**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<br>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

2.1.1 Software-Tests . . . . . 3

**3 Anforderungsanalyse / Fragestellung** **5**

3.1 Allgemeine Anforderungen . . . . . 5

3.2 Fortlaufendes Beispiel . . . . . 6

3.2.1 Voraussetzungen . . . . . 6

3.2.2 Gewählte Problemstellung . . . . . 6

3.2.3 Beispiel-Use-Cases . . . . . 8

3.3 Modellierungsvarianten der Testdaten für DbUnit . . . . . 8

3.3.1 XML-Dataset . . . . . 9

3.3.2 Default-Dataset . . . . . 10

3.3.3 SB Testing DB . . . . . 11

**4 Modellierung der Test-Daten** **15**

4.1 DSL-Entwürfe . . . . . 15

4.1.1 Entwurf 1 . . . . . 15

4.1.2 Entwurf 2 . . . . . 16

4.1.3 Entwurf 3 . . . . . 16

4.2 Implementierung . . . . . 17

4.2.1 Implementierungsvarianten . . . . . 18

**5 Generieren von Testdaten** **19**

**6 Proof of Concept** **21**

**7 Zusammenfassung und Ausblick** **23**

# 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...

### 2.1.1 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. [5, 810f]

Alle Voraussetzungen und Vorbedingungen für einen Testlauf werden unter der Bezeichnung *Test Fixture* zusammengefasst. Es repräsentiert den Zustand des SUT vor den Tests. [5, 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. ([5, 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. ([5, 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. [5, 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)

# Kapitel 3

# Anforderungsanalyse / Fragestellung

## 3.1 Allgemeine Anforderungen

Die Firma *Seitenbau GmbH* verwendet für die Java-basierten Datenbankanwendungen das Framework *JUnit* und die Erweiterung *DbUnit*. Da die Modellierung der Testdaten mit DbUnit-eigenen Mitteln einige Nachteile hat, hat Seitenbau die Bibliothek *SB Testing DB* entwickelt. Allerdings kann SB Testing DB nicht alle Nachteile wett machen, so bleibt die Modellierung von Beziehungen unübersichtlich. Eine genauere Betrachtung der Vor- und Nachteile verschiedener Modellierungsvarianten in Zusammenhang mit DbUnit wird in Abschnitt 3.3 beschrieben.

Die allgemeinen Anforderungen an die zu entwickelnde Sprache 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.

- **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**:

- **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.

## 3.2 Fortlaufendes Beispiel

Eine einheitliche und fortlaufende Problemstellung 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 2.1.1).

### 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:

- Eine Lehrveranstaltung muss von genau einem Professor geleitet werden, ein Professor kann beliebig viele (also auch keine) Lehrveranstaltungen leiten.
- Eine Prüfung ist genau einer Lehrveranstaltung zugeordnet. Eine Lehrveranstaltung kann mehrere Prüfungen haben (z.B. Nachschreibprüfung).
- Eine Prüfung muss mindestens von einem Professor beaufsichtigt werden, ein Professor kann in beliebig vielen Prüfungen Aufsicht haben.
- Jeder Student kann beliebig vielen Lehrveranstaltungen besuchen. Lehrveranstaltungen benötigen jedoch mindestens drei Besucher um stattzufinden und sind aus Kapazitätsgründen auf 100 Teilnehmer begrenzt.

- Jeder Student kann bei beliebig vielen Lehrveranstaltungen Tutor sein und jede Lehrveranstaltung kann beliebig viele Tutoren haben.

- Jeder Student kann an beliebig vielen Prüfungen teilnehmen und umgekehrt eine Prüfung von einer beliebigen Anzahl von Studenten geschrieben werden.

- Ein Raum ist genau einem Professor zugewiesen. Ein Professor kann genau einen oder keinen Raum haben.

Abbildung 3.1 zeigt die Problemstellung grafisch in Form eines ER-Diagramms. **To do** (3) **To do** (4) **To do** (5)

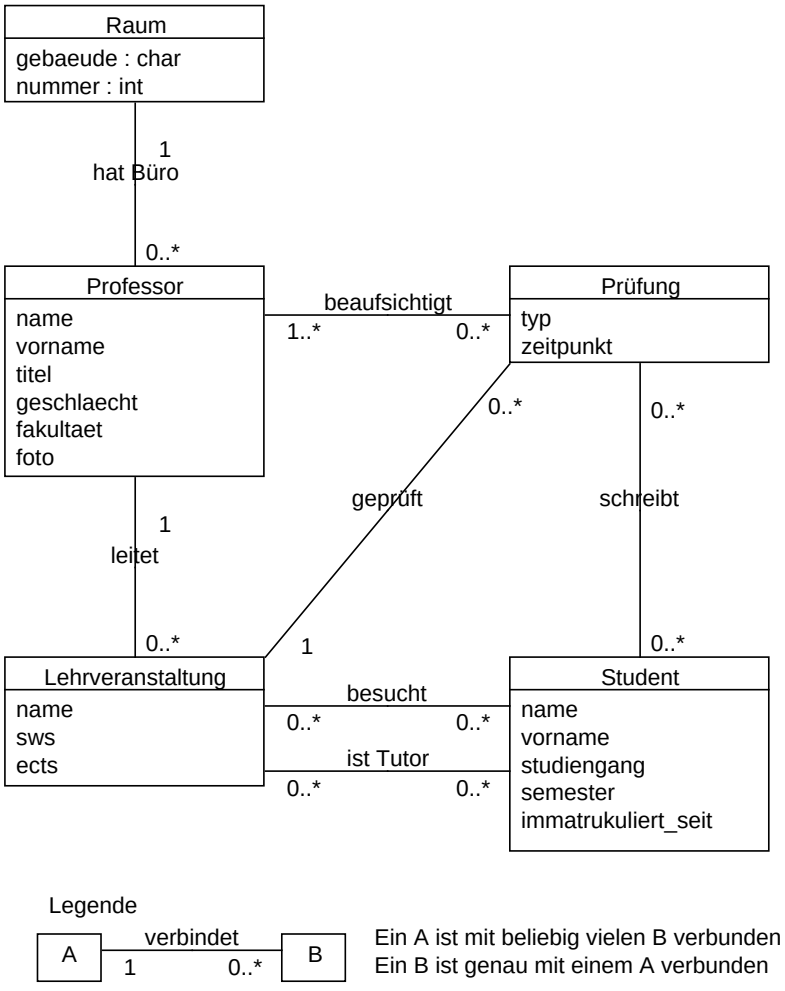


Abbildung 3.1: ER-Diagramm des fortlaufenden Beispiels

Das entsprechende relationale Modell wird in Abbildung 3.2 dargestellt. Die Notationen orientieren sich an den Stil von Ambler in [1].

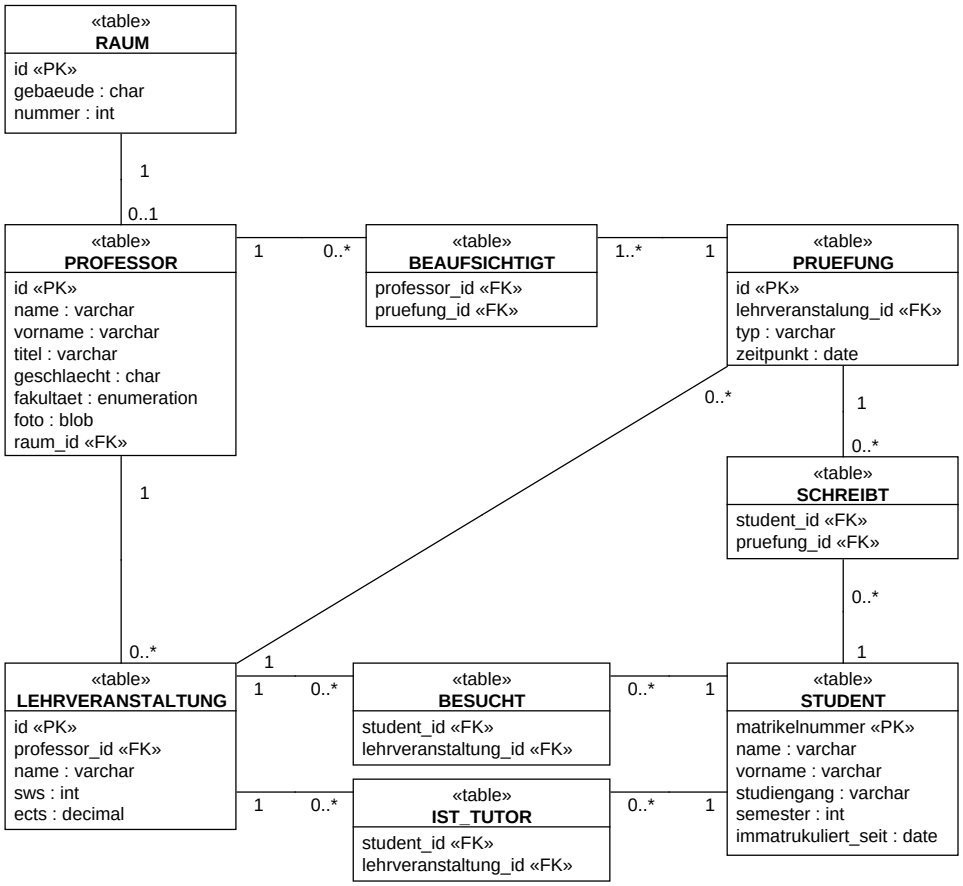


Abbildung 3.2: Relationales Modell des fortlaufenden Beispiels

### 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*, das 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.

```
<!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>
            <value>2</value>
            <value>2</value>
            <value>Design Patterns</value>
            <value>4</value>
            <value>3</value>
        </row>
    </table>
  ...
</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 (Beschrei-

bung 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).

FlatXmlDataSet

```
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" />
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

- Meta-Informationen als Sprachelement - Immer noch eine Zeile pro Wert (starkes vertikales Wachsen) - Deutlich kompakter, dafür ggf. eigene DTD notwendig

### 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.

```
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 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

Die Bibliothek *SB Testing DB* der Firma *Seitenbau GmbH* versucht Nachteile der Xml- und Default-Datasets aufzufangen. Ein Generator erzeugt aus Meta-Informationen zu den Ta-

bellen eine einfache Java-DSL. Über diese DSL können die Testdaten modelliert werden. 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[1]. Die Syntax ist kompakter und dennoch ausdrucksstärker als bei beiden vorherigen Varianten.

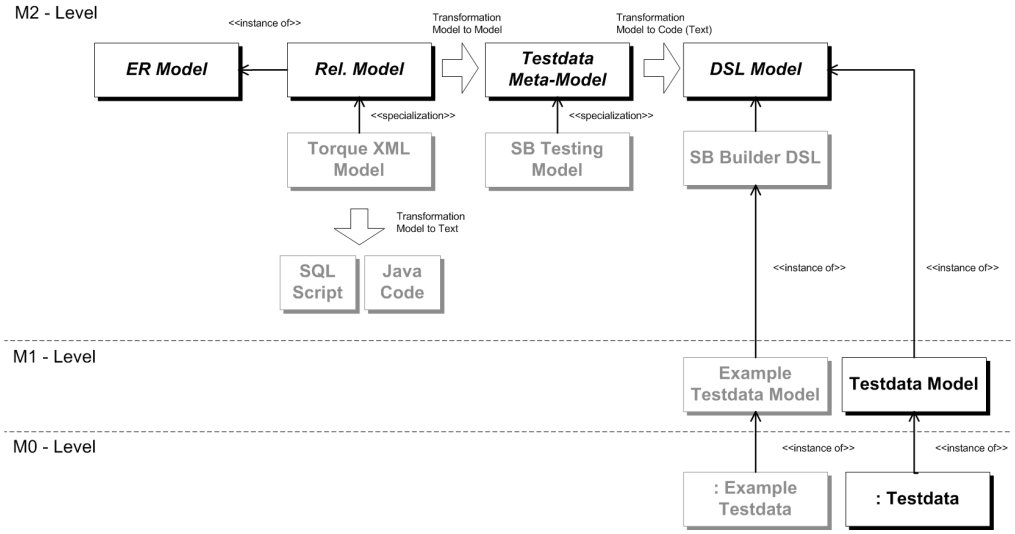


Abbildung 3.3: Modell-Beschreibung

```
table_Professor
  .insertRow()
    .setId(Parameters.Professor.HAASE_ID)
    .setName("Haase")
    .setVorname("Oliver")
    .setTitel("Prof. Dr.")
    .setFakultaet("Informatik")
  .insertRow()
    .setId(Parameters.Professor.WAESCH_ID)
    .setName("Wäsch")
    .setVorname("Jürgen")
    .setTitel("Prof. Dr.-Ing.")
    .setFakultaet("Informatik");

table_Lehrveranstaltung
  .insertRow()
    .setId(Parameters.Lehrveranstaltung.VSYSTEME_ID)
    .setProfessorId(Parameters.Professor.HAASE_ID)
    .setName("Verteilte Systeme")
    .setSws(4)
    .setEcts(5)
  .insertRow()
    .setId(Parameters.Lehrveranstaltung.DESIGN_PATTERNS_ID)
    .setProfessorId(Parameters.Professor.HAASE_ID)
    .setName("Design Patterns")
    .setSws(4)
    .setEcts(3);
```

Listing 3.4: SB Testing Dataset (1)

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

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 (siehe Listing 3.5).

```
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: SB Testing Dataset (2)

# Kapitel 4

# Modellierung der Test-Daten

## 4.1 DSL-Entwürfe

### 4.1.1 Entwurf 1

Eine DSL, die sich stark an *SB Testing DB* 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: 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.

### 4.1.2 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.

```
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: Mögliche DSL (2)

### 4.1.3 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).

```
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.

Der Double-Pipe-Operator (||) soll die Spalte mit dem Entitätsidentifikatoren visuell von den Datenspalten trennen.

## 4.2 Implementierung

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 von Java ab. **To do** (6) 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 [4, 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.

### 4.2.1 Implementierungsvarianten

Eine DSL kann auf unterschiedliche Art 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[2]. Beide Ansätze bieten individuelle Vorteile, die im folgenden diskutiert werden.

#### Laufzeit-Meta-Programmierung

Die Tabellen mit den Daten können als Closures an einen Parser zum Interpretieren übergeben werden. - use zum „Einblenden“ von Methoden - Operator-Überladen der Einfachheit halber statisch (anders nur extrem umständlich) - da statische Operatoren Speichern der Ergebnisse in ThreadLocal-Variable - Kontext des Closures geändert, um lokalen Operatoren Vorrang zu geben - getProperty-Aufruf bei Variablen/Referenzen

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 4.5). Ein Platzhalter (Unterstrich) verhindert Syntax-Fehler, wenn in einer Spalte kein Wert vorkommt (siehe Zeile 4, Spalte „leitet“). Aus Sicht des Parsers stellt der dieser eine Variable dar.

```
1  def fixture = [
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 4.5: 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 bietet 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. kann er

**To do** (7)

[2] AST ist die Abkürzung für *Abstract Syntax Tree*

# 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. [6]
2. [8, 20ff]
3. [3]
4. [7]
5. [2]

## Bild


Abbildung 7.1: Der Titel

## Bildgruppe


(a) Text 1


(b) Text 2

(c) Text 3

Abbildung 7.2: Gemeinsamer Titel

# Abbildungsverzeichnis

3.1 ER-Diagramm des fortlaufenden Beispiels . . . . . 7
3.2 Relationales Modell des fortlaufenden Beispiels . . . . . 8
3.3 Modell-Beschreibung . . . . . 12

7.1 Der Titel . . . . . 26
7.2 Gemeinsamer Titel . . . . . 26

# Listings

3.1 XML Dataset . . . . . 9
3.2 Flat XML Dataset . . . . . 10
3.3 Default Dataset . . . . . 10
3.4 SB Testing Dataset (1) . . . . . 12
3.5 SB Testing Dataset (2) . . . . . 13
4.1 Mögliche DSL (1) . . . . . 15
4.2 Mögliche DSL (2) . . . . . 16
4.3 Mögliche DSL (3) . . . . . 16
4.4 Vereinfachung von Ausdrücken in Groovy . . . . . 17
4.5 DSL-Entwurf 3 für Laufzeit-Meta-Programmierung angepasst . . . . . 18
7.1 Der Titel . . . . . 25

# 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] 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.

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

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

[5] 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.

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

[7] Thomas Stahl. *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. 2., aktualisierte und erw. Aufl. Heidelberg: dpunkt-Verl., 2007, XV, 441 S. ISBN: 978-3-89864-448-8. URL: http://www.gbv.de/dms/ilmenau/toc/528370707.PDF.

[8] 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. 7): Beispiel erweitern für 1:1-Beziehungen
- ☐ 4 (p. 7): Attribute einführen
- ☐ 5 (p. 7): Diagramm evtl in Chen-Notation
- ☐ 6 (p. 17): Closures erklären
- ☐ 7 (p. 18): „Muster“ für 1:1, 1:n und m:n