Modellierung und Generierung von Test-Daten für Datenbank-basierte Anwendungen*

Nikolaus Moll†
Jürgen Wäsch, Christian Baranowski
HTWG Konstanz / SEITENBAU GmbH
stu@dev.nikolaus-moll.de
juergen.waesch@htwg-konstanz.de
christian.baranowski@seitenbau.com

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 übersichtliche Beschreibung von Daten und deren Beziehungen sowie ein Generator zur automatischen Erzeugung von Testdaten. Die in Java entwickelte Software steht unter der Apache License 2.0 zur Verfügung.

1 Problemstellung und Überblick

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 i.A. sehr umfangreich und komplex und somit aufwändig und fehleranfällig. Die Komplexität ergibt sich v.a. aus der Beschreibung der Beziehungen zwischen den einzelnen Datensätzen. Diese unterliegen einer Menge komplexer fachlicher Regeln, die sich aus dem Domänen-Modell und der Geschäftslogik der Anwendung ergeben. Besonders bei Systemen mit großen oder komplexen Datenbank-Schemata kann ein Testdaten-Set schnell unübersichtlich werden.

Übergreifendes Ziel der hier beschriebenen Arbeit [Mol13] war es, die Spezifikation von Testdaten für Datenbank-basierte Java-Anwendungen zu vereinfachen. Zum einen wurde hierzu eine geeignete Domänen-spezifische Sprache (DSL) für Testdaten entwickelt. Die DSL erlaubt eine übersichtliche Spezifikation von Testdaten, ist einfach zu nutzen und integriert sich in gängige Entwicklungsumgebungen. Zum anderen wurde ein Generator zur automatischen Erzeugen von Testdaten implementiert. Das Ziel hierbei war es, mit möglichst wenig Datensätzen viele Grenzfälle bei Beziehungen abzudecken. 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

*Diese Arbeit wurde im Seerhein-Lab (www.seerhein-lab.org/) durchgeführt, einer Kooperation der HTWG Konstanz und der Firma SEITENBAU GmbH.

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

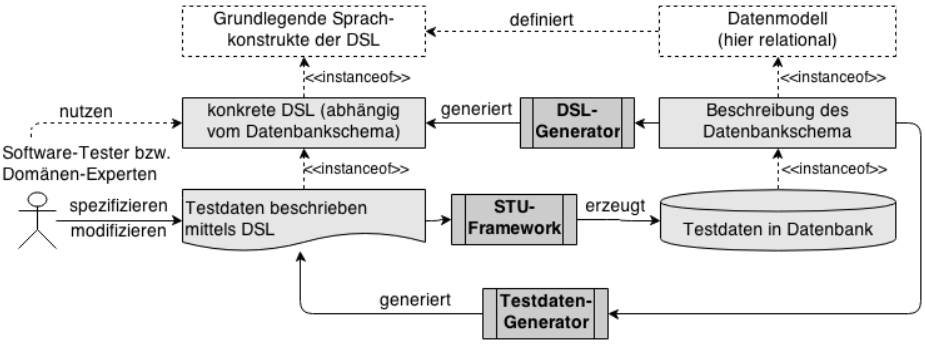

Abbildung 1: Überblick über den gewählten Ansatz.

License 2.0 und wird federführend von der Firma Seitenbau entwickelt. Für Tests von Datenbank-basierten Anwendungen setzt STU auf der Bibliothek DbUnit auf.

Abbildung 1 gibt einen Überblick über den im Projekt gewählten Ansatz. Ausgangspunkt ist eine speziellen Beschreibung des relationalen Datenbankschemas (Details siehe [Mol13]). Diese kann mittels eines Tools (nicht in der Abb. dargestellt) manuell erstellt bzw. aus einer existierenden Datenbank extrahiert und ergänzt werden. Aus der Beschreibung des Datenbankschemas wird die schema-abhängige Testdaten-DSL generiert. Diese DSL kann dann von den Software-Testern genutzt werden, um verschiedene Testdaten-Sets zu beschreiben und diese mittels STU in ihre Unit-Tests einzubinden. Die mittels der DSL spezifizierten Testdaten werden dabei durch das STU-Frameworks automatisch in die Datenbank eingespielt (Backdoor-Manipulation [Mes07]). Ausgehend von der Schema-Beschreibung können in der DSL beschriebene Testdaten-Sets automatisch generiert werden. Die generierten Testdaten können ggfls. vor Verwendung noch modifiziert werden.

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 verschiedene tabellarischen Beschreibungsformen für die Testdaten. Nach einer Evaluation wurde eine tabellarische Beschreibungsform gewählt, die über das STU-Framework genutzt werden kann. Diese Art der Testdatenmodellierung ist übersichtlich, syntaktisch einfach und kann von einer IDE wie Eclipse unterstützt werden. Die grundlegende Idee für die tabellarische Darstellung stammt vom Testframework Spock [N+12]. Die EBNF der DSL ist in [Mol13] zu finden.

Listing 1 zeigt beispielhaft einen Auszug aus einer DSL-Beschreibung für Testdaten einer Anwendung zur Verwaltung von Büchern (Datenbank-Schema siehe Abb. 2). 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

```
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: Mittels DSL beschriebenes Testdaten-Set (Table Builder API).

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 dabei auf Groovy, einer dynamisch typisierten Programmiersprache für die Java Virtual Machine. Die DSL kann eingebettet zusammen mit Java in den Tests, z.B. mit JUnit, genutzt werden. ...

Details zur Implementierung sind in [Mol13] zu finden.

ToDo: Evtl. nicht mehr notwendig: Definition eines Datenbank-Schemas, Details Generierung der DSL sowie Unit-test Beispiele etc. ...

3 Generierung von Testdaten

Um das Testen von Datenbank-basierten Anwendungen zu erleichtern, soll es möglich sein, automatisch Testdaten für funktionale Tests aus dem Datenbankschema generieren zu lassen. Die generierten Testdaten können direkt bzw. als Basis für die Erstellung von Testdatensätzen genutzt werden (vgl. Abb. 1). Das Ziel sind kleine Datasets, die für möglichst viele Unit-Tests verwendet werden können. Kleine Datasets lassen sich von Testern einfacher verstehen. Einheitliche Testdaten erleichtern den Umgang mit Unit-Tests, da sich der Tester nicht immer wieder in verschiedene Datasets hineinversetzen muss.

Eine umfassende Literaturanalyse und die Analyse existierender Testdatengeneratoren zeig-

ten, dass bisher keine passende Lösung für diese Anforderungen existiert. In der Wissenschaft beschriebene Ansätze und Algorithmen generieren meist für eine zu testende SQL-Abfrage einen passenden Testdatenbestand. Einer SQL-Abfrage liegt dabei eine formale Spezifikation zu Grunde, die allerdings für ein Anwendungsprogramm normalerweise nicht vorhanden ist. Existierende Software-Werkzeuge fokussieren sich auf die Generierung von Massendaten, die v.a. für Performanz-Tests und nicht für funktionale Tests geeignet sind. Dies zeigt sich auch daran, dass diese Werkzeuge Beziehungen zwischen Entitäten nur zufällig erzeugen und im Allgemeinen wesentlich mehr Daten generieren als für einen Menschen noch einfach verständlich sind. Weiterhin deuten Untersuchungen im Projekt darauf hin, dass die Komplexität der Testdatengenerierung im allgemeinen Fall nicht-polynomial ist.

Aus diesem Grund wurde ein neuer, effizienter Algorithmus zur Testdatengenerierung für die Projektproblemstellung entworfen. Der entwickelte Algorithmus berücksichtigt lokal alle Kombinationen der unteren und oberen Grenzwerte von binären Beziehungstypen n..N↔m..M und versucht gleichzeitig die Anzahl der generierten Entitäten und Beziehungen zu minimieren. Globale, d.h. ”transitive” Abhängigkeiten über mehrere Beziehungstypen hinweg, die zu einer kombinatorischen Explosion führen können, bleiben dabei (augenblicklich) unberücksichtigt.

Der Algorithmus betrachtet das Datenbank-Schema als Graph. Die Tabellen stellen die Knoten und die Beziehungen stellen die Kanten dar. Da assoziative Tabellen ebenfalls Beziehungen ausdrücken, werden diese vom Algorithmus als besondere Kante behandelt. Der Graph wird ausgehend von eines Knotens traversiert. Alle Kanten des aktuellen Knotens und die damit verbundenen Knoten werden rekursiv besucht. Der Algorithmus erzeugt zu jeder Kante Entitäten der beiden beteiligten Tabellen. Jede Kante und jede Tabelle werden genau genau einmal durchlaufen.

Der Algorithmus soll an dem in Abb. 2 dargestellten Datenbank-Schema einer Bücherverwaltung veranschaulicht werden. Ein Buch gehört genau zu einem Verlag und wird von beliebig vielen (mindestens einer) Autoren geschrieben. Ein Verlag kann selbst beliebig viele und sogar keine Bücher veröffentlichen und ein Autor kann an keinem oder beliebig Büchern beteiligt sein.

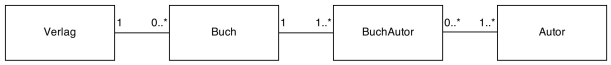

Abbildung 2: Datenbank-Schema der Bücherverwaltung

Als Ausgangspunkt für die Traversierung wird der Knoten Buch verwendet. Von diesem Knoten aus werden alle Kanten besucht, angefangen mit der Kante zum Knoten Verlag. Dieser repräsentiert eine 1:0..*-Beziehung. Um möglichst alle Grenzfälle abzudecken, wird eine Entität von Verlag erzeugt, die keine Bücher verlegt. Außerdem wird ein Verlag generiert, der genau ein Buch verlegt, und ein Verlag, der viele Bücher veröffentlicht. Anstelle von * kann ein konfigurierbarer Wert verwendet werden, in dem Beispiel wird dafür 4 verwendet. Für diese Kante werden also drei Entitäten des Typs Verlag und fünf Entitäten des Typs Buch erzeugt.

Der Knoten Verlag hat keine nicht-besuchten Kanten, weshalb die Traversierung in Buch

fortgesetzt wird mit der Kante zum Knoten BuchAutor. Die Kante hat eine assoziative Tabelle als Ziel, weshalb hier andere Schritte notwendig sind wie bei der letzten Kante. Die Tabelle BuchAutor drückt eine 0..*:1..*-Beziehung zwischen Buch und Autor aus. Der Algorithmus sieht vor, die vier möglichen min/max-Kombinationen generiert werden. Jede Beziehung zwischen einem Buch und einem Autor führt zu einer Entität in der Tabelle BuchAutor. Es wird ein Autor benötigt, der kein Buch geschrieben hat, und ein Autor, der genau an einem Buch beteiligt ist (min:min). Außerdem wird ein Autor benötigt, der vier Bücher schreibt (max:min), und ein Buch, das vier Autoren hat (min:max). Und schließlich werden vier Bücher benötigt, die jeweils die gleichen vier Autoren haben (max:max). Für die assoziative Tabelle BuchAutor werden folglich zehn Entitäten des Typs Buch und elf Entitäten des Typs Autor benötigt, wobei die bereits erzeugten Entitäten des Typs Buch hier weiterverwendet werden. Die Traversierung endet hier, da jede Kante besucht wurde.

Die fünf im vorausgegangenen Schritt erzeugten Entitäten des Typs Buch sind allerdings noch ungültig, da sie noch nicht zu einem Verlag gehören. Solche Entitäten werden im letzten Schritt behandelt. Alle Entitäten werden auf Gültigkeit bzgl. ihrer Beziehungen überprüft, und bei Bedarf werden weitere Beziehungen und falls notwendig weitere Entitäten erzeugt. Abb. 3 stellt die generierten Entitäten und ihre Beziehungen grafisch dar. Eine Entität wird als Kreis dargestellt, die Verbindungslinien zwischen den Entitäten stellen ihre Beziehungen dar. Dazu gehören auch die genierten Entitäten in der assoziativen Tabelle BuchAutor.

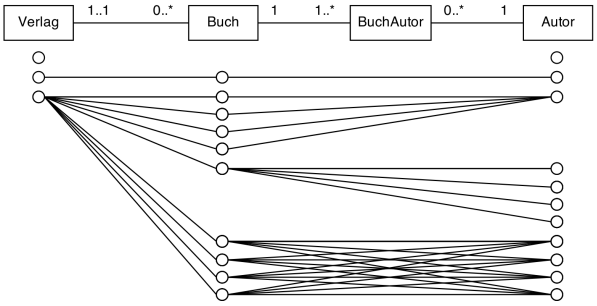

Abbildung 3: Generierte Entitäten und Beziehungen

Der Algorithmus wurde in Java als Teil von STU implementiert. Details zum Algorithmus und sein Pseudo-Code, zur Implementierung und zur Generierung von Daten für die einzelnen Entitäten sind in [Mol13] zu finden. Verschiedene Evaluationen haben gezeigt, dass sich die erzeugten Testdaten in Unit-Tests nutzen lassen und auch in die Datenbank einspielen lassen. Das Ergebnis der Generierung ist bei der aktuellen Implementierung unabhängig von der Reihenfolge der Traversierung.

4 Fazit

Die in dieser Arbeit vorgestellten Konzepte und die daraus resultierende Software wurden in das vorhandene STU-Framework integriert und vereinfachen das Testen von Datenbank-basierten Anwendungen erheblich. Die entwickelte Testdaten-DSL wurde bespielsweise in

der Qualitätssicherung von mehreren produktiven Softwareprojekten eingesetzt. Der Spezifikationsaufwand und die Fehlerrate konnte im Vergleich zur früheren Vorgehensweise dadurch deutlich reduziert werden. Der Testdatengenerator wurde erfolgreich auf mehrere Datenbankschema (mit teilweise mehr als 80 Tabellen) angewandt. Unser Verfahren zur Testdatengenerierung konnte (innerhalb von Sekunden) in jedem Fall einen konsistenten, übersichtlichen Testdaten-Set erzeugen. Die erzeugten Testdaten-Sets boten eine sehr gute Startbasis für die Anwendungstests.

Der in dem Projekt entwickelte Code steht unter der Apache License 2.0 zur Verfügung unter unter `https://github.com/Seitenbau/stu/`. Details zur Implementierung und ausführliche Code-Beispiele zur Verwendung sind in [Mol13] zu finden.

Literatur

[Mes07] Gerard Meszaros. *XUnit Test Patterns: Refactoring Test Code*. The Addison-Wesley Signature Series. Addison-Wesley, 2007.

[Mol13] Nikolaus Moll. Testdaten-Modellierung und -Generierung für Datenbank-basierte Anwendungen. Diplomarbeit, HTWG Konstanz / SEITENBAU GmbH, October 2013. `http://xxx.xxx.xx/`.

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