Software-Migration ist keine Endstation

Automatisiertes Refactoring von generiertem Java-Code

Felix Graßler, Denis Uhlig
pro et con Innovative Informatikanwendungen GmbH, Reichenhainer Straße 29a, 09126 Chemnitz
{felix.grassler,denis.uhlig}@proetcon.de

Abstract

In der Diskussion um die Vor- und Nachteile einer Software-Migration wird immer wieder die Behauptung aufgestellt, dass automatisch generierter Programmcode schwer wartbar sei. Diese Behauptung wurde schon vielfach in der Praxis widerlegt [1]. Automatisch migrierter Quelltext kann seine Herkunft nicht verleugnen und ist in seiner Ausprägung immer ein Kompromiss zwischen Wiedererkennbarkeit und Anpassung an Designprinzipien der Zielsprache. Die Autoren haben in der Vergangenheit eine Reihe von Software-Migrationsprojekten begleitet, in deren Mittelpunkt eine Sprachkonvertierung von COBOL nach Java mit dem Konvertierungswerkzeug "CoJaC" (COBOL-Java-Converter) stand [2]. Diese migrierten Softwaresysteme unterliegen natürlich - ebenso wie andere Softwaresysteme auch - einer ständigen Wartung und Weiterentwicklung. Es wurden Werkzeuge entwickelt, mit denen in diesem Prozess durch automatisiertes Refactoring für die migrierten Java-Programme ein Qualitätszuwachs in Bezug auf "Java-typischen" Quelltext erreicht werden kann. Deren Wirkungsweise wird in diesem Beitrag beschrieben.

1 Annotations statt Boilerplate

In der automatischen COBOL-Java-Konvertierung mit "CoJaC" werden aus COBOL konvertierte Datenstrukturen in Java mit einer Reihe von Methoden angereichert, welche das Verhalten der COBOL-Datentypen in Java emulieren. Sie dienen vor allem der Serialisierung der Datenstrukturen. Ihr Aufbau besitzt immer die gleiche, einfache Grundstruktur. Dieser sogenannte Boilerplate-Code erschwert die Lesbarkeit und die Arbeit mit den generierten Datenstrukturen im Rahmen der Wartung.

Diese Methoden können durch Annotations ersetzt werden. Ein Annotation-Prozessor verarbeitet diese beim Compilieren. Aus den Annotations werden neue Unterklassen generiert, welche die ersetzten Methoden beinhalten. Der Boilerplate-Code wird damit ausgelagert. Die Datenstrukturen selbst werden somit übersichtlicher und die manuelle Wartung dieser Methoden entfällt:

```
public class AlteStruktur{
  public CobolNumber number;
  public CobolString string;

  public void write(Factory fact){
    fact.store(number);
    fact.store(string);
  }
  // weitere Boilerplate-Methoden
}
```

```
public class NeueStruktur{
  @DataField(10)
  public CobolNumber number;
  @DataField(20)
  public CobolString string;
}
```

Die Ersetzung von Boilerplate durch Annotations in den Java-Programmen ist auf Basis einfacher Mustererkennung möglich. Damit werden schlankere und besser wartbare Definitionen von Datenstrukturen erreicht.

2 Geht das nicht kürzer?

2.1 Motivation

In großen Programmsystemen kommen auch entsprechend komplexe, verschachtelte Datenstrukturen zum Einsatz. In COBOL ist es möglich und auch üblich, bei Zugriffen auf verschachtelte Datenelemente die Zwischenstufen wegzulassen, solange die Eindeutigkeit bewahrt bleibt. In Java muss hingegen stets der vollständig qualifizierte Name angegeben werden. So entstehen bei tief verschachtelten Datenstrukturen lange Zugriffsketten, welche die Lesbarkeit des Codes beeinträchtigen. Diese Zugriffe können jedoch automatisiert durch kurze Getter- und Setter-Methoden ersetzt werden. Das geschieht in zwei Stufen. Die bereits existierenden Zugriffe werden automatisiert durch Getter bzw. Setter ersetzt und damit verkürzt. Die verwendeten Datenelemente werden mit entsprechenden Annotations versehen. Das ist ein einmaliger Prozess. Die verwendeten Setter/Getter existieren zu diesem Zeitpunkt noch nicht. Diese werden erst beim Compilieren anhand der gesetzten Annotations erzeugt. Dadurch können auch bei zukünftiger Wartung verkürzte Zugriffe generiert werden, bspw. beim Hinzufügen eines neues Attributs in eine Datenstruktur. Diese sind dann genauso aufgebaut, wie die einmalig erzeugten Zugriffe.

2.2 Einmalige Vorarbeiten

Hierfür wird das Eclipse Java Developement Toolkit (JDT) zu Hilfe genommen, mit dem Syntaxbäume für die Java-Sourcen erstellt und manipuliert werden können. Für die Ersetzungen müssen komplexe Zusammenhänge zwischen den Programmen und Datenstrukturen analysiert werden. Das ist rechenintensiv. Da diese Ersetzungen aber nur einmalig durchgeführt werden, ist die Laufzeit nicht entscheidend. Zudem wird der Arbeitsaufwand im Annotation Processing verringert und die Compile-Zeit verkürzt. Im nachfolgenden Beispiel existiert ein langer set-Zugriff auf das Attribut *firstName* der Klasse *PersName*. Letztere ist eine innere Klasse von *EmployeeDetails*. Der Zugriff wird auf einen Setter verkürzt und *firstName* mit @Setter annotiert.

```
/* innerhalb eines Programms: */
  EmployeeDetails employee = new
      EmployeeDetails();
  // alter und neuer Zugriff auf firstName
  employee.person.persName.firstName.setValue("Bob");
  employee.setFirstName("Bob");


/* Datenstruktur: EmployeeDetails */
public class EmployeeDetails {
  public Person person = new Person();

  public class Person {
    public PersName persName = new PersName();
```

```
    public class PersName {
      @Setter
      public CobolString firstName;
      public CobolString lastName;
    }
  }
}
```

2.3 Annotation Processing

Alle Annotations einer Strukturklasse werden zusammengetragen. Das betrifft die Annotations der direkten Attribute der jeweiligen Klasse sowie die Attribute der inneren Strukturen. Anschließend wird für jedes annotierte Attribut mindestens eine Methode generiert, entsprechend der Annotation eine Setter- bzw. Getter-Methode. Für jede Strukturklasse mit mindestens einer Annotation wird jeweils eine neue Unterklasse generiert, die alle benötigten Setter und Getter beinhaltet. Damit ist auch hier der Boilerplate-Code gekapselt und die Lesbarkeit der Datenstrukturen wird nicht beeinflusst. In den Programmen werden anstelle der bisherigen Strukturklassen die generierten Unterklassen verwendet, damit die erzeugten Methoden auch nutzbar sind. Da außer den Getter und Setter keine Veränderungen vorgenommen werden, können die Unterklassen ansonsten genauso wie die Originale genutzt werden. Die Methoden werden nach einem festen Schema generiert: Zunächst "get" bzw. "set", gefolgt vom Namen des Attributs, auf welches zugegriffen wird. Für eine einfachere Verwendung wird eine Javadoc erzeugt, in welcher der komplette Zugriffspfad steht. Damit ist auch aus den Programmen heraus ersichtlich, auf welches Element konkret zugegriffen wird.

```
/**
  * person.persName.firstName.setValue(value)
  */
public void setFirstName(String value) {
  person.persName.firstName.setValue(value);
}
```

2.4 Zugriffsarten

In COBOL werden Unterstrukturen nicht immer direkt innerhalb der Hauptstruktur definiert, sondern sie können auch in eigene Quelltextdateien ausgelagert werden. Das ist immer dann notwendig, wenn die gleiche Unterstruktur in mehreren Hauptstrukturen verwendet wird. Da auch in diesem Fall verkürzte Zugriffe erzeugt werden sollen, können die Annotations mit einem Parameter versehen werden. Dieser bestimmt die gewünschte Art des Zugriffs auf das annotierte Attribut.

Im Standardfall bezieht sich die Annotation auf das Element selbst und die Annotation wird der Hauptstruktur in der jeweiligen Quelltextdatei zugeordnet. Dieses Verhalten ist im Beispiel aus Kapitel 2.2 dargestellt. Bei ausgelagerten Unterstrukturen muss die Verarbeitung mehrere Quelltextdateien überspannen. Dafür können *PROVIDE*- und *INLUDE*- Zugriffstypen verwendet werden. Bei *PROVIDE* wird die Annotation zwar der jeweiligen (Unter-)Struktur zugeordnet, allerdings wird kein verkürzter Zugriff erzeugt, sondern die Annotation nur "exportiert" bzw. bereitgestellt. In der Hauptstruktur kann die Instanz der ausgelagerten Unterstruktur mit einer *INLUDE*- Annotation markiert werden. In diesem Fall wird nicht für das Attribut selbst ein verkürzter Zugriff erzeugt, sondern stattdessen werden alle exportierten Zugriffe der Unterstruktur verarbeitet.

```
public class EmployeeDetails {
  @Setter(Access.INCLUDE)
  public Person person = new Person();
}

public class Person {
  public PersName persName = new PersName();
  public class PersName {
    @Setter(Access.PROVIDE)
    public CobolString firstName;
  }
}
```

Im Beispiel exportiert die Klasse *Person* die Setter-Annotation des Elements *firstName*. In der Klasse *EmployeeDetails* wird diese importiert. Der Annotation-Prozessor erzeugt eine Unterklasse von *EmployeeDetails*, welche eine Setter-Methode für das Attribut *firstName* enthält.

```
public class _EmployeeDetails_ extends
    EmployeeDetails {
  /** person.persName.firstName.setValue(value)
      */
  public void setFirstName(String value) {
    person.persName.firstName.setValue(value);
  }
}
```

3 Zusammenfassung

Das automatisierte Ersetzen von Boilerplate-Code durch Annotations erzeugt übersichtlicheren, kompakteren und somit besser wartbaren Java-Code. Mit dem JDT können alle notwendigen Zusammenhänge zwischen Programmen, Variablen und Datenstrukturen umfassend analysiert werden, und komplexe Manipulationen sind möglich. Eine automatisierte Anpassung muss auch komplizierte Sonderfälle behandeln, wie z. B. Namenskollisionen bei identisch benannten Attributen in mehreren Unterstrukturen. Diese Sonderfälle können aber ebenso automatisiert behandelt werden und stehen als nächste Entwicklungspunkte auf der Agenda. Ein wesentliches Ziel ist aber erreicht. Es wurde nachgewiesen, dass es automatisiert möglich ist, migrierten Java-Code durch verschiedene Refactoring-Maßnahmen zu verbessern und somit in Wartungsphasen nach der Software-Migration auf Anforderungen und Wünsche von Kunden einzugehen.

Literaturverzeichnis

[1] Becker, C.; Kaiser, U.: Wartung von automatisch generiertem Java-Code nach einer Software-Migration. Softwaretechnik-Trends, Band 38, Heft 2, 2018.

[2] Kaiser, U.; Uhlig, D.: Erfolgreiche BS2000-Migration. In: it-dayli.net, Oktober 2016.