Jens Kosiol

Ausgabe: 8. Mai 2023
Abgabe: 22. Mai 2023

Graph- and Model-Driven Engineering
Übungsblatt 2

1 Entwurf Graphprogramm (8 Punkte)

Schreiben Sie ein Graphprogramm in Groove, das für einen gegebenen (zusammenhängenden) Graphen einen Spannbaum berechnet. Die Eingabegraphen sollen hierbei einfach über dem Typgraphen getypt sein, der einen Knoten node und eine Kante edge enthält. Zur Berechnung des Spannbaums fügen Sie eine Kante selected hinzu, die markieren soll, dass eine Kante in den Spannbaum aufgenommen wurde; der Spannbaum wird also berechnet, indem Kanten von diesem Typ selected in den Eingabegraphen eingefügt werden. Ihr Programm soll die folgenden Eigenschaften haben:

a) Das Programm soll für jeden zusammenhängenden Eingabegraphen terminieren. Zum Abschluss sollen die eingefügten selected-Kanten einen Spannbaum bilden. Insbesondere sollen keine parallelen selected-Kanten geschaffen werden und selected-Kanten dürfen nur parallel zu Kanten des Ausgangsgraphen verlaufen.

b) Das Programm soll grundsätzlich in der Lage sein, jeden *ungerichteten* Spannbaum eines zusammenhängenden Eingabegraphen zu berechnen (beachten Sie, dass die Eingabegraphen in Groove automatisch gerichtet sind).

Sie dürfen Konzepte wie *flags* und *NACs* in den Regeln nutzen.

2 Modellierung mit Multiregeln (8 Punkte)

a) Entwerfen Sie einen Typgraphen *TG*, der das Modellieren von (vereinfachten) Klassendiagrammen erlaubt. Es soll Klassen, Attribute und Methoden geben; diese Objekte sollen jeweils einen Namen haben können. Klassen sollen voneinander erben können und Attribute und Methoden haben. Methoden können von anderen Methoden und von Attributen abhängig sein. Attribute sind von einem der primitiven Datentypen, die Groove bereitstellt (bool, int, string, real).

b) Spezifizieren Sie eine einfache Variante des Refactorings *Pull Up Method* (vgl. z. B. hier) als Graphtransformationsregel; verwenden Sie dazu das Konzept einer Multiregel. Ihre Regel soll alle Methoden eines gleichen Namens aller *direkten* Unterklassen einer ausgewählten Klasse löschen und stattdessen eine neue Methode mit diesem Namen in der Ausgangsklasse schaffen. Der Name soll Eingabeparameter der Regel sein. Sie können auch den Abschnitt über *Reguläre Ausdrücke* (Kap. 3.2)

in der Anleitung zu Groove lesen und eine Regel spezifizieren, die alle Methoden eines konkreten Namens aus *allen* (transitiven) Unterklassen der gewählten Klasse löscht.

3 Graphtransformationssysteme aus Code (12 Punkte)

Gegeben seien die unten folgenden Java-Klassen, die gemeinsam einen Algorithmus zum gegenseitigen Ausschluss zweier nebenläufiger Prozesse beim Zugriff auf eine Ressource implementieren. Der Algorithmus durchläuft dabei die folgenden Phasen:

- Anfordern: Einer der Prozesse fordert den Zugriff auf die Ressource an.
- Warten: Der Prozess wartet, bis er Zugriff auf die Ressource bekommt.
- Eintritt in Kritische Phase: Der Prozess bekommt Zugriff auf die Ressource.
- Veränderung durchführen: Der Prozess verändert die Ressource.
- Wechseln des nächsten Prozesses: Es wird festgelegt, welcher Prozess als nächstes Zugriff bekommt.
- Anfordern beenden: Der Prozess, der Zugriff hatte zieht seine Anforderung auf Ressourcenzugriff zurück.

Leiten Sie aus der Implementierung mit Groove ein typisiertes Graphtransformationssystem ab, das diesen Algorithmus für einen *UIProcess*, einen *LogicProcess*, eine *Ressource* und einen *Mutex* nachbildet. Sie dürfen entweder Kontrollstrukturen verwenden, die Groove für Graphprogramme bereitstellt (siehe S.25 ff. in der Dokumentation), oder den Ablauf implizit steuern, indem Sie sich im Graphen zusätzliche Eigenschaften merken, die sich nicht eins zu eins im Java-Code wiederfinden. Um sich in typisierten Knoten Eigenschaften zu „merken“, können Sie z.Ḃ. *flags* einsetzen.

Hinweis: Unter Race-Condition versteht man, dass zwei Prozesse zur gleichen Zeit auf Daten (z. B. eine Variable) zugreifen möchten. Dann entscheidet der Prozess-Scheduler, welcher Prozess zuerst Rechenzeit bekommt und daher welcher Prozess zuerst auf die Variable zugreifen kann. Daraus resultiert, dass der Wert der Variablen unvorhersehbar wird. Um dies zu verhindern, verwenden wir in Java die Schlüsselworte *volatile* und *synchronized*. Um die Lesbarkeit zu erhöhen wird auf diese Schlüsselworte im Code

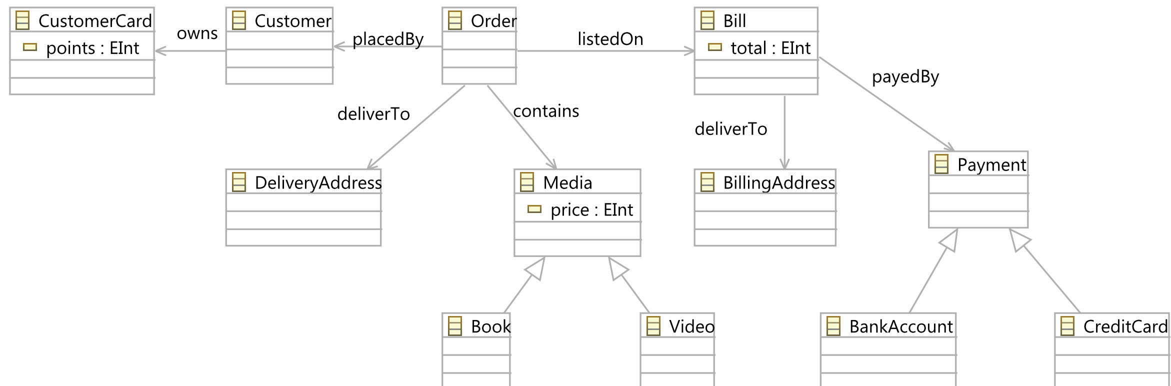

Abbildung 1: Ontologie für den Verkauf von Büchern und Videos

verzichtet. Sie können trotzdem annehmen, dass beide Prozesse den gleichen Zustand aller Variablen sehen und dass die Methoden atomar ausgeführt werden, also keine Race-Conditions bei ihrer Ausführung auftreten können.

4 Service Matching (12 Punkte)

Gegeben sei die Ontologie für den Verkauf von Büchern und Videos aus Abbildung 1 (Erweiterung gegenüber der Ontologie aus der Vorlesung).

Ein Service Requestor sucht nach den drei Services aus Abbildung 2 (beachten Sie die gegenüber Groove abweichende farbliche Syntax!) und findet einen Provider, der sechs Services anbieten. Die Services sind hierbei wie folgt informell beschrieben:

- **deleteCard_P:** Die Kundenkarte eines bereits im System vorhanden Kunden, eine Kreditkarte und eine Rechnungsadresse werden gelöscht.
- **create_Bill_P:** Es wird eine neue Rechnung erstellt, die eine schon im System vorhandene Rechnungsadresse und Bezahlmethode enthält. Der Rechnungsbetrag ist 0.

Jens Kosiol

Ausgabe: 8. Mai 2023
Abgabe: 22. Mai 2023

Graph- and Model-Driven Engineering
Übungsblatt 2

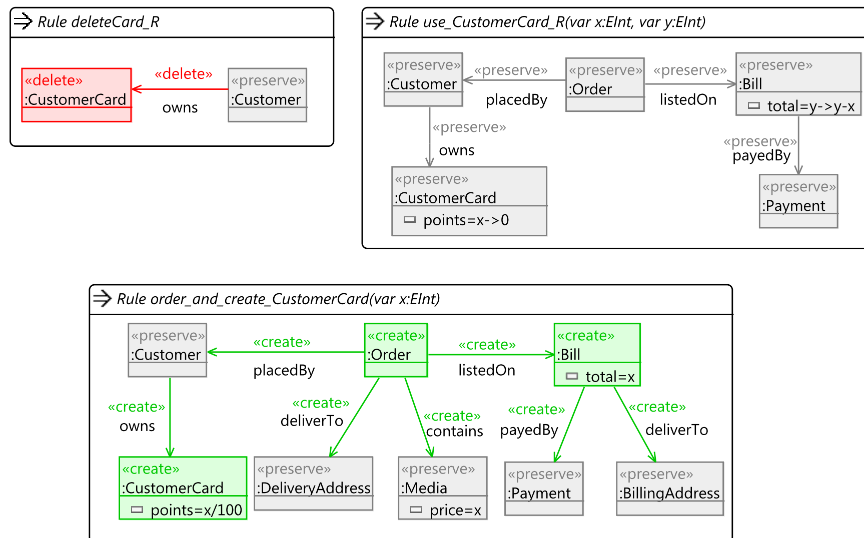

Abbildung 2: Drei Service Requests

- **use_Customer_Card_P:** Es werden keine Elemente erzeugt oder gelöscht. Die Bonuspunkte der Kundenkarte eines Kunden werden vom Betrag einer Rechnung abgezogen. Die Bonuspunkte der Kundenkarte werden auf 0 gesetzt.
- **create_Customer_Card_P:** Ein Kunde erhält eine neue Kundenkarte, die 0 Bonuspunkte enthält.
- **offer_Bonuspoints_P:** Ein Kunde bekommt die Bonuspunkte seiner Bestellung auf seine Kundenkarte gutgeschrieben. Hierbei wird 1% des Rechnungsbetrages zu dem bereits auf der Kundenkarte vorhandenen Betrag hinzugefügt.
- **create_Order_P:** Ein Kunde erstellt eine neue Bestellung. Diese enthält: Den georderten Artikel, eine Lieferadresse, eine Rechnung, die eine Bezahlmethode und eine Rechnungsadresse enthält. Der Preis des Artikels wird dem bereits auf der Rechnung vorhandenen Betrag hinzugefügt.

a) Erstellen Sie Regeln für die oben beschriebenen Services und stellen Sie diese grafisch dar.

b) Geben Sie für jeden vom Requestor gesuchten Service an, ob dieser direkt durch den Provider erfüllt werden kann. Begründen Sie Ihre Antworten informell, anhand der oben beschriebenen Services und geben Sie an, wie die notwendigen Abbildungen zu den von Ihnen in Aufgabenteil a) erstellten Regeln definiert sind bzw. warum die notwendige Abbildung nicht existieren kann.

c) Geben Sie an, ob der Service *order_and_create_CustomerCard* inkrementell durch den Provider mit den von Ihnen erstellten Regeln erfüllt werden kann. Falls ja, begründen Sie Ihre Antwort, indem Sie die beteiligten Schritte nennen und angeben, wie die jeweils notwendigen Abbildungen definiert sind. Argumentieren Sie andernfalls, warum dies mit Ihren Regeln nicht möglich ist.

Code für Aufgabe 3

```
1  import java.util.Set;
2
3  public class Mutex {
4    private Resource resource;
5    private Process first;
6    private Process second;
7    private Set<Process> requestedBy = new HashSet<Process>();
8    private Process nextGrantedProcess = null;
9
10   public Mutex(Resource resource, Process first, Process second) {
11     this.resource = resource;
12     this.first = first;
13     this.second = second;
14     nextGrantedProcess = first;
15   }
16
17   public void requestResource(Process proc) {
18     requestedBy.add(proc);
19   }
20
21   public void revokeRequest(Process proc) {
22     requestedBy.remove(proc);
23   }
24
25   public boolean isAccessGranted(Process proc) {
26     return (requestedBy.size()==1 && requestedBy.contains(proc));
27   }
28
29   public Process getGrantedProcess() {
30     return nextGrantedProcess;
31   }
32
33   public void switchGrantedProcess(Process proc) {
```

```
34       if (proc == first) {
35         nextGrantedProcess = second;
36       } else {
37         nextGrantedProcess = first;
38       }
39     }
40     public Resource getResource() {
41       return resource;
42     }
43   }
```

```
1  public abstract class Process extends Thread {
2
3    public void changeResource(Mutex mut) {
4
5      mut.requestResource(this);
6      while (!mut.isAccessGranted(this)) {
7        if (mut.getGrantedProcess() != this) {
8          mut.revokeRequest(this);
9          while (mut.getGrantedProcess() != this) {
10         }
11         mut.requestResource(this);
12       }
13     }
14
15     applyChange(mut.getResource());
16
17     mut.switchGrantedProcess(this);
18     mut.revokeRequest(this);
19   }
20
21   protected abstract void applyChange(Resource res);
22 }
```

```java
public class Resource {

  private int value = 0;

  public void increaseValue() {
    value++;
  }

  public void decreaseValue() {
    value--;
  }

  public int getValue() {
    return value;
  }
}
```

```java
public class UIProcess extends Process {

  @Override
  protected void applyChange(Resource res) {
    res.increaseValue();
  }

}
```

```java
public class LogicProcess extends Process {

  @Override
  protected void applyChange(Resource res) {
    res.decreaseValue();
  }

}
```