Automatisierte Code-Refaktorierung in der Praxis

Andres Koch (akoch@objeng.ch), Remo Koch (rkoch@objeng.ch)
Object Engineering GmbH, CH-8142 Uitikon-Waldegg

Abstract

Programm-Code wurde in den letzten Dekaden durch Heerscharen von *Entwicklerinnen* und *Entwicklern* produziert. Neue Anforderungen konnten nicht mehr oder nur ungenügend mit dem bestehenden Design abgedeckt werden. Für ein Redesign und der dazugehörigen Code-Refaktorierung wurde wenig oder keine Zeit eingeräumt. Schnell kann der Überblick über diese grossen Mengen an Code verlorengehen.

In der vorliegenden Fallstudie wurde ein teilautomatisiertes Refaktorierungs-Verfahren auf eine Java EE Code-Basis angewendet. Die damit gesammelten Erfahrungen haben gezeigt, dass oft die "kleinen Dinge" die grössten Zeitersparnisse für die Entwickler bringen. Der Erfolg der automatisierten Code-Refaktorierung hängt von einer engen Zusammenarbeit mit dem Kunden, einer klaren und bestimmten Zielarchitektur und einem iterativen und pragmatischen Vorgehen ab.

Vorgehen

Im vorliegenden Projekt ging es darum bestehenden Code entsprechend der neuen Architektur neu zu organisieren und umzustrukturieren. Der Programm-Code, bestehend aus mehr als 6000 Klassen, musste extrahiert werden und nach Domänen, sowie spezifischen Ziel-Eigenschaften in separate Klassen verschoben werden. Es war quasi eine Transformation von einer vertikalen in eine horizontale Struktur. Damit der Aufwand für die manuelle Nachbearbeitung minimiert werden konnte, wurden abhängige Referenzen importiert, private Methoden migriert und Hilfestellung für Entwickler in Form von direkten Code-Links und Kommentaren, sowie ausführliche Reports zum Ursprungscode generiert.

Dieses Verfahren beinhaltet folgende Schritte:

1. Einlesen (Parsing) der gesamten Code-Basis (ca. 6000 Klassen) und Erstellen eines abstrakten Syntax-Baumes (AST) für jede Klasse.

2. Durchlaufen des AST mit Erkennung der projekt- und sprachenspezifischen Regeln und Speichern der relevanten Informationen im Meta-Modell.

3. Wiederholtes Durchlaufen des Metadaten-Baumes und Ergänzung zusätzlicher Metainformationen und Verbindungen zu anderen Entitäten (Klassen, Methoden, Statements) im gleichen Modell.

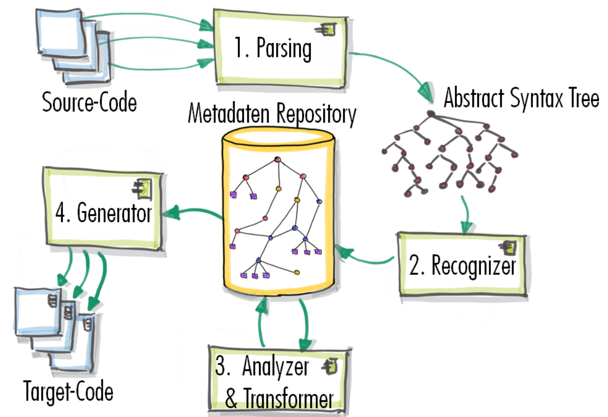

Abbildung 1: Vorgehen

4. Finales Durchlaufen des Metadaten-Baumes und Generierung der neuen Artefakte.

5. Kontrolle der generierten Artefakte und Einfügen in ein Versionskontroll-System (Git) zum einfachen Vergleich zwischen verschiedenen Generierungs-Versionen.

6. Einsatz der generierten Artefakte im System mit der neuen Architektur.

Erkennung

Jede Klasse wird von einem Java-Parser verarbeitet. In einem ersten Durchlauf werden nur Klassen und Methoden ins Meta-Modell gespeichert. Im zweiten Durchlauf werden die projektspezifischen Erkennungs-Regeln angewendet und der Code bis auf Anweisungs-Ebene verarbeitet. Beim Erkennen der spezifischen Regeln werden alle relevanten Informationen im Meta-Modell gespeichert. Dies wurde sowohl im ersten wie auch im zweiten Teil mit Hilfe eines Visitors über den AST durchgeführt.

Nach diesen Schritten (1. und 2. oben) steht die ganze Problem-Domäne der bestehenden Programme im Modell zur Verfügung.

Struktur-Analyse

In mehreren nachfolgenden Analyse-Schritten werden die Meta-Entitäten untereinander verknüpft, Attribute werden aufgrund von erkannten Eigenschaften definiert und weitere Daten werden gesammelt sowie Strukturen aufgebaut. Für jeden dieser Aspekte wird ein eigener Analysator verwendet. Dies erwies sich als vorteilhaft für die während dem Refaktorierungs-Prozess ändernden Anforderungen.

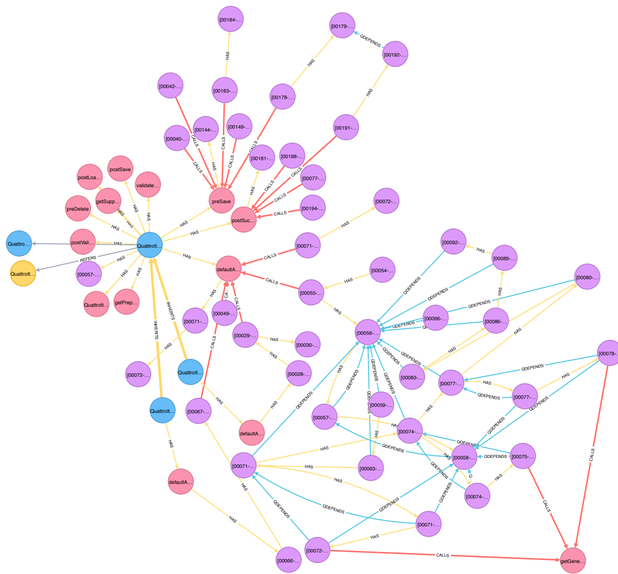

Abbildung 2: Auszug aus dem Graphen des analysierten Meta-Modell

Generierungs-Schritt

Die vorangehenden Schritte haben das Meta-Modell so aufbereitet, dass es für die Generierung der Ziel-Artefakte vorbereitet ist. Die Aufwände für die Aufbereitung der Meta-Daten sind wesentlich grösser als für die finale Generierung des Ziel-Codes. Das Entwickeln des Generators setzt ein umfassendes Know-How über die vorhandenen Meta-Daten voraus, um effizient zu sein.

Grenzen und Pragmatik

Wie erwähnt, ist dieser Ansatz eine *teilautomatisierte* Refaktorierung. Entsprechend muss man die Grenzen der Automatisierung erkennen und sich auch daran halten. Zum Beispiel: da es sich um Code handelt von dem man weiss, dass dieser von einem Compiler fehlerfrei eingelesen werden kann, kann man sich Fehlerbehandlungen zum Grossteil sparen.

Wenn man Code-Situationen antrifft, welche nicht eindeutig erkannt oder bearbeitet werden können, sollten diese mindestens protokolliert werden, damit diese manuell nachbearbeitet werden können. Die Protokolle werden als strukturierte Tabellen abgelegt, die von den Entwicklern nach Belieben sortiert und ausgewertet werden können.

Manuelle Nachbearbeitung

Die Überprüfung und das Nachbearbeiten des generierten Codes sollte für die ausführende Person so gut wie möglich unterstützt werden. Im vorliegenden Projekt wurde ein Git-Repository verwendet, in welches nach jeder Generierung (inklusive Code-Änderungen der Tool-Software) die erzeugten Artefakte eingefügt wurden. So konnte sowohl die generierende Software, wie auch Regeländerungen einfach und verlässlich überprüft werden.

Würde man die gleiche Refaktorierung ausschliesslich manuell lösen, würde man feststellen, dass der grösste Zeitaufwand beim Zusammensuchen und Erkennen von Code-Teilen, sowie dem Verschieben von Code an neue Stellen entsteht. Dazu käme das Suchen von (selbst) eingebauten Fehlern dazu.

Damit bei der Nachbearbeitung schnell auf den Ursprungs-Code zurückgegriffen werden kann, wurden im generierten Code Links eingebaut, über welche man in der verwendeten IDE mit einem Klick zum Original-Code springen kann. Eingefügte Kommentare und Such-Referenzen in Form von eindeutigen Identitäts-Schlüsseln erlauben es auch aus der Protokoll-Tabelle auf den Code zu referenzieren.

Es ist sehr wichtig, dass den Entwicklern jegliche Unterstützung geboten wird, welche wertvolle Zeit einsparen lässt. Es sind manchmal die kleinen Dinge, wie ein Kommentar oder ein Link, welche den grossen Unterschied ausmachen. Für Situationen mit hoher Häufigkeit lohnt es sich im Generator- bzw. im Analyse-Schritt mehr Zeit und Denkarbeit aufzuwenden.

Ziel-Architektur

Dass vor einer automatischen Generierung die Ziel-Architektur vorhanden ist, ist unabdingbar. Gerade in diesem Projekt hat der Kunde vor der automatisierten Refaktorierung die Zielarchitektur und das Design erstellt und dazu auch manuell die ersten später zu generierenden Klassen erstellt. Das hat dazu geführt, dass man die Erkennungs-Regeln finden konnte und auch Probleme besser erkennen konnte, die später bei der Automatisierung relevant waren. Die Kompetenz und das Domänen-Wissen des Kunden, seine guten Spezifikationen mit Abnahmekriterien und die enge und gute Zusammenarbeit haben erlaubt, das Projekt innerhalb der geplanten Zeit und innerhalb des budgetierten Aufwandes erfolgreich abzuschliessen.

Werkzeug-Konzept

Die für die Refaktorierung des Codes verwendete Software teilte sich in eine generische Basis-Komponente und eine zusätzliche projektspezifisch entwickelte Komponente auf (ca. 10'000 LOC). Dieses Konzept erlaubt eine effiziente und zielgerichtete Umsetzung für Kunden-Projekte.

Referenzen

[1] Kai-Uwe Herrmann, Bison Schweiz AG. *Teilautomatisiertes Architektur-Reengineering in einem Java-EE-Monolithen.* WSRE 2019, 2019.

[2] Andres Koch, Remo Koch, Object Engineering GmbH. *Metadaten basiertes, teilautomatisiertes Software-Reengineering.* WSRE 2018, 2018.