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

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

Abstract:

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

†Bis 31.12.2012 Student im Master-Studiengang Informatik / akademischer Mitarbeiter der HTWG Konstanz; seit 15.01.2014 tätig bei der Pentasys AG in München.

1 Einleitung und Motivation

Aus FORUM Artikel

Zum Testen von Software-Anwendung, die Daten persistent in einem Datenbanksystem verwalten, werden Testdaten für die Datenbank benötigt. Für Anwendungen mit einem komplexen Datenbankschema ist die Spezifikation dieser Testdaten meist nicht trivial, da neben den Entitäten auch deren Beziehungen betrachtet werden müssen. Diese Beziehungen unterliegen in der Regel einer Menge komplexer fachlicher Regeln, die sich aus dem Domänen-Modell der Anwendung ergeben.

In dieser Arbeit wird untersucht, wie die zum Test von Datenbank-basierten Anwendungen notwendigen Testdaten auf einfache Weise beschrieben und automatisch erzeugt werden können.

Eine zu prüfende Anwendung wird im Kontext von Software-Tests als System under Test (SUT) bezeichnet. Alle Voraussetzungen für einen Test werden dabei als sog. Test Fixture bezeichnet. Idealerweise soll die in einem Test Fixture beinhaltete Datenmenge für einen funktionalen Test eine hohe Testabdeckung bieten, dabei aber so klein und übersichtlich wie möglich sein. Somit sind die Testdaten einfacher zu verstehen und für eine große Anzahl von funktionalen Tests zu verwenden.

Ein üblicher Ansatz, ein SUT in Verbindung mit einer Datenbank zu testen, ist das Testmuster Back Door Manipulation [Mes07]. Idee hierbei ist, dass das Einspielen des Test Fixture in die Datenbank direkt durch den Test am SUT vorbei, d.h. sozusagen durch eine Hintertür, geschieht. Im ersten Schritt, dem Setup, wird die Datenbank über direkten Zugriff an dem SUT vorbei in den Anfangszustand des Test Fixture gebracht. Anschließend können im Exercise-Schritt die zu testenden Operationen am SUT durchgeführt werden. Die Überprüfung, ob sich das SUT richtig verhalten hat, findet im Verify-Schritt statt. Dabei kann der Zustand der Datenbank mit dem erwarteten Zustand des SUT verglichen, ebenfalls über die Test-eigene Datenbankverbindung. Abschließend können im vierten optionalen Schritt, Teardown, noch Aufräumarbeiten in der Datenbank implementiert sein.

Basis für die nachfolgend beschriebenen Projektarbeiten ist 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 von der Firma Seitenbau entwickelt. Für Tests von Datenbank-basierten Anwendungen setzt STU auf der Bibliothek DbUnit [2] auf. DbUnit ist ein Framework zum Testen von Datenbank-basierten Java-Anwendungen.

Ziel des Projekts ist es u.a., funktionale Tests durch die Spezifikation eines Java-API und einer speziellen Testdatenbeschreibungssprache so zu vereinfachen, dass eine Datenbank einfach zu Beginn des Tests – über den Test selbst – in den wohldefinierten Anfangszustand das Test Fixture versetzt werden kann. Des Weiteren soll der erwartete Datenbank-Zustand am Testende einfach auf Basis des Test Fixture beschrieben werden können. Nach Ausführung des Tests soll automatisch der über das SUT erzeugte Datenbank-Zustand mit dem erwarteten Datenbank-Zustand verglichen und somit die Korrektheit der Anwendung geprüft werden können.

2 Die Modellierungssprache

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 Test-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 [Spo]. 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. ...). 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.

```
class HochschuleDataSet extends HochschuleBuilder
{
  def tables() {
    professorTable.rows {
      REF    | name      | vorname   | titel            | fakultaet
      WAESCH | "Waesch"  | "Juergen" | "Prof. Dr.-Ing." | "Informatik"
      HAASE  | "Haase"   | "Oliver"  | "Prof. Dr."      | "Informatik"
    }
    lehrveranstaltungTable.rows {
      REF        | name                | sws | ects
      VSYS       | "Verteilte Systeme" | 4   | 5
      DPATTERNS  | "Design Patterns"   | 4   | 3
    }
    pruefungTable.rows {
      REF          | typ   | zeitpunkt
      P_VSYS       | "K90" | DateUtil.getDate(2013, 4, 1, 14, 0, 0)
      P_DPATTERNS  | "M30" | DateUtil.getDate(2013, 1, 6, 12, 0, 0)
    }
    studentTable.rows {
      REF         | matrikelnummer | name         | vorname    | studiengang
      MUSTERMANN  | 123456         | "Mustermann" | "Max"      | "BIT"
      MOLL        | 287336         | "Moll"       | "Nikolaus" | "MSI"
    }
  }

  def relations() {
    WAESCH.beaufsichtigt(P_VSYS)
    HAASE.leitet(VSYS, DPATTERNS)
    HAASE.beaufsichtigt(P_DPATTERNS)
    P_VSYS.stoffVon(VSYS)
    DPATTERNS.hatPruefung(P_DPATTERNS)
    MOLL.schreibt(P_VSYS)
    MOLL.besucht(VSYS)
    VSYS.hatTutor(MOLL)
    MUSTERMANN.besucht(DPATTERNS)
  }

}
```

Listing 1: DataSet modelliert mit dem Table Builder API

Die Implementierung der Testdaten-DSL basiert dabei auf Groovy [4], 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. ...

Definition eines Datenbank-Schemas, Generierung der DSL sowie Unit-test Beispiele etc.: siehe Master Thesis.

3 Generierung von Testdaten

Aus FORUM Artikel

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. 3).

Bei der Testdatengenerierung sind u.a. folgende Anforderungen zu berücksichtigen:

- Die Testdaten müssen dem Datenbankschema der Anwendung entsprechen, um diese Daten überhaupt in die Datenbank einspielen zu können.
- Die Testdaten sollen einerseits genügend Daten enthalten, um die fachliche Struktur der Anwendungsdaten zu erhalten; andererseits sollen möglichst wenige Daten erzeugt werden, damit die eingespielten Daten übersichtlich und für die Software-Tester verständlich bleiben.
- Für die Durchführung und Wartung von Tests ist es von Vorteil, wenn möglichst viele Tests dieselben Testdaten verwenden können, d.h. wenn die generierten Testdaten eine möglichst große Testabdeckung erzielen. Somit sind insgesamt weniger Testdatensätze zu verwalten, was z.B. bei Änderung des zugrundeliegenden Datenbankschemas von Vorteil ist und die Übersichtlichkeit erhöht.
- Die Beschreibung des Datenbankschemas, aus dem die Testdaten generiert werden, muss so ausdrucksmächtig sein, dass auch Sachverhalte beschrieben werden können, die über die strukturelle Abbildung, z.B. auf Datenbank-Tabellen, hinausgehen, damit die Testdaten auch den weitergehenden Anforderungen des Systems under Test genügen und realistische Szenarien abbilden. Beispielsweise kann eine Anwendung eine 1¡-¿1..* Beziehung definieren; in einem relationalen Datenbanksystem ist jedoch nur eine 1¡-¿0..* Beziehung durch Fremdschlüssel- und Not-Null-Constraints strukturell abbildbar.
- Die generierten Daten sollen in der entwickelten DSL beschrieben werden, damit diese direkt im STU Test-Framework nutzbar sind.

Interessanterweise stellte sich bei einer umfassenden Literaturanalyse und Analyse existierender Testdatengeneratoren heraus, dass bisher keine passende Lösung für die beschriebene Problemstellung 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. Anleihen konnten dabei aus [5] gezogen werden. 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. ”transitiveÄbhängigkeiten über mehrere Beziehungstypen hinweg, die zu einer kombinatorischen Explosion führen können, bleiben dabei (augenblicklich) unberücksichtigt. Weitere Details zu dem im Projekt entwickelten Verfahren zur Testdatengenerierung (u.a. Beschreibung des Algorithmus in Pseudo-Code) sind [6] zu entnehmen.

Outline

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.

3.1 Beschreibung des Algorithmus

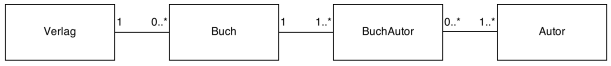

Abbildung 1: Datenbank-Diagramm

3.2 Implementierung und Evaluation

4 Fazit

Note: include DSL Fazit

Der entwickelte Testdatengenerator wurde auf ein Datenbankschema eines produktiven Anwendungssystems mit 12 Datenbanktabellen und einigen fachlichen Einschränkungen angewendet. Durch unser Verfahren zur Testdatengenerierung wurde ein konsistenter, übersichtlicher Satz an Testdaten erzeugt, der eine gute Startbasis für Anwendungstest bietet.

Der Code des DSL-Interpreters und des Testdatengenerators ist verfügbar unter https://github.com/Seitenbau/stu/.

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.

[Spo] Spock. *Spock*. Spock. `docs.spockframework.org/`.