HOCHSCHULE KONSTANZ TECHNIK, WIRTSCHAFT UND GESTALTUNG
UNIVERSITY OF APPLIED SCIENCES

Modellierung von Testdaten

Nikolaus Moll
287336
Konstanz, 11. 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 von Testdaten**

Verfasser: Nikolaus Moll
TODO
TODO TODO

1. Prüfer: TODO TODO
TODO
TODO
TODO TODO

2. Prüfer: PRUEFERBTITLE PRUEFERB
TODO
TODO
TODO TODO

Abgabedatum: 11. Oktober 2013

Abstract

Thema:	Modellierung von Testdaten
Verfasser:	Nikolaus Moll
Betreuer:	TODO TODO PRUEFERBTITLE PRUEFERB
Abgabedatum:	11. Oktober 2013

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

Ehrenwörtliche Erklärung

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

(1) meine Master-Arbeit mit dem Titel

Modellierung von Testdaten

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, 11. Oktober 2013

Nikolaus Moll

Inhaltsverzeichnis

Abstract **v**

Ehrenwörtliche Erklärung **vii**

1 Einleitung **1**

2 Grundlegende Konzepte **3**

2.1 Modellgetriebene Software-Entwicklung 3

2.2 Software-Tests 3

2.3 DbUnit 4

2.4 SB Testing DB 4

2.5 Konventionen 6

2.5.1 Datenbank ER-Diagramme 6

3 Anforderungsanalyse / Fragestellung **9**

3.1 Allgemeine Anforderungen 9

3.2 Fortlaufendes Beispiel 10

3.2.1 Voraussetzungen 10

3.2.2 Gewählte Problemstellung 10

3.2.3 Beispiel-Use-Cases 12

3.3 Modellierungsvarianten der Testdaten für DbUnit 13

3.3.1 XML-Dataset 13

3.3.2 Default-Dataset 15

3.3.3 SB-Testing-DB-DataSet 16

4 Modellierung der Test-Daten **19**

4.1 Architektur der generierten Klassen 19

4.2 Entwurf der DSL 20

4.2.1 DSL-Entwürfe 21

4.2.2 Entscheidung 23
4.3 Implementierungsvorbereitung 23
4.3.1 Implementierungsvarianten 24
4.3.2 Implementierungsentscheidung 27
4.4 Änderungen am Generator-Modell 27
4.4.1 Spalten-Flags 27
4.4.2 Modellierung von Relationen über Builder-Klassen 28
4.4.3 Alte und neue Builder-Klassen im Vergleich 28
4.5 Realisisierung 29
4.5.1 Neue DataSet-Builder-Klassen 30
4.5.2 Tabellenparser 30
4.5.3 Referenzen und Scopes 31
4.5.4 Beispiel-DataSet in Groovy 32
4.5.5 Nutzung des DataSets in Unit-Tests 34
4.5.6 Komposition von DataSets 35
4.5.7 Erweiterungen in generierter API 36
4.5.8 JavaDoc 38
4.5.9 Verhalten bei Fehlern in den Tabellendefinitionen 41
4.5.10 Nicht umgesetzt 41

5 Generieren von Testdaten **43**

6 Proof of Concept **45**

7 Zusammenfassung und Ausblick **47**

Abkürzungsverzeichnis **51**

Abbildungsverzeichnis **53**

Listings **56**

Literaturverzeichnis **58**

Kapitel 1

Einleitung

Kapitel 2

Grundlegende Konzepte

noch zu erklären: - benutzte Terminologie - Modellgetriebene Software-Entwicklung - Tests, Testdatengenerierung - Literatur nutzen

2.1 Modellgetriebene Software-Entwicklung

- **M0**: Konkrete Information
- **M1**: Meta-Daten zum Beschreiben der Information. Auch als *Modell* bezeichnet.
- **M2**:
- **M3**:

- Modell, Meta-Modell, Modell-Ebenen - http://www.omg.org/spec/MOF/ISO/19502/PDF/ spricht von den klassischen 4 schichten... - DSL, intern vs. extern

2.2 Software-Tests

Das zu testende System wird im Rahmen von Software-Tests als *System Under Test* (abgekürzt SUT) bezeichnet. Dabei kann es sich je nach Test und Kontext auf Klassen, Objekte, Methoden, vollständige Anwendungen oder Teile davon beziehen. [10, 810f]

Alle Voraussetzungen und Vorbedingungen für einen Testlauf werden unter der Bezeichnung *Test Fixture* zusammengefasst. Es repräsentiert den Zustand des SUT vor den Tests. [10, S. 814] Es gibt verschiedene Arten von Test Fixtures. Die im Rahmen dieser Arbeit relevanten sind *Standard Fixture* und *Minimal Fixture*.

Ein Test Fixture wird als Standard Fixture bezeichnet, wenn es für alle bzw. fast alle Tests verwendet werden kann. Ein Standard Feature reduziert nicht nur den Aufwand zum Entwerfen von Testdaten für die einzelnen Tests, sondern verhindert darüber hinaus, dass der Test-Ingenieur sich bei verschiedenen Tests immer wieder in unterschiedliche Test-Daten hineinversetzen muss. Nur in Ausnahmefällen sollten Tests modifizierte oder eigene Testdaten verwenden. ([10, S. 305])

Minimal Fixtures stellen Test Fixtures dar, deren Umfang auf ein Minimum reduziert wurde. Dadurch lassen sich Minimal Fixtures im Allgemeinen leichter verstehen. Das Reduzieren der Daten kann auch zu Leistungsvorteilen bei der Ausführung der Tests führen. ([10, S. 302])

Eine übliche Vorgehensweise, Systeme in Verbindung mit Datenbanken zu testen, stellt *Back Door Manipulation* dar. Dabei wird die Datenbank über direkten Zugriff, vorbei am zu testenden System, in den Anfangszustand gebracht. Anschließend können die zu testenden Operationen am System durchgeführt werden. Um zu überprüfen, ob sich das System richtig verhalten hat, wird der Zustand der Datenbank mit dem erwarteten Zustand verglichen - ebenfalls am zu testenden System vorbei. [10, 327ff]

To do (1)

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. 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. 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 (2)

2.3 DbUnit

To do (3)

2.4 SB Testing DB

Die Firma Seitenbau GmbH verwendet für die Java-basierten Datenbankanwendungen das Framework JUnit mit der Erweiterung DbUnit. Da die Modellierung der Testdaten mit DbUnit-eigenen Mitteln einige Nachteile hat (siehe Abschnitt 3.3), hat Seitenbau die Bibliothek *SB Testing DB* entwickelt. Sie verfolgt keinen generischen Ansatz, der eine Standard-API für die Modellierung von DataSets definieren würde. Stattdessen wird für jedes Datenbank-Modell ein individuelles API bzw. eine interne Java-DSL generiert.

Abbildung 2.1 stellt grafisch dar, wie aus einem Datenbank-Modell die Java-DSL erzeugt wird. Ausgangspunkt ist ein relationales Datenbankmodell. Üblicherweise liegt es bei Seitenbau als *Apache-Torque*-Modell im XML-Format vor. Dieses muss vor der eigentlichen Code-Erzeugung in ein für den Generator interpretierbares Modell (das SB-Testing-DB-Modell) transformiert werden, das die Meta-Informationen zur Datenbank enthält. Zu den notwendigen Meta-Informationen gehören Tabellennamen und Daten zu den Spalten, z.B. Name und Datentypen. Das SB-Testing-DB-Modell kann manuell gepflegt und auch vollständig manuell entwickelt werden. Mit Hilfe des SB-Testing-DB-Modells erzeugt der Generator die interne Java-DSL (DSL Model).

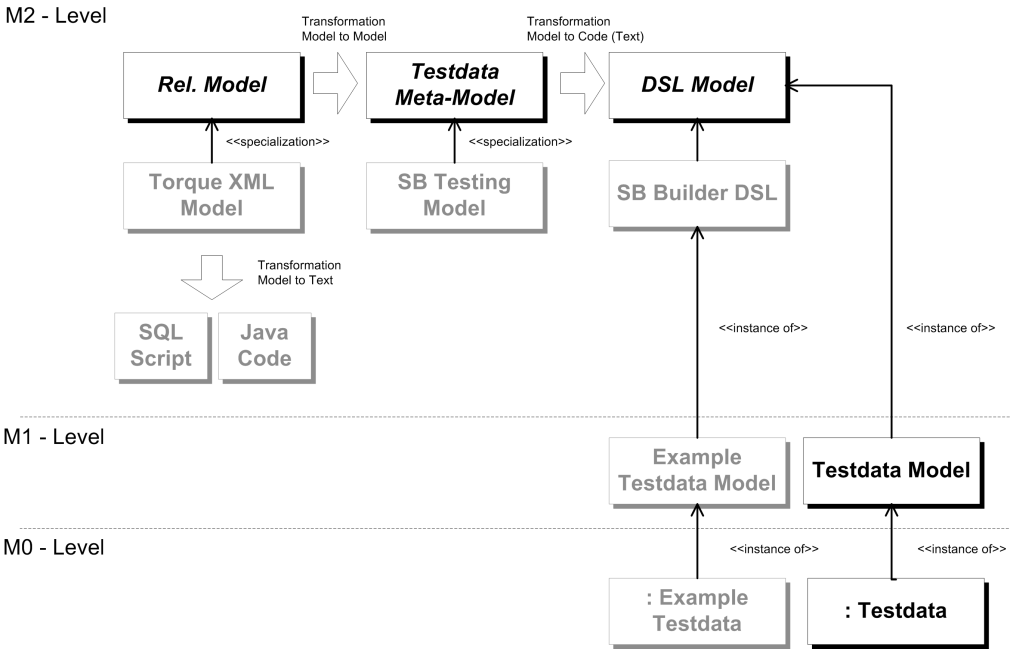

Abbildung 2.1: Modell-Beschreibung

Das SB-Testing-DB-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 Exception 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 erzeugt die Java-Klassen mit Hilfe von *Apache Velocity*. Velocity ist eine sogenannte Template-Engine, die aus Templates Dokumente erzeugt. Die Vorlagen können Platzhalter enthalten, die von Velocity durch konkrete Werte ausgetauscht werden, und auch von Velocity interpretierte Steueranweisungen, z.B. Verzweigungen und Schleifen.

Die generierten Klassen setzen das Builder-Pattern mit einem Fluent API um. **To do** (4)

Die Namen der generierten Klassen hängen vom Modell ab. Unter anderem werden Klassen der folgenden Kategorien erzeugt:

- **DataSet**: 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 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**: Für jede Tabelle wird eine entsprechende Klasse generiert. Der Klassenname setzt sich aus dem Namen der Tabelle und dem Suffix „Table“ zusammen. Die Klasse stellt Methoden zum Hinzufügen und zum Löschen von Tabellenzeilen bereit. Für

Tabellenzeilen gibt es eine passende RowBuilder-Klasse. Außerdem kann nach Zeilen gesucht werden. Dafür gibt es einerseits eine Klasse (`FindWhere`, Beschreibung siehe unten) für einfache Werte-Suchen. Andererseits gibt es auch die Möglichkeit, eine Zeile in Form eines RowBuilders als Such-Parameter zu verwenden. Die gefundenen Zeilen müssen in allen auf dem RowBuilder gesetzten Werten in den jeweiligen Spalten übereinstimmen.

Da die Klasse das DbUnit-Interface `ITable` implementiert, kann sie direkt in DbUnit-DataSets verwendet werden. Dafür muss die Klasse auch Meta-Informationen zu den Spalten beinhalten.

- **RowBuilder**: Zu jeder Tabelle wird eine Klasse zur Repräsentation einer Tabellenzeile generiert. Die Klasse 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. **To do** (5)

- **FindWhere**: Für einfache Suchanfragen gibt es für jede Tabelle die innere Klasse `FindWhere`. Sie ermöglicht die Suche nach einem Wert in einer Spalte. Es wird eine Liste von Tabellenzeilen zurückgeliefert. Nicht zu unrecht trägt sie den Ausdruck „Find“ im Namen. Sie dient dem Zweck, Zeilen zu finden, von denen bekannt ist, dass es sie gibt. Entsprechend wird eine Exception geworfen, wenn keine Zeile entsprechend dem Suchkriterium gefunden wird.

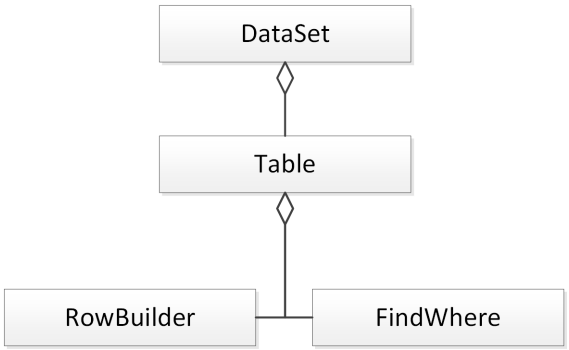

Abbildung 2.2: Klassendiagramm: SB Testing DataSet Builder

2.5 Konventionen

2.5.1 Datenbank ER-Diagramme

Für die Darstellung von Datenbank-Diagrammen wird ein einheitlicher Stil verwendet. Dieser orientiert sich an Ambler aus [1]. Auf die Angabe von Stereotypen wird sowohl bei den Tabellen, als auch bei den Beziehungen zwischen Tabellen verzichtet.

Erklären: - Spalten/PK/FK - Kardinalitäten

Abbildung 2.3 zeigt ein Diagramm mit zwei Tabellen. Tabelle 2 enthält einen Fremdschlüssel, der einem Primärschlüssel aus Tabelle 1

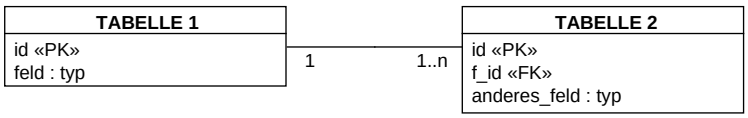

Abbildung 2.3: Datenbank-Diagramm-Stil nach Ambler

Kapitel 3

Anforderungsanalyse / Fragestellung

Einleiten: Zwei Fragestellungen, DSL und Generierung

3.1 Allgemeine Anforderungen

Da die Modellierung der Testdaten mit DbUnit-eigenen Mitteln einige Nachteile hat, hat die Seitenbau GmbH die Bibliothek *SB Testing DB* entwickelt. Allerdings kann *SB Testing DB* nicht alle Nachteile wett machen. So bleibt unter anderem die Modellierung von Beziehungen unübersichtlich. Eine genauere Betrachtung der Vor- und Nachteile verschiedener Modellierungsvarianten in Zusammenhang mit DbUnit wird in Abschnitt 3.3 beschrieben.

To do (6) Kurz Ziele

Die allgemeinen Anforderungen an die zu entwickelnde DSL sind wie folgt:

- **Integration in Werkzeugkette**: Eine der wichtigsten Anforderungen an die DSL ist, dass sie sich in die bestehende Werkzeugkette der Firma Seitenbau integrieren lassen muss. Daraus folgt die Anforderung, dass sie sich in Java nutzen lassen soll. Ähnlich wie bei *SB Testing DB* sollen Datasets auch nachträglich veränderbar sein.
- **IDE-Support**: Die DSL sollte sich gut in gängige Entwicklungsumgebungen wie Eclipse und IntelliJ integrieren lassen.
- **Schlankheit**: Die Sprache soll auf syntaktischen Ballast verzichten und einen übersichtlichen Code zur Modellierung der Daten ermöglichen. Meta-Informationen sollten ausschließlich in Form von Sprachelementen auftauchen.
- **Beziehungen**: Beziehungen sollen sich einfach und typsicher modellieren lassen. Es soll nicht mehr notwendig sein, symbolische Java-Konstanten z.B. für die Definition von ID-Nummern zu verwenden.
- **Typ-Sicherheit**: Beim Test müssen falsche Typen (z.B. bei Beziehungen) erkannt werden und den Test scheitern lassen. Idealerweise sollten die Typen allerdings schon zur Compilier-Zeit überprüft werden können.
- **Funktionen als Werte**: - Datumswerte - Berechnungen - Einlesen von BLOBs
- **Gültigkeitsbereiche**:

- **Zielgruppe**: Die Zielgruppe für die DSL sind überwiegend Software-Entwickler. Anwender, die versiert sind im Umgang mit Datenbanken, sollten zumindest keine Probleme beim Lesen und Verstehen der DSL haben.
- **-Diskussion-**: Sollen sich auch „ungültige Daten“ modellieren lassen?

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

- **Kompatiblität**: Die Generierung der Testdaten soll nicht nur bei Nutzung der neuen Modellierungssprache verwendet werden können. Der Testdaten-Generator soll auch DataSets auf Basis der bisherigen SB-Testing-DB-Builder erstellen können.

3.2 Fortlaufendes Beispiel

Ein einheitliches und fortlaufendes Beispiel soll der Arbeit als Grundlage dienen. Die Problemstellung besteht aus einem Modell und einem Satz von Testdaten. Alle im weiteren Verlauf diskutierten Modellierungsvarianten werden diese Problemstellung umsetzen und die Testdaten modellieren.

3.2.1 Voraussetzungen

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).

3.2.2 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 (also auch 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**: Ein Raum ist genau einem Professor zugewiesen. Ein Professor kann genau einen oder keinen Raum haben.

Abbildung 3.1 zeigt das Beispiel grafisch in Form eines ER-Diagramms. **To do** (7) **To do** (8) **To do** (9)

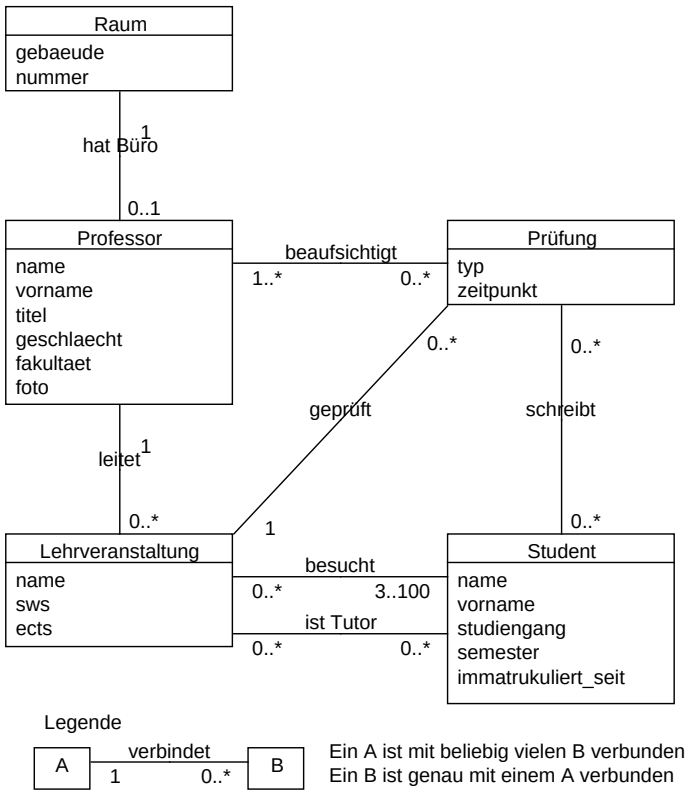

Abbildung 3.1: ER-Diagramm des fortlaufenden Beispiels

Das entsprechende Datenbank-Diagramm wird in Abbildung 3.2 dargestellt.

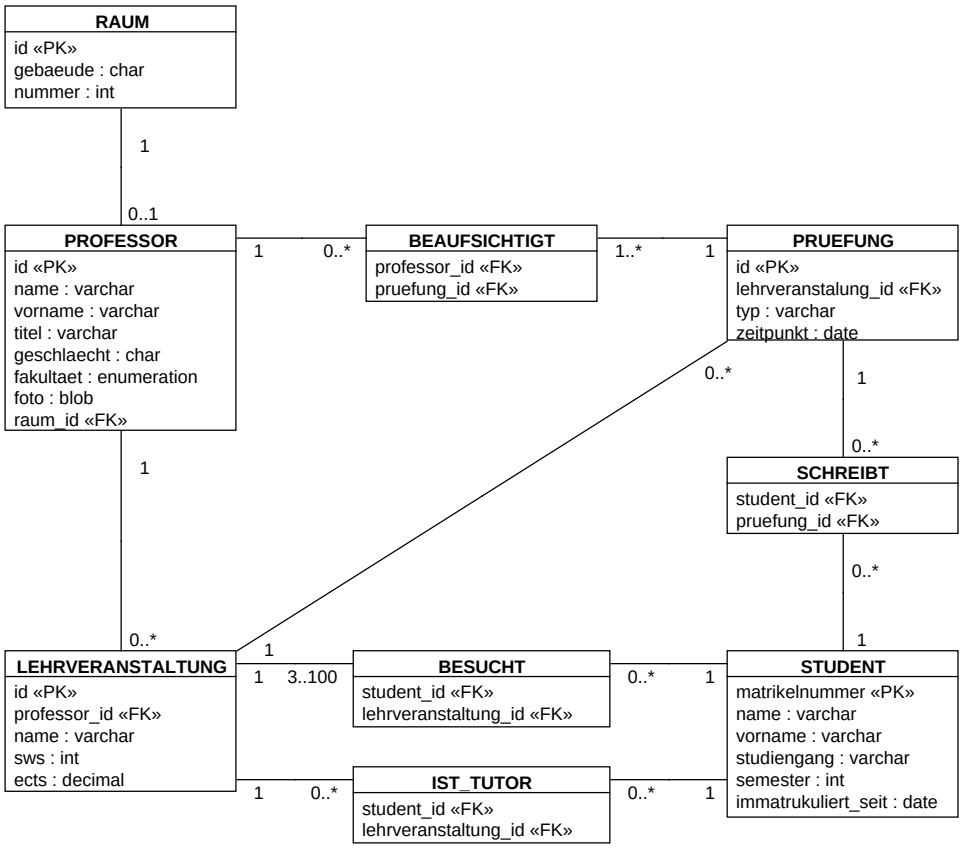

Abbildung 3.2: Datenbank-Diagramm des fortlaufenden Beispiels

To do (10)

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 *Binary Large Object* (BLOB) dargestellt.

3.2.3 Beispiel-Use-Cases

Um den einen Kompromiss für die Komplexität der Testdaten zu finden, werden vier Fragestellungen definiert. Diese Fragen sollen dabei helfen, den Umfang der Testdaten bestimmen zu können. Die Fragen stellen sich wie folgt dar:

1. Welcher Professor unterrichtet die meisten Studenten?
2. Welcher Student nimmt an den meisten Prüfungen teil?
3. Welcher Student ist Tutor und nimmt gleichzeitig an der Prüfung teil?
4. Welcher Professor macht die wenigste Aufsicht in Fremdveranstaltungen (Lehrveranstaltungen eines anderen Professors)?

3.3 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. Datasets aus DbUnit bieten allerdings nicht die Möglichkeit, aus einem bestehenden Dataset ein zweites zu erzeugen, das die Änderungen an der Datenbank beinhaltet.

Im Folgenden werden verschiedene Modellierungsarten für DbUnit-Datasets diskutiert. Die Erkenntnisse sollen in die Anforderungen an die DSL einfließen.

3.3.1 XML-Dataset

Eine Variante, ein Dataset für DbUnit zu modellieren, stellt XML dar. DbUnit bietet dazu die Klasse *XmlDataSet*, die eine XML-Datei nach einem vorgegebenen Dokumententyp einlesen kann. Das Listing 3.1 zeigt einen Ausschnitt einer solchen XML-Datei, in dem die beiden Tabellen *Professor* und *Lehrveranstaltung* definiert werden.

BLOBs lassen sich bei den meisten Modellierungsvarianten nur mit zusätzlichem Aufwand realisieren. Eine direkte Darstellung ist in den meisten Sprachen nicht möglich, allerdings in Bezug auf die Übersicht auch nicht unbedingt erwünscht. Egal ob sie in die XML-Datei eingebettet werden mit Hilfe einer XML-kompatiblen Codierung, oder ob ein Bezug auf externe Ressourcen genommen werden soll, solche Funktionen müssen für DbUnit-Datasets manuell implementiert werden. Hier bietet die Modellierung über eine Programmiersprache wie Java die meisten Freiheiten: Das Datenobjekt kann auf beliebige Art und Weise zur Laufzeit erzeugt werden (Einlesen aus Datei, Generierung, etc.).

```
<!DOCTYPE dataset SYSTEM "dataset.dtd">
<dataset>
    <table name="PROFESSOR">
        <column>id</column>
        <column>name</column>
        <column>vorname</column>
        <column>titel</column>
        <column>fakultaet</column>
        <row>
            <value>1</value>
            <value>Wäsch</value>
            <value>Jürgen</value>
            <value>Prof. Dr.-Ing.</value>
            <value>Informatik</value>
        </row>
        <row>
            <value>2</value>
            <value>Haase</value>
            <value>Oliver</value>
            <value>Prof. Dr.</value>
            <value>Informatik</value>
        </row>
    </table>
    <table name="LEHRVERANSTALTUNG">
        <column>id</column>
        <column>professor_id</column>
        <column>name</column>
        <column>sws</column>
        <column>ects</column>
        <row>
            <value>1</value>
            <value>2</value>
            <value>Verteilte Systeme</value>
            <value>4</value>
            <value>5</value>
        </row>
        <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 positiven Eigenschaften bei der Modellierung in 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.

Leider können die Werkzeuge kaum erkennen, ob in den einzelnen Zellen die richtigen Typen verwendet werden. Die in der XML-Datei enthaltenen Meta-Informationen (Beschreibung der Spalten, Zeilen 4-8 und 25-29) reichen dafür nicht aus. Die Meta-Informationen sind redundant und erschweren die Pflege.

Das Modellieren von Referenzen findet auf einer niedrigen Abstraktionsebene statt und ist damit unübersichtlich und fehleranfällig. Primär- und Fremdschlüssel müssen von Hand gepflegt werden. In umfangreicheren Datasets sind unkommentierte Beziehungen für Betrachter nur schwer nach zu vollziehen, da ein Schlüsselwert üblicherweise keinen unmittelbaren Rückschluss auf den referenzierten Datensatz erlaubt.

Ein großer Nachteil von XML-Datasets 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.

Datasets in XML wachsen schnell in vertikaler Richtung und enthalten unter Umständen auch viel syntaktischen Overhead. Von den rund 30 gezeigten Zeilen enthalten nur zehn Zeilen wirkliche Daten bzw. drücken Beziehungen aus (Zeilen 21 und 26).

Es gibt noch eine zweite Möglichkeit, Datasets via XML zu erzeugen: Das *FlatXmlDataSet*. 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 diesem Punkt 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" />
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" />
```

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

```
    <LEHRVERANSTALTUNG id="2"
        professor_id="2"
        name="Design Patterns"
        sws="4"
        ects="3" />
...
</dataset>
```

Listing 3.2: Flat-XML-Dataset

Wie auch beim XmlDataSet sollte der Übersicht wegen für jeden Wert eine Zeile verwendet werden. Durch die fehlende Hierarchie wirkt das FlatXmlDataSet etwas unübersichtlich.

3.3.2 Default-Dataset

Um einige der Probleme zu vermeiden, die in Verbindung mit XML-Datasets auftreten, kann das Default-Dataset verwendet werden. Dieses lässt sich programmatisch, also dynamisch zur Laufzeit, erstellen. Durch die Nutzung von symbolischen Konstanten als Schlüsselwerte können Beziehungen ausdrucksstärker modelliert werden. Das Erzeugen des Datasets, das den nach einem Test erwarteten Datenbankzustand repräsentiert, bleibt umständlich, ist aber auf Java-Ebene mit weniger Redundanz lösbar.

```
DefaultTable professor = new DefaultTable(
    "professor",
    new Column[] {
      new Column("id", DataType.INTEGER),
      new Column("name", DataType.VARCHAR),
      new Column("vorname", DataType.VARCHAR),
      new Column("titel", DataType.VARCHAR),
      new Column("fakultaet", DataType.VARCHAR),
    }
  );
professor.addRow(new Object[] {
      Parameters.Professor.WAESCH_ID,
      "Wäsch",
      "Jürgen",
      "Prof. Dr.-Ing.",
      "Informatik",
    });
professor.addRow(new Object[] {
      Parameters.Professor.HAASE_ID,
      "Haase",
      "Oliver",
      "Prof. Dr.",
      "Informatik",
    });
dataSet.addTable(professor);

DefaultTable lehrveranstaltung = new DefaultTable(
    "lehrveranstaltung",
    new Column[] {
      new Column("id", DataType.INTEGER),
      new Column("professor_id", DataType.INTEGER),
      new Column("name", DataType.VARCHAR),
      new Column("sws", DataType.INTEGER),
      new Column("ects", DataType.INTEGER),
    }
  );
lehrveranstaltung.addRow(new Object[] {
      Parameters.Lehrveranstaltung.VSYSTEME_ID,
      Parameters.Professor.HAASE_ID,
      "Verteilte Systeme",
      4,
      5,
    });
lehrveranstaltung.addRow(new Object[] {
      Parameters.Lehrveranstaltung.DESIGN_PATTERNS_ID,
      Parameters.Professor.HAASE_ID,
      "Design Patterns",
      4,
```

```
49          3,
50      });
51  dataSet.addTable(lehrveranstaltung);
```

Listing 3.3: Default-Dataset

Diese Umsetzung löst allerdings nicht alle Probleme. So müssen immer noch Meta-Informationen über die Tabellen modelliert werden (Zeilen 3-9 und 29-36). Obwohl diese sogar Typinformationen beinhalten, werden Typ-Fehler erst zur Laufzeit erkannt. Der Einsatz von symbolischen Konstanten erleichtert zwar die Pflege des Datasets, dennoch lassen sich Konstanten doppelt belegen oder auch Primärschlüssel einer falschen Datenbank als Fremdschlüssel angegeben werden.

Ähnlich wie für die Modellierung über XML-Dateien sind für eine übersichtliche Formatierung viele Zeilen notwendig und umfangreiche Datensets werden schnell unübersichtlich. Insgesamt bietet die Nutzung der Java-Datasets in dieser Art nur wenig Vorteile gegenüber den XML-Datasets.

3.3.3 SB-Testing-DB-DataSet

Die Bibliothek *SB Testing DB* der Firma Seitenbau GmbH versucht Nachteile der XML- und Default-DataSets aufzufangen. In Abschnitt 2.4 wird diese Bibliothek beschrieben. Ein Generator erzeugt aus Meta-Informationen zu den Tabellen eine einfache Java-DSL. Über diese DSL lassen sich die Testdaten modellieren. Im Gegensatz zu DbUnit-Datasets unterliegt dieses Modell weniger strikten Einschränkungen in Bezug auf Modifikationen, und erlaubt auch das Löschen von Datensätzen. Um die modellierten Daten in Verbindung mit DbUnit zu verwenden, kann aus dem Modell ein DbUnit-Dataset erzeugt werden. Der Vorteil dieses zusätzlichen Modells ist, dass sich daraus verhältnismäßig einfach Varianten von DbUnit-Datasets erzeugen lassen, z.B. ein Dataset mit dem Ausgangszustand, und ein Dataset mit dem erwarteten Zustand am Ende des Tests. Die Java-DSL sorgt für Typsicherheit zur Compilierzeit[2]. Die Syntax ist kompakter und dennoch ausdrucksstärker als bei beiden vorherigen Varianten.

```
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);
```

[2]Gängige Entwicklungsumgebungen wie Eclipse zeigen falsche Typen bereits während der Entwicklung an.

Listing 3.4: SB Testing Dataset (1)

Die Modellierung von Referenzen stellt sich als ähnlich problematisch wie bei den bisherigen Java-Datasets dar (siehe Abschnitt 3.3.2). Nach wie vor wächst das Dataset vertikal in der Datei.

Zumindest das Problem mit den Referenzen kann durch eine Erweiterung auf M2-Ebene etwas entschärft werden. Ein um Beziehungen erweitertes Modell ermöglicht typsichere Referenzen auf andere Entitäten (siehe Listing 3.5, Zeilen 20 und 27). Bei dieser Variante kann unter Umständen darauf verzichtet werden, Primärschlüssel manuell zu vergeben.

```
1  RowBuilder_Professor haase =
2    table_Professor
3      .insertRow()
4        .setName("Haase")
5        .setVorname("Oliver")
6        .setTitel("Prof.␣Dr.")
7        .setFakultaet("Informatik");
8  RowBuilder_Professor waesch =
9    table_Professor
10     .insertRow()
11       .setName("Wäsch")
12       .setVorname("Jürgen")
13       .setTitel("Prof.␣Dr.-Ing.")
14       .setFakultaet("Informatik");
15
16 RowBuilder_Lehrveranstaltung vsys =
17   table_Lehrveranstaltung
18     .insertRow()
19       .setName("Verteilte␣Systeme")
20       .refProfessorId(haase)
21       .setSws(4)
22       .setEcts(5);
23 RowBuilder_Lehrveranstaltung design_patterns =
24   table_Lehrveranstaltung
25     .insertRow()
26       .setName("Design␣Patterns")
27       .refProfessorId(haase)
28       .setSws(4)
29       .setEcts(3);
```

Listing 3.5: SB Testing Dataset (2)

Kapitel 4

Modellierung der Test-Daten

Die Erweiterungen und andere Verbesserungen fließen nicht in die bisher genutzte Bibliothek *SB Testing DB* ein. Stattdessen wird der Quellcode dieser Bibliothek als Ausgangspunkt für das neue Projekt *STU* (Simple Test Utils, https://github.com/Seitenbau/stu) verwendet. *STU* steht unter der Open-Source-Lizenz XYZ[To do (11)]

Die bisherigen Schnittstellen sollen so weit möglich nur ergänzt und nicht verändert werden, so dass auf *SB Testing DB* basierende Tests möglichst leicht auf *STU* portiert werden können. Neue bzw. angepasste Tests können jedoch von den neuen Möglichkeiten profitieren.

Bei den Schnittstellen zur Beschreibung des Datenbankmodells für den Generator wird jedoch auf Kompatibilität verzichtet.

4.1 Architektur der generierten Klassen

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

- Das **Fluent Builder API** ist ein **Java**-basiertes API. Es nutzt das Builder-Pattern in Verbindung mit einem Fluent Interface (ref builder pattern).
- Das **Table Builder API** ist das **Groovy**-basierte API, das es erlaubt, die Testdaten tabellarisch zu modellieren.

Abbildung 4.1 stellt die Architektur grafisch dar.

To do (12)

Die neue Table Builder API stellt eine Schicht über der bisherigen Fluent Builder API dar. Neue Funktionen müssen jedoch nicht zwangsläufig in der 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**: Es gibt verschiedene Ansätze, Klassen um neue Funktionen zu erweitern oder ihr Verhalten zu ändern. Unabhängig davon, ob auf Vererbung oder Delegation gesetzt wird, werden neue Datentypen benötigt.

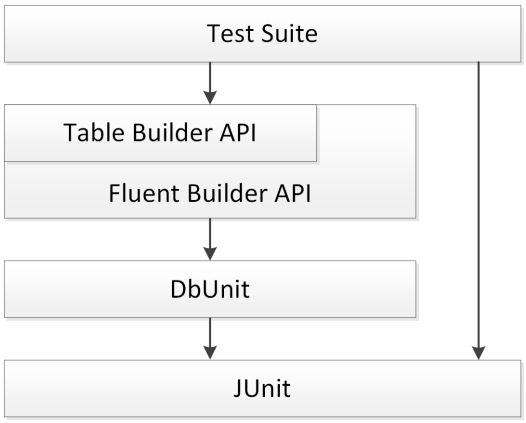

Abbildung 4.1: Architektur

Soll die Schicht der Fluent Builder API nicht verändert werden, stellt Vererbung keine Option zur Erweiterungen von den Klassen dar, die von der Fluent-Builder-API-Schicht selbst instantiiert werden. Eine Lösung könnten Adapter-Klassen sein, die sämtliche Methoden der zu adaptierenden Klasse beinhalten, aber auch die Erweiterungen. Eine solche Adapter-Klasse kann aus einer Vielzahl an Methoden bestehen, die nichts anderes machen, als die Aufgabe weiter zu delegieren.

Insgesamt stellen Adapterklassen in Kombination mit Delegation keine elegante Lösungen dar. Die unübersichtlicheren und aufgeblähten Klassenhierarchie ist dabei noch das kleinere Problem. Gravierender ist, dass innerhalb der Table Builder API konsequenterweise nur noch die Adapter-Klasse statt der ursprünglichen Klasse als Rückgabetyp von Methoden in Frage kommen darf. Das würde bedeuten, dass weitere Klassen adaptiert werden müssten, nur um den Rückgabetyp anzupassen.

Vererbung hat ähnliche Nachteile was die Klassenhierarchie betrifft und würde außerdem noch Änderungen in der Fluent-Builder-API-Schicht nach sich ziehen. Wenn allerdings Änderungen innerhalb dieser Schicht gemacht werden, dann können die Erweiterungen auch direkt in dieser Schicht, also den bisherigen Klassen gemacht werden. So lange nur neue Funktionen hinzukommen und das Verhalten bestehender Methoden nicht verändert wird, müssen bestehende Tests nicht an die neuen Schnittstellen angepasst werden.

- **Mehrwert gegenüber *SB Testing DB***: Auch wenn auf das neue Table Builder API verzichtet wird, bietet das Fluent Builder API einen Mehrwert gegenüber der bisherigen SB-Testing-DB-Implementierung.

- **Einheitliches Verhalten**: Beide APIs zeigen auf diese Weise ein einheitlicheres Verhalten.

4.2 Entwurf der DSL

... Vor- und Nachteile einzelner Entwürfe ...

4.2.1 DSL-Entwürfe

Entwurf 1

Eine DSL, die sich stark an *SB Testing DB* orientiert, könnte wie folgt aussehen:

```
HAASE = professor {
  name       "Haase"
  vorname    "Oliver"
  titel      "Prof. Dr."
  fakultaet  "Informatik"
}

WAESCH = professor {
  name       "Wäsch"
  vorname    "Jürgen"
  titel      "Prof. Dr.-Ing."
  fakultaet  "Informatik"
}

VSYS = lehrveranstaltung {
  name       "Verteilte Systeme"
  sws        4
  ects       5
}

DPATTERNS = lehrveranstaltung {
  name       "Design Patterns"
  sws        4
  ects       3
}

...

HAASE leitet VSYS
HAASE leitet DPATTERNS
HAASE beaufsichtigt P_DPATTERNS
WAESCH beaufsichtigt P_VSYS
...
```

Listing 4.1: Mögliche DSL (1)

Diese DSL kommt ohne manuell vergebene ID-Nummern aus und verwendet Variablennamen für die Modellierung von Beziehungen. Da für jeden Wert eine eigene Zeile verwendet wird, werden umfangreiche Daten schnell unübersichtlich. Die Beschreibung der Beziehungen abseits der Definition der Daten erschwert den Umgang mit den Daten und die Übersicht ebenfalls.

Entwurf 2

Ein leicht abgewandelter Entwurf zeigt, wie sich die Beziehungen näher an den eigentlichen Daten beschreiben lassen könnten. An dem Problem, dass die Daten relativ schnell in vertikaler Richtung wachsen, ändert das jedoch nichts.

```
HAASE = professor {
  name       "Haase"
  vorname    "Oliver"
  titel      "Prof. Dr."
  fakultaet  "Informatik"
  leitet     VSYS, DPATTERNS
  beaufsichtigt P_DPATTERNS
}

WAESCH = professor {
  name       "Wäsch"
  vorname    "Jürgen"
  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: Mögliche DSL (2)

Entwurf 3

Der dritte Entwurf versucht die Daten durch eine tabellarische Struktur übersichtlich zu gestalten. Sie kommt mit wenig syntaktischem Ballast aus. 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: Mögliche DSL (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.

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. Auf Tabulatoren sollte unter Umständen verzichtet werden, da sie von verschiedenen Editoren unterschiedlich dargestellt werden können. Bei vielen Spalten wächst diese Darstellung horizontal. Bei optionalen Spalten bzw. kaum genutzte Spalten kann die tabellarische Darstellung unübersichtlich werden.

Einige Entwicklungsumgebungen wie Eclipse bieten spezielle Block-Bearbeitungsfunktionen an, die beim Arbeiten an einer Tabellen-DSL hilfreich sein kann. So können beispielsweise in einer Spalte über mehrere Zeilen hinweg Leerzeichen eingefügt oder entfernt werden.

Zur besseren Übersicht kann es bei größeren Tabellen sinnvoll sein, den Tabellenkopf zu wiederholen.

4.2.2 Entscheidung

Der dritte Entwurf zeigt, dass eine tabellarische Schreibweise viele Schwächen der anderen Varianten ausmerzt. Die Darstellung wirkt übersichtlich, da Tabellen ... **To do** (13)

+ Möglichkeit, Beziehungen über eine spezielle Syntax wie in Entwurf 1 auszudrücken

To do (14)

4.3 Implementierungsvorbereitung

Da sich die DSL in die bisherige Werkzeug-Kette von Seitenbau integrieren lassen soll (siehe Abschnitt 3.1), sollte die DSL in Java nutzbar sein. Zwar kann eine DSL grundsätzlich auch in Java realisiert werden, doch die Möglichkeiten diesbezüglich sind relativ eingeschränkt und die DSL sieht immer noch nach Java aus. Es lassen sich allerdings auch andere Sprachen im Java-Umfeld nutzen.

Eine davon ist Groovy. Groovy ist eine dynamisch typisierte Sprache[1], die direkt in Java-Bytecode übersetzt wird und damit auch in einer Java Virtual Machine ausgeführt wird. Sie teilt sich das Objekt-Modell mit Java, so dass aus Groovy heraus instantiierte Objekte auch in der Host-Anwendung nutzbar sind (und umgekehrt). Auch wenn Java-Code bis auf wenige Ausnahmen gültiger Groovy-Code und sich dort gleich verhält, enthält Groovy Techniken, die den Code mehr wie eine natürliche Sprache aussehen lassen. So kann oftmals auf die Semikolons am Ende einer Anweisung verzichtet werden, und auch auf das Einklammern von Parametern kann bei Methoden aufrufen verzichtet werden (wenn die Methode genau einen Parameter erwartet). Außerdem kann statt dem Punkt zwischen Objekt und Methode beim Aufruf verzichtet werden.

Listing 4.4 zeigt einen Befehl einmal in typischer Java-Syntax und einmal mit den Syntax-Vereinfachungen von Groovy:

```
1 myList.append("value␣1").append("value␣2");
2 myList append "value␣1"  append "value␣2"
```

Listing 4.4: Vereinfachung von Ausdrücken in Groovy

Groovy hebt sich ferner durch die Möglichkeit Operatoren zu überladen und durch Closures (Funktionsabschlüsse) von Java ab. Ein Closure 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. [8, S. 40] **To do** (15) Die Unterstützung zur Meta-Programmierung stellt sich beim Implementieren einer DSL ebenfalls als nützlich heraus. Dadurch ist es z.B. möglich, abgeschlossene Klassen innerhalb von Groovy um Methoden zu erweitern oder auf den Zugriff von nicht definierten Klassenelementen zu reagieren.

Aus diesen Gründen empfiehlt Ghosh in [6, S. 148] Groovy als Host für DSLs in Verbindung mit Java-Anwendungen.

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

4.3.1 Implementierungsvarianten

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, letzteres in Form von AST-Transformationen. Beide Ansätze bieten individuelle Vorteile, die im folgenden diskutiert werden.

Laufzeit-Meta-Programmierung

Eine Möglichkeit, die DSL mit Hilfe von Laufzeit-Meta-Programmierung zu implementieren sieht eine Klasse zum Parsen von Closures vor, die eine Tabelle beinhalten. Diese Klasse, `TableParser`, enthält dafür die Methode `parseTableClosure`. Die Methode soll als Ergebnis eine Liste von Tabellenzeilen zurückliefern. Da an dieser Stelle noch keinerlei Interpretation der Tabellenwerte durchgeführt wird, stellt eine Tabellenzeile selbst ebenfalls eine Liste dar - aus den Objekten der Spalten.

Der Ansatz ist, Operator-Überladen für das Parsen zu verwenden. Soll ein binärer Operator implementiert werden, ist die übliche Vorgehensweise in Groovy, die Klasse des linken Operanden um eine entsprechende Methode für den Operator zu erweitern. Diese Methode trägt einen vorgegebenen Namen und erwartet als binärer Operator den rechten Operanden als Parameter (eine Übersicht findet sich beispielsweise in [8, 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. **To do** (16)

Groovy bietet allerdings auch eine zweite Möglichkeit für das Operator-Überladen an. Anstatt den Operator als Methode dem linken Operand (bzw. der Klasse) hinzuzufügen, wird er als statische Methode (in einer beliebigen Klasse) realisiert. 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 die Kategoriemethoden in einem Closure verwendet werden. [8, S. 192]

Listing 4.5 zeigt das Grundgerüst des Tabellenparsers:

```
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

Die Methode `or` erwartet zwei Parameter vom Typ `Object`. Obwohl in Groovy alle Typen von `Object` abgeleitet sind, gibt es Oder-Ausdrücke, bei denen diese Methode nicht aufgerufen wird. Ein in der Klasse definierter Operator mit passenden Datentypen wird dieser

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

allgemeinen Methode bevorzugt, z.B. bei zwei `Integer`-Werten. Doch auch solche Operationen lassen sich überschreiben, wenn für die Datentypen passende Kategoriemethoden definiert werden.

Der Parser in der Form kann noch nicht mit selbst definierten Spaltennamen und Bezeichner für die einzelnen Entitäten umgehen. Der Compiler versucht, diese Ausdrücke aufzulösen und sucht nach entsprechenden Properties. Properties können normale Variablen sein oder parameterlose Get-Methoden, die beim Aufruf in Groovy ohne das Präfix `get` und den runden Klammern verwendet werden können. Kann Groovy eine Property nicht auflösen ruft es in der aktuellen Klasse (bzw. dem Ausführungskotext) die Methode `propertyMissing` auf. Durch Überschreiben dieser Methode kann auf nicht auflösbare Bezeichner reagiert werden. Damit Groovy die gewünschte Methode bei der Ausführung eines Closures aufruft, kann der Ausführungskontext von Closures gesetzt werden. Auf diese Weise werden Properties und Methoden von der als Ausführungskontext festgelegten Instanz verwendet. In diesem Prototyp wird der Kontext auf den Tabellenparser selbst gesetzt. Die Änderungen sind in Listing 4.6 dargestellt.

```
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 Methoden haben keinen Zugriff auf Instanz-Variablen der Klasse *TableParser*. Ihre Ergebnisse können sie demnach auch nur in statische Elementen aufbewahren. Um die Klasse Thread-sicher zu machen, d.h. das gleichzeitige Parsen von Tabellen aus verschiedenen Threads heraus, wird für die Ergebnisse eine threadlokale Liste verwendet. **To do** (17) [7]

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. Aus Sicht des Parsers stellt der Unterstrich eine Variable dar.

```
def dataset = [
  professor: {
    REF    | name    | vorname  | titel            | fakultaet    | leitet           | beaufsichtigt
    WAESCH | "Wäsch" | "Jürgen" | "Prof. Dr.-Ing." | "Informatik" | _                | P_VSYS
    HAASE  | "Haase" | "Oliver" | "Prof. Dr."      | "Informatik" | VSYS & DPATTERNS | P_DPATTERNS
  },

  lehrveranstaltung: {
    REF       | name                | sws | ects
    VSYS      | "Verteilte Systeme" | 4   | 5
    DPATTERNS | "Design Patterns"   | 4   | 3
  },

  ...
]
```

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.

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 ([5, 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 interessante Statements sind von den 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. Drei Arten von Ausdrücken sind dabei für das Parsen relevant:

- **`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 linken und rechten Operanden, die selbst vom Typ `ExpressionStatement` sind, rekursiv behandelt.
- **`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.

Insgesamt muss viel Aufwand betrieben werden, um den AST zu analysieren. Dabei wirft die IDE-Unterstützung noch einige Fragezeichen auf: Die Labels werden kaum über Auto-Vervollständigung oder Vorschläge innerhalb der IDE nutzbar sein.

4.3.2 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. Die Laufzeit-Meta-Programmierung erlaubt zwar weniger Anpassungen an die Sprache, ist aber für die gewünschte DSL ausreichend und die Umsetzung einfacher. AST-Transformationen könnten zwar Mehrwert bieten, z.B. könnte auch der Double-Pipe-Operator genutzt werden und Labels, allerdings überwiegen die Nachteile des weniger ausdrucksstarken und damit wartungsunfreundicheren Codes. Als Konsequenz wird die einfachere Laufzeit-Meta-Programmierung verwendet.

4.4 Änderungen am Generator-Modell

Bei den Klassen für die Modellierung des zu Grunde liegenden Datenbankmodells wird auf Abwärtskompatibilität verzichtet. Während das alte API auf überladene Methoden mit vielen Parametern setzt, ist das neue API entsprechend dem Builder-Pattern umgesetzt [2, 11ff]. So enthält das alte API neun Methoden zum Hinzufügen einer Spalte in einer Tabelle enthält, wovon eine als *deprecated* eingestuft ist. Dieses Design ist unübersichtlich und nur schwer erweiterbar. Jeder weitere optionale Parameter könnte die Anzahl der Methoden verdoppeln. Demgegenüber gibt es beim Builder-Pattern für jeden optionalen Parameter eine einzelne Set-Methode.

Die neuen Builder-Klassen decken den Funktionsumfang der alten API ab. Dabei werden Flags für Spalten nicht mehr über ein `EnumSet` festgelegt, sondern über Methoden für die vordefinierten Flags. In Abschnitt 4.4.1 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 4.5.8).

To do (18)

4.4.1 Spalten-Flags

SB Testing DB sieht verschiedene Flags für Spalten vor, die in einem *Enum* zusammengefasst sind. 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:

- **Identifikator-Spalte**: Als Identifikator-Spalte wird die Spalte bezeichnet, die einen für die Zeile einmaligen Wert enthält, über die eine Zeile zweifelsfrei identifiziert werden kann. In dieser Eigenschaft ähnelt sie dem Primärschlüssel in Datenbanken. Es bietet sich an, Primärschlüssel als Identifikator-Spalte zu modellieren. Dennoch gibt es keine Kausalität zwischen Identifikator-Spalte und Primärschlüssel: Ein Primärschlüssel muss nicht als Identifikator-Spalte modelliert werden, und eine Identifikator-Spalte muss kein Primärschlüssel sein.

  Die Identifikator-Spalte ist die Spalte, die bei Beziehungen verwendet wird, wenn das Ziel eine Tabelle und nicht eine Spalte der Tabelle ist. Das Flag wird über die

Methode `identifierColumn` aktiviert. Dabei werden die Flags Unveränderbar und Einmalig ebenfalls aktiviert.

- **Next-Value-Methode**: *SB Testing DB* (und damit auch *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.

- **Automatischer Aufruf der Next-Value-Methode**: 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.

- **Auto Increment**: ... DBUNIT-Flag ... implizit addNextMethod

- **Unveränderbar**: 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()`.

- **Einmalig**: Dieses Flag gibt an, dass die Werte einer Spalte nur jeweils ein Mal vorkommen, und sie deshalb zur Identifikation einer Zeile verwendet werden können. Das Flag sollte auch nur dann verwendet werden, wenn eine solche Identifikation erwünscht ist.

  Aufgrund des Anwendungszwecks wird das Flag Unveränderbar implizit aktiviert. Das Einmalig-Flag wird über die Methode `unique()` aktiviert und setzt keine Unique-Eigenschaft in der Datenbank voraus.

4.4.2 Modellierung von Relationen über Builder-Klassen

4.4.3 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 4.8 zeigt die Modellierung in *SB Testing DB*, während Listing 4.9 die in *STU* eingeführten Builder veranschaulicht.

```
1  database("Hochschule");
2  packageName("com.seitenbau.sbtesting.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 4.8: Beispiel SB-Testing-DB-Builder

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

Beide Varianten ähneln sich, Unterschiede liegen - mit Ausnahme der Flags und Relationen - eher im Detail. Die kürzeren Parameterlisten und die zusätzlichen Funktionen führen dazu, dass die selben Modelle in *STU* einige Zeilen länger werden. Die gewonnene Ausdrucksstärke macht diesen Nachteil allerdings mehr als wett.

```
1  database("Hochschule");
2  packageName("com.seitenbau.stu.dbunit.hochschule");
3  enableTableModelClassesGeneration();
4
5  Table professoren = table("professor")
6      .description("Die Tabelle mit den Professoren der Hochschule")
7      .column("id", DataType.BIGINT)
8        .identifierColumn()
9        .autoInvokeNext()
10     .column("name", DataType.VARCHAR)
11     .column("vorname", DataType.VARCHAR)
12     .column("titel", DataType.VARCHAR)
13     .column("fakultaet", DataType.VARCHAR)
14   .build();
15
16 Table lehrveranstaltungen = table("lehrveranstaltung")
17     .description("Die Tabelle mit den Lehrveranstaltungen der Hochschule")
18     .column("id", DataType.BIGINT)
19       .identifierColumn()
20       .autoInvokeNext()
21     .column("professor_id", DataType.BIGINT)
22       .reference
23         .local
24           .name("geleitetVon")
25           .description("Gibt an, von welchem Professor eine
                 Lehrveranstaltung geleitet wird.")
26         .foreign(professoren)
27           .name("leitet")
28           .description("Gibt an, welche Lehrveranstaltungen ein Professor
                 leitet.")
29     .column("name", DataType.VARCHAR)
30     .column("sws", DataType.INTEGER)
31     .column("ects", DataType.DOUBLE)
32   .build();
```

Listing 4.9: Beispiel *STU*-Builder

4.5 Realisisierung

Im Folgenden wir die Realisierung der DSL beschrieben. Dabei werden einige Implementierungsdetails beschrieben und auch gezeigt, wie die DSL praktisch genutzt werden kann. Die DSL sollte möglichst guten Support durch die IDE bieten, um die Arbeit mit den Tabellen zu vereinfachen. Dazu gehört, dass Bezeichner wie Tabellen- und Spaltennamen nicht nur erkannt werden, sondern auch automatisch vervollständigt werden können. Die in Listing 4.7 gezeigte Variante kann diesem Anspruch nicht genügen. Falsche Tabellennamen können erst zur Laufzeit festgestellt werden und auch für die Spaltenbezeichner kann es so keinen IDE-Support geben, da sie von der Tabelle abhängig. Der IDE-Support wird über die neuen DataSet-Builder-Klassen realisiert.

4.5.1 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 4.5.2).

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 4.2 dargestellt.

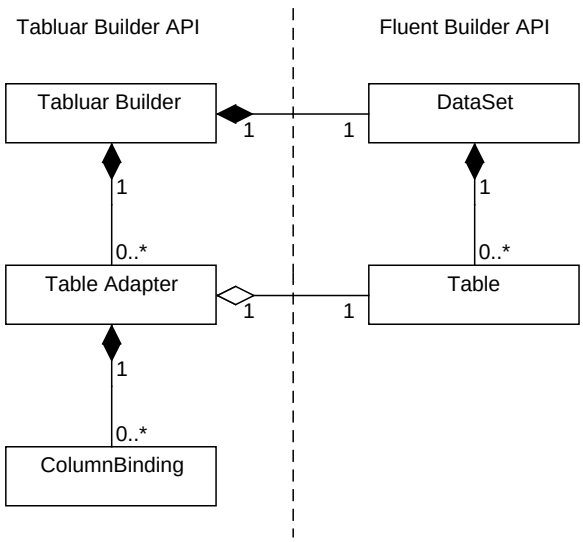

Abbildung 4.2: Klassendiagramm der DataSet-Builder

4.5.2 Tabellenparser

Der Code zum Parsen der Tabellen-Closures basiert auf dem dem in Abschnitt 4.3.1 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 für 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.

Außerdem kann der Parser auch Informationen zu der Spalte abfragen, z.B. ob es sich um ein Feld mit einmaligen Werten handelt, die zur Identifizierung genutzt werden können. Sofern das Flag aktiviert ist, ist eine Such-Methode auf dem `ColumnBinding` implementiert, die es erlaubt den zu einem Wert gehörenden RowBuilder zu finden.

- TableParser (Groovy) - TableParserContext - TableParserCallback - zeilenweise wegen Exceptions - Groovy-Anteil minimiert - Diagramm :-)

4.5.3 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 4.5.7 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()`. Listings 4.10 und 4.11 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 4.10: Binden von Referenzen (Table Builder API)

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

Listing 4.11: Binden von Referenzen (Fluent Builder API)

Da Referenzen die zugehörigen RowBuilder kennen, können ihre Werte auch direkt auf der Referenz abgefragt werden (siehe Listing 4.12).

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

Listing 4.12: 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 4.13 zeigt ein Beispiel,wie die Relation zwischen einem Professor und einer Prüfung modellieren lässt.

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

Listing 4.13: 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.

4.5.4 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 4.14 zeigt, wie sich die selben Daten mit der neuen DSL modellieren lassen - diesmal allerdings in vollem Umfang.

```
class HochschuleDataSet extends HochschuleBuilder
{

```

```
4   def tables() {
5
6     professorTable.rows {
7       REF    | name    | vorname  | titel            | fakultaet
8       WAESCH | "Wäsch" | "Jürgen" | "Prof. Dr.-Ing." | "Informatik"
9       HAASE  | "Haase" | "Oliver" | "Prof. Dr."      | "Informatik"
10    }
11
12    lehrveranstaltungTable.rows {
13      REF        | id  | name                | sws | ects
14      VSYS       | 1   | "Verteilte Systeme" | 4   | 5
15      DPATTERNS  | 2   | "Design Patterns"   | 4   | 3
16    }
17
18    pruefungTable.rows {
19      REF          | id  | typ   | zeitpunkt
20      P_VSYS       | 1   | "K90" | DateUtil.getDate(2013, 4, 1, 14, 0, 0)
21      P_DPATTERNS  | 2   | "M30" | DateUtil.getDate(2013, 1, 6, 12, 0, 0)
22    }
23
24    studentTable.rows {
25      REF         | matrikelnummer | name         | vorname    | studiengang
26      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 4.14: DataSet modelliert mit Table Builder API

Insgesamt ist die Darstellung sehr übersichtlich und kommt ohne syntaktischen Ballast aus. Die DataSet-Klasse erweitert die generierte Klasse `HochschuleBuilder` und überschreibt die beiden Methoden `tables` und `relations`. In diesen Methoden sollen die Tabellendaten bzw. die Beziehungen der Entitäten modelliert werden.

Der Code wurde aufgrund der eingeschränkten Seitenbreite leicht angepasst und die Tabelle mit den Studenten in zwei Blöcke aufgeteilt (Zeilen 24 bis 32). In diesem Beispiel ist diese Darstellung eher unüblich, aber der Parser unterstützt auch die Definition von Teiltabellen mit unterschiedlichen Spalten innerhalb eines Closures.

Auffällig ist, dass keine der assoziativen Tabellen explizit auftaucht, diese werden implizit durch die Beziehungen modelliert. D.h. assoziativer Beziehungen können mit Hilfe der DSL nicht nur auf Datenbank-Ebene, sondern auch auf der abstrakteren ER-Ebene ausgedrückt werden. Es wäre allerdings auch möglich, die assoziativen Beziehungen innerhalb der Methode `tables` zu definieren (siehe Listing 4.15).

```
1  class HochschuleDataSet extends HochschuleBuilder
2  {
3
4    def tables() {
5
6      ...
```

```
7
8       beaufsichtigtTable.rows {
9         professor | pruefung
10        WAESCH    | P_VSYS
11        HAASE     | DPATTERNS
12      }
13
14    }
15
16    ...
17
18  }
```

Listing 4.15: Assoziative Beziehung über Tabelle

4.5.5 Nutzung des DataSets in Unit-Tests

Wie das Beispiel-DataSet aus Listing 4.14 in einem JUnit-Test verwendet werden kann, zeigt Listing 4.16. Das System Under Test (siehe Abschnitt 2.2) ist ein Spring-Service, der von der Variable `sut` (Zeile 20) repräsentiert wird. **To do** (19)

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 macht, 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.

```
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 4.16: 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 4.17 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 4.17: Test-Methode in Groovy

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

4.5.6 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 die DataSet-Klassen, die als Basis für das gerade definierte DataSet dienen sollen, über die Hook-Methoden `extendsDataSet` (bei einem Basis-DataSet) bzw. `extendsDataSets` (für mehrere Basis-DataSets) zurückgeliefert werden. Listing 4.18 zeigt, wie ein DataSet ein anderes als Basis verwendet.

```
class ExtendedHochschuleDataSet extends HochschuleBuilder {

  def extendsDataSet() { HochschuleDataSet }

  def tables() {

   lehrveranstaltungTable.rows {
      REF       | id   | name                | sws | ects
      PROGR     | 3    | "Programmieren"     | 4   | 6.0
    }

  }

  def relations() {
    HAASE.leitet(PROGR)
  }

}
```

Listing 4.18: Erweitertes DataSet

Die Syntax für die Komposition aus den drei DataSet-Klassen `DataSet1`, `DataSet2` und `DataSet3` ist in Listing 4.19 dargestellt:

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

Listing 4.19: Erweitertes DataSet

Das erweiterte DataSet kann in den selben Unit-Tests verwendet werden. Dabei reicht es aus, die Annotation `DatabaseSetup` entsprechend anzupassen (siehe Listing 4.20).

```
  @Test
  @DatabaseSetup(prepare = ExtendedHochschuleDataSet)
  public void assignedLehrveranstaltungen() throws Exception {
    // prepare
    Professor haase = new Professor();
    haase.setId(HAASE.id);

    // execute
    List<Lehrveranstaltung> items = sut.findLehrveranstaltungen(haase);

    // verify
    def findWhere = dataSet.lehrveranstaltungTable.findWhere
    int count = findWhere.professorId(HAASE).rowCount
    assertThat(items).hasSize(count);
  }
```

Listing 4.20: Test auf erweiterem DataSet

4.5.7 Erweiterungen in generierter API

Die meisten Erweiterungen an der Fluant-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 4.21 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 threated as future value
  def tutors(LehrveranstaltungRef ref) {
    return {
      // findWhere throws an exception if no rows are found
      if (isttutorTable.getWhere.lehrveranstaltungId(ref).present) {
        def rows = isttutorTable.findWhere.lehrveranstaltungId(ref);
        return rows.getRowCount()
      }
      return 0;
    }
  }
}
```

Listing 4.21: 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. In Groovy können Closures verwendet werden, ...

- **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 4.22 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 4.22: Such-Beispiele

- **getWhere**: - Zusätzlich zu findWhere
- **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 4.23 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 4.23: Beispiel für find

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

- **foreach**:

```
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 4.24: Beispiel für foreach

4.5.8 JavaDoc

Zum guten IDE-Support gehört auch, dass der Tester beim Erstellen der Tests durch aussagekräftige JavaDoc unterstützt wird.

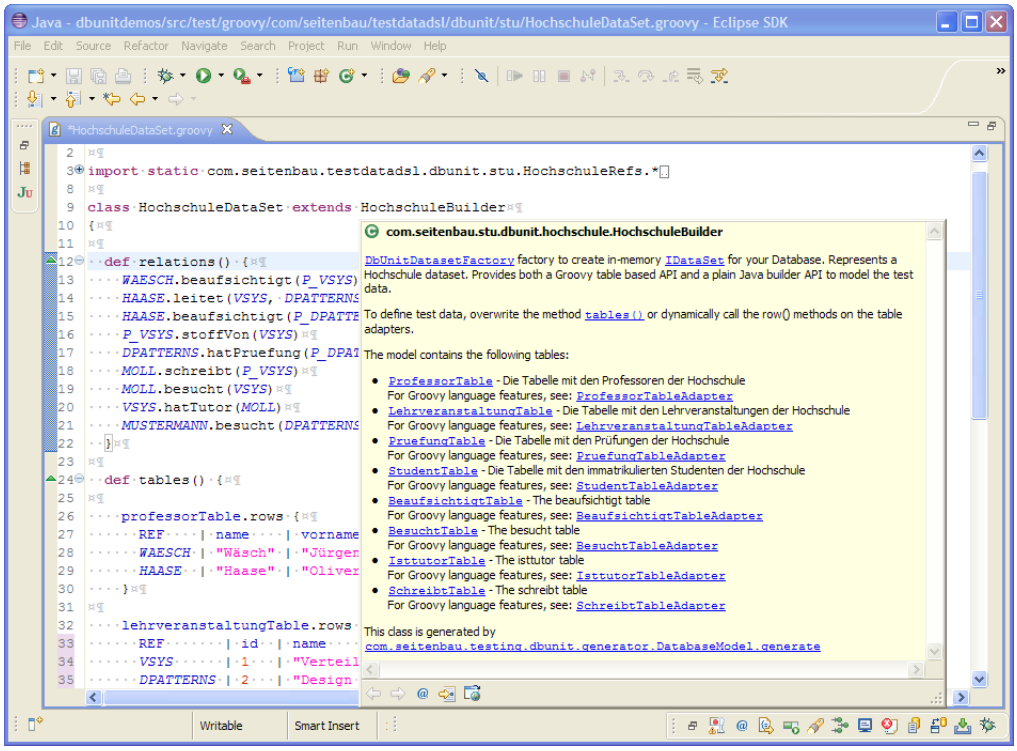

Abbildung 4.3: Tooltip Builder

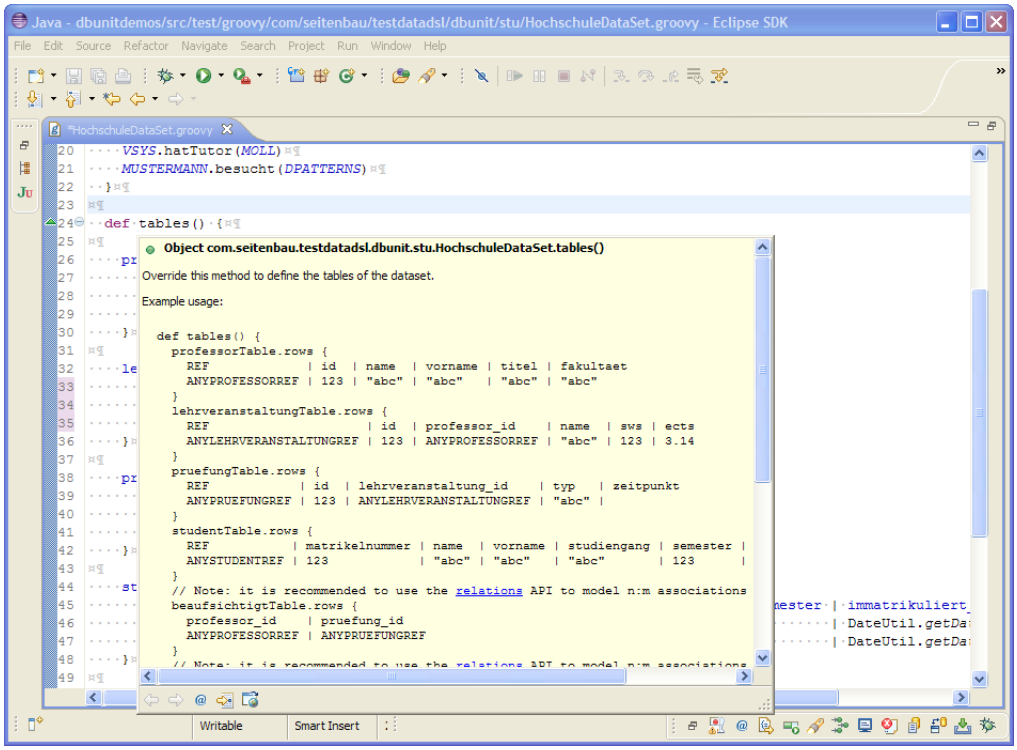

Abbildung 4.4: Tooltip tables()

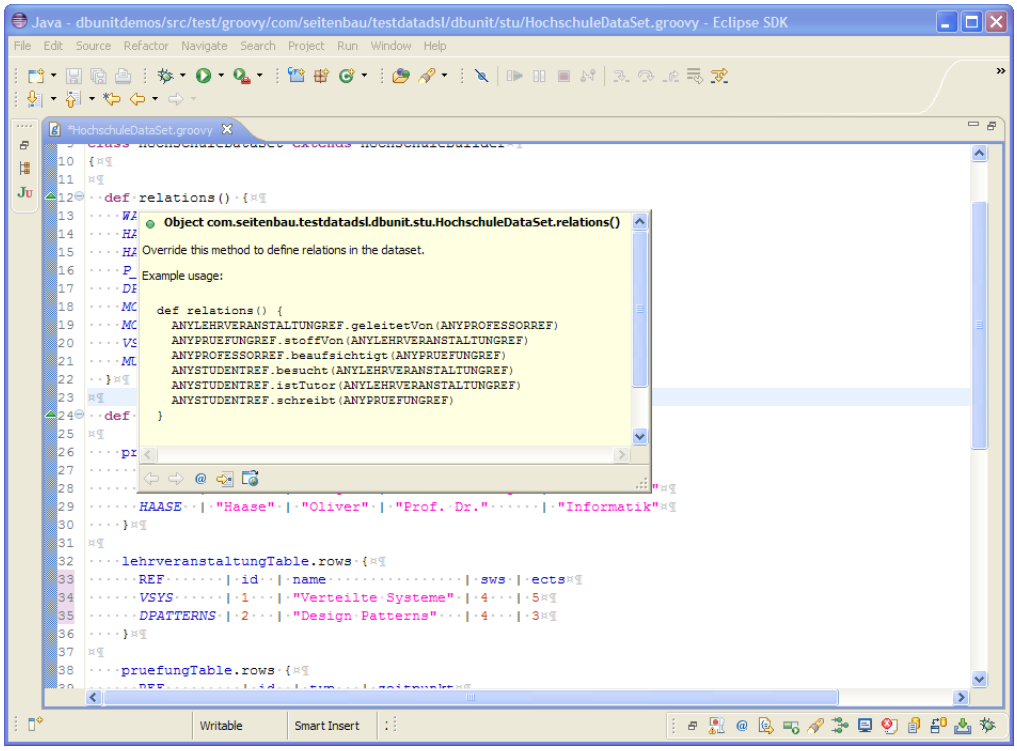

Abbildung 4.5: Tooltip relations()

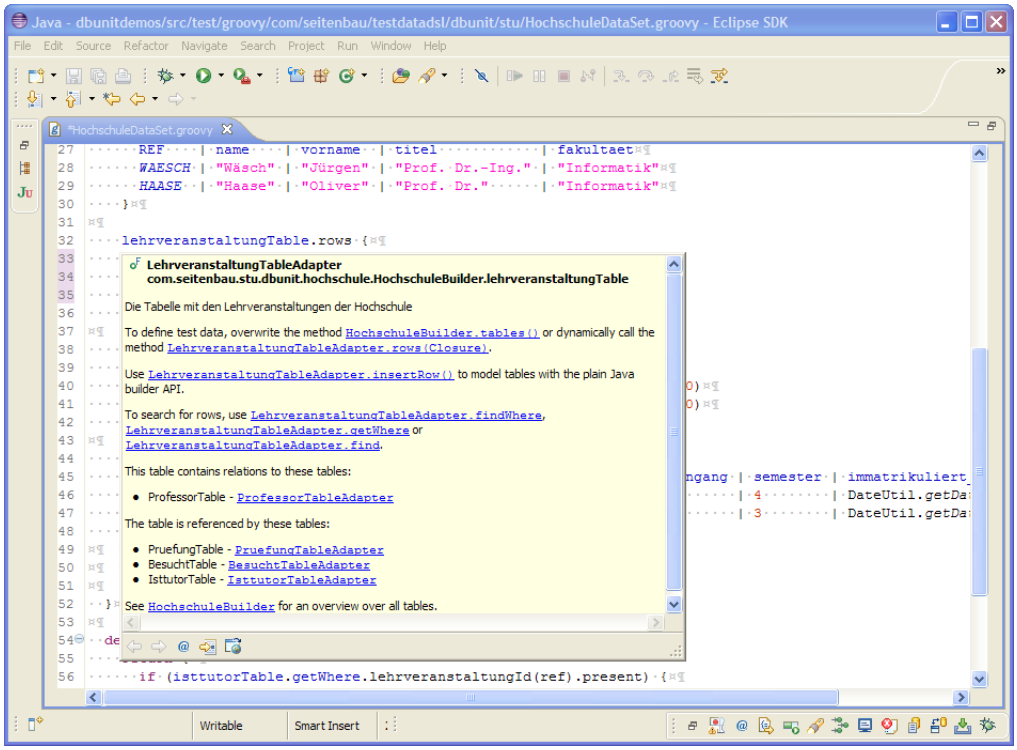

Abbildung 4.6: Tooltip Tabelle

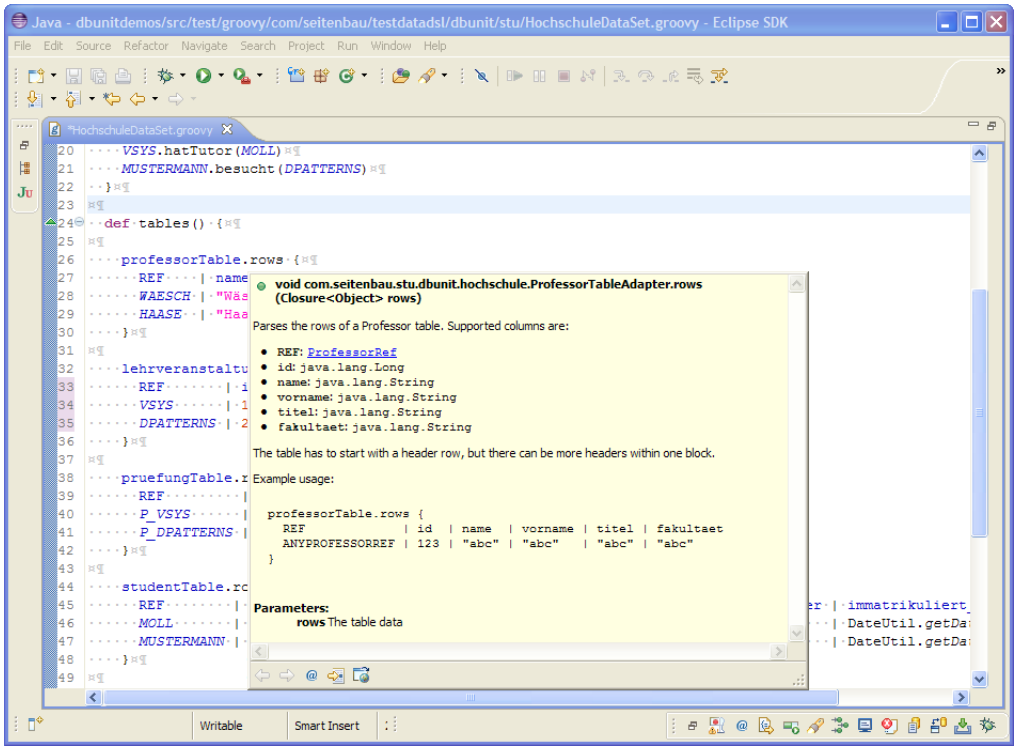

Abbildung 4.7: Tooltip Zeilen

4.5.9 Verhalten bei Fehlern in den Tabellendefinitionen

4.5.10 Nicht umgesetzt

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 4.25 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 4.25: 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 [9, 65f]. Außerdem würde es sich um einen unerwarteten Nebeneffekt handeln [9, 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?

To do (20)

Kapitel 5

Generieren von Testdaten

Fragen: - Wie muss dass Modell „angereichert“ werden? - Wie können Daten sinnvoll generiert werden?

Kapitel 6

Proof of Concept

Kapitel 7

Zusammenfassung und Ausblick

Titel

Untertitel

Stichpunkte

- **Item 1**: Text
- **Item 2**: Text

Aufzählung

1. **Item 1**: Text
2. **Item 2**: Text

Abkürzung

Quellcode

```
Code
```

Listing 7.1: Der Titel

Verweise

1. siehe 7.1
2. siehe Listing 7.1
3. siehe Abb. 7.1
4. siehe Abschnitt 7
5. siehe Kapitel 7

Zitate

1. [11]
2. [13, 20ff]
3. [4]
4. [12]
5. [3]

Bild

Abbildung 7.1: Der Titel

Bildgruppe

(a) Text 1 (b) Text 2 (c) Text 3

Abbildung 7.2: Gemeinsamer Titel

Abkürzungsverzeichnis

AST	Abstract Syntax Tree
BLOB	Binary Large Object)
GUI	Graphical User Interface (Grafische Benutzeroberfläche)
SUT	System Under Test (siehe Abschnitt 2.2)

Abbildungsverzeichnis

2.1 Modell-Beschreibung 5
2.2 Klassendiagramm: SB Testing DataSet Builder 6
2.3 Datenbank-Diagramm-Stil nach Ambler 7

3.1 ER-Diagramm des fortlaufenden Beispiels 11
3.2 Datenbank-Diagramm des fortlaufenden Beispiels 12

4.1 Architektur 20
4.2 Klassendiagramm der DataSet-Builder 30
4.3 Tooltip Builder 39
4.4 Tooltip tables() 39
4.5 Tooltip relations() 40
4.6 Tooltip Tabelle 40
4.7 Tooltip Zeilen 41

7.1 Der Titel 50
7.2 Gemeinsamer Titel 50

Listings

3.1 XML-Dataset 13
3.2 Flat-XML-Dataset 14
3.3 Default-Dataset 15
3.4 SB Testing Dataset (1) 16
3.5 SB Testing Dataset (2) 17
4.1 Mögliche DSL (1) 21
4.2 Mögliche DSL (2) 21
4.3 Mögliche DSL (3) 22
4.4 Vereinfachung von Ausdrücken in Groovy 23
4.5 Tabellen-Parser Grundgerüst mit Operator-Überladen 24
4.6 Tabellen-Parser Grundgerüst mit Operator-Überladen 25
4.7 DSL-Entwurf 3 für Laufzeit-Meta-Programmierung angepasst 26
4.8 Beispiel SB-Testing-DB-Builder 28
4.9 Beispiel *STU*-Builder 29
4.10 Binden von Referenzen (Table Builder API) 32
4.11 Binden von Referenzen (Fluent Builder API) 32
4.12 Zugriff auf Werte über Referenzen 32
4.13 Definition von Beziehungen über Referenzen 32
4.14 DataSet modelliert mit Table Builder API 32
4.15 Assoziative Beziehung über Tabelle 33
4.16 JUnit-Tests (reiner Java-Code) 34
4.17 Test-Methode in Groovy 35
4.18 Erweitertes DataSet 36
4.19 Erweitertes DataSet 36
4.20 Test auf erweiterem DataSet 36
4.21 Beispiel Lazy Valunes 37
4.22 Such-Beispiele 37

4.23 Beispiel für find 38
4.24 Beispiel für foreach 38
4.25 Löschen von Zeilen 41
7.1 Der Titel 49

Literatur

[1] 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.

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

[3] Eric Evans. *Domain-driven Design: Tackling Complexity in the Heart of Software*. Addison Wesley Professional, 2004. ISBN: 978-0-321-12521-7. URL: http://books.google.de/books?id=7dlaMs0SECsC.

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

[5] Erich Gamma u. a. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995. ISBN: 978-0-201-63361-0. URL: http://books.google.de/books?id=6oHuKQe3TjQC.

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] Thomas Stahl. *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. 2., aktualisierte und erw. Aufl. dpunkt-Verl., 2007. ISBN: 978-3-89864-448-8. URL: http://www.gbv.de/dms/ilmenau/toc/528370707.PDF.

[13] Mario Winter u. a. *Der Integrationstest*. Hanser, 2012. ISBN: 978-3-446-42564-4. URL: http://www.dpunkt.de/buecher/4075.html.

To do…

- ☐ 1 (p. 4): Grafik Back Door Manipulation
- ☐ 2 (p. 4): Layer test erklären
- ☐ 3 (p. 4): DataSets erklären
- ☐ 4 (p. 5): weiter erklären
- ☐ 5 (p. 6): Raw-Setter erklären?
- ☐ 6 (p. 9): Kurz Ziele
- ☐ 7 (p. 11): Beispiel erweitern für 1:1-Beziehungen
- ☐ 8 (p. 11): Attribute einführen
- ☐ 9 (p. 11): Diagramm evtl in Chen-Notation
- ☐ 10 (p. 12): Legende
- ☐ 11 (p. 19): Welche Lizenz?
- ☐ 12 (p. 19): Neues Bild für Schichten
- ☐ 13 (p. 23): Hier wäre eine Quelle super, dass Menschen vertraut mit Tabellen sind
- ☐ 14 (p. 23): Einfache Sprachdefinition, Grammatik
- ☐ 15 (p. 23): Quelle Kent Beck Smalltalk Best Practice Patterns
- ☐ 16 (p. 24): Mögliche ungewollte Seiteneffekte
- ☐ 17 (p. 25): thread local erklären mit quelle
- ☐ 18 (p. 27): Abhängigkeitsdiagramm der neuen Builder-Klassen?
- ☐ 19 (p. 34): Quelle Spring Service
- ☐ 20 (p. 42): „Muster“ für 1:1, 1:n und m:n