Modellierung und Generierung von Testdaten für Datenbank-basierte Anwendungen

Nikolaus Moll*
Christian Baranowski, Thomas Fox, Jürgen Wäsch
Seerhein-Lab, Konstanz† (www.seerhein-lab.org)
stu@dev.nikolaus-moll.de

Abstract: In dieser Arbeit wird ein Ansatz zur Vereinfachung der Spezifikation von Testdaten für Datenbank-basierte Anwendungen vorgestellt. Dies beinhaltet eine DSL zur einfachen und übersichtlichen Beschreibung von Daten und deren Beziehungen sowie einen Generator zur automatischen Erzeugung von Testdaten.

1 Problemstellung und Ansatz

Softwaretests sind ein wichtiger Baustein für die Qualitätssicherung von Softwareprojekten. Für Tests von Datenbank-basierten Anwendungen müssen u.a. Testdaten für die Datenbank spezifiziert werden, auf deren Basis das Verhalten der zu testenden Software geprüft werden kann. Die Spezifikation dieser Testdaten ist leider augenblicklich sehr umfangreich und komplex und somit aufwändig und fehleranfällig. Die Komplexität ergibt sich v.a. aus der Beschreibung der Beziehungen zwischen den einzelnen Entitäten. Diese unterliegen einer Menge komplexer fachlicher Regeln, die sich aus dem Domänen-Modell und der Geschäftslogik der Anwendung ergeben.

Übergreifendes Ziel der hier beschriebenen Arbeit [Mol13] war es, die Spezifikation von Testdaten für Datenbank-basierte Java-Anwendungen zu vereinfachen. Hierzu wurde zum einen eine geeignete Domänen-spezifische Sprache (DSL) für Testdaten entwickelt. Zum anderen wurde ein Generator zur automatischen Erzeugung von Testdaten implementiert. Basis der Entwicklungsarbeiten war die Java-Bibliothek Simple Test Utils for JUnit & Co. (STU) zur Vereinfachung von Unit-Tests für Java-Anwendungen. STU steht unter der Apache License 2.0 und wird federführend von der SEITENBAU GmbH entwickelt.

Abb. 1 gibt einen Überblick über den gewählten, modellgetriebenen Ansatz. Ausgangspunkt ist eine formale Beschreibung des relationalen Datenbankschemas (Details siehe [Mol13]). Diese kann mittels eines Tools (nicht dargestellt) manuell erstellt bzw. aus einer existierenden Datenbank extrahiert und ergänzt werden. Aus der Schema-Beschreibung wird die Schema-abhängige Testdaten-DSL generiert. Diese DSL kann dann von den Testern genutzt werden, um verschiedene Testdaten-Sets zu beschreiben und diese mittels STU in ihre Unit-Tests einzubinden. Die Testdaten-Sets werden bei den Tests durch das STU-Framework automatisch in die Datenbank eingespielt. Auf Basis der Schema-Beschreibung können auch in der DSL beschriebene Testdaten-Sets generiert werden. Die generierten Testdaten können ggfls. vor Verwendung noch angepasst werden.

*Bis 31.12.2013 Student im Master-Studiengang Informatik / akademischer Mitarbeiter der Hochschule Konstanz (HTWG); seit 15.01.2014 tätig bei der PENTASYS AG in München.

†Das Seerhein-Lab ist eine Kooperation der HTWG Konstanz und der Firma SEITENBAU GmbH.

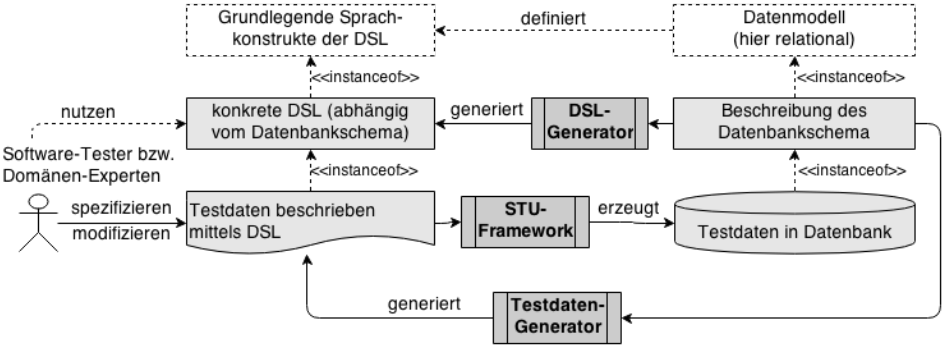

Abbildung 1: Überblick über den gewählten Ansatz.

2 Testdaten-DSL

Es wurden verschiedene Ansätze zur Entwicklung einer DSL für Testdaten untersucht. Der Fokus lag u.a. auf der Fachlichkeit der Datenstruktur, der typsicheren Beschreibung der Testdaten und der einfachen Spezifikation von Beziehungen zwischen Entitäten. Untersucht wurden verschiedene XML-basierte Darstellungen, wie z.B. in DbUnit benutzt, programmatische Spezifikationen und tabellarische Beschreibungsformen. Nach einer Evaluation wurde eine tabellarische Beschreibungsform gewählt. Diese Art der Testdatenmodellierung ist übersichtlich und syntaktisch einfach. Die grundlegende Idee stammt vom Testframework Spock [N+12]. Die EBNF der DSL ist in [Mol13] zu finden.

Listing 1 zeigt beispielhaft die Testdaten-DSL für eine Bücherverwaltung (Datenbankschema siehe Abb. 2 oben). In der tabellarischen Darstellung (`tables`) enthält die erste Zeile die Spaltennamen der Tabelle, die anderen Zeilen enthalten die einzufügenden Daten. Die erste Spalte einer Datenzeile enthält jeweils einen symbolischen Namen (`REF`) für den Tabelleneintrag, der zur Referenzierung und somit Spezifikation von Beziehungen (`relations`) zwischen Datensätzen genutzt werden kann.

Die Implementierung der Testdaten-DSL basiert auf der dynamischen Programmiersprache Groovy und verwendet Laufzeit-Metaprogrammierung in Verbindung mit Operator-Überladen [Mol13]. Die DSL kann eingebettet zusammen mit Java in den Tests (z.B. mit JUnit) genutzt werden und integriert sich sehr gut in gängige IDEs wie Eclipse. Die Spaltennamen sind in der DSL definiert, so dass Autovervollständigung unterstützt wird. Über die REF-Namen können Beziehungen typsicher modelliert und konkrete Werte abgefragt werden. Details zur Implementierung, zur Generierung der DSL für ein Datenbankschema und zur Nutzung der DSL in Softwaretests mit STU sind in [Mol13] zu finden.

3 Generierung von Testdaten

Zielsetzung war die Generierung von möglichst kleinen Testdaten-Sets, die für möglichst viele fachliche Tests verwendet werden können, d.h. eine hohe Testabdeckung bieten. Eine umfassende Literaturanalyse und die Evaluation existierender Werkzeuge ergab, dass bisher keine passende Lösung für diese Anforderung existiert. Aus diesem Grund wurde ein neuer Algorithmus zur Testdatengenerierung entworfen. Anleihen konnten dabei aus [HTW06] gezogen werden. Idee ist, durch Nutzung von Äquivalenzklassen und das gezielte Generieren von Grenzfällen bei Beziehungen eine hohe Testabdeckung zu erreichen.

```
class BookDatabaseGroovyDataSet extends BookDatabaseBuilder {
  def tables() {
    buchTable.rows {
      REF           | name
      CLEANCODE     | "Clean Code"
      EFFECTIVEJAVA | "Effective Java"
      DESIGNPATTERNS | "Design Patterns" }
    verlagTable.rows {
      REF           | name
      PRENTICE      | "Prentice Hall International"
      ADDISONWESLEY | "Addison-Wesley" }
    autorTable.rows {
      REF        | vorname     | nachname
      UNCLEBOB   | "Robert C." | "Martin"
      BLOCH      | "Joshua"    | "Bloch"
      GAMMA      | "Erich"     | "Gamma"
      HELM       | "Richard"   | "Helm"
      JOHNSON    | "Ralph"     | "Johnson"
      VLISSIDES  | "John"      | "Vlissides"  }
  }
  def relations() {
    PRENTICE.verlegt(CLEANCODE)
    ADDISONWESLEY.verlegt(EFFECTIVEJAVA, DESIGNPATTERNS)
    CLEANCODE.geschriebenVon(UNCLEBOB)
    EFFECTIVEJAVA.geschriebenVon(BLOCH)
    DESIGNPATTERNS.geschriebenVon(GAMMA, HELM, JOHNSON, VLISSIDES)
  }
}
```

Listing 1: Beispiel eines mittels DSL beschriebenen Testdaten-Sets.

Der Algorithmus betrachtet das Datenbankschema als Graph. Tabellen stellen Knoten, Beziehungen (Fremdschlüssel) stellen Kanten dar. Da assoziative Tabellen ebenfalls Beziehungen ausdrücken, werden diese als besondere Kanten behandelt. Ausgehend von einem beliebigen Startknoten werden die Kanten und die damit verbundenen Knoten rekursiv traversiert. Der Algorithmus erzeugt zu jeder Kante Entitäten der beteiligten Tabellen und mindestens Beziehungen für alle Kombinationen der unteren und oberen Grenzwerte, entsprechend den spezifizierten Unter- und Obergrenzen. Gleichzeitig wird versucht, die Anzahl der generierten Entitäten und Beziehungen zu minimieren. Aus diesem Grund werden auch nicht alle, über mehrere Kanten hinweg, mögliche Kombinationen berücksichtigt, da dies zu einer kombinatorischen Explosion führen würde.

Der Algorithmus soll an dem Datenbankschema aus Abb. 2 (UML-Darstellung mit Multiplizitäten) veranschaulicht werden. Als Startknoten wird im Beispiel *Buch* verwendet. Von hier aus werden alle Kanten besucht, hier angefangen mit der Kante (1..1:0..*) zum Knoten *Verlag*. Um möglichst alle Grenzfälle bzw. Äquivalenzklassen abzudecken, wird erzeugt: (1) eine Verlags-Entität, die keine Bücher verlegt, (2) ein Verlag, der genau ein Buch verlegt und (3) ein Verlag, der viele Bücher veröffentlicht[1]. Der Knoten *Verlag* hat keine nichtbesuchten Kanten, weshalb die Traversierung in *Buch* fortgesetzt wird mit der Kante zum Knoten *BuchAutor* (assoziative Tabelle, die eine 0..*:1..*-Assoziation zwischen *Buch* und *Autor* ausdrückt). Der Algorithmus sieht vor, die vier möglichen min/max-Kombinationen zwischen *Buch* und *Autor* zu generieren. Jede Beziehung zwischen einem *Buch* und einem *Autor* resultiert in einer Entität in der Tabelle *BuchAutor*. Existierende Entitäten in *Buch* und *Autor* werden für diese generierten Beziehungen soweit möglich wiederverwendet, bei Bedarf werden neue Entitäten in *Buch* und *Autor* generiert. Die Traversierung des Graph endet nun, da jede Kante besucht wurde. Allerdings sind einige der bis hierhin erzeugten *Buch*-Entitäten noch ungültig, da sie noch nicht zu einem *Verlag* gehören. Solche Entitäten

[1] Für * wird ein konfigurierbarer Wert verwendet, im Beispiel 4.

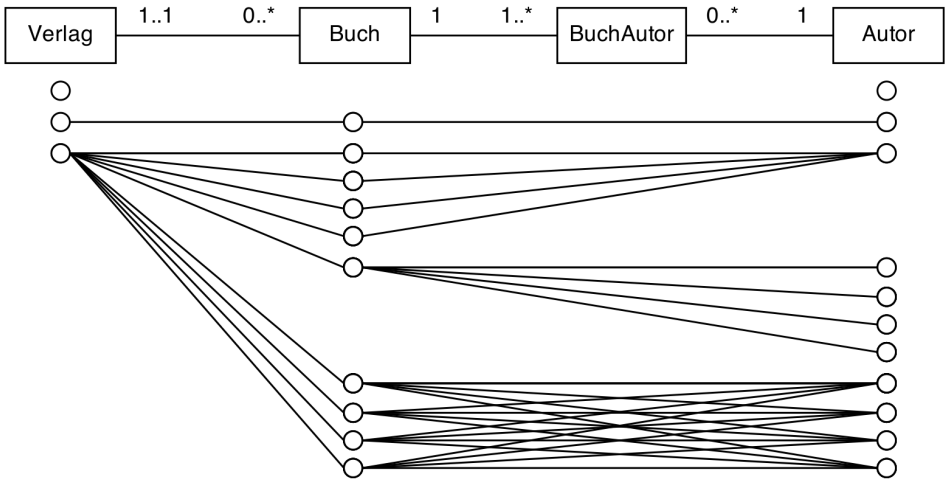

Abbildung 2: Beispiel-Datenbankschema und daraus generierte Entitäten und Beziehungen.

werden im letzten Schritt des Algorithmus behandelt. Alle Entitäten werden auf Gültigkeit bzgl. ihrer Beziehungen überprüft und bei Bedarf werden weitere Beziehungen und weitere Entitäten erzeugt. Abb. 2 stellt das im Beispiel generierte Testdaten-Set (Entitäten und ihre Beziehungen) grafisch dar.

Details zum Algorithmus (Pseudocode) und zur Java-Implementierung sind in [Mol13] zu finden. Evaluationen haben gezeigt, dass die generierten Testdaten unabhängig von der Reihenfolge der Traversierung und bzgl. des Datenbankschemas immer gültig sind. Zur Erzeugung der Werte von Entitäten wurden verschiedene Wertegeneratoren implementiert.

4 Fazit

Die in dieser Arbeit vorgestellten Konzepte und die daraus resultierende Software wurden in das vorhandene STU-Framework integriert. Der Code steht unter der Apache License 2.0 unter `https://github.com/seitenbau/stu/` zur Verfügung.

Die entwickelte Testdaten-DSL wurde bereits in der Qualitätssicherung von mehreren produktiven Softwareprojekten eingesetzt. Der Spezifikationsaufwand und die Fehlerrate konnte im Vergleich zur früheren Vorgehensweise deutlich reduziert werden. Der Testdatengenerator wurde dabei auf Datenbankschemata mit teilweise mehr als 80 Tabellen angewandt. Der Testdatengenerator konnte in jedem Fall einen konsistenten, übersichtlichen Testdaten-Set erzeugen, der eine sehr gute Startbasis für die Anwendungstests ergab.

Literatur

[HTW06] Kenneth Houkjaer, Kristian Torp und Rico Wind. Simple and realistic data generation. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, 2006.

[Mol13] Nikolaus Moll. Testdaten-Modellierung und -Generierung für Datenbank-basierte Anwendungen. Masterthesis, HTWG Konstanz / SEITENBAU GmbH, Oktober 2013. `http://nikolaus-moll.de/Masterthesis.pdf`.

[N+12] Peter Niederwieser et al. *Spock - the enterprise ready specification framework*, 2012. `http://spockframework.org/`.