Generierte Unit-Tests zur Absicherung automatisiert migrierter Validierungs-Regeln

Kai-Uwe Herrmann (kai-uwe.herrmann@bison-group.com)
Bison Schweiz AG; Allee 1A; CH-6210 Sursee

Abstract

Bei der Modernisierung unserer JavaEE-basierten ERP-Software Bison Process *wird Geschäftslogik aus der derzeitigen "Big-Ball-of-Mud"-Architektur in eine an fachlichen Grenzen geschnittene Komponenten-Architektur überführt. Die Code-Einzelteile bleiben dabei erhalten, sie werden nur neu strukturiert. Für die Migration des Codes verfolgen wir einen Teilautomatisierungs-Ansatz (siehe [2], [3]). Tests stellen sicher, dass die migrierte Logik gleich bleibt. Mangelnde existierende Testabdeckung erfordert die Neuerstellung solcher Tests. Wir beschäftigten uns mit der Frage, ob es möglich sei, den Legacy-Code auch als Basis für die Generierung von Unit-Tests für die neuen Klassen zu nutzen.*

1 Ausgangslage

In [2] und [3] präsentierten wir, wie wir für einen Teil der Geschäftslogik unseres ERP-Systems *Bison Process* ein Architektur-Reengineering durchführen. Wir zeigten Quell- und Ziel-Design und stellten vor, wie wir mit Hilfe von OMAN[1] den Überführungsprozess teilautomatisieren konnten. Entwickler sparen sich einen grossen Teil manueller Arbeit. Wir entschieden daher, den Ansatz auszuweiten und weitere Automatisierungsschritte zu gehen.

2 Teststrategie

Definitionsgemäss wird beim Reengineering die Softwarequalität verändert, während die Funktionalität meist gleich bleibt. So ist es auch bei unseren Reengineering-Projekten. Ziel ist es, die Wart- und Erweiterbarkeit zu verbessern, die Konfiguriertbarkeit einzuschränken, dabei jedoch die Funktionalität zu erhalten. Jede solche Reengineering-Maßnahme wird deshalb durch entsprechende funktionale Tests abgesichert.

Für Use-Cases, die Geschäftsobjekte verändern (siehe Abbildung 1), eignet sich ein integratives Testen des gesamten Prozesses (zum Beispiel das Stornieren einer Bestellung). Wir definieren Test-Cases für jeden Use-Case, führen den Testfall mit der Legacy-Logik aus, speichern Anfangs- und End-Zustand des Geschäftsobjektes und prüfen den Testfall später mit der modernisierten Logik gegen die gespeicherten Ergebnisse.

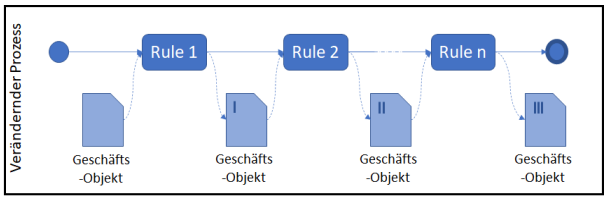

Abbildung 1: *Verändernder Prozess. Prozess verändert das Geschäftsobjekt mit jedem Schritt*

Die Validierungslogik, mit deren Migration wir uns derzeit beschäftigen, ist für diesen Testansatz nicht geeignet. Der Validierungsprozess ist in Abbildung 2 dargestellt: Er hat viele Ausgänge, die relevant und zu testen sind. Der Gesamtablauf kann nicht mit einem einzelnen Geschäftsobjekt geprüft werden. Für eine umfassende Testabdeckung muss vielmehr ein Geschäftsobjekt für jede Regel konstruiert werden, um insbesondere den NOK-Ausgang jeder Regel zu testen. Je später die Regel im Gesamtprozess liegt, desto aufwändiger wird die Konstruktion des validierten Geschäftsobjektes, da die davor liegenden Regeln ohne Fehler durchlaufen werden müssen.

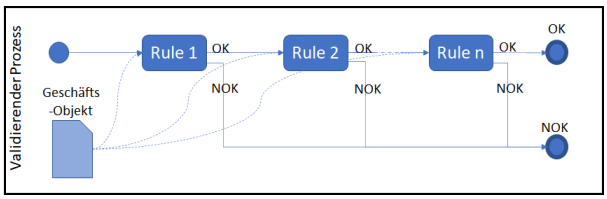

Abbildung 2: *Validierender Prozess. Geschäftsobjekt geht unverändert in jede Validierungsregel. Regel entscheidet ob OK oder Fehler (NOK). NOK führt zum sofortigen Abbruch.*

Deshalb streben wir für diesen Prozess einen Unit-Test-basierten Ansatz an, der den Logikerhalt jeder einzelnen Regel prüft. Da wir jedoch die Struktur des Codes ändern, also die Validierungsregeln vor der Migration gänzlich anders in Klassen verteilt sind, als im Ziel-Design, ist die Herausforderung dieses Vorgehens, tatsächlich eine Vorher-/Nachherprüfung zu erreichen. Wir kommen im Abschnitt 4 darauf zurück.

[1] OMAN: Analyse- und Generierungswerkzeug; Object Engineering GmbH; https://www.objeng.ch

3 Ziel-Design der Validatoren

Die Validierungslogik zerlegen wir in kleine Einheiten, die Validatoren. Ähnliche Validierungs-Aspekte werden gruppiert und gemeinsam zu einem Validator zusammengefasst. Schliesslich führt eine Engine alle Validatoren nacheinander aus.

4 Unit-Tests erstellen

Für jeden Validator soll ein Unit-Test erzeugt werden, der zwei Aufgaben erfüllt: 1. die Absicherung der Migration des Validierungs-Codes (vorher vs. nachher); 2. die spätere, ständige Regressions-Prüfung der Validierungslogik.

Ein solcher Unit-Test kann wegen des unterschiedlichen Klassen-Schnitts nicht gegen Legacy-Klassen und modernisierte Klassen gleichermaßen funktionieren. Die Unit-Tests erstellen wir deshalb für die neuen Validatoren. Damit sie genau die Logik abprüfen, die in den Validator migriert wird, erstellen wir sowohl Validator als auch Unit-Test aus dem gleichen Legacy-Code-Input (siehe Abbildung 3).

Die zwei Generatoren ermöglichen es uns, die Unit-Test-Klassen inklusive aller Test-Case-Varianten zu generieren, einschliesslich des Testccodes zur Erfüllung/Nicht-Erfüllung der Validierungsbedingungen, soweit dies nach Kosten/Nutzen-Aspekten sinnvoll ist. Wo nicht rentabel, werden dennoch die Setter-Statements generiert, die der Entwickler dann manuell um konkrete Werte ergänzt.

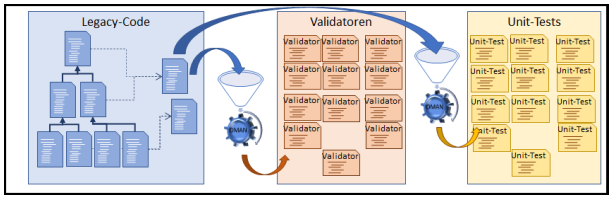

Abbildung 3: *Aus dem Legacy-Code werden sowohl Validatoren als auch Unit-Tests generiert.*

5 Unit-Tests konstruieren

Jeder Unit-Test prüft genau die Logik, die der zugehörige Validator realisiert. Außerhalb befindliche Logik wird simuliert. Das erfordert zum einen *Design for testability*, zum anderen benötigen wir Stubbing bzw. Mocking, um die abhängigen Logik-Teile zu simulieren. Wir nutzen Mockito für das Mocking abhängiger Logik und ein bereits existierendes eigenes Stubbing-Framework für die Geschäftsobjekt-Graphen in Bison Process.

Die Bedingungen der Validierungs-Regeln zerlegen wir in ihre Teilbedingungen. Jede sinnvolle Kombination von Wahrheitswerten, die die Teilbedingungen annehmen können, um die Gesamtbedingung zu erfüllen bzw. nicht zu erfüllen, führt zu einem Test-Case für diese Validierungs-Regel und für jeden Test-Case müssen die Werte im Test so gesetzt werden, dass die Teilbedingungen beim Durchlaufen den gewünschten Ziel-Wahrheitswert annehmen.

Für die Test-Cases, in denen die Gesamtbedingung *wahr* ist, erwarten und überprüfen wir im Test den von der Validierungs-Regel erzeugten Fehler. Im *falsch*-Fall prüfen wir dagegen, auch wirklich keinen Fehler zu erhalten.

Beispiel

```
if ( o.getA().isSpecial()
     && ! o.getA().hasChanged()
     && c.isGuiContext()
   ) {
   throw new BusinessLogicException(ERR);
}
```

In diesem einfachen Beispiel sind folgende Wahrheitswert-Kombinationen sinnvoll und führen zu Test-Cases.

Gesamtbed.	a.special	a.changed	c.gui
true	true	false	true
false	true	false	false
false	true	true	*don't care*
false	false	*don't care*	*don't care*

Der Unit-Test für den ersten dieser Test-Cases könnte also z.B. folgende Statements enthalten, um die Teilbedingungen entsprechend zu erfüllen.

```
obj.getA().setSpecial();
ctx.setGuiContext(true);
try {
   validator.execute(obj, ctx);
   fail("Exception expected");
} catch (BusinessLogicExcepton e) {
   assertEquals(ERR, e.getError());
}
```

6 Fazit

Mit dem generativen Ansatz können wir nicht nur Geschäftslogik migrieren, sondern auch Tests teilautomatisiert erstellen. Wir schätzen mit der Investition in den Generator mehr als die Hälfte des Aufwandes für die Erstellung der Unit-Tests einzusparen. Es geht dabei insgesamt um ca. 1000 Unit-Test-Klassen.

Literatur

[1] Andres Koch, Remo Koch, "Metadaten basiertes, teilautomatisiertes Software-Reengineering", Proceedings zum 20. Workshop Software-Reengineering und -Evolution, GI-Fachgruppe Software-Reengineering, 2018

[2] Kai-Uwe Herrmann, "Teilautomatisiertes Architektur-Reengineering in einem JavaEE Monolithen", Proceedings zum 21. Workshop Software-Reengineering und -Evolution, GI-Fachgruppe Software-Reengineering, 2019

[3] Andres Koch, Remo Koch, "Automatisierte Code-Refaktorierung in der Praxis", Proceedings zum 21. Workshop Software-Reengineering und -Evolution, GI-Fachgruppe Software-Reengineering, 2019