Teilautomatisiertes Architektur-Reengineering in einem JavaEE Monolithen

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

Abstract

Im Rahmen eines umfangreichen Sanierungs-Programmes für ein großes JavaEE basiertes ERP-System verfolgen wir einen technischen Reengineering-Ansatz: bestehender Code wird dabei in neue architektonische Strukturen überführt. Für die Code-Migration wollen wir eine Teilautomatisierung erreichen, um die manuellen Aufwände zu minimieren. Dabei greifen wir auf die Arbeit von Object-Engineering[1] [1] zurück, die im Rahmen des WSRE 20 im Jahr 2018 vorgestellt wurde. Gemeinsam mit dem Unternehmen haben wir einen Generator für unsere spezifische Aufgabestellung erstellt, basierend auf deren generischem Werkzeug OMAN.

1 Ausgangslage

Das ERP-System *Bison Process* hat eine Code-Basis von ca. 1.5 mloc. Seine Entwicklung begann in den späten 1990er Jahren. Qualitätsziele wie Plattformunabhängigkeit und Anpassbarkeit standen im Vordergrund. Sie führten zu umfassenden Konfigurations- und Scripting-Möglichkeiten im Produkt. Der intensive Gebrauch dieser Anpassungsmöglichkeiten in Kundenprojekten hatte steigende Wartungs- und Releasing-Kosten sowie Stabilitätsprobleme zur Folge. Ursachen dafür waren vor allem mangelnde Testbarkeit, schwer verständliche, generische Strukturen und Code sowie eine monolithische Architektur ohne Komponentenbildung und ein darüber hinaus schnelles Wachstum der Code-Basis.

2015 wurde die Strategie für das Produkt geändert und neue Qualitätsziele definiert. Primäre Ziele sind die Senkung der Betriebs- und Wartungskosten sowie die Steigerung von Stabilität und Wartbarkeit. Dazu setzte man ein umfangreiches Modernisierungs-Programm auf. Zu dessen Beginn wurden die Haupt-Kostentreiber identifiziert, Maßnahmen definiert um diese zu adressieren und die Ziel-Architektur festgelegt. Die Maßnahmen werden seitdem in diversen Projekten umgesetzt.

2 Legacy Architektur

Eines dieser Projekte hat das Ziel, die Kern-Geschäftslogik aus einer umfangreichen Vererbungshierarchie mit einem zusätzlichen Geflecht an Hilfsklassen (Abbildung 1) neu zu strukturieren.

Das gilt insbesondere für die Defaulting-Funktion, die Benutzereingaben am UI dazu nutzt, weitere Feldwerte vorzubelegen. Um diese Funktion geht es im vorliegenden Papier: Eine Defaulting-Funktion für Wert x, dessen Berechnung abhängig ist von Wert y muss in der Berechnungsreihenfolge immer nach der Berechnungregel für Attribut y stehen. In der sequentiellen Codestruktur mit sehr vielen solcher Regeln sind diese Abhängigkeiten nicht explizit und oft nur schwer erkenn- und nachvollziehbar. Hohe Aufwände bei Änderungen der Abhängigkeiten oder der Einführung neuer Regeln sind die Folge.

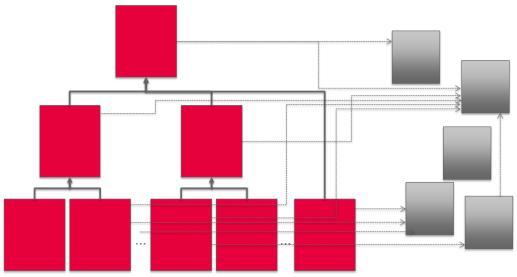

Abbildung 1: Vereinfachte Darstellung der heutigen Vererbungshierarchie (rot) mit einem Geflecht an weiteren abhängigen Hilfsklassen (grau)

3 Zielarchitektur

Mit der neuen Architektur entsteht eine Fachdomänen-orientierte Komponentenstruktur und zusätzlichen domänenübergreifenden Library-Komponenten zur Wiederverwendung gemeinsamer Logik. Sie bieten ihre Funktionen über ein API als grobgranulare Controller-Services an. Diese orchestrieren feingranulare X-lets (=Mini-Services) innerhalb der Komponenten (Abbildung 2). Im bisherigen Code war der Code lediglich durch Java Packages (Namespaces) segregiert. Da es so faktisch kein Information-Hiding gab, birgt der Legacy-Code ein komplexes Geflecht an Abhängigkeiten. Mit der Komponentenbildung zielen wir auf Entkopplung mit den bekannten positiven Effekten auf Wart- und Erweiterbarkeit.

Legacy-Funktionen, die nicht in die Zielstruktur

[1] Object Engineering GmbH; Birmensdorfer Strasse 32; CH-8142 Uitikon-Waldegg (ZH); https://www.objeng.ch

überführt werden (fachlicher Schnitt), bleiben an ihrer ursprünglichen Stelle und werden mittels Interfaces und Abhängigkeitsumkehr abstrahiert. (Abbildung 2).

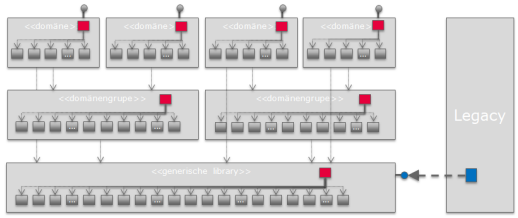

Abbildung 2: Vereinfachte Darstellung des Ziel-Designs: Die Controller-Services sind rot dargestellt, die X-Lets in dunkelgrau; blau: Abhängigkeitsumkehr für benötigte Funktionen, die im Legacy-Code verbleiben

4 Vorgehen beim Reengineering

Entscheidend für den Erfolg eines solchen Projektes ist es, einen effizienten Weg zu finden, den Code aus dem ursprünglichen Klassen-Geflecht ins Ziel-Design zu überführen ohne dabei die Geschäftslogik zu verändern. Wir haben unser Vorgehen für die Migration der Defaulting-Logik zunächst grob geplant, manuell erprobt und anschließend weiter optimiert.

Das erarbeitete Vorgehens-Muster konnte nun wiederholt angewendet werden: jeweils ausgehend von einer Einstiegs-Methode in der Legacy-Struktur ist der Code in die Tiefe zu verfolgen, relevante Statements zu identifizieren und in das richtige X-let zu verschieben; dabei ist der Kontext dieser Statements ebenfalls relevant: Bedingungen, Zwischenvariablen und deren Berechnung. Abzugrenzen ist am Übergang zu technischem Code, der unverändert bleibt sowie bei frachlichem Code, der nicht zur betrachteten fachlichen Domäne gehört.

5 Automatisierung

Zunehmende Routine führt zu der Frage nach Automatisierungsmöglichkeiten für den Migrationsprozess. Wir nutzten die Vorarbeit [1] von Object Engineering. Basierend auf deren Modell und Werkzeug haben wir in Zusammenarbeit mit dem Unternehmen eine Generator-Lösung erarbeitet und konnten so eine Teilautomatisierung erreichen.

Im konkreten Fall wird ein X-let pro zu schreibendem Attribut erzeugt. Das Werkezeug erkennt im Input-Code die Statements, die ein relevantes Attribut schreiben. Aus dem Code, der auf dem Weg zu diesen Statements liegt (Bedingungen, Berechnungen) werden Abhängigkeiten zu anderen Attributen ermittelt und Code-Snippets aufgesammelt. Methodenaufrufe werden nachverfolgt und deren Code ebenfalls berücksichtigt. Auf diese Weise ist es dem Generator möglich im ersten Schritt ein abstraktes Graphen-Modell der Logik mit allen relevanten Eigenschaften, Abhängigkeiten und Code-Snippets zu erstellen. Aus diesem Modell wird in einem zweiten Schritt mittels Graph-Durchwanderung der neue Code pro Zielattribut gesammelt und in die jeweilige X-let-Klasse generiert.

Die ursprünglich beherbergende Klasse eines schreibenden Statements für ein Attribut definiert die Zuordnung zur fachlichen Domäne und damit den Namespace des X-lets. Die X-lets folgen einem Command-Pattern. Die für ein Zielattribut relevanten Code-Snippets werden im Generierungs-Schritt in der Command-Methode in ihrer ursprünglichen Reihenfolge abgelegt. So enthält am Ende jedes X-let sämtliche für sein jeweiliges Ziel-Attribut notwendige Logik inklusive aller Bedingungen und Berechnungen.

Die generierten Defaulting-X-Lets sind nach der Generierung aber noch nicht fertig. Der Wirtschaftlichkeit geschuldet bleiben noch manuelle Arbeiten, wie z.B. die Anbindung an abgegrenzte Funktionalität, die im Legacy-Code verbleibt. Diese ist per Definition (Ziel-Architektur) aus den neuen Komponenten nicht mehr direkt zugänglich (s.o.). Die Service-Infrastruktur der Komponenten wird aus dem gleichen Grund nicht mit generiert. Ebenso sind Refactorings und Anpassungen an die neue Umgebung ebenfalls noch manuell vorzunehmen. Durch die Generierung können Code-Duplikationen entstehen, die eventuell wieder zusammengeführt werden. Ein Generierungs-Protokoll weist auf fragwürdige Szenarien hin und gibt so Hinweise auf Stellen, die evtl. überarbeitet werden müssen. Es verbleiben in unserem Szenario ca. 50% des ursprünglichen manuellen Aufwandes.

6 Fazit

Wer sich für den Bau eines solchen Generator-Werkzeugs entscheidet, darf kein Plug-and-Play erwarten. Eine sehr gute Kenntnis der Original-Code-Basis und ein klares Bild des Ziel-Designs ist entscheidend. Zudem muss das Migrationsvorgehen manuell gut erprobt sein und Migrationsregeln müssen identifiziert werden. Der Aufwand bei der Erstellung eines solchen Analyse/Generierungs-Werkzeugs hängt stark von der Qualität des Eingangs-Codes ab: in der vorliegenden Code-Basis gab es die erwähnten architektonischen Schwächen, der Source-Code folgt aber klaren Konventionen - so waren Migrationsregeln überschaubar und gut zu formalisieren.

Wir konnten den Ansatz für einige unserer Problemstellungen als eine pragmatische Methode erkennen um für die Generierung sehr vieler gleichartiger Code-Elemente einen deutlichen Effizienzgewinn bei der Migration zu erreichen.

Literatur

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