**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**
Hochschule für Technik, Wirtschaft und Gestaltung

Fakultät Informatik
Studiengang 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

# Kapitel 1

# Einleitung

# Kapitel 2

# Grundlagen

## 2.1 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.

### 2.1.1 Voraussetzungen

Der Schwerpunkt der Modellierung liegt bei der Darstellung von Beziehungen zwischen Entitäten. Dabei soll die Problemstellung einerseits nicht zu komplex sein, damit sie überschaubar bleibt. Andererseits soll sie komplex genug sein, um möglichst alle Bezeihungensarten zwischen Entitäten abzudecken.

Die Testdaten sollten so gewählt werden, dass idealerweise für alle Tests die selben Daten verwendet werden können. Einheitiche Daten sorgen dafür, dass sich der Entwickler (verbessern) nicht bei verschiedenen Tests in unterschiedliche Testdaten hineinversetzten muss. Nur in Ausnahmefällen sollten Tests modifizierte oder eigene Testdaten verwenden. Um dem Entwickler entgegen zu kommen, sollte der Umfang der Testdaten nicht größer sein als erforderlich.

### 2.1.2 Gewählte Problemstellung

Das gewählte Beispiel stellt eine starke Vereinfachung des Prüfungswesens an Hochschulen dar. Auf eine praxisnahe Umsetzung wird zugunsten der Komplexität verzichtet. Es beinhaltet die folgenden vier Entitäten:

- **Professor**: Ein Professor leitet Lehrveranstaltugnen.
- **Lehrveranstaltung**: Eine Lehrveranstaltng wird von einem Professor geleitet. Es kann zu jeder Lehrveranstaltung eine Prüfung geben.
- **Prüfung**: Eine Prüfung ist einer Lehrveranstaltung zugeordnet. Außerdem hat mindestens ein Pofessor Aufsicht.

- **Student**: Studenten können an Lehrveranstaltungen und an Prüfungen teilnehmen. Studenten haben außerdem die Möglichkeit, Tutoren von Lehrveranstaltugen zu sein.

Die Beziehungen der Entitäten 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. Nachschreibeprüfung). 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 und Lehrveranstalungen von beliebig vielen Studenten besucht werden. Die gleiche Bezeihung gilt für Tutoren: Jeder Student kann bei beliebig vielen Lehrveranstaltungen Tutor sein und jede Lehrveranstaltung kann beliebig viele Tutoren haben. Schließlich kann jeder Student auch an beliebig vielen Prüfungen teilnehmen und umgekehrt eine Prüfung von einer beliebigen Anzahl von Studenten geschrieben werden.

Abbildung 2.1 stellt die Problemstellung grafisch dar. Die Abbildung zeigt, dass es keine 1:1 Beziehung gibt. Eine 1:1-Beziehung kann jedoch als Spezialfall einer 1:n-Beziehung angesehen werden.

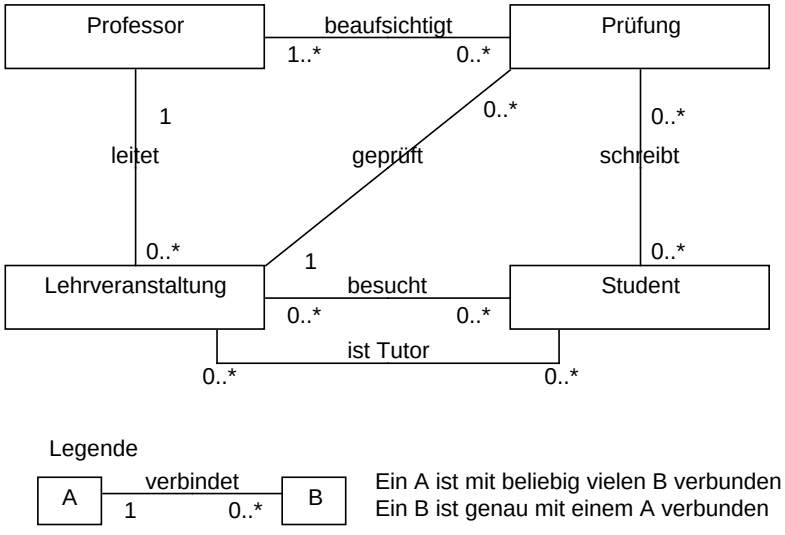


Abbildung 2.1: ER-Diagramm des fortlaufenden Beispiels

Das entsprechende relationale Modell sieht folgendermaßen aus:

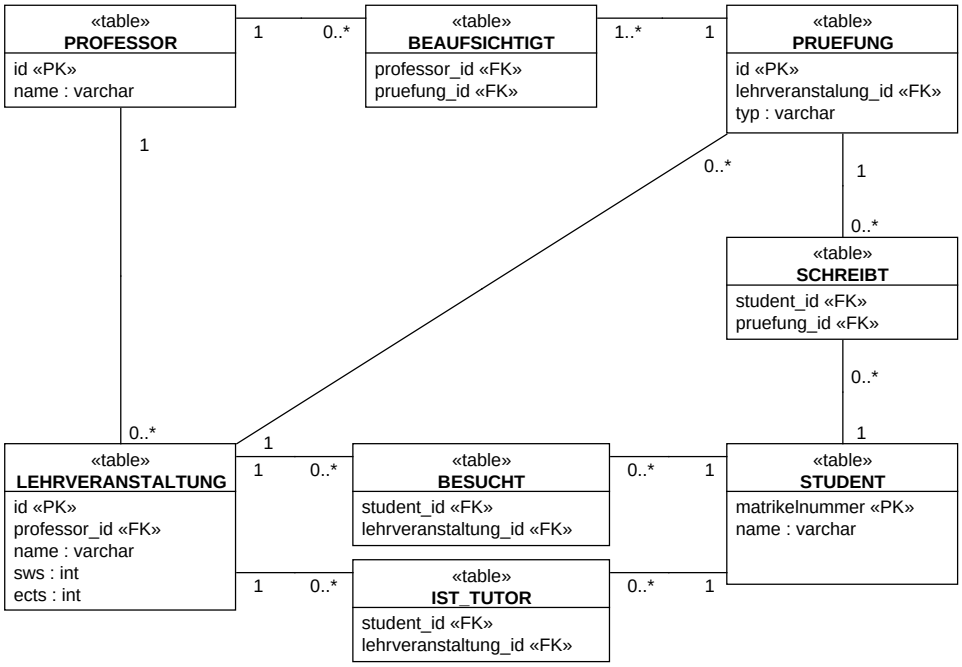


Abbildung 2.2: Relationales Modell des fortlaufenden Beispiels

### 2.1.3 Wahl der Testdaten

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)?

**To do** (1)

## 2.2 Modellierung der Testdaten in DbUnit

**To do** (2) Vorteile Bac Door Manioulation vs Layer Test

- schneller, v.a. wenn SUT „langsam“ ist
- SUT kann Fehler „verberegen“ (schreibt und liest in falscher Spalte)

- Flexibler, beliebige Zustände implementierbar (auch fehlerhafte)

Nachteile

- Plattform-spezifisch

DbUnit-Tests basieren durchlaufen im Allgemeinen folgenden Schritte:

- Dataset aufbauen und in Datenbank einspielen
- Änderung an Datenbank durchführen (zu testende Operation durchführen)
- Zweites Dataset aufbauen, das den neuen Zustand der Datenbank widerspiegelt
- Aktuelle Datenbank mit weitem Dataset vergleichen

Dabei stellt es mitunter ein Problem dar, aus einem bestehenden DbUnit-Dataset ein zweites Dataset zu erzeugen, das die Änderungen an der Datenbank beinhaltet. So gibt es keine Möglichkeit in einem solchen Dataset, eine Zeile zu löschen.

**To do** (3)

### 2.2.1 XML-Dataset

Eine Variante, ein Dataset für *DbUnit* zu modellieren, stellt XML dar. *DbUnit* biete dazu die Klasse *XmlDataSet*, das eine XML-Datei nach einem vorgegebenen Dokumententyp einlesen kann. Das Listing 2.1 zeigt einen Ausschnitt einer solchen XML-Datei, in dem die beiden Tabellen *professor* und *lehrveranstaltung* definiert weden.

```
<!DOCTYPE dataset SYSTEM "dataset.dtd">
<dataset>
  <table name="PROFESSOR">
    <column>id</column>
    <column>name</column>
    <row>
      <value>1</value>
      <value>Jürgen Wäsch</value>
    </row>
    <row>
      <value>2</value>
      <value>Oliver Haase</value>
    </row>
  </table>
  <table name="LEHRVERANSTALTUNG">
    <column>id</column>
    <column>professor_id</column>
    <column>name</column>
    <row>
      <value>1</value>
      <value>2</value>
      <value>Verteilte Systeme</value>
    </row>
    <row>
      <value>2</value>
      <value>2</value>
      <value>Design Patterns</value>
    </row>
  </table>
```

```
30   ...
31 </dataset>
```

Listing 2.1: XML Dataset

Die positiven Eigenschaften bei der Modellierung in XML sind unter anderem, dass es für XML 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 4-5 und 16-18) reichen dafür nicht aus. Die Meta-Informationen sind redundnat 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 angegeben werden, die Werte tauchen redundant auf und sind ohne Kommentare in umfangreicheren Datasets für Betrachter nur schwer nachzuvollziehen. Die Beziehungen werden auf einer sehr niedrigen Abstraktionsebene ausgedrückt und

Ein großer Nachteil von XML-Datasets ist, dass der erwartete Datenbankzustand selbst wieder den kompletten Datenbankbestand umfassen muss. Eine inkrementelle bzw. differnetielle Definition der erwarteten Änderungen ist über XML-Datasets 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).

### 2.2.2 Java-Dataset

Um einige der Probleme zu vermeiden, die in Verbindung mit XML-Datasets auftreten, können Java-Datasets verwendet werden. Diese lassen sich dynamisch erstellen. Java-Datasets lösen einen Teil der Redundanz-Probleme, da symbolische Konstanzen anstelle von numerischen verwendet werden können. Das Erzeugen des Datasets, das den 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      }
7    );
8  professor.addRow(new Object[] {
9      Parameters.Professor.WAESCH_ID,
10     "Jürgen␣Wäsch"
11   });
12 professor.addRow(new Object[] {
13     Parameters.Professor.HAASE_ID,
14     "Oliver␣Haase"
15   });
16 dataSet.addTable(professor);
17
18 DefaultTable lehrveranstaltung = new DefaultTable(
```

```
19      "lehrveranstaltung",
20      new Column[] {
21        new Column("id", DataType.INTEGER),
22        new Column("professorid", DataType.INTEGER),
23        new Column("name", DataType.VARCHAR),
24      }
25    );
26  lehrveranstaltung.addRow(new Object[] {
27      Parameters.Lehrveranstaltung.VERTEILTE_SYSTEME_ID,
28      Parameters.Professor.HAASE_ID,
29      "Verteilte Systeme"
30    });
31  lehrveranstaltung.addRow(new Object[] {
32      Parameters.Lehrveranstaltung.DESIGN_PATTERNS_ID,
33      Parameters.Professor.HAASE_ID,
34      "Concurrency and Design Patterns"
35    });
36  dataSet.addTable(lehrveranstaltung);
```

Listing 2.2: Java Dataset

Diese Umsetzung löst allerdings nicht alle Probleme. So müssen immer noch Meta-Informationen über die Tabellen odelliert werden. Obwohl diese sogar Typinformationen beinhalten, werden Typ-Fehler erst zur Laufzeit erkannt. Der Einsatz von symbolischen Konstanten verringert die Redundanz zwar und erleichtert damit die Pflege des Datasets, dennoch lassen sich Konstanten doppelt belegen und es können 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 unübersichtlch. Insgesamt bietet die Nutzung der Java-Datasets in dieser Art nur wenig Vorteile gegenüber den XML-Datasets.

### 2.2.3 SB Testing DSL

**To do** (4) Generator erzeugt DSL auf Basis von Meta-Daten (M2).

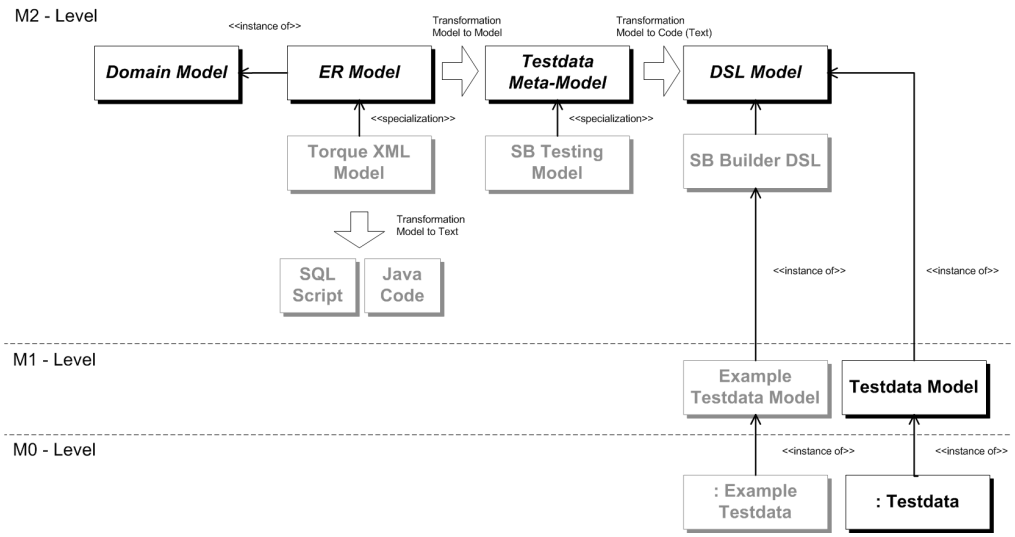


Abbildung 2.3: Modell-Beschreibung

```
table_Professor
  .insertRow()
     .setId(Parameters.Professor.HAASE_ID)
     .setName("Oliver Haase")
  .insertRow()
    .setId(Parameters.Professor.WAESCH_ID)
    .setName("Jürgen Wäsch");

table_Lehrveranstaltung
  .insertRow()
    .setId(Parameters.Lehrveranstaltung.
       VERTEILTE_SYSTEME_ID)
    .setProfessorId(Parameters.Professor.HAASE_ID)
    .setName("Verteilte Systeme")
  .insertRow()
    .setId(Parameters.Lehrveranstaltung.
       DESIGN_PATTERNS_ID)
    .setProfessorId(Parameters.Professor.HAASE_ID)
    .setName("Design Patterns");
```

Listing 2.3: SB Testing Dataset (1)

Die SB Testing DSL versucht die Nachteile der Java-Datasets zu umgehen. Die Syntax ist kompakter und dennoch ausdrucksstärker als die beiden vorherigen Varianten. Außerdem können falsche Typen bereits zur Compilierzeit erkannt werden.

Die Modellierung von Referenzen stellt sich als ähnlich problematisch wie bei den bisherigen Java-Datasets dar (siehe Abschnitt 2.2.2). 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 2.4).

```
RowBuilder_Professor haase =
  table_Professor
    .insertRow()
      .setName("Oliver Haase");
RowBuilder_Professor waesch =
  table_Professor
    .insertRow()
      .setName("Jürgen Wäsch");

RowBuilder_Lehrveranstaltung vsys =
  table_Lehrveranstaltung
    .insertRow()
      .setName("Verteilte Systeme")
      .refProfessorId(haase);
RowBuilder_Lehrveranstaltung design_patterns =
  table_Lehrveranstaltung
    .insertRow()
      .setName("Design Patterns")
      .refProfessorId(haase);
```

Listing 2.4: SB Testing Dataset (2)

## 2.3 Die DSL

### 2.3.1 Anforderungen an die DSL

Die Testdaten sollen in einer *Domain Specific Language* (DSL) beschrieben werden.

**To do** (5)

- Integration in bestehende Tool-Kette
- In Java nutzbar (interne DSL), Dataset soll zur Laufzeit erweiterbar bleiben in selber Syntax
- Zugriff auf Daten aus Java heraus
- „Meta-Informtionen als Sprachelement“
- Symbolische Konstanten überflüssig machen
- „Dekorieren“ von Werten (before(date))
- wenig syntaktischer Ballast
- Typ-Prüfungen zu Compilierzeit
- übersichtliche Darstellung ermöglichen (v.a. vertikal)
- Namensräume/Scopes
- Beziehungen sollen sich einfach und typsicher modellieren lassen

### 2.3.2 Zielgruppe

### 2.3.3 DSL-Entwürfe

# Kapitel 3

# Zusammenfassung und Ausblick

# Kapitel 4

# Titel

## 4.1 Untertitel

### 4.1.1 Stichpunkte

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

### 4.1.2 Aufzählung

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

### 4.1.3 Abkürzung

### 4.1.4 Quellcode

```
1 Code
```

Listing 4.1: Der Titel

### 4.1.5 Verweise

1. siehe 4.1
2. siehe Listing 4.1
3. siehe Abb. 4.1
4. siehe Abschnitt 4.1
5. siehe Kapitel 4

### 4.1.6 Zitate

1. [2]
2. [3]
3. [1]
4. [5]
5. [6, 20ff]
6. [4]

### 4.1.7 Bild


Abbildung 4.1: Der Titel

### 4.1.8 Bildgruppe


(a) Text 1


(b) Text 2

(c) Text 3

Abbildung 4.2: Gemeinsamer Titel

# Abbildungsverzeichnis

# Listings

# 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] Author. *Title*. Organisation. 2009. URL: http://www.gidf.de (besucht am 05. 06. 2011).

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

[4] 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.

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

[6] 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. 5): Daten beschreiben
- ☐ 2 (p. 5): Back Door Manipulation
- ☐ 3 (p. 6): DbUnit kurz beschreiben (Erweiterung von JUnit)
- ☐ 4 (p. 8): SB Testing DSL
- ☐ 5 (p. 9): Anforderungen