



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
2.2.1 Datenbank-Tests	4
2.3 Konventionen	5
2.3.1 ER-Diagramme	5
2.3.2 Relationale Datenbank-Diagramme	5
2.4 STU (Simple Test Utils)	5
3 Anforderungsanalyse / Fragestellung	9
3.1 Allgemeine Anforderungen	9
3.2 Modellierungskonzepte für Beziehungen	10
3.2.1 1:1-Beziehungen	10
3.2.2 1:n-Beziehungen	10
3.2.3 n:m-Beziehungen	11
3.2.4 Andere Beziehungen	11
3.3 Kriterien für Sprachbewertung	11
3.4 Fortlaufendes Beispiel	11
3.4.1 Anforderungen an das Beispiel	11
3.4.2 Gewählte Problemstellung	12
3.4.3 Beispiel-Use-Cases	14
3.5 Modellierungsvarianten der Testdaten für DbUnit	15
3.5.1 XML-DataSet	15
3.5.2 Default-DataSet	17
3.5.3 STU-DataSet	18

4	Entwurf einer Modellierungssprache für Test-Daten	21
4.1	Entwurf der DSL	21
4.1.1	Beispiel-Entwurf 1	21
4.1.2	Beispiel-Entwurf 2	22
4.1.3	Beispiel-Entwurf 3	23
4.1.4	Finaler DSL-Entwurf	24
4.2	Wahl der Technologie	24
4.2.1	Implementierungsvarianten mit Groovy	25
4.2.2	Implementierungsentscheidung	28
4.3	Definition der Sprache	28
4.4	Beispiel-DataSet in Groovy	29
5	Realisierung der Sprache	31
5.1	Änderungen am Generator-Modell	32
5.1.1	Spalten-Eigenschaften	33
5.1.2	Modellierung von Relationen über Builder-Klassen	34
5.1.3	Alte und neue Builder-Klassen im Vergleich	34
5.2	Neue DataSet-Builder-Klassen	36
5.3	Tabellenparser	36
5.4	Referenzen und Scopes	38
5.5	Nutzung des DataSets in Unit-Tests	39
5.6	Komposition von DataSets	40
5.7	Erweiterungen in generierter API	41
5.8	JavaDoc	44
5.9	Verhalten bei Fehlern in den Tabellendefinitionen	44
5.10	Nicht umgesetzt	45
5.10.1	Zusammengesetzte Schlüssel	45
5.10.2	Unterstützung für weitere Beziehungstypen	45
5.10.3	Komfortfunktionen	47
6	Generieren von Testdaten	49
6.1	Generieren von Beziehungen	49
6.1.1	Klasse der 1:1-Beziehungen	49
6.1.2	Klasse der 1:n-Beziehungen	51
6.1.3	Klasse der n:m-Beziehungen (assoziative Beziehungen)	51
6.2	Der Algorithmus	51
6.3	Grenzen des Algorithmus	52
6.4	Erweiterungen am Modell	52
6.5	-Warum Generierung	52

INHALTSVERZEICHNIS

7 Proof of Concept	53
8 Zusammenfassung und Ausblick	55
Abbildungsverzeichnis	57
Listings	60
Literaturverzeichnis	62

Kapitel 1

Einleitung

Ungenutzte Quellen

1. [SL10]
2. [Win+12, 20ff]
3. [Sta07]
4. [Eva04]

Kapitel 2

Grundlegende Konzepte

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

2.1 Modellgetriebene Software-Entwicklung

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

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

To do (1)

- DSL, intern vs. extern

2.2 Software-Tests

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

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

- **Standard Fixture:** Ein Test Fixture wird als Standard Fixture bezeichnet, wenn es für alle bzw. fast alle Tests verwendet werden kann. Ein Standard Feature reduziert nicht nur den Aufwand zum Entwerfen von Testdaten für die einzelnen Tests, sondern

verhindert darüber hinaus, dass der Tester sich bei verschiedenen Tests immer wieder in unterschiedliche Test-Daten hineinversetzen muss. Nur in Ausnahmefällen sollten Tests modifizierte oder eigene Testdaten verwenden. [Mes07, S. 305]

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

2.2.1 Datenbank-Tests

Ein übliches Muster, Systeme in Verbindung mit Datenbanken zu testen, ist *Back Door Manipulation*. Dieses Muster besteht aus vier Schritten. Im ersten Schritt, dem *Setup*, wird die Datenbank über direkten Zugriff, vorbei am zu testenden System, in den Anfangszustand gebracht. Anschließend können im *Exercise*-Schritt die zu testenden Operationen am System durchgeführt werden. Die Überprüfung, ob sich das System richtig verhalten hat, findet im als *Verify* bezeichneten Schritt statt. Dabei wird der Zustand der Datenbank mit dem erwarteten Zustand verglichen, ebenfalls am zu testenden System vorbei. Abschließend kann der vierte Schritt, *Teardown*, noch Aufräumarbeiten durchführen. Abbildung 2.1 stellt Back Door Manipulation grafisch dar. [Mes07, 327ff]

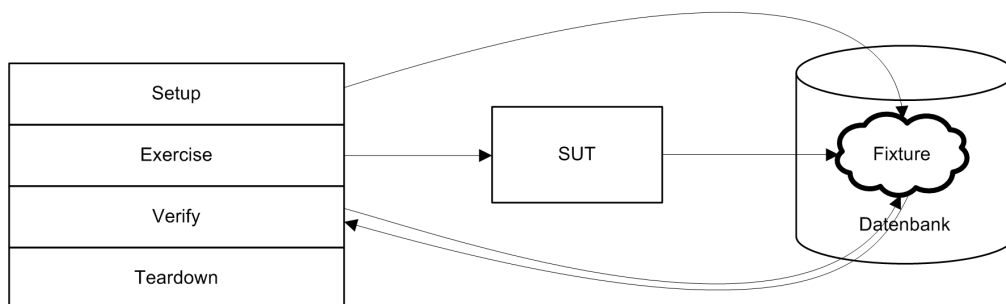


Abbildung 2.1: Back Door Manipulation

Es gibt mehrere Vorteile, die Datenbank nicht über das zu testende System in den Anfangszustand zu bringen. Einerseits können semantische Fehler im zu testenden System unter Umständen nur so gefunden werden. Andererseits kann der Zustand mitunter schneller in die Datenbank geschrieben werden, wenn nicht der Weg über das zu testende System gemacht wird. Außerdem bietet es in Bezug auf die Zustände eine höhere Flexibilität: Die Datenbank kann auch in Zustände gebracht werden, die über das System nicht erreicht werden können. Dafür leidet die Flexibilität an einer anderen Stelle: Die Tests sind abhängig vom konkret verwendeten Datenbank-System. Wird die Datenbank von SQL auf NoSQL umgestellt, müssen die Tests angepasst werden. Außerdem setzt der direkte Zugriff auf die Datenbank voraus, dass die Semantik der zu testenden Anwendung berücksichtigt wird. Aus Sicht der Anwendung dürfen sich von der Anwendung eingespielte Daten in ihrer Form nicht von den manuell in die Datenbank geschriebenen Daten unterscheiden.

To do (2)

2.3 Konventionen

2.3.1 ER-Diagramme

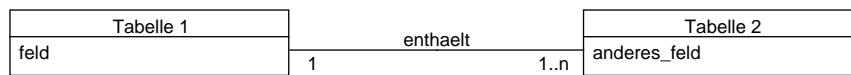


Abbildung 2.2: Stil von ER-Diagrammen

2.3.2 Relationale Datenbank-Diagramme

Alle in dieser Arbeit abgebildeten relationalen Datenbank-Diagramme verwenden einen einheitlichen Modellierungsstil, der auch von Ambler in Ambler in [AS06] verwendet wird. Dieser Stil erweitert das dem UML-konform das Klassendiagramm.

Tabellen werden durch Boxen repräsentiert. Diese Boxen sind unterteilt in zwei Bereiche: den Tabellenbezeichner (oben) und die Attribute der Tabelle (unten). Attribute werden in der Form *Bezeichner : typ* beschrieben. Es gibt die beiden Stereotypen *PK* für *Primary Key* und *FK* für *Foreign Key*, um die Attribute entsprechend zu klassifizieren.

Beziehungen zwischen Tabellen werden durch Verbindungslinien dargestellt. Die Kardinalitäten beschreiben die Art der Verbindung. Die Bedeutung der Kardinalitäten lassen sich am besten über ein Beispiel erklären (Abbildung 2.3). Eine Entität aus TABELLE 1 ist mit mindestens einer und beliebig vielen Entitäten aus TABELLE 2 in einer Beziehung, jede Entität aus TABELLE 2 ist mit genau einer Entität aus TABELLE 1 in Beziehung. Beziehungen können zusätzlich mit einem Bezeichner versehen werden.

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

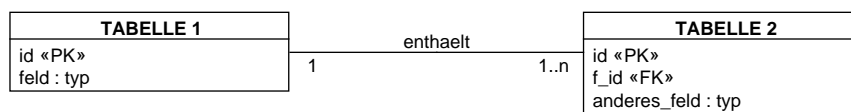


Abbildung 2.3: Stil relationaler Datenbank-Diagramme nach Ambler

2.4 STU (Simple Test Utils)

STU ist eine Bibliothek zur Vereinfachung von Unit-Tests für Java-basierte Anwendungen. *STU* steht unter der *Apache License 2.0* [Fou04] und wird maßgeblich von der Firma SEITENBAU entwickelt. Diese Arbeit thematisiert nur den Teil von *STU*, der Tests von Datenbank-Anwendungen vereinfachen soll.

Für Unit-Tests von Datenbank-Anwendungen nutzt *STU* die Bibliothek *DbUnit*. *DbUnit* stellt eine Erweiterung für JUnit dar und stellt Funktionen speziell für den Test von

Datenbank-Anwendung zur Verfügung [Mes07, S. 748]. Mit Hilfe von *DbUnit* kann eine Datenbank zu Beginn eines Tests in einen definierten Zustand gebracht werden. Dazu kann *DbUnit* DataSets in die Datenbank einspielen. ^{To do} (3) Außerdem kann *DbUnit* den Inhalt einer Datenbank mit einem DataSet vergleichen. So kann nach dem Test überprüft werden, ob das SUT die richtigen Modifikationen an den Daten vorgenommen hat.

— Nachteile von *DbUnit*-DataSets - vorwegnehmen ? —

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

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

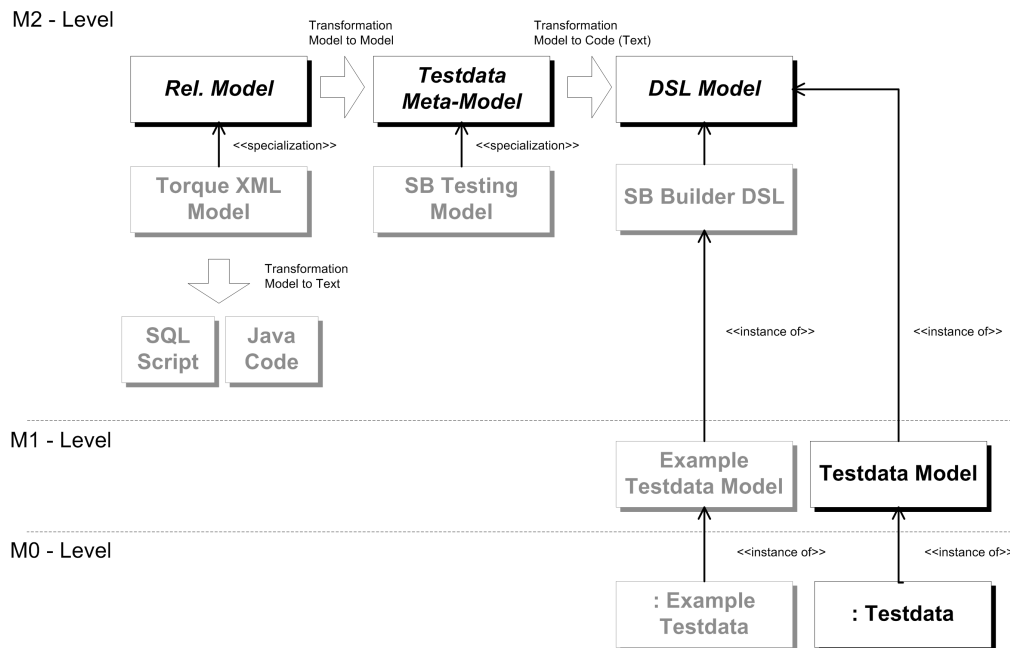


Abbildung 2.4: Modell-Beschreibung

Das STU-Modell enthält keine Datenbank-Constraints. Eine Abbildung dieser würde keine wesentliche Vorteile bringen. Das API bzw. die erzeugten DataSets sind ausschließlich für den Einsatz im Test-Umfeld gedacht. Sollte ein DataSet Daten enthalten, die gegen die in der Datenbank definierten Constraints verstoßen, scheitert das Einspielen des DataSets in die Datenbank und eine Fehlermeldung wird ausgelöst. Aus Sicht des Testers ist dieses Ver-

halten ausreichend, da die Exception zum Scheitern der Unit-Tests führen wird. Der Mehrwert, dass ungültige DataSets schon vor dem Einspielen als solches zu erkennen, ist gering im Vergleich zu dem Aufwand, Constraint-Mechanismen verschiedener Datenbanksysteme nachzubauen.

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

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

- **DataSet:** Ein DataSet repräsentiert eine Menge von Testdaten. Es umfasst alle Tabellen eines Datenbankmodells. Es wird eine abstrakte DataSet-Klasse generiert. Der Zugriff auf die Tabellen erfolgt über öffentliche Felder. Die Klasse enthält die Methode `createDBUnitDataSet`, um die für die Unit-Tests benötigten DbUnit-Datasets zu erzeugen. Dabei werden Template-Methoden **To do** ⁽⁴⁾ 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:** Eine Table-Klasse fasst alle Testdaten in Form von Zeilen einer Tabelle zusammen. Für jede Tabelle wird eine individuelle Klasse generiert. Der Klassenname setzt sich aus dem Namen der Tabelle und dem Suffix „Table“ zusammen. Die Klasse stellt Methoden zur Modellierung und für den Zugriff auf die Tabellendaten bereit. Zur Integration in DbUnit implementiert sie das DbUnit-Interface `ITable`.
- **RowBuilder:** Eine Zeile einer Tabelle wird von einem RowBuilder repräsentiert. Zu jeder Tabelle wird eine individuelle RowBuilder-Klasse erzeugt. Sie beinhaltet für jede Spalte mehrere Methoden zum Setzen und Abfragen des jeweiligen Wertes. Die Methodennamen setzen sich zusammen aus der Aufgabe (`get` bzw. `set`) und dem Spaltennamen.
- **FindWhere:** Für einfache Suchanfragen gibt es für jede Tabelle die innere Klasse `FindWhere`. Sie ermöglicht die Suche nach einem Wert in einer Spalte und liefert eine Liste von Tabellenzeilen. Die Methode ist zur Suche nach Zeilen vorgesehen, von denen erwartet wird, dass es sie gibt.

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

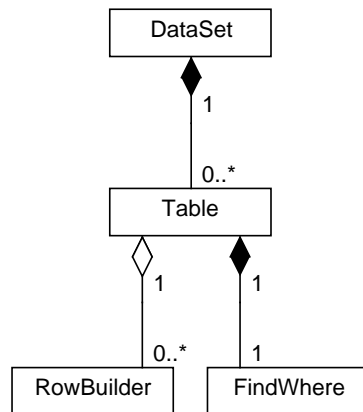


Abbildung 2.5: Logisches Klassendiagramm der STU-DataSet-Klassen

Kapitel 3

Anforderungsanalyse / Fragestellung

Einleiten: Zwei Fragestellungen, DSL und Generierung

3.1 Allgemeine Anforderungen

Die Hauptziele dieser Arbeit stellen sich wie folgt dar:

1. Vereinfachen der Modellierung von Beziehungen
2. Modellierung von Test-Daten übersichtlicher machen
3. Automatisches Generieren von Test-Daten

Für die Modellierung gelten diese allgemeinen Anforderungen:

- **Integration in bestehende Werkzeugkette:** Die Lösung sollte sich nach Möglichkeit in die bestehende Werkzeugkette von SEITENBAU integrieren lassen.
- **IDE-Integration:** Bedienbarkeit für den Tester stellt eine der wichtigsten Anforderungen dar. Daten sollen komfortabel modelliert werden können. Die Integration in Entwicklungsumgebungen wie Eclipse oder IntelliJ IDEA muss gegeben sein.
- **Beziehungen:** Beziehungen sollen einfach modellieren werden können.
- **Gültigkeitsbereiche:** To do (5)
- **Veränderbarkeit von DataSets:** DataSets sollen sich bei der Modellierung beliebig verändern lassen.
- **Komposition:** DataSets sollen sich aus anderen DataSets zusammensetzen lassen.
- **Typ-Sicherheit:** Die Beschreibung der Daten sollte typsicher erfolgen. Idealerweise sollten falsche Typen schon während des Compilierns erkannt werden.
- **Funktionen als Werte:** Es soll möglich sein, Hilfsfunktionen zur Berechnung von Werten zu verwenden, z.B. zum Einlesen von Binary Large Objects (BLOBs) aus Dateien.

- **Zielgruppe:** Die Zielgruppe für die DSL sind Software-Entwickler und Tester. Der Code zur Modellierung der Daten sollte auch für andere Projekt-Mitglieder lesbar und verständlich sein.
- **Ungültige Daten:** Es sollen sich auch aus Sicht der Datenbank oder des SUT ungültige Daten modellieren lassen.

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

- **Deterministische Generierung:** Auch wenn die Test-Daten aus Zufallsdaten bestehen, sollen sie deterministisch generiert werden können. Das heißt, dass die Generierung des Modells mit den selben Einstellungen auch zum selben Ergebnis führt.
- **Kompatibilität:** Die Generierung der Testdaten soll nicht nur bei Nutzung der neuen Modellierungssprache verwendet werden können. Der Testdaten-Generator soll auch DataSets auf Basis der bisherigen STU-Builder erstellen können.

3.2 Modellierungskonzepte für Beziehungen

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

3.2.1 1:1-Beziehungen

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

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

3.2.2 1:n-Beziehungen

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

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

3.2.3 n:m-Beziehungen

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

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

3.2.4 Andere Beziehungen

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

3.3 Kriterien für Sprachbewertung

- Zeilen
- Spalten - für Bildschirm angemessen?
- Zeichenverhältnis
- Benutzbarkeit: Verständlichkeit, Erlernbarkeit
- Modifizierbarkeit

3.4 Fortlaufendes Beispiel

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

3.4.1 Anforderungen an das Beispiel

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

3.4.2 Gewählte Problemstellung

Das gewählte Beispiel stellt eine starke Vereinfachung des Prüfungswesens an Hochschulen dar. Auf eine praxisnahe Umsetzung wird zugunsten der Komplexität verzichtet. Personenbezogene Begriffe werden in der maskulinen Form verwendet, ohne dabei Aussagen über das Geschlecht der repräsentierter Personen zu machen. Es beinhaltet die folgenden vier Entitätstypen:

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

Die Beziehungen der Entitätstypen stellen sich wie folgt dar:

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

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

3.4. FORTLAUFENDES BEISPIEL

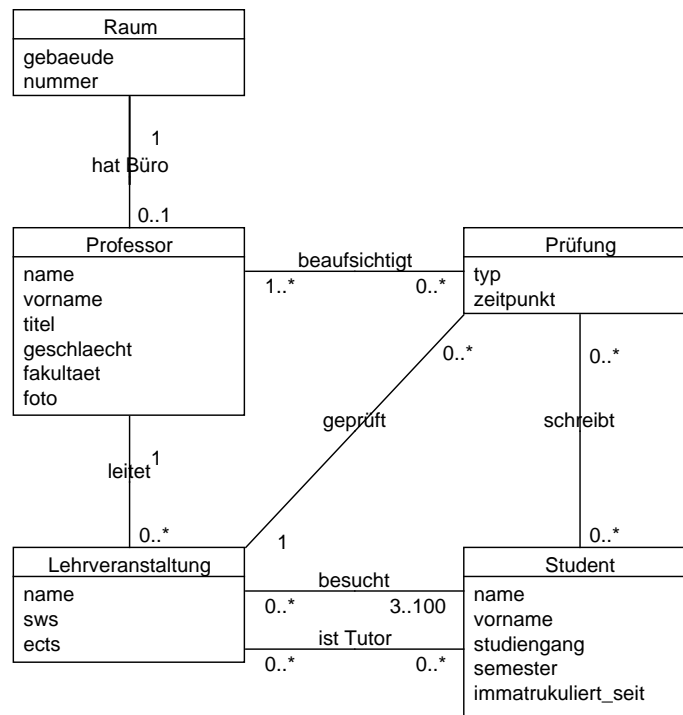


Abbildung 3.1: ER-Diagramm des fortlaufenden Beispiels

Das entsprechende relationale Datenbank-Diagramm wird in Abbildung 3.2 dargestellt.



Abbildung 3.2: Relationales Datenbank-Diagramm des fortlaufenden Beispiels

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

3.4.3 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)?

To do (6)

3.5 Modellierungsvarianten der Testdaten für DbUnit

In *DbUnit* werden die Datenbankzustände durch *DataSets* repräsentiert. Für einen Test werden gewöhnlich zwei *DataSets* benötigt: das erste für den Anfangszustand, das zweite für den erwarteten Zustand. Allerdings bieten *DbUnit*-*DataSets* nur begrenzte Möglichkeiten, das *DataSet* mit dem erwarteten Zustand aus dem *DataSet* mit dem Anfangszustand zu erzeugen.

Im Folgenden werden verschiedene Modellierungsarten für *DbUnit*-*DataSets* diskutiert. Die Ergebnisse stellen die Grundlage für die konkretere Zielsetzung dar.

3.5.1 XML-DataSet

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

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

```

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

```

Listing 3.1: XML-DataSet

Die positiven Eigenschaften bei der Modellierung mit XML sind unter anderem, dass für XML ein breites Angebot an Werkzeugen zur Verfügung steht. Diese können über den Dokumententyp prüfen, ob die Datei den Regeln entspricht.

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

Beziehungen zwischen Datensätzen werden über numerische Konstanten realisiert. Die referenzierten Schlüssel müssen in der entsprechenden Fremdschlüssel-Spalte verwendet werden. Die manuelle Pflege der Schlüssel kann unübersichtlich und damit fehleranfällig 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 bei der Nutzung von `XmlDataSet` ist, dass der erwartete Datenbankzustand selbst wieder den kompletten Datenbankbestand umfassen muss. `DbUnit` erlaubt zwar mehrere DataSets zu einem zusammenzufassen, das Entfernen von Datensätzen ist darüber aber nicht möglich. Mehrere XML-Dateien mit ähnlichen, überwiegend sogar gleichen Daten, sorgen für ein hohes Maß an Redundanz. Darüber hinaus sieht `DbUnit` keinen Mechanismus für die Komposition von XML-DataSets auf Modellierungsebene vor, d.h. es geht aus einer solchen XML-Datei nicht hervor, dass sie auf anderen DataSets aufbaut und diese erweitert.

`DbUnit`-konforme XML-Dateien 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).

Das `FlatXmlDataSet` stellt die zweite Variante dar. Hierbei gibt es keine von `DbUnit` vorgegebene DTD, da die Tags den Tabellen-Namen entsprechen¹. Eine solche XML-Datei kommt ohne explizite Meta-Informationen zu den Tabellen aus. Stattdessen stellen sie eine Art Sprachelement dar und werden für die Zuweisung der Werte verwendet. In Bezug auf die Meta-Informationen ist das `FlatXmlDataSet` übersichtlicher als das `XmlDataSet` (siehe Listing 3.2).

```

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

```

¹Es ist möglich, eine eigene DTD zu definieren.

3.5. MODELLIERUNGSVARIANTEN DER TESTDATEN FÜR DBUNIT

```
20     name="Design_Patterns"
21     sws="4"
22     ects="3" />
23     ...
24 </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.

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

3.5.2 Default-DataSet

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

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

```
1  DefaultTable professor = new DefaultTable(
2      "professor",
3      new Column[] {
4          new Column("id", DataType.INTEGER),
5          new Column("name", DataType.VARCHAR),
6          new Column("vorname", DataType.VARCHAR),
7          new Column("titel", DataType.VARCHAR),
8          new Column("fakultaet", DataType.VARCHAR),
9      }
10 );
11 professor.addRow(new Object[] {
12     Parameters.Professor.WAESCH_ID,
13     "Wäsch",
14     "Jürgen",
15     "Prof._Dr.-Ing.",
16     "Informatik",
17 });
18 professor.addRow(new Object[] {
19     Parameters.Professor.HAASE_ID,
20     "Haase",
21     "Oliver",
22     "Prof._Dr.",
23     "Informatik",
24 });
25 dataSet.addTable(professor);
26
27 DefaultTable lehrveranstaltung = new DefaultTable(
28     "lehrveranstaltung",
29     new Column[] {
30         new Column("id", DataType.INTEGER),
31         new Column("professor_id", DataType.INTEGER),
32         new Column("name", DataType.VARCHAR),
33         new Column("sws", DataType.INTEGER),
34         new Column("ects", DataType.INTEGER),
35     }
36 );
```

```

37 | lehrveranstaltung.addRow(new Object[] {
38 |     Parameters.Lehrveranstaltung.VSYSTEME_ID,
39 |     Parameters.Professor.HAASE_ID,
40 |     "Verteilte_Systeme",
41 |     4,
42 |     5,
43 | });
44 | lehrveranstaltung.addRow(new Object[] {
45 |     Parameters.Lehrveranstaltung.DESIGN_PATTERNS_ID,
46 |     Parameters.Professor.HAASE_ID,
47 |     "Design_Patterns",
48 |     4,
49 |     3,
50 | });
51 | dataSet.addTable(lehrveranstaltung);

```

Listing 3.3: Default-DataSet

Diese Variante löst allerdings nicht alle Probleme: So müssen immer noch Meta-Informationen zu den Tabellen modelliert werden (Zeilen 3-9 und 29-36). Obwohl diese sogar Typinformationen beinhalten, werden Typ-Fehler erst zur Laufzeit beim Einspielen in die Datenbank 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 Formattierung viele Zeilen notwendig und umfangreiche Datensets werden daher unübersichtlich. Insgesamt bietet die Nutzung dieser Java-Datasets wenig Vorteile gegenüber den XML-Datasets.

3.5.3 STU-DataSet

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

STU führt eine eigene DataSet-Klasse ein, über die die Daten modelliert werden. Diese DataSet-Klasse kann bei Bedarf von den aktuellen Daten ein DbUnit-DataSet erzeugen. Auf diese Weise können DataSets aus *STU* einfacher und umfangreicher als DbUnit-Datasets modifiziert werden, wie z.B. das Löschen von Zeilen.

Auf diese Weise können mit *STU* verhältnismäßig einfach Varianten eines DbUnit-Datasets erzeugt werden, z.B. ein DataSet mit dem Ausgangszustand und ein DataSet mit dem erwarteten Zustand am Ende des Tests.

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

```

1 | table_Professor
2 |   .insertRow()
3 |     .setId(Parameters.Professor.HAASE_ID)
4 |     .setName("Haase")
5 |     .setVorname("Oliver")
6 |     .setTitel("Prof._Dr.")
7 |     .setFakultaet("Informatik")
8 |   .insertRow()
9 |     .setId(Parameters.Professor.WAESCH_ID)
10 |     .setName("Wäsch")
11 |     .setVorname("Jürgen")
12 |     .setTitel("Prof._Dr.-Ing.")

```

3.5. MODELLIERUNGSVARIANTEN DER TESTDATEN FÜR DBUNIT

```
13     .setFakultaet("Informatik");
14
15 table_Lehrveranstaltung
16     .insertRow()
17     .setId(Parameters.Lehrveranstaltung.VSYSTEME_ID)
18     .setProfessorId(Parameters.Professor.HAASE_ID)
19     .setName("Verteilte_Systeme")
20     .setSws(4)
21     .setEcts(5)
22     .insertRow()
23     .setId(Parameters.Lehrveranstaltung.DESIGN_PATTERNS_ID)
24     .setProfessorId(Parameters.Professor.HAASE_ID)
25     .setName("Design_Patterns")
26     .setSws(4)
27     .setEcts(3);
```

Listing 3.4: STU DataSet (1)

Die Modellierung von Beziehungen stellt sich als ähnlich problematisch wie bei den bisherigen Java-Datasets dar (siehe Abschnitt 3.5.2). Nach wie vor wächst das DataSet vertikal in der Datei.

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

```
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: STU DataSet (2)

Kapitel 4

Entwurf einer Modellierungssprache für Test-Daten

Ein Ziel dieser Arbeit ist es, die Modellierung von Testdaten zu vereinfachen. DbUnit ist eine bewährte Bibliothek zur Unterstützung von Unit-Tests datenbankbasierter Anwendungen. Welche Schwächen DbUnit-Datasets in Bezug auf die Modellierung haben, wurde bereits in Abschnitt 3.5 thematisiert.

Ein üblicher Weg, bestehende Schnittstellen zu vereinfachen, stellen Fassaden dar. Eine solche Fassade kann auf unterschiedliche Arten realisiert werden.

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

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

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

4.1 Entwurf der DSL

Als Grundlage für die Definition der Modellierungssprache sollen verschiedene Beispiel-Entwürfe dienen. Dabei wird das fortlaufende Beispiel aus Kapitel 3 mit unterschiedlichen Entwürfen modelliert und die Vor- und Nachteile analysiert.

Mit Hilfe dieser Analyse wird die finale Modellierungssprache definiert.

4.1.1 Beispiel-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)

Die in Listing 4.1 gezeigte 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 Beispiel-Entwurf 2

Ein leicht abgewandelter Entwurf (siehe Listing 4.2) 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 Beispiel-Entwurf 3

Listing 4.3 zeigt den dritten Entwurf einer DSL. Es wird versucht die Daten durch eine tabellarische Struktur übersichtlich zu gestalten. Die Syntax der Sprache ist sehr schlicht und ausdrucksstark. Ein Label vor einer Tabelle drückt aus, welche Daten folgen (Zeilen 1 und 6). Die Tabelle selbst beginnt mit einer Kopfzeile, die die Spaltenreihenfolge beschreibt (Zeilen 2 und 7). Einzelne Spalten werden vom Oder-Operator (|) getrennt. Die erste Spalte nimmt Zeilen-Identifikatoren auf und ist von den Daten mit Hilfe des Double-Pipe-Operators (||) abgegrenzt.

```

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.

Bei umfangreichen Tabellen, die möglicherweise nicht mehr auf eine Bildschirmseite passen, kann es sinnvoll sein, den Tabellenkopf zu wiederholen. Dies sollte von der Implementierung genauso unterstützt werden wie die Definition neuer Tabellenköpfe mit unterschiedlichen Spalten.

4.1.4 Finaler DSL-Entwurf

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

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

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

4.2 Wahl der Technologie

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

Groovy ist eine dynamisch typisierte Sprache¹, die direkt in Java-Bytecode übersetzt wird und damit auch in einer Java Virtual Machine (JVM) ausgeführt wird. Dies ermöglicht die Nutzung von Groovy-Klassen und Groovy-Objekten in Java und umgekehrt.

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

Listing 4.4 zeigt die selben Anweisungen einmal in typischer Java-Syntax (Zeile 1) und einmal mit den Syntax-Vereinfachungen von Groovy (Zeile 2):

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

Listing 4.4: Vereinfachung von Ausdrücken in Groovy

Groovy hebt sich ferner durch die Möglichkeit Operatoren zu überladen und durch Closures von Java ab. Ein Closure (Funktionsabschluss) ist ein Codeblock, der wie eine Funktion aufgerufen und genutzt werden kann. In Java lassen sich Closures mit syntaktisch umfangreicheren Methoden-Objekten nachbilden. Ein Methoden-Objekt stellt eine Instanz einer (möglicherweise anonymen) Klasse dar, die nur eine Methode implementiert. [Kön07, S. 40] ^{To do (7)}

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.

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

4.2.1 Implementierungsvarianten mit Groovy

Eine DSL kann auf unterschiedliche Arten implementiert werden. Groovy bietet dafür zwei Möglichkeiten der Meta-Programmierung an: Laufzeit-Meta-Programmierung und Compiler-Zeit-Meta-Programmierung, 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 [Kön07, S. 58]).

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

Groovy bietet 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. [Kön07, 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

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

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 allgemeinen Methode bevorzugt, z.B. bei zwei `Integer`-Werten. Doch auch solche Operationen lassen sich überschreiben, wenn für die Datentypen passende Kategoriemethoden definiert werden. Diese und weitere Anpassungen sind in Listing 4.6 dargestellt.

Im zu parsenden Closure sollen Variablen genutzt werden, die nicht im Closure selbst sondern in einer anderen Klasse definiert werden. Zu diesen Variablen gehören die Bezeichner der Spalten und die der Zeilen. Groovy erlaubt es Closures im Kontext eines Delegates auszuführen. In diesem Beispiel wird die aktuelle Instanz der Klasse `TableParser` verwendet (Zeile 22). Dieser Delegat wird bei der Auflösung von Variablen- und Methoden-Bezeichnern verwendet. In Zeile 23 wird festgelegt, dass bei der Auflösung zuerst im Delegaten gesucht wird.

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

```

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

```

Listing 4.6: Tabellen-Parser Grundgerüst mit Operator-Überladen

Die statischen 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 Erlauben von gleichzeitigem Parsen von Tabellen aus verschiedenen Threads heraus, wird für die Ergebnisse eine threadlokale Liste verwendet. [Goe09, S. 45]

Die Laufzeit-Meta-Programmierung kann die Syntax der Sprache nicht beliebig erweitern. Groovy kennt keinen Double-Pipe-Operator. Deshalb kann dieser weder überladen noch über Laufzeit-Meta-Programmierung eingeführt werden. Folglich ist es nicht möglich, den

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

```

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

```

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

AST-Transformation

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

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

Allerdings muss zum Auswerten einer Tabelle bei AST-Transformationen ein relativ großer Aufwand betrieben werden. Ein AST-Transformationsklasse, erhält über das Visitor-Pattern Zugriff auf die abstrakten Syntaxbäume einzelner Module ([Gam+95, 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. Die folgenden drei Arten von Ausdrücken sind relevant für das Parsen:

- **BinaryExpression:** Ein binärer Ausdruck besteht aus zwei Operanden und einem Operator. Wenn es sich beim Operator um einen Pipe oder Double-Pipe-Operator handelt, werden die 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. Möglicherweise kann durch die Nutzung von sogenannten DSL Descriptoren für die Groovy-Plugins gängiger IDEs auch eine IDE-Unterstützung für Labels und Spalten erreicht werden. Dieser Frage wird allerdings im weiteren Verlauf nicht nachgegangen.

4.2.2 Implementierungsentscheidung

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

Die Entscheidung fällt auf die Laufzeit-Meta-Programmierung. Der Grund dafür ist, dass sie einfacher zu verwenden ist. AST-Transformationen würden bezogen auf die Anforderungen keinen Mehrwert bieten. Darüber hinaus dürfte der Code zur AST-Transformation komplexer und wartungsunfreundlicher ausfallen.

4.3 Definition der Sprache

Die Entscheidung zugunsten der Laufzeit-Meta-Programmierung führt zu einigen Änderungen an der Sprache aus dem dritten Entwurf (siehe Abschnitt 4.1.3). Wie beschrieben, wird auf den Double-Pipe-Operator verzichtet. Anstelle der Labels treten vordefinierte Variablen, deren Namen sich nach jeweiligen Tabellenbezeichnungen richten. Auf diesen Variablen kann zum Definieren von Tabellendaten die Methode `rows` aufgerufen werden. Die Daten werden in Form eines Closures übergeben.

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

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

```

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

```

Listing 4.8: EBNF der Tabellen

```

1 Relations  = { Relation, NewLine };
2 Relation   = Ref, ".", RelationName, "(", RefList, ")", [ Attributes ];
3 RefList    = Ref, [ { ",", Ref } ];
4 Ref        = ? Bezeichner einer Ref-Variable ?;
5 Attributes = { ".", Attribute, "(", Value, ")" };
6 Attribute  = ? Von Beziehung abhängiger Attribut-Bezeichner ?;
7 Value      = ? numerische Literale, symbolische Konstanten,
8             Methodenaufrufe ?;
9 NewLine    = "\n";

```

Listing 4.9: EBNF der Relationen

4.4 Beispiel-DataSet in Groovy

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

```

1  class HochschuleDataSet extends HochschuleBuilder
2  {
3
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 | name | sws | ects
14             VSYS | "Verteilte_Systeme" | 4 | 5
15             DPATTERNS | "Design_Patterns" | 4 | 3
16         }
17
18         pruefungTable.rows {
19             REF | typ | zeitpunkt
20             P_VSYS | "K90" | DateUtil.getDate(2013, 4, 1, 14, 0, 0)
21             P_DPATTERNS | "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 4.10: DataSet modelliert mit Table Builder API

Insgesamt ist die Darstellung sehr übersichtlich. 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.

Innerhalb der Methode `tables` werden die Daten der einzelnen Tabellen auf Datenbank-Ebene modelliert. Die Methode `relations` erlaubt die Modellierung von Beziehungen auf der höheren ER-Ebene. Sogar Beziehungen, die über assoziative Tabellen realisiert werden, können so modelliert werden.

Alternativ könnten Beziehungen auch auf der Datenbank-Ebene modelliert werden. Listing 4.11 veranschaulicht diese Variante.

```
1  class HochschuleDataSet extends HochschuleBuilder
2  {
3
4      def tables() {
5
6          ...
7
8          lehrveranstaltungTable.rows {
9              REF      | professor
10             VSYS     | HAASE
11             DPATTERNS | HAASE
12         }
13
14         beaufsichtigtTable.rows {
15             professor | pruefung
16             WAESCH    | P_VSYS
17             HAASE     | DPATTERNS
18         }
19     }
20 }
21
22 ...
23
24 }
```

Listing 4.11: Beziehungen innerhalb von Tabellen

Kapitel 5

Realisierung der Sprache

Im Folgenden wird 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.

SB Testing DB stellt eine sinnvolle Grundlage für Erweiterungen und Verbesserung dar. Es verfolgt ein ähnliches Ziel und versucht ebenfalls, die Modellierung von Test-Daten zu vereinfachen und bietet bereits einige Verbesserungen gegenüber DbUnit. Außerdem stellt dies eine gewisse Kompatibilität sicher, so dass bestehende Tests zumindest nach geringfügigen Anpassungen mit den Erweiterungen funktionieren sollten. Aus einem domänenspezifischen Datenbank-Modell erzeugt *SB Testing DB* ein individuelles API zur Modellierung von DataSets. Um die Modellierung zu vereinfachen, vor allem in Bezug auf die Beziehungen, sollen die generierten Klassen um eine Fassade ergänzt werden. Eine Fassade stellt eine Schnittstelle auf höherer Abstraktionsebene dar, um das System einfach zu verwenden [Gam+95, S. 185].

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

Um Probleme bezüglich Abwärtskompatibilität zu vermeiden, fließen die Anpassungen nicht in *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.

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

- Das **Fluent Builder API** ist ein **Java**-basiertes API. Der Name spiegelt wieder, dass es ein Java Fluent API bereit stellt (siehe auch Abschnitt 2.4). Ein solches API wird auch als interne DSL bezeichnet.

- Das **Table Builder API** ist das **Groovy**-basierte API bzw. die neue DSL. Über diese DSL können die Testdaten tabellarisch modelliert werden.

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

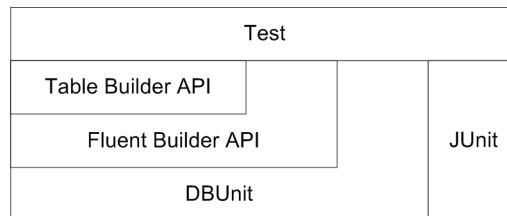


Abbildung 5.1: Architektur

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

- **Code-Qualität:** Die neuen Funktionen können direkt in bestehende Klassen integriert werden, anstatt neue Typen einzuführen. Auf Adapterklassen und Delegation kann auf diese Weise verzichtet.
- **Mehrwert gegenüber *SB Testing DB*:** Auch wenn auf das neue Table Builder API verzichtet wird, kann das bietet das Builder API so einen Mehrwert gegenüber der bisherigen *SB-Testing-DB*-Implementierung bieten. Z.B. können verbesserte Möglichkeiten zur Modellierung von Beziehungen im Fluent Builder API integriert werden und diese Funktionen auch in reinen Java-basierten Tests nutzbar machen.
- **Einheitlicher Funktionsumfang:** Funktionen, die in das Fluent Builder API integriert werden, können in beiden APIs genutzt werden. Die Folge ist, dass der Funktionsumfang nicht bzw. weniger stark von der genutzten API abhängt.

5.1 Änderungen am Generator-Modell

Die hinzugekommenen Funktionen erfordern Erweiterungen in den Klassen zur Modellierung der zu Grunde liegenden Datenbank. Da das API in *SB Testing DB* überladene Methoden mit vielen Parametern zur Beschreibung von Spalten nutzt (es gibt dafür zwölf reguläre und eine als *deprecated* eingestufte Methoden), soll in *STU* das anwenderfreundlichere Builder-Pattern verwendet werden.

Durch den Einsatz dieses Patterns sollen die Klassen wartbarer und erweiterbarer bleiben. Beim Überladen von Methoden würde sich bei jedem weiteren optionalen Parameter die Anzahl an Methoden unter Umständen verdoppeln. Außerdem sind lange Parameterlisten für Programmierer nicht immer intuitiv: Die Reihenfolge lässt sich oft nur schwer merken. 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 Eigenschaften für Spalten nicht mehr über ein `EnumSet` festgelegt, sondern über Methoden für die vordefinierten Flags. In Abschnitt 5.1.1 wird weiter auf das Thema Flags eingegangen. Darüber hinaus bieten die neuen Klassen die Möglichkeit, Beschreibungen zu Tabellen und Spalten hinzu zu fügen. Diese werden bei der Code-Generierung für die Erstellung von JavaDoc-Kommentaren verwendet (siehe Abschnitt 5.8).

To do (8)

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

5.1.1 Spalten-Eigenschaften

SB Testing DB sieht verschiedene Eigenschaften, sogenannte 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:

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

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

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

Listing 5.1: Beispiel für unveränderliche Identifikatoren

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

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

- **Add Next Method:** *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.
- **Auto Invoke Next:** Ist dieses Flag aktiviert, wird die Next-Value-Methode beim Anlegen einer neuen Tabellenzeile automatisch aufgerufen. Beim Setzen des Flags über die Builder-Methode `autoInvokeNext()` wird automatisch auch das Flag zum Generieren der Next-Value-Methode gesetzt.
- **Immutable:** Ist dieses Flag gesetzt, kann ein Wert in einer Spalte nur ein Mal gesetzt werden, und danach nicht mehr verändert werden. Wenn das Flag zum automatischen Aufruf der Next-Value-Methode aktiviert ist, kann der automatisch erzeugte Wert allerdings überschrieben werden. Die Methode zum Aktivieren des Flags heißt `immutable()`.

5.1.2 Modellierung von Relationen über Builder-Klassen

In Bezug auf die Modellierung des Datenbank-Modells für den Generator stellt das neue API zur Modellierung der Relationen die größte Veränderung in *STU* gegenüber *SB Testing DB* dar. Über `reference` kann der Builder zur Beschreibung der Relation aufgerufen werden. Die Beschreibung findet auf zwei Ebenen statt:

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

Beide Ebenen erlauben die Angabe eines Bezeichners, der die Beziehung in die jeweilige Richtung beschreibt, der Ausgangspunkt wird durch die Ebene bestimmt. Dieser Bezeichner werden für die Methoden zur Modellierung der Beziehungen verwendet. Daneben können auch noch Beschreibungstexte angegeben werden, die für die JavaDoc genutzt werden.

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

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

5.1.3 Alte und neue Builder-Klassen im Vergleich

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

5.1. ÄNDERUNGEN AM GENERATOR-MODELL

```
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 5.2: Beispiel SB-Testing-DB-Builder

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

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

```
1 database("Hochschule");
2 packageName("com.seitenbau.stu.dbunit.hochschule");
3
4 Table professoren = table("professor")
5     .description("Die_Tabelle_mit_den_Professoren_der_Hochschule")
6     .column("id", DataType.BIGINT)
7     .identifierColumn()
8     .autoInvokeNext()
9     .column("name", DataType.VARCHAR)
10    .column("vorname", DataType.VARCHAR)
11    .column("titel", DataType.VARCHAR)
12    .column("fakultaet", DataType.VARCHAR)
13    .build();
14
15 Table lehrveranstaltungen = table("lehrveranstaltung")
16     .description("Die_Tabelle_mit_den_Lehrveranstaltungen_der_Hochschule")
17     .column("id", DataType.BIGINT)
18     .identifierColumn()
19     .autoInvokeNext()
20     .column("professor_id", DataType.BIGINT)
21     .reference
22         .local
23             .name("geleitetVon")
24             .description("Gibt_an,_von_welchem_Professor_eine_Lehrveranstaltung_geleitet_wird.")
25         .foreign(professoren)
26             .name("leitet")
27             .description("Gibt_an,_welche_Lehrveranstaltungen_ein_Professor_leitet.")
28     .column("name", DataType.VARCHAR)
29     .column("sws", DataType.INTEGER)
30     .column("ects", DataType.DOUBLE)
31     .build();
```

Listing 5.3: Beispiel STU-Builder

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

```

1 associativeTable("besucht")
2   .column("student_id", DataType.BIGINT)
3   .reference
4     .foreign(studenten)
5     .name("besucht")
6     .description("Die_Lehrveranstaltungen,_die_ein_Student_besucht.")
7   .column("Lehrveranstaltung_id", DataType.BIGINT)
8   .reference
9     .foreign(lehrveranstaltungen)
10    .name("besuchtVon")
11    .description("Die_Studenten,_die_eine_Lehrveranstaltung_besuchen.")
12 .build();

```

Listing 5.4: Beispiel für assoziative Tabelle

5.2 Neue DataSet-Builder-Klassen

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

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

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

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

5.3 Tabellenparser

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

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

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

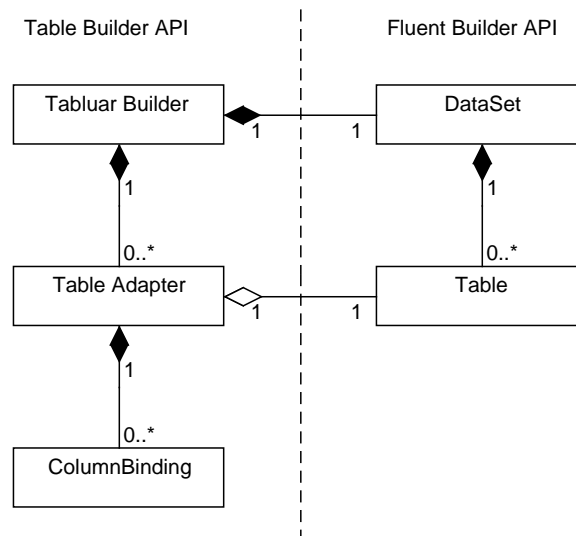


Abbildung 5.2: Klassendiagramm der DataSet-Builder

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

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

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

Mit automatischen Typumwandlungen bietet `STU` eine Komfortfunktion, die die Lesbarkeit weiter verbessert. So versucht `STU`, Werte dynamisch beim Parsen in den vom Modell erwarteten Datentyp zu bringen. Im Beispiel ist der Spalte `ects` in der Lehrveranstaltungstabelle der Datentyp `Double` zugeordnet. Die in der Spalte auftauchenden `Integer`-Werte werden automatisch in `Double`-Werte umgewandelt. Da der Parser nicht mit primitiven

Datentypen sondern nur auf Objekten arbeitet, geht die Umwandlung über einen einfachen Type-Cast hinaus.

5.4 Referenzen und Scopes

Neben der Möglichkeit, Daten tabellarisch zu modellieren, gehören die neuen Referenz-Datentypen zu der wichtigsten Erweiterung. In *STU* ist eine Referenz eine Art Stellvertreter für eine Entität (Tabellenzeile). Die Referenz kann bei der Modellierung oder auch bei Such-Anfragen anstelle konkreter Werte (wie Primärschlüssel) verwendet werden. Die Such-Anfrage-Möglichkeiten werden in Abschnitt 5.7 erläutert.

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

```

1 professorTable.rows {
2   REF      | name      | vorname   | titel      | fakultaet
3   WAESCH   | "Wäsch"   | "Jürgen"  | "Prof. Dr.-Ing." | "Informatik"
4   ...
5 }

```

Listing 5.5: Binden von Referenzen (Table Builder API)

```

1 table_Professor.insertRow()
2   .bind(WAESCH)
3   .setName("Wäsch")
4   .setVorname("Jürgen")
5   .setTitle("Prof. Dr.-Ing.")
6   .setFakultaet("Informatik")
7   ...

```

Listing 5.6: Binden von Referenzen (Fluent Builder API)

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

```

1 WAESCH.getName() // Java style
2 WAESCH.name      // Groovy style

```

Listing 5.7: Zugriff auf Werte über Referenzen

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

```

1 WAESCH.beaufsichtigt(P_VSYS)

```

Listing 5.8: Definition von Beziehungen über Referenzen

Die Referenzen müssen vor ihrer Nutzung definiert (also deklariert und instantiiert) werden. Zwar könnten in Groovy auch nicht explizit definierte Referenzen verwendet werden, allerdings würde Tool-Unterstützung verloren gehen (z.B. beim Umbenennen von Referenzen, Erkennen von Tippfehlern bei Bezeichnen). Außerdem könnten sie auch nicht im normalen Java-Code verwendet werden. Es bietet sich an, sie als globale Variablen zu definieren.

Verschiedene DataSets (mit dem selben Datenbank-Modell) können die selben Referenzen nutzen, auch wenn sie unterschiedliche Werte repräsentieren.

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

5.5 Nutzung des DataSets in Unit-Tests

Wie das Beispiel-DataSet aus Listing 4.10 in einem JUnit-Test verwendet werden kann, zeigt Listing 5.9. Das System Under Test (siehe Abschnitt 2.2) ist ein Spring-Service, der von der Variable `sut` (Zeile 20) repräsentiert wird. **To do** (9)

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 wurde (Zeilen 22 und 23). Außerdem wird dieses DataSet auch bei der `DataSetRegistry` als aktives DataSet registriert und die Daten in die Datenbank eingespielt. Dadurch kommen die Test-Methoden ohne Verwaltungsaufgaben aus. Der Test `removeStudent` testet, ob das System die richtigen Änderungen in der Datenbank vornimmt, wenn der Student MUSTERMANN entfernt wird. Da dem Service die zu löschende Entität übergeben werden muss (Zeile 38), wird in den Zeilen 29 bis 35 eine entsprechende Instanz erstellt und konfiguriert.

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

```

1  @RunWith(SpringJUnit4ClassRunner.class)
2  @ContextConfiguration(classes=HochschuleContext.class)
3  public class HochschuleDataSetDatabaseTest {
4
5      @Autowired
6      DataSource dataSource;
7
8      @Rule
9      public DatabaseTesterRule dbTester =
10         new DatabaseTesterRule(new Future<DataSource>() {
11             @Override
12             public DataSource getFuture()
13             {
14                 return dataSource;
15             }
16         }).addCleanAction(new ApacheDerbySequenceReset()
17             .autoDerivateFromTablename("_SEQ"));
18
19     @Autowired
20     HochschuleService sut;
21
22     @InjectDataSet
23     HochschuleBuilder dataSet;
24
25     @Test
26     @DatabaseSetup(prepare = HochschuleDataSet.class)

```

```

27 public void removeStudent() throws Exception {
28     // prepare
29     Student student = new Student();
30     student.setMatrikelnummer(MUSTERMANN.getMatrikelnummer());
31     student.setVorname(MUSTERMANN.getVorname());
32     student.setName(MUSTERMANN.getName());
33     student.setStudiengang(MUSTERMANN.getStudiengang());
34     student.setSemester(MUSTERMANN.getSemester());
35     student.setImmatriculiertSeit(MUSTERMANN.getImmatriculiertSeit());
36
37     // execute
38     sut.removeStudent(student);
39
40     // verify
41     dataSet.studentTable.deleteRow(MUSTERMANN);
42     dataSet.besuchtTable.deleteAllAssociations(MUSTERMANN);
43
44     dbTester.assertDataBase(dataSet);
45 }
46
47 ...
48
49 }

```

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

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

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

```

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

```

Listing 5.10: Test-Methode in Groovy

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

5.6 Komposition von DataSets

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

5.7. ERWEITERUNGEN IN GENERIERTER API

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

```
1 class ExtendedHochschuleDataSet extends HochschuleBuilder {
2
3     def extendsDataSet() { HochschuleDataSet }
4
5     def tables() {
6
7         lehrveranstaltungTable.rows {
8             REF | id | name | sws | ects
9             PROGR | 3 | "Programmieren" | 4 | 6.0
10        }
11    }
12
13    def relations() {
14        HAASE.leitet(PROGR)
15    }
16
17 }
18 }
```

Listing 5.11: Erweitertes DataSet

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

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

Listing 5.12: Erweitertes DataSet

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

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

Listing 5.13: Test auf erweiterem DataSet

5.7 Erweiterungen in generierter API

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

- **RowBuilder:** Die Erweiterungen der `RowBuilder` betreffen vor allem die verbesserten Möglichkeiten Relationen auszudrücken. So gibt es für Spalten, die eine Relation

zu einer anderen Spalte enthalten, nun neben einem Setter für den konkreten Wert (z.B. des Fremdschlüssels) einen Setter zum Setzen des entsprechenden Ref-Typs.

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

1. **Reihenfolge:** Die Modellierung der Daten ist in diesem Fall keiner strengen Reihenfolge unterworfen. Es ist egal, ob die Zeile, auf die Bezug genommen wird, überhaupt schon initialisiert wurde.
 2. **Konsistenz:** Die Werte werden nicht redundant gespeichert. Wird der Wert an einer Stelle geändert, ist dieser Wert unmittelbar im gesamten DataSet so sichtbar.
- **Future Values:** Eine der wenigen Erweiterungen, die nicht auf die Einführung der Ref-Typen zurückzuführen sind, sind Future Values. Dabei handelt es sich um Werte, die erst beim Abfragen ausgewertet werden. Dies kann nützlich sein, wenn sich Werte abhängig von anderen Daten ändern. Listing 5.14 zeigt ein Beispiel, in der die Lehrveranstaltungstabelle um eine Spalte erweitert wurde. Diese Spalte soll die Anzahl der Tutoren aufnehmen, die die Lehrveranstaltung betreuen.

```

1 class HochschuleDataSet extends HochschuleBuilder
2 {
3
4     def tables() {
5
6         lehrveranstaltungTable.rows {
7             REF      | name | sws | ects | tutoren
8             VSYS      | "Verteilte_Systeme" | 4 | 5 | tutors(VSYS)
9             DPATTERNS | "Design_Patterns" | 4 | 3 | tutors(DPATTERNS)
10        }
11
12        ...
13    }
14
15    ...
16
17    // returns a Closure which is treated as future value
18    def tutors(LehrveranstaltungRef ref) {
19        return {
20            def rows = istsutorTable.quietFindWhere.lehrveranstaltungId(ref)
21            return rows.rowCount
22        }
23    }
24 }

```

Listing 5.14: Beispiel Lazy Valunes

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

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

```

1 dataSet.table_Professor.findWhere.vorname("Oliver");
2 dataSet.table_Professor.findWhere.vorname(HAASE);

```

Listing 5.15: Such-Beispiele

- **quietFindWhere:** In manchen Fällen kann es sinnvoll sein, bei einer Suche ohne Treffer keine Ausnahme auszulösen. Ein Beispiel dafür ist das Closure in Listing 5.14. Eine Lehrveranstaltung ohne Tutoren kann in diesem Beispiel normal sein.
- **getWhere:** Wenn davon auszugehen ist, dass eine Such-Anfrage genau eine Zeile als Ergebnis liefert, kann `getWhere` verwendet werden. Im Gegensatz zu `findWhere` liefert es das Ergebnis nicht in Form einer Liste, sondern als `Optional`-Wert zurück [Com13]. Gibt es auf eine Suchanfrage mehr als einen Treffer, wird eine `Exception` ausgelöst.
- **find:** Sind die einfachen Such-Anfragen über `findWhere` bzw. `getWhere` nicht mächtig genug, können mit Hilfe von `find` Filter-basierte Suchen durchgeführt werden. In Listing 5.16 wird ein Filter gezeigt, der alle Professoren findet, deren Vorname die Länge sechs hat.

```

1 Filter<RowBuilder_Professor> FILTER =
2     new Filter<RowBuilder_Professor>()
3     {
4         @Override
5         public boolean accept(RowBuilder_Professor value)
6         {
7             return value.getVorname().length() == 6;
8         }
9     };
10
11 RowCollection_Professor profs = dataSet.professorTable.find(FILTER);

```

Listing 5.16: Beispiel für find

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

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

```

1 Action<RowBuilder_Professor> ACTION =
2     new Action<RowBuilder_Professor>()
3     {
4         @Override
5         public void call(RowBuilder_Professor value)
6         {
7             System.out.println("Professor:_" + value.getName());
8         }
9     };
10
11 dataSet.professorTable.foreach(ACTION);

```

Listing 5.17: Beispiel für foreach

5.8 JavaDoc

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

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

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

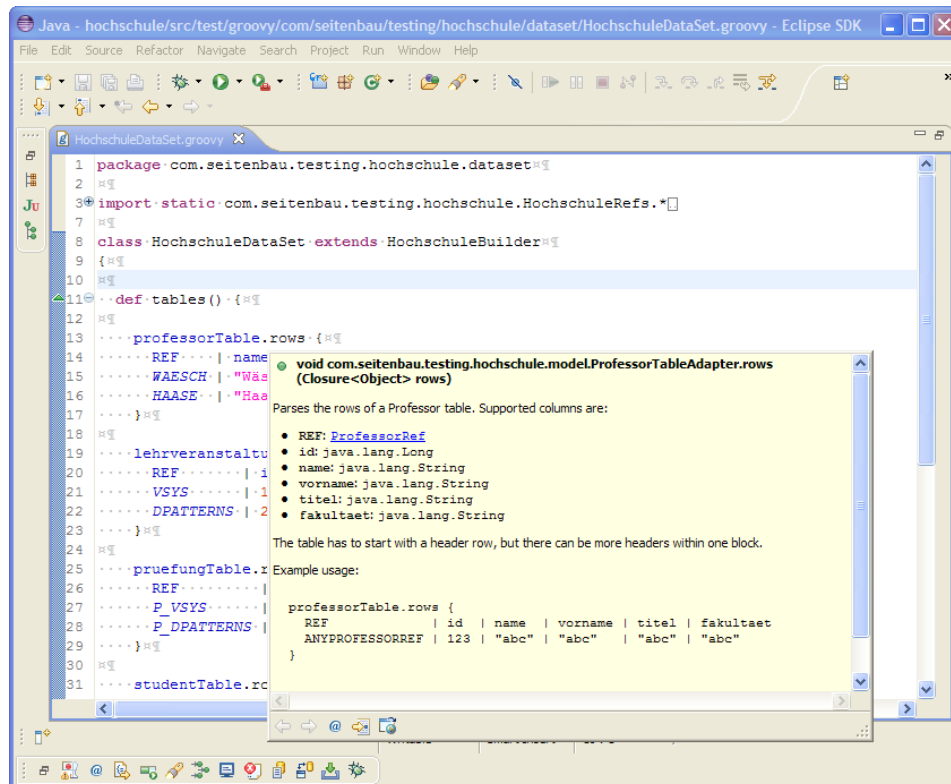


Abbildung 5.3: Beispiel JavaDoc-Tooltip

5.9 Verhalten bei Fehlern in den Tabellendefinitionen

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

Fehler in der eigentlichen Tabellenstruktur, z.B. eine abweichende Anzahl von Spalten, kann der Standard-Compiler nicht erkennen, genauso wie ungültige Werte bzw. ungültige Typen. Solche Fehler werden in der gegenwärtigen Implementierung zur Laufzeit erkannt und führen zum Scheitern der Tests. Dazu wirft der Tabellen-Parser eine Exception

der Klasse `TableParserException`. Wenn ein falscher Typ verwendet wird, könnte die Meldung der Exception so aussehen: *Cannot set value <5> of type java.lang.Integer; expected class java.lang.String in [TableRowModel: <JobsRef> | 5 | "Creating software"]*

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

5.10 Nicht umgesetzt

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

5.10.1 Zusammengesetzte Schlüssel

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

5.10.2 Unterstützung für weitere Beziehungstypen

Folgende Beziehungstypen müssen manuell umgesetzt werden.

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

```

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

```

Listing 5.18: Reflexive Beziehungen manuell

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

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

```

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

```

Listing 5.19: Reflexive Beziehungen mit Hilfe assoziativer Tabelle

- **Zirkuläre Beziehungen** Reflexive Beziehungen stellen eine besondere Form von zirkulären Beziehungen dar. Die Probleme sind relativ ähnlich.

```

1 Table event = table("event")
2   .column("id", DataType.BIGINT)
3   .defaultIdentifier()
4   .column("name", DataType.VARCHAR)
5   .column("organizer", DataType.BIGINT)
6   .build();
7
8 Table person = table("person")
9   .column("id", DataType.BIGINT)
10  .defaultIdentifier()
11  .column("name", DataType.VARCHAR)
12  .column("participates", DataType.BIGINT)
13  .reference
14  .foreign(event)
15  .build();

```

Listing 5.20: Zirkuläre Beziehungen manuell

```

1 Table person = table("person")
2   .column("id", DataType.BIGINT)
3   .defaultIdentifier()
4   .column("name", DataType.VARCHAR)
5   .build();
6
7 Table event = table("event")
8   .column("id", DataType.BIGINT)
9   .defaultIdentifier()
10  .column("name", DataType.VARCHAR)
11  .column("organizer", DataType.BIGINT)
12  .build();
13
14 associativeTable("participations")
15   .column("event", DataType.BIGINT)
16   .reference
17    .foreign(event)
18   .column("participant", DataType.BIGINT)
19   .reference
20    .foreign(person)
21   .build();

```

Listing 5.21: Zirkuläre Beziehungen mit assoziativer Tabelle

- **Ternäre Beziehungen**

5.10.3 Komfortfunktionen

Die Realisierung könnte an manchen Stellen dem Test-Ingenieur mehr manuelle Arbeit abnehmen. So wird darauf verzichtet, beim Löschen einer Zeile aus einer Tabelle auch alle beteiligten Beziehungen zu entfernen. Listing 5.22 zeigt, wie ein Professor aus der Professoren-Tabelle entfernt wird. Die erste Zeile entfernt keine Einträge in anderen Tabellen wie z.B. der Beaufsichtigt-Tabelle. Folglich müssen die Relationen (mehr oder weniger) manuell aus anderen Tabellen entfernt werden.

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

Listing 5.22: 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 Ausdruckstärke des Tests.
- **Code-Qualität:** Eine Funktion (bzw. Methode) sollte genau eine Aufgabe erledigen. Wenn `deleteRow` zusätzlich beteiligte Relationen auflöst, erledigt diese Funktion mehr als nur eine Aufgabe [Mar09, 65f]. Außerdem würde es sich um einen unerwarteten Nebeneffekt handeln [Mar09, 75f].
- **Klarheit:** Es ist nicht eindeutig, wie beim Entfernen von Zeilen vorgegangen werden soll, wenn sie Teil einer Relation sind. Bei einer n:m-Relation könnte sich die Regel ableiten lassen, dass beim Löschen einer Zeile auch alle assoziierten n:m-Relationen entfernt werden können. Aber was ist bei einer 1:n-Relation? Wenn ein Professor entfernt wird, was soll mit Lehrveranstaltungen passieren, die ihm zugeordnet sind?

Kapitel 6

Generieren von Testdaten

Das Generieren von Test-Daten ist kein neues Thema. Es gibt verschiedene Ansätze dafür. Einige Programme generieren Massendaten. Die Erwartung ist, dass durch die Vielzahl zufälliger Daten auch alle Arten von Beziehungen ausreichend generiert werden. Ein solcher Massendaten-Generator lässt sich vergleichsweise leicht realisieren. Für Unit-Tests mit überschaubaren Test-Daten ist dieser Ansatz eher nicht geeignet.

Ein anderer Ansatz ist für Unit-Tests entgegenkommender. Dort werden Test-Daten anhand von konkreten Abfragen generiert. Allerdings ist es nicht Ziel der Arbeit, für jeden Test individuelle Daten zu generieren. Außerdem können Anfragen vom SUT auch abstrahiert werden, beispielsweise wenn ein Spring-Service verwendet wird.

Für die Generierung von Test-Daten aus einem Datenbank-Modell bietet sich die erste Variante an. Da die Menge der generierten Test-Daten gering gehalten werden soll, soll ein Algorithmus entwickelt werden, der Beziehungen nicht rein zufällig modelliert. Stattdessen soll versucht werden, möglichst alle Grenzfälle zu berücksichtigen. Äquivalenzklassenbildung und Grenzwertanalyse sind ein bewährtes Vorgehen, um die Menge von Test-Daten zu reduzieren.

Im Folgenden wird beschrieben, welche Fälle für verschiedene binäre Beziehungen auf jeden Fall generiert werden müssen.

6.1 Generieren von Beziehungen

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

Die Abbildungen stellen dar, welche Entitäten *mindestens* generiert werden sollten.

6.1.1 Klasse der 1:1-Beziehungen

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

1..1:1..1

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



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

0..1:1..1

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

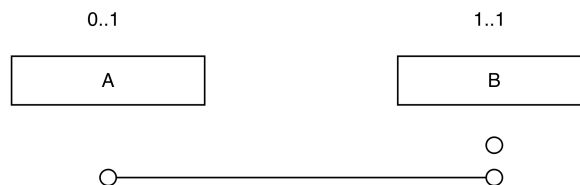


Abbildung 6.2: Beziehungen nach dem Schema 0..1:1..1

Der Generator muss mindestens zwei Entitäten des Typs B erzeugen, und eine des Typs A, um sicherzustellen, dass alle Fälle für diese Beziehung abgedeckt sind. Die Beziehung 1..1:0..1 ist symmetrisch zu dieser.

0..1:0..1

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

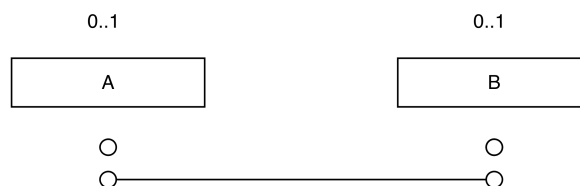


Abbildung 6.3: Beziehungen nach dem Schema 0..1:0..1

6.1.2 Klasse der 1:n-Beziehungen

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

In den Abbildungen wird als Grenze n der Wert 3 verwendet.

1..1:1..n

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

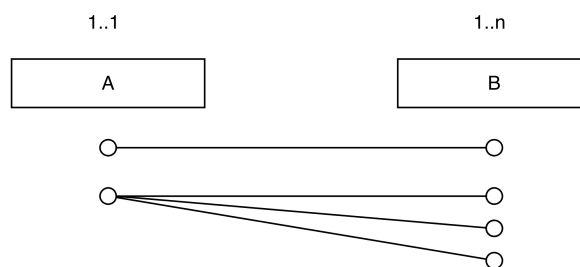


Abbildung 6.4: Beziehungen nach dem Schema 1..1:1..n

0..1:1..n

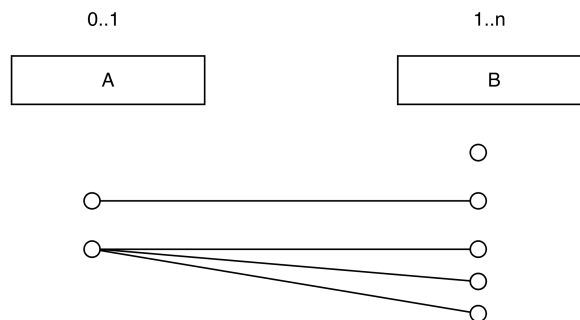


Abbildung 6.5: Beziehungen nach dem Schema 0..1:1..n

1..1:0..n

0..1:0..n

6.1.3 Klasse der n:m-Beziehungen (assoziative Beziehungen)

6.2 Der Algorithmus

1. Start-Tabelle festlegen 2. wenn Tabelle noch nicht behandelt, dann 3. Kanten bestimmen und behandeln (ausgehende bevorzugt) 4. Schritte 2-4 rekursiv für die verbundene Tabelle wiederholen 5. Prüfen ob alle Entitäten gültige Beziehungen haben, ggf. Entitäten erstellen

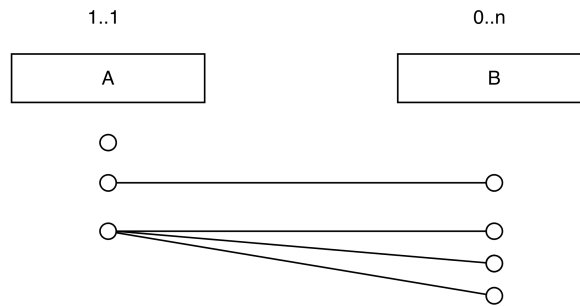


Abbildung 6.6: Beziehungen nach dem Schema 1..1:0..n

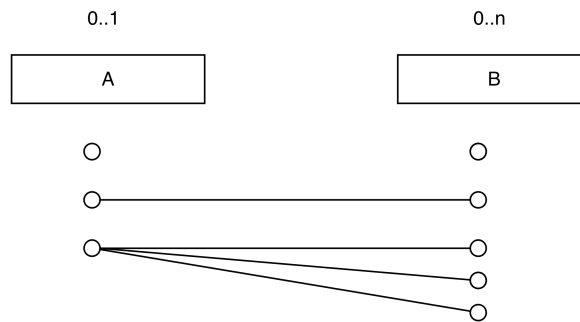


Abbildung 6.7: Beziehungen nach dem Schema 0..1:0..n

Assoziative Tabellen werden separat behandelt Jede Kante und jede Tabelle wird nur einmal in den Schritten 2-4 besucht

6.3 Grenzen des Algorithmus

6.4 Erweiterungen am Modell

Red Gate SQL Data Generator: - Fragt Datenbank-Schema aus SQL-Server ab - Modelliert Beziehungen(!), allerdings zufällig und bedeutungslos - Anzahl generierter Zeilen pro Tabelle steuerbar - Zufall pro Spalte über Seed steuerbar - Eher für große Datenmengen gedacht

DTM Data Generator - Weniger bequem als Red Gate, z.B. keine Erkennung der Spaltentypen -

- Beziehungstypen, wie werden sie generiert - Assoziative Tabellen

6.5 -Warum Generierung

Möglichkeiten: - Generierung - Real-Daten, Extraktion, Anonymisierung (bei Unit-Tests wenig praktikabel, da Tests vor Inbetriebnahme laufen sollen)

Je nach Anwendungsfall unterschiedliche Data-Sets sinnvoll - Unit-Tests eher klein - Performance-Tests, Regressionstests eher groß

Kapitel 7

Proof of Concept

Kapitel 8

Zusammenfassung und Ausblick

Abbildungsverzeichnis

2.1	Back Door Manipulation	4
2.2	Stil von ER-Diagrammen	5
2.3	Stil relationaler Datenbank-Diagramme nach Ambler	5
2.4	Modell-Beschreibung	6
2.5	Logisches Klassendiagramm der STU-DataSet-Klassen	8
3.1	ER-Diagramm des fortlaufenden Beispiels	13
3.2	Relationales Datenbank-Diagramm des fortlaufenden Beispiels	14
5.1	Architektur	32
5.2	Klassendiagramm der DataSet-Builder	37
5.3	Beispiel JavaDoc-Tooltip	44
6.1	Beziehungen nach dem Schema 1..1:1..1	50
6.2	Beziehungen nach dem Schema 0..1:1..1	50
6.3	Beziehungen nach dem Schema 0..1:0..1	50
6.4	Beziehungen nach dem Schema 1..1:1..n	51
6.5	Beziehungen nach dem Schema 0..1:1..n	51
6.6	Beziehungen nach dem Schema 1..1:0..n	52
6.7	Beziehungen nach dem Schema 0..1:0..n	52

Listings

3.1	XML-DataSet	15
3.2	Flat-XML-DataSet	16
3.3	Default-DataSet	17
3.4	STU DataSet (1)	18
3.5	STU DataSet (2)	19
4.1	Mögliche DSL (1)	22
4.2	Mögliche DSL (2)	22
4.3	Mögliche DSL (3)	23
4.4	Vereinfachung von Ausdrücken in Groovy	24
4.5	Tabellen-Parser Grundgerüst mit Operator-Überladen	25
4.6	Tabellen-Parser Grundgerüst mit Operator-Überladen	26
4.7	DSL-Entwurf 3 für Laufzeit-Meta-Programmierung angepasst	27
4.8	EBNF der Tabellen	28
4.9	EBNF der Relationen	28
4.10	DataSet modelliert mit Table Builder API	29
4.11	Beziehungen innerhalb von Tabellen	30
5.1	Beispiel für unveränderliche Identifikatoren	33
5.2	Beispiel SB-Testing-DB-Builder	35
5.3	Beispiel <i>STU</i> -Builder	35
5.4	Beispiel für assoziative Tabelle	36
5.5	Binden von Referenzen (Table Builder API)	38
5.6	Binden von Referenzen (Fluent Builder API)	38
5.7	Zugriff auf Werte über Referenzen	38
5.8	Definition von Beziehungen über Referenzen	38
5.9	JUnit-Tests (reiner Java-Code)	39
5.10	Test-Methode in Groovy	40
5.11	Erweitertes DataSet	41

5.12	Erweitertes DataSet	41
5.13	Test auf erweiterem DataSet	41
5.14	Beispiel Lazy Valunes	42
5.15	Such-Beispiele	43
5.16	Beispiel für find	43
5.17	Beispiel für foreach	43
5.18	Reflexive Beziehungen manuell	45
5.19	Reflexive Beziehungen mit Hilfe assoziativer Tabelle	46
5.20	Zirkuläre Beziehungen manuell	46
5.21	Zirkuläre Beziehungen mit assoziativer Tabelle	46
5.22	Löschen von Zeilen	47

Literatur

- [AS06] Scott W. Ambler und Pramod J. Sadalage. *Refactoring Databases, Evolutionary Database Design*. The Addison-Wesley Signature Series. Addison-Wesley, 2006. ISBN: 978-0-3212-9353-4. URL: <http://www.addison-wesley.de/main/main.asp?page=aktionen/bookdetails&ProductID=108888>.
- [Blo08] Joshua Bloch. *Effective Java Second Edition*. The Java Series. Addison-Wesley, 2008. ISBN: 9780321356680. URL: <http://books.google.com/books?id=ka2VUBqHiWkC&pg>.
- [Com13] Google Guava Community. *Optional (Guava: Google Core Libraries for Java 15.0-SNAPSHOT API)*. Google Guava Community, 2013. URL: <http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/base/Optional.html> (besucht am 19.07.2013).
- [Eva04] 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>.
- [Fou04] The Apache Software Foundation. *Apache License, Version 2.0*. The Apache Software Foundation, 2004. URL: <http://www.apache.org/licenses/LICENSE-2.0.html> (besucht am 01.01.2004).
- [Fow10] Martin Fowler. *Domain-Specific Languages*. The Addison-Wesley Signature Series. Addison-Wesley, 2010. ISBN: 978-0321712943. URL: <http://martinfowler.com/books/dsl.html>.
- [Gam+95] 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>.
- [Gho10] Debasish Ghosh. *DSLs in Action*. Manning, 2010. ISBN: 978-1-935182-45-0. URL: <http://www.manning.com/ghosh/>.
- [Goe09] Brian Goetz. *Java concurrency in practice*. 7. print. Addison-Wesley, 2009. ISBN: 978-0-321-34960-6. URL: <http://www.gbv.de/dms/ilmenau/toc/601225643.PDF>.
- [HM05] Rob Harrop und Jan Machacek. *Pro Spring*. Apress, 2005. ISBN: 978-1-59059-461-2. URL: http://books.google.de/books/about/Pro_Spring.html?id=q2PIjAJoxSAC&redir_esc=y.

- [Kön07] Dierk König. *Groovy im Einsatz*. Fachbuchverl. Leipzig im Carl-Hanser-Verl., 2007. ISBN: 978-3-446-41238-5. URL: http://deposit.d-nb.de/cgi-bin/dokserv?id=2948820&prov=M&dok_var=1&dok_ext=htm.
- [Mar09] Robert C. Martin. *Clean Code: Refactoring, Patterns, Testen und Techniken für sauberen Code*. mitp-Verlag, 2009. ISBN: 978-3-8266-5548-7. URL: <http://www.it-fachportal.de/shop/buch/Clean%20Code%20-%20Refactoring,%20Patterns,%20Testen%20und%20Techniken%20f%C3%BCr%20sauberen%20Code/detail.html,b164659>.
- [Mes07] Gerard Meszaros. *XUnit Test Patterns: Refactoring Test Code*. The Addison-Wesley Signature Series. Addison-Wesley, 2007. ISBN: 978-0-13-149505-0. URL: <http://xunitpatterns.com/index.html>.
- [Mof] *Meta Object Facility (MOF) Specification 1.4.1*. 2005. URL: <http://www.omg.org/spec/MOF/ISO/19502/PDF/> (besucht am 17.07.2011).
- [SL10] Andreas Spillner und Tilo Linz. *Basiswissen Softwaretest*. dpunkt.verlag, 2010. ISBN: 978-3-89864-642-0. URL: <http://www.dpunkt.de/buecher/4075.html>.
- [Sta07] 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/ilmeneau/toc/528370707.PDF>.
- [Win+12] 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. 3): ergänzen
- ☐ 2 (p. 4): Layer test erklären
- ☐ 3 (p. 6): Kurz erklären, was ein DataSet ist
- ☐ 4 (p. 7): REF GOF
- ☐ 5 (p. 9): Gültigkeitsbereiche erklären
- ☐ 6 (p. 14): Überleitung
- ☐ 7 (p. 24): Quelle Kent Beck Smalltalk Best Practice Patterns
- ☐ 8 (p. 33): Abhängigkeitsdiagramm der neuen Builder-Klassen?
- ☐ 9 (p. 39): Quelle Spring Service