Modellbasierte Testdatenspezifikation und -generierung mittels Äquivalenzklassen und SQL

Mario Friske[1], Dierk Ehmke[2]

[1]Friske Consulting, Berlin
[2]periplus instruments, Darmstadt

Zusammenfassung. Für automatisierte Softwaretests werden komplexe Testdaten benötigt. Deren Erstellung ist aufwendig, deshalb werden Verfahren zur effizienten Testdatenspezifikation und -generierung benötigt. In diesem Beitrag stellen wir zwei entsprechende Ansätze vor und diskutieren, wie diese gewinnbringend kombiniert werden können.

Zunächst stellen wir ein modellbasiertes Vorgehen vor, welches auf interaktiver Zuordnung von Äquivalenzklassen und typischen Repräsentanten basiert. Anschließend präsentieren wir einen zweiten Ansatz, der auf Modellen in SQL-Notation aufsetzt. Aus diesen Modellen werden mit Angaben aus abstrakten Testfällen konkrete automatisierte Testfälle inklusive Testdaten und -orakel erzeugt.

Anhand eines Praxisbeispiels werden Vor- und Nachteile der beiden Vorgehensweisen sowie Kombinationsmöglichkeiten erläutert.

1 Hintergrund

Die Erzeugung von Testdaten für den Systemtest ist in vielen Fällen nicht trivial. Typische Anforderungen an Testdaten sind, dass diese repräsentativ und kompatibel mit bestehenden Testumgebungen sind.

Bei der Testfallerzeugung wird zwischen logischen und konkreten Testfällen unterschieden [1].

- Ein logischer oder auch abstrakter Testfall ist nach [2] ein Testfall ohne konkrete Ein- und Ausgabewerte für Eingabedaten und vorausgesagte Ergebnisse. Er verwendet logische Operanden, weil die konkreten noch nicht definiert oder verfügbar sind. Oft werden logische Testfälle auf Äquivalenzklassenebene spezifiziert, d.h., es werden Mengen potenzieller Testdaten mithilfe charakteristischer Eigenschaften beschrieben.

- Demgegenüber enthält ein konkreter Testfall tatsächliche Werte für Eingaben und vorausgesagte Ergebnisse. Logische Operanden der abstrakten Testfälle werden darin durch konkrete Werte ersetzt, d. h. die Äquivalenzklassen der logischen Testfälle werden in den konkreten Testfällen mit Repräsentanten besetzt.

Auch konkrete Testfälle sind nicht notwendigerweise direkt ausführbar, da sie abstrakte Schnittstellen und Operationen enthalten. In ausführbaren Testfällen sind diese durch konkrete Schnittstellenbezeichnungen und durch Anweisungen einer Programmier- oder Skriptsprache ersetzt.

2 Problembeschreibung

Mit obigem Hintergrund tritt in der Praxis häufig folgendes Problem auf: Es sollen unter Berücksichtigung vorgegebener Abdeckungskriterien systematisch logische Testfälle erzeugt werden, entweder manuell oder automatisiert. Die abstrakten Testfälle sind mit typischen Repräsentanten zu besetzen.

Wir sehen zwei Gründe, warum dieser Schritt schwierig sein kann:

1. Der konkrete Repräsentant lässt sich nicht unmittelbar aus der Äquivalenzklassenspezifikation herleiten. Es sind weitere Hilfsmittel nötig, um den Datensatz zu ermitteln, oder er muss dem Tester bereits bekannt sein.
2. Es ist entweder nicht möglich oder zu aufwändig, den erforderlichen Datensatz dynamisch im Test zu erzeugen. Beispielsweise können erforderliche Verarbeitungszeiten das Anlegen während des Tests unpraktikabel machen.

Oft wird die Repräsentantenauswahl daher durch einen Tester übernommen, der mithilfe seines Wissens über die Domäne und Testumgebung die Daten bestimmt: Im ersten Fall legt er ein Testdatensatz mit bekannten Daten an. Im zweiten Fall wählt der Tester einen passenden Datensatz aus und passt ihn ggf. an.

Es stellt sich die Frage, wie Testdaten effizient spezifiziert und generiert werden können.

3 Fallbeispiel IBAN-Erzeugung

Am Beispiel der International Bank Account Number (IBAN) wollen wir zwei sich ergänzende Ansätze zur Testdatenspezifikation und -generierung erläutern.

Abbildung 1 zeigt den Aufbau von IBANs für den Wirtschaftsraum DACH, d.h. Deutschland, Österreich und die Schweiz. IBANs bestehen aus bis zu 34 alphanumerischen Zeichen und setzen sich aus einem 2-stelligen Ländercode, einer 2-stelligen Prüfsumme und einer maximal 30-stelligen Kontoidentifikation, der Basic Bank Account Number (BBAN) zusammen. Die Länge und der konkrete Aufbau der BBAN ist länderspezifisch.

Austria	ATkk bbbb bccc cccc cccc
Germany	DEkk bbbb bbbb cccc cccc cc
Switzerland	CHkk bbbb bccc cccc cccc c

b = National Bank code c = Account number

Abbildung 1: Aufbau von IBANs im Raum DACH

BBANs sind in Deutschland nach dem Format 8n+10n, in Österreich nach dem Format 5n+11n und in der Schweiz nach dem Format 5n+12c aufgebaut, wobei n für numerische und c für alphanumerische Zeichen steht.

4 Modellbasierte Spezifikation von Testdaten

In dem ersten hier vorgestellten Ansatz werden Testdaten mithilfe von Metamodellen repräsentiert.

Dazu verwenden wir modellbasierte Varianten bestehender Techniken: Äquivalenzklassen werden mithilfe einer Variante der Klassifikationsbäume [3] und Repräsentanten mithilfe attributierter Klassifikationsbäume [4] repräsentiert. In beiden Fällen definieren wir die Datenstrukturen mittels Metamodellen, so dass eine leichte Weiterverarbeitung der Testdaten mit den Methoden und Werkzeugen der Model-Driven Architecture (MDA) erfolgen kann.

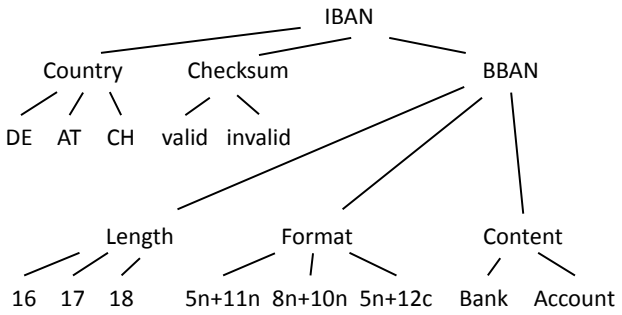

Abbildung 2: Klassifikationsbaum für DACH-IBANs

Ein Klassifikationsbaum für IBANs im Raum DACH ist in Abbildung 2 dargestellt. Darin werden sowohl testrelevante Aspekte (Klassifikationen), z.B. Ländercode, als auch mögliche Ausprägungen (Klassen), z.B. „DE“, dargestellt.

Äquivalenzklassenspezifikation. In einem Überweisungssystem sollen länderspezifische Workflows getestet werden. Die Eingabemasken werden dynamisch auf Basis der eingegebenen IBAN spezifisch an das Land des Empfängers angepasst.

Zur Testfallermittlung verwenden wir die in [5] dargestellte UCV-Methode. Dabei werden die Use Cases mit einem Modell unterlegt und einzelne Pfade durch das Modell mit Äquivalenzpartitionen aus dem separat aufzubauenden Klassifikationsbaum verknüpft.

Für einen bestimmten Workflow, d.h. Pfad durch das Modell, lässt sich so schnell eine benötigte Äquivalenzklassenkombination ermitteln. In unserem Beispiel wäre das beispielsweise eine Schweizer IBAN. Konkrete Repräsentanten werden an dieser Stelle noch nicht ermittelt - das erfolgt erst im folgenden Schritt.

Repräsentantenbestimmung. Unter Berücksichtigung der eingangs erwähnten Randbedingungen sollen nun zu jeder Äquivalenzklasse zugehörige Repräsentanten bestimmt werden.

Wir unterscheiden drei verschiedene Verfahren, die prinzipiell dazu einsetzbar sind:

1. Interaktive Eingabe durch den Tester,
2. Auswahl aus einem existierenden Datenbestand,
3. regelbasierte Generierung.

Die interaktiven Eingabe durch den Tester ist das elementarste Verfahren und kann jederzeit angewendet werden, insbesondere auch dann, wenn die komplexeren Verfahren zu keinem Ergebnis führen.

Bei der zweiten Möglichkeit sehen wir zwei Vorgehensweisen als Erfolg versprechend: Zum einen ist es prinzipiell möglich, aus formalisierten Äquivalenzklassenbeschreibungen SQL-Kommandos zu generieren, die genau die gewünschte Menge aller Repräsentanten aus dem Datenbestand selektieren. Zum anderen kann unter Nutzung von Metadaten passende Repräsentanten aus einer existierenden Testdatenbasis interaktiv ausgewählt werden.

Bei dem Vorgehen nach [5] wird in allen vorangegangenen Fällen das Modell mit dem entsprechenden Datensatz verknüpft. Die Daten sind jedoch oft partiell und müssen ggf. noch ergänzt werden.

Die dritte Möglichkeit zur Repräsentantenbestimmung ist die regelbasierte Generierung, die im Folgenden im Detail betrachtet wird.

5 SQL-basierte Testdatengenerierung

Der zweite hier vorgestellte Ansatz ist in der Lage den Output des ersten weiter zu verarbeiten. Dieser Ansatz arbeitet mit in SQL beschriebenen Modellen.

SQL wurde für die Beschreibung komplexer Datenflüsse entworfen und ist dadurch hervorragend als Modellierungssprache für Testdatengenerierung geeignet. Es ist weit verbreitet, sowohl unter Entwicklern, als auch bei Testdatenspezialisten und bei technischen Testern. SQL ist deklarativ und bietet ein gutes Abstraktionsniveau.

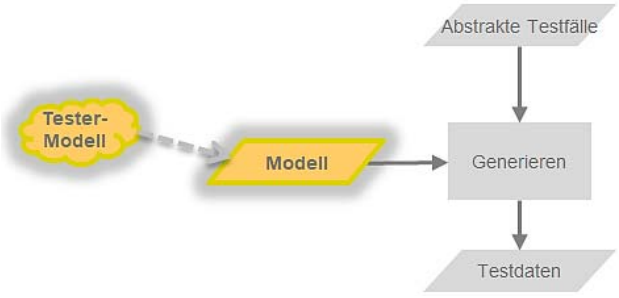

Abbildung 3: Grober Aufbau der modellbasierten Testdatengenerierung

Der Testdatenmodellierer formuliert sein mentales Modell über das System under Test (SUT) in SQL, wie in Abbildung 3 dargestellt. Testdatenbeschreibungen, die nur die Testaspekte berücksichtigen, werden

```
CREATE TABLE ACCOU(
 IBAN       CHAR(34),
 BANK_CODE   DECIMAL(20,0),
 ACCOUN      DECIMAL(20,0),
 COUNTRY     CHAR(100),
 CONSTRAINT check_iban
   CHECK( COUNTRY IN('GERMANY','AUSTRIA','SWITZERLAND') )
);

CREATE TABLE ACCOU2(
 IBAN       CHAR(34),
 BANK_CODE   DECIMAL(20,0),
 ACCOUN      DECIMAL(20,0),
 COUNTRY     CHAR(100),
 CHECKNO     DECIMAL(5,0),
 COUNTRYMARK  CHAR(4),
 COUNTRY_NO   DECIMAL(6,0),
 ACC_FACTOR   DECIMAL(20,0),
 BC_FACTOR   DECIMAL(20,0),
 NP_FORMAT    CHAR(34),
 NUMERIC_PART CHAR(34) not null,
 CONSTRAINT check_iban2
 CHECK(
     ( BANK_CODE >= 1 )
  AND ( ACCOUN >= 1 )
  AND ( CHECKNO >= 0 )
  AND ( COUNTRYMARK IN('AT','CH','DE') )
  AND ( COUNTRY_NO IN(1314,1029,1217) )
  AND ( ACC_FACTOR IN(10000000000,100000000000,1000000000000) )
  AND ( BC_FACTOR IN(100000000,100000) )
  AND ( NP_FORMAT IN('00000000000000000000',
              '00000000000000000',
              '000000000000000000')
    )
  AND ( BANK_CODE <= BC_FACTOR - 1 )
  AND ( ACCOUN <= ACC_FACTOR - 1 )
  AND ( CHECKNO = 98 - mod ( ( BANK_CODE * ACC_FACTOR * 1000000 +
      ACCOUN * 1000000 + COUNTRY_NO * 100 ), 97 ) )
  AND ( CHECKNO * BC_FACTOR * ACC_FACTOR + BANK_CODE * ACC_FACTOR +
      ACCOUN = TO_NUMBER( NUMERIC_PART, NP_FORMAT ) )
  AND ( IBAN = CONCAT( COUNTRYMARK, NUMERIC_PART ) )
  AND (
     (     ( COUNTRYMARK = 'DE' )
       AND ( COUNTRY = 'GERMANY' )
       AND ( COUNTRY_NO = 1314 )
       AND ( ACC_FACTOR = 10000000000 )
       AND ( BC_FACTOR = 100000000 )
       AND ( NP_FORMAT = '00000000000000000000' )
     )
    XOR
     (
         ( COUNTRYMARK = 'AT' )
       AND ( COUNTRY = 'AUSTRIA' )
       AND ( COUNTRY_NO = 1029 )
       AND ( ACC_FACTOR = 100000000000 )
       AND ( BC_FACTOR = 100000 )
       AND ( NP_FORMAT = '000000000000000000' )
     )
    XOR
     (
      ...
     )
    )
  )
);

INSERT INTO ACCOU2 SELECT IBAN, BANK_CODE, ACCOUN, COUNTRY FROM ACCOU A;
```

Abbildung 4: SQL-Modell für die IBAN-Berechnung

mit unserem Werkzeug periplus autogen zusammen mit dem Modell zu vollständigen Testdatensätzen generiert. Konkret wird dies anhand des IBAN Beispiels erläutert: Ziel ist die Generierung von IBANs, die im Test verwendet werden können.

Abbildung 4 zeigt das SQL-Modell für die IBAN-Berechnung.

Tabelle ACCOUN soll die zu generierenden Testdaten aufnehmen und enthält folgende Angaben zu einem Konto: IBAN, BLZ, Kontonummer und Name des Landes, zu dem die Bank gehört. Tabelle ACCOUN2 ist eine Hilfstabelle, die diese und weitere Hilfsspalten enthält. Ein INSERT-Statement verbindet beide Tabellen. Die Berechnungen werden in Tabellen-Constraints, in der Select-Liste und der WHERE-Klausel des INSERT-Statements formuliert. Schließlich sind das Ausgabeformat für ACCOUN und die Testdatenspezifikationen festzulegen.

Der SQL-Code enthält

- die Definition der Tabellen mit
- der Vereinbarung der Spalten und ihren Datentypen
- den Constraints, die deren Eigenschaften beschreiben, und
- drei Constraints, die die Berechnung der Prüfnummer, des numerischen Teils der IBAN und dem Zusammenbau des Länderkennzeichens und dieses numerischen Teils enthalten.

Die Konstanten für die Schweiz sind aus Platzgründen nicht aufgeführt.

Die Beispiele in Abbildung 5 erläutern die Funktionsweise des Ansatzes:

- In Zeile 1 wird aus BLZ, Kontonummer und Land eine IBAN errechnet.
- In Zeile 2 wird lediglich die BLZ angegeben, sie passt ausschließlich zu deutschen IBANs. Dazu passend werden die anderen Angaben berechnet.
- Zeilen 3 bis 6 demonstrieren das für die übrigen Spalten.

In der letzten Zeile schließlich werden keine Daten angegeben. Auch dafür werden alle Daten ergänzt und eine für den Test akzeptable IBAN geliefert. Dieser Fall demonstriert, dass man in Testfällen viele Details weglassen kann, die für die Testidee eines abstrakten Testfalls nicht wesentlich sind – sie werden generiert. Das spart viel Arbeit und auch Wartungsaufwand, da im Fall, dass Spalten wegfallen oder Constraints für sie geändert werden, man nur dort anpassen muss, wo Angaben gemacht werden.

6 Diskussion

Modellbasierter Test. Zunächst war unser Ziel, modellgetrieben Testdaten zu generieren. Betrachtet man Testfälle und das Orakel ebenfalls als Testdaten, dann leistet dieser Ansatz auch modellgetriebenes Testen. Aus dem Modell lassen sich auch Testfälle generieren. Unter anderem generiert autogen Parameter und deren Werte für die Pairwise-Methode. Berücksichtigt werden dabei nur die Spalten, für die nur eine begrenzte Anzahl diskreter Werte gültig sind. Diese

No	Comment	BANK_CODE	ACCOUN	COUNTRY	IBAN
1	BANK_CODE, ACCOUN and COUNTRY preset	64090	12345678901	AUSTRIA	AT896409012345678901
2	only BANK_CODE preset, fits only to DE	64090100	1	GERMANY	DE886409010000000001
3	only ACCOUN preset, fits to DE	1	1234567890	GERMANY	DE170000000011234567890
4	only ACCOUN preset, fits to CH	1	123456789012	SWITZERLAND	CH4300001123456789012
5	only COUNTRY preset	1	1	SWITZERLAND	CH1700001000000000001
6	only IBAN preset	20050550	1015871393	GERMANY	DE02200505501015871393
7	smoke test, nothing preset	1	1	SWITZERLAND	CH1700001000000000001

preset generated

Abbildung 5: Beispieltabelle für partielle Testdatendefinitionen und Generierung

werden mit einem freien Pairwise-Werkzeug PICT [6] kombiniert.

Modellgetriebener Test unterstützt nicht nur bei der Testfallerstellung, sondern auch bei der Testautomatisierung. Für das SUT werden alle notwendigen Daten und Dokumente erzeugt. Eine Testautomatisierung vorausgesetzt, können die Testfälle unmittelbar ausgeführt werden. Idealerweise werden sämtliche nachfolgende Diagnoseschritte ebenfalls automatisiert, sodass sich Durchlauf- und Analysezeiten verkürzen.

Erfahrungen im agilen Umfeld. Nach unserer Erfahrung erfordert die Ermittlung der fachlichen Anforderungen 50% des Aufwands für Testdatenmanagement. Die Formulierung als Modell erfordert nur einen Bruchteil dessen.

Probleme bei der Generierung der Testfälle zeigen Fehler im Modell auf, so gibt es schnelles Feedback, ob die Fachlichkeit richtig verstanden wurde.

In agilen Projekten ist die Fachlichkeit in der Regel auf viele unterschiedlich aktuelle Tickets verteilt. Ein aktuelles Modell wird nicht nur für Tester zur wichtigen Referenz. Darüber hinaus ändern sich Testbasis, Architektur, Schnittstellen und Testware öfter als in nicht agilen Projekten. Folglich sind Modell, Testfälle, Testdaten und Ausgabeformate häufig anzupassen.

Ein in SQL formuliertes Modell ist nach unseren Erfahrungen gut wartbar. Partielle Testdatendefinitionen und eine saubere Trennung der Ausgabeschnittstelle sind weitere Gründe, warum Wartung von autogen sehr gut unterstützt wird.

Mockregeln, fixierte Testdaten, Testaspekte und Generierung von Suchmasken werden problemlos in autogen formuliert. Damit wird eine durchgängige Testautomatisierung erreicht, angefangen von der Testfallerzeugung über die Testdatengenerierung bis zur automatischen Ausführung.

7 Fazit und Ausblick

Der erste Teil des skizzierten kombinierten Verfahrens ermöglicht es, Testdatenspezifikationen zu erzeugen, indem Äquivalenzklassenbeschreibungen definiert und benötigte Repräsentanten spezifiziert werden.

Im zweiten Teil werden benötigte Äquivalenzklassenrepräsentanten erzeugt, indem die partiellen Testdatenspezifikationen genutzt werden, um mittels SQL komplette Datensätze zu generieren.

Die gewinnbringende Kombination der beiden Verfahren sehen wir als eine gute Möglichkeit, innovative modellbasierte Testgenerierungsmethoden in bestehenden industriellen Kontexten mit jeweils vorhandenen Werkzeugketten und Datenbeständen einzusetzen. Wir hoffen, dass wir einen Beitrag dazu leisten können, systematisches und insbesondere modellbasiertes Testen weiter zu verbreiten.

Literatur

[1] Andreas Spillner und Tilo Linz. *Basiswissen Softwaretest*. Dpunkt Verlag, 2005.

[2] German Testing Board e.V. ISTQB/GTB Standardglossar der Testbegriffe, Version 2.3, 2014.

[3] Matthias Grochtmann und Klaus Grimm. Classification trees for partition testing. *Software Testing, Verification and Reliability*, 3(2):63–82, 1993.

[4] Stefan Lützkendorf und Klaus Bothe. Attributierte Klassifikationsbäume zur Testdatenbestimmung. *Softwaretechnik-Trends*, 23(1), 2003.

[5] Mario Friske. Anwendungsfallbasierte Testfallerstellung mit der UCV-Methode. *Softwaretechnik-Trends*, 35(1), 2015.

[6] PICT. https://github.com/microsoft/pict.