Von JOBOL zu JAVOL – Refaktorisierung migrierter Java-Programme

Nils Bauer, Christian Becker, Uwe Erdmenger, Felix Graßler, Denis Uhlig
pro et con Innovative Informatikanwendungen GmbH, Reichenhainer Straße 29a, 09126 Chemnitz
nils.bauer,christian.becker,uwe.erdmenger,felix.grassler,denis.uhlig@proetcon.de

Abstract

Die Firma pro et con realisiert Softwaremigrationsprojekte, bspw. von COBOL nach Java, als 1:1-Migration. Die konvertierten Programme enthalten i. d. R. COBOL-typische und kundenspezifische Codemuster, welche sich im Laufe der Wartung als störend herauskristallisieren können. Dieser Bericht beschreibt Werkzeuge zur Refaktorisierung solcher Muster, welche in einer separaten Phase am Ende eines Migrationsprojektes oder auch Jahre später noch eingesetzt werden können.

1 Motivation

pro et con realisiert Softwaremigrationsprojekte toolgestützt mit der eigenentwickelten *pecBOX*, einem Werkzeugkasten, der verschiedene Migrations- und Metawerkzeuge sowie Codegeneratoren zusammenfasst. Eines der Migrationswerkzeuge ist der COBOL to Java Converter (*CoJaC*), welcher einen Automatisierungsgrad von über 98% erreicht und bereits in mehreren, kommerziellen Migrationsprojekten zum Einsatz kam. Nach Projektabschluss werden die migrierten Java-Programme gewartet und weiterentwickelt, woran pro et con zum Teil beteiligt ist. Erfahrungen aus verschiedenen Projekten und der Wartung migrierter Systeme haben gezeigt, dass der Quelltext zwar gut lesbar ist, allerdings migrationsbedingt dennoch Elemente enthält, welche in Java unüblich sind und die Lesbarkeit erschweren können. Diese sind:

1. technischer Code, der für die korrekte Funktionsweise notwendig ist,
2. COBOL-typische Codemuster, welche durch die Ursprungssprache vorgegeben sind,
3. kundenspezifische Programmiermuster, basierend auf dem Stil der ursprünglichen Entwickler.

Punkt 1 kann durch eine *technische* Refaktorisierung verbessert werden. Dazu zählt bspw. die Umstellung von migrationsbedingt langen Zugriffsketten auf verkürzte Bezeichner [1]. Hierbei erfolgt keine inhaltliche Veränderung der Programme und der Programmlogik. Die geänderten Programme sind vor und nach der Refaktorisierung im Prinzip identisch.

Bei den folgenden beiden Punkten handelt es sich dagegen um potentiell *inhaltliche* Verbesserungen. Die Punkte 2 und 3 werden ebenso wie Punkt 1 in einer separaten Reengineering-Phase nach der Migration bearbeitet.

Allerdings muss diese nicht direkt im Anschluss an die Migration erfolgen. Die Entwickler des Kunden benötigen erfahrungsgemäß eine gewisse Einarbeitungszeit, um sich an den migrierten Code und den neuen Entwicklungsprozess nach der Migration zu gewöhnen. Erst mit steigender Erfahrung im Laufe der Wartung kristallisieren sich häufig Codemuster heraus, die neben einer verbesserten Les- und Wartbarkeit vor allem wegen einer besseren Akzeptanz durch Java-Entwickler überarbeitet werden sollten.

Die Erkenntnisse aus verschiedenen Migrationsprojekten fließen stetig als Verbesserung in die Migrationswerkzeuge ein, wovon jedoch die Kunden bereits abgeschlossener Projekte i. d. R. nicht profitieren. Aus diesem Grund wurde eine Reihe von Refaktorisierungswerkzeugen entwickelt, welche den migrierten Code von bereits abgeschlossenen Migrationsprojekten unter Beachtung von COBOL-typischen und kundenspezifischen Codemustern automatisiert verbessern können. Sie arbeiten unabhängig voneinander, sind optional und können je nach Kunde und dessen Anforderungen beliebig miteinander kombiniert werden.

2 Arbeitsweise und Architektur

Der Ablauf einer Refaktorisierung findet in mehreren Schritten statt. Als erstes wird ein Java-Package oder ein vollständiges Java-Projekt mit allen darin befindlichen Java-Sourcen analysiert. Mit Hilfe des Eclipse Java Development Toolkit (JDT) wird für jedes Sourcefile ein Java-Syntaxbaum erstellt. Dabei werden die zu refaktorisierenden Klassen in den Syntaxbäumen selektiert. Dies geschieht anhand von zuvor definierten Kriterien, bspw. einer bestimmten Elternklasse, des Typs oder eines implementierten Interfaces. Im nächsten Schritt wird der Syntaxbaum mit dem Visitor-Pattern traversiert. Einzelne Java-Statements (als Teilbäume mit Knoten) werden in Ausdrücke zerlegt. Es wird geprüft, ob ein solcher Teilbaum einem bestimmten Muster entspricht. Als Beispiel dient die von COBOL nach Java migrierte Anweisung `someVar.compute(someVar.getValue()+1)` zum Inkrementieren einer Variablen um den Wert 1. Deren Teilbaum wird wie folgt ausgewertet:

```
1|  tree == expression1.call(parameter)
2|  parameter == sum(expression1.call2, '1')
3|  call  == 'compute'
4|  call2 == 'getValue'
```

In Zeile 1 wird geprüft, ob der Teilbaum dem Muster `expression1.call(parameter)` entspricht. Die Zeilen 2 bis 4 enthalten weitere Prüfungen, ob die restlichen Ausdrücke dem gesuchten Muster entsprechen. Sind alle Bedingungen erfüllt, dann ist ein relevanter Kandidat für eine Refaktorisierung gefunden. Die Nutzung von Syntaxbäumen bietet den Vorteil, dass Kontextinformationen in einer strukturierten Form nutzbar sind, wodurch falsch positive Kandidaten minimiert werden können. Es entsteht so eine Liste von zu refaktorisierenden Kandidaten und aller vorzunehmenden Änderungen in den Sourcen. Letztere können Ergänzungen, Verschiebungen, Löschungen oder sonstige Umformungen sein. Im letzten Schritt erfolgt die Änderung durch Bildung neuer Teilbäume:

```
1|  call3 = 'increment'
2|  tree  = expression1.call3
```

Für das obige Beispiel ergibt sich nach der Refaktorisierung das folgende Ergebnis: `someVar.increment()`. Jede dieser Umformungen ist als separates Modul konzipiert, so dass verschiedene, aufeinander aufbauende Kombinationen möglich sind.

3 Anwendung in bestehendem Projekt

Die Refaktorisierungswerkzeuge wurden u. a. auf ein bereits erfolgreich abgeschlossenes COBOL-Java-Migrationsprojekt mit 210 Programmen und 1.400.000 Lines of Code (Java) angewendet, bei dem pro et con an der Wartung und Weiterentwicklung beteiligt ist. Daraus ergaben sich in den letzten Jahren neue Erfahrungen bezüglich der Codemuster, die aufgrund der Ursprungssprache COBOL oder dem Programmierstil der originalen Entwickler aus Java-Sicht schwer lesbar bzw. unüblich sind. Nachfolgend zwei Beispiele dazu:

Negation in Schleifenbedingungen (COBOL-typisch): Schleifenbedingungen werden in COBOL in einer negierten Form angegeben (s. Zeile 1 `PERFORM`):

```
1|  PERFORM UNTIL a > b
2|    COMPUTE a = a + 1
3|  END-PERFORM
```

Durch die 1:1-Migration entsteht folgender Java-Code:

```
1|  while(!(a > b)) {
2|     a.compute(a.getValue()+1);
3|  }
```

Die Negation der Bedingung `!(a > b)` ist unnötig schwer lesbar und für Java-Code unüblich. Das Refaktorisierungswerkzeug analysiert die Vorkommen solcher Codemuster und entfernt die Negation durch Umstellung des inneren Operators >. Das Ergebnis (inkl. des Inkrement-Beispiels) sieht wie folgt aus:

```
1|  while(a <= b) {
2|     a.increment();
3|  }
```

Diese relativ einfache Umformung sorgt bereits für eine bessere Lesbarkeit und erleichtert das Codeverständnis solcher häufig vorkommenden Schleifenbedingungen, da die unnötige Negation entfällt. Im Beispielprojekt wurden ca. 1.700 Schleifenbedingungen und 9.000 Increment-Anweisungen umgeformt.

Zuweisung 88er Stufen (kundenspezifisch): 88er Stufen in COBOL sind sogenannte Bedingungsnamen mit einem bestimmten Wert. Diese können in `IF`-Anweisungen anstelle von `Magic Numbers` genutzt werden:

```
1|  01 SYSTEM-AKTIV PIC X.
2|   88 AKTIV       VALUE 1. * siehe IF
3|
4|  MOVE 1 TO SYSTEM-AKTIV   * Variante 1
5|  SET AKTIV TO TRUE        * Variante 2
6|
7|  IF AKTIV THEN ...
```

Im obigen Beispiel besitzt die Variable `SYSTEM-AKTIV` (Zeile 1) den Bedingungsnamen `AKTIV` (Zeile 2). Der Wert von `SYSTEM-AKTIV` kann über `MOVE` (Zeile 4) oder `SET` (Zeile 5) zugewiesen werden. Hat `SYSTEM-AKTIV` den Wert 1, dann ist die Bedingung im `IF` in Zeile 7 erfüllt. Das Setzen von Werten für Bedingungsnamen wie in Zeile 4 ist fehleranfällig, da auch ungültige Werte zugewiesen werden können. Diese Variante ist auch schlechter lesbar, da die Entwickler ggf. nicht wissen, was `MOVE 1 TO SYSTEM-AKTIV` für eine fachliche Bedeutung hat. Die Variante 2 ist eigentlich bereits in COBOL zu bevorzugen, wird jedoch nicht immer verwendet. Nach der 1:1-Migration entsteht für das aktuelle Beispiel folgender Java-Code:

```
4|  systemAktiv.setValue(1); // Variante 1
5|  systemAktiv.setAktiv();  // Variante 2
6|
7|  if( systemAktiv.isAktiv() ) { ...
```

Die Refaktorisierungswerkzeuge unterstützen die Umformung der Variante 1 in die Variante 2. Dies ist ein komplexerer Anwendungsfall, da hier sowohl der Typ der Variable `systemAktiv` als auch alle Zuweisungen (`systemAktiv.setValue(...)`) und alle zugewiesenen Werte mit den möglichen Bedingungsnamen (`setAktiv()`) abgeglichen werden müssen. Dieses Muster ist kundenspezifisch und die Anzahl der Vorkommen hängt vom Programmierstil der ursprünglichen COBOL-Entwickler ab. Im Beispielprojekt wurden ca. 900 Stellen refaktorisiert.

Neben diesen Beispielen werden bisher insgesamt 13 einfachere und komplexere Refaktorisierungen unterstützt, welche sich optional je nach Kundenwunsch anwenden lassen. Der Test erfolgt äquivalent zum Test regulärer Erweiterungen im Rahmen der Wartung. Der Kunde nutzt hierfür eine Testsuite, die i. d. R. im Rahmen eines Migrationsprojektes entsteht und im Laufe der Wartung erweitert wird.

4 Schlussfolgerungen

Die Erfahrungen aus den letzten Migrationsprojekten zeigen, dass die Optimierung der migrierten Programme eine wichtige Phase in einem Migrationsprojekt darstellt. Dabei stehen neben offensichtlichen Gründen wie der Verbesserung der Les- und Wartbarkeit der Quelltexte auch die Steigerung der Akzeptanz durch neue Entwickler im Vordergrund. Die Refaktorisierungswerkzeuge, welche die Sourcen inhaltlich ändern, können auch Jahre nach dem Migrationsprojekt und mit steigender Erfahrung der Entwickler angewendet werden. Dadurch ist es auch möglich, dass Kunden bereits abgeschlossener Projekte von den in nachfolgenden Projekten gewonnenen Erfahrungen und daraus resultierenden Verbesserungen der Migrationstechnologie profitieren. Der Aufwand für die manuelle Umsetzung solcher Refaktorisierungen in dieser Größenordnung und Häufigkeit in großen Projekten wäre i. d. R. wirtschaftlich nicht gerechtfertigt. Projekte dieser Größenordnung sind nur mit Hilfe automatisierter Werkzeuge möglich.

Literaturverzeichnis

[1] Graßler, F.; Uhlig, D.: Software-Migration ist keine Endstation. Softwaretechnik-Trends, Band 40, Heft 2, August 2020