHRVERANSTALTUNG_3
  STUDENT_6  | LEHRVERANSTALTUNG_2
```

```
      STUDENT_6  | LEHRVERANSTALTUNG_3
      STUDENT_7  | LEHRVERANSTALTUNG_4
      STUDENT_8  | LEHRVERANSTALTUNG_4
      STUDENT_9  | LEHRVERANSTALTUNG_4
      STUDENT_10 | LEHRVERANSTALTUNG_4
      STUDENT_11 | LEHRVERANSTALTUNG_4
      STUDENT_12 | LEHRVERANSTALTUNG_4
      STUDENT_13 | LEHRVERANSTALTUNG_4
      STUDENT_14 | LEHRVERANSTALTUNG_4
      STUDENT_15 | LEHRVERANSTALTUNG_4
      STUDENT_16 | LEHRVERANSTALTUNG_4
      STUDENT_17 | LEHRVERANSTALTUNG_5
      STUDENT_17 | LEHRVERANSTALTUNG_6
      STUDENT_18 | LEHRVERANSTALTUNG_5
      STUDENT_18 | LEHRVERANSTALTUNG_6
      STUDENT_19 | LEHRVERANSTALTUNG_5
      STUDENT_19 | LEHRVERANSTALTUNG_6
      STUDENT_20 | LEHRVERANSTALTUNG_5
      STUDENT_20 | LEHRVERANSTALTUNG_6
      STUDENT_21 | LEHRVERANSTALTUNG_5
      STUDENT_21 | LEHRVERANSTALTUNG_6
      STUDENT_22 | LEHRVERANSTALTUNG_5
      STUDENT_22 | LEHRVERANSTALTUNG_6
      STUDENT_23 | LEHRVERANSTALTUNG_5
      STUDENT_23 | LEHRVERANSTALTUNG_6
      STUDENT_24 | LEHRVERANSTALTUNG_5
      STUDENT_24 | LEHRVERANSTALTUNG_6
      STUDENT_25 | LEHRVERANSTALTUNG_5
      STUDENT_25 | LEHRVERANSTALTUNG_6
      STUDENT_26 | LEHRVERANSTALTUNG_5
      STUDENT_26 | LEHRVERANSTALTUNG_6
      STUDENT_27 | LEHRVERANSTALTUNG_7
      STUDENT_28 | LEHRVERANSTALTUNG_7
      STUDENT_29 | LEHRVERANSTALTUNG_7
      STUDENT_30 | LEHRVERANSTALTUNG_8
      STUDENT_30 | LEHRVERANSTALTUNG_9
      STUDENT_31 | LEHRVERANSTALTUNG_8
      STUDENT_31 | LEHRVERANSTALTUNG_9
      STUDENT_32 | LEHRVERANSTALTUNG_8
      STUDENT_32 | LEHRVERANSTALTUNG_9
      STUDENT_33 | LEHRVERANSTALTUNG_10
      STUDENT_34 | LEHRVERANSTALTUNG_10
      STUDENT_35 | LEHRVERANSTALTUNG_10
      STUDENT_36 | LEHRVERANSTALTUNG_10
      STUDENT_37 | LEHRVERANSTALTUNG_10
      STUDENT_38 | LEHRVERANSTALTUNG_10
      STUDENT_39 | LEHRVERANSTALTUNG_10
      STUDENT_40 | LEHRVERANSTALTUNG_10
      STUDENT_41 | LEHRVERANSTALTUNG_10
      STUDENT_42 | LEHRVERANSTALTUNG_10
      STUDENT_43 | LEHRVERANSTALTUNG_11
      STUDENT_43 | LEHRVERANSTALTUNG_12
      STUDENT_44 | LEHRVERANSTALTUNG_11
      STUDENT_44 | LEHRVERANSTALTUNG_12
      STUDENT_45 | LEHRVERANSTALTUNG_11
      STUDENT_45 | LEHRVERANSTALTUNG_12
      STUDENT_46 | LEHRVERANSTALTUNG_11
      STUDENT_46 | LEHRVERANSTALTUNG_12
      STUDENT_47 | LEHRVERANSTALTUNG_11
      STUDENT_47 | LEHRVERANSTALTUNG_12
      STUDENT_48 | LEHRVERANSTALTUNG_11
      STUDENT_48 | LEHRVERANSTALTUNG_12
      STUDENT_49 | LEHRVERANSTALTUNG_11
      STUDENT_49 | LEHRVERANSTALTUNG_12
      STUDENT_50 | LEHRVERANSTALTUNG_11
      STUDENT_50 | LEHRVERANSTALTUNG_12
      STUDENT_51 | LEHRVERANSTALTUNG_11
      STUDENT_51 | LEHRVERANSTALTUNG_12
      STUDENT_52 | LEHRVERANSTALTUNG_11
      STUDENT_52 | LEHRVERANSTALTUNG_12
    }

    isttutorTable.rows() {
      student_id | lehrveranstaltung_id
      STUDENT_1  | LEHRVERANSTALTUNG_1
      STUDENT_2  | LEHRVERANSTALTUNG_2
      STUDENT_2  | LEHRVERANSTALTUNG_3
      STUDENT_3  | LEHRVERANSTALTUNG_4
      STUDENT_4  | LEHRVERANSTALTUNG_4
      STUDENT_5  | LEHRVERANSTALTUNG_5
      STUDENT_5  | LEHRVERANSTALTUNG_6
      STUDENT_6  | LEHRVERANSTALTUNG_5
      STUDENT_6  | LEHRVERANSTALTUNG_6
      STUDENT_7  | LEHRVERANSTALTUNG_7
      STUDENT_8  | LEHRVERANSTALTUNG_8
      STUDENT_8  | LEHRVERANSTALTUNG_9
      STUDENT_9  | LEHRVERANSTALTUNG_10
      STUDENT_10 | LEHRVERANSTALTUNG_10
      STUDENT_11 | LEHRVERANSTALTUNG_11
      STUDENT_11 | LEHRVERANSTALTUNG_12
      STUDENT_12 | LEHRVERANSTALTUNG_11
      STUDENT_12 | LEHRVERANSTALTUNG_12
    }

    schreibtTable.rows() {
      student_id | pruefung_id | versuch
      STUDENT_1  | PRUEFUNG_1  | 2
      STUDENT_2  | PRUEFUNG_2  | 2
      STUDENT_2  | PRUEFUNG_3  | 1
      STUDENT_3  | PRUEFUNG_4  | 2
      STUDENT_4  | PRUEFUNG_4  | 3
      STUDENT_5  | PRUEFUNG_5  | 1
      STUDENT_5  | PRUEFUNG_6  | 2
      STUDENT_6  | PRUEFUNG_5  | 3
```

```
284         STUDENT_6  | PRUEFUNG_6  | 2
285         STUDENT_7  | PRUEFUNG_7  | 3
286         STUDENT_8  | PRUEFUNG_8  | 3
287         STUDENT_8  | PRUEFUNG_9  | 1
288         STUDENT_9  | PRUEFUNG_10 | 2
289         STUDENT_10 | PRUEFUNG_10 | 3
290         STUDENT_11 | PRUEFUNG_11 | 3
291         STUDENT_11 | PRUEFUNG_12 | 2
292         STUDENT_12 | PRUEFUNG_11 | 2
293         STUDENT_12 | PRUEFUNG_12 | 3
294       }
295
296     }
297
298 }
```

Listing B.1: Generierte DSL

To do…

- ☐ 1 (p. v): Abstract schreiben
- ☐ 2 (p. 3): Modellgetriebene Software-Entwicklung ergänzen: Beispiele, DSL, intern vs extern
- ☐ 3 (p. 5): Datenbank-Tests: Layer Test erklären
- ☐ 4 (p. 9): Gültigkeitsbereiche erklären
- ☐ 5 (p. 71): Praktischer Einsatz / Evaluation ergänzen
- ☐ 6 (p. 76): Praktischer Nutzen für Seitenbau