Objektorientierte Programmierung Blatt 5

Institut für Softwaretechnik und Programmiersprachen | Sommersemester 2024
Matthias Tichy, Raphael Straub und Florian Sihler

Abgabe (git) bis
2. Juni 2024

Inheritance, Interfaces, and Generics

6 ⭐ 2 🟦 4 🟢

- Das Konzept von Vererbung verstehen und von Komposition unterscheiden können
- das Konzept von Sichtbarkeit verstehen
- Ein Blick auf Generics
- Enumerationen kennenlernen

Aufgabe 1: Structuring My Code

Durch unseren Ausflug in die wundervolle Welt der Objekt-Orientierung haben wir bereits einige Konzepte kennengelernt, die es uns erlauben unsere Software zu strukturieren. In dieser Aufgabe wollen wir nun einen ersten Blick auf die diversen Möglichkeiten werfen, durch die wir Klassen miteinander Kombinieren können.

a) Vererbung ⭐⭐

Vererbung erlaubt es uns eine Klasse zu definieren, die die Attribute und Methoden einer anderen (Super-/Eltern-)Klasse erbt. Dies bedeutet, dass die erbende Klasse alle Attribute und Methoden der Basisklasse übernimmt, diese aber bei Bedarf überschreiben kann.

Betrachten Sie für die Aufgabe die folgende Klasse, die Sie auch unter `src/main/java/exercise1/Animal.java` in Ihrem Repository finden können:

```java
public class Animal {
   private String name;
   private int age;

   public Animal(final String name, final int age) {
      this.name = name;
      this.age = age;
   }

   public String getName() {
      return name;
   }

   public int getAge() {
      return age;
   }

   public void print() {
      System.out.println("Ich bin ein Tier, mein Name ist " + name
            + " und ich bin " + age + " Jahre alt.");
```

```
23      }
24  }
```

Schreiben Sie die drei Klassen `Dog`, `Cat` und `Penguin`, welche von der Klasse `Animal` erben. Jede dieser Klassen soll die Methode `Animal::print` so überschreiben, dass Sie zusätzlich zum Namen und dem Alter auch die Art des Tieres ausgibt! Der Pinguin soll dabei noch ein weiteres Attribut `cuteness` haben, welches ebenfalls von der Methode `print` ausgegeben werden soll. Achten Sie auf geeignete Sichtbarkeiten und Konstruktoren für die jeweiligen Klassen!

b) Sichtbarkeiten und Komposition ★★

Im Gegensatz zur Vererbung, bei der eine Klasse die Attribute und Methoden einer anderen Klasse erbt, kann eine Klasse auch Attribute mit dem Typ von (eventuell anderen) Klassen besitzen, was wir als Komposition bezeichnen.

In dieser Aufgabe möchten wir ein Regal implementieren. Wie schon auf dem letzten Blatt möchten wir, dass Sie passende Klassen erstellen, die alle notwendigen Attribute und Methoden besitzen, verlangen aber nicht, dass Sie die Methoden tatsächlich implementieren — versuchen Sie hierbei explizit sinnvolle Attribute und Methoden zu finden. Dieses Mal achten wir dabei besonders

1. auf Ihre Verwendung von Modifikatoren (sowohl für die Sichtbarkeit, wie `public` und `private` als auch für die Veränderbarkeit von Variablen wie `final`),
2. darauf, welche Eigenschaften Sie in welchen Klassen implementieren und
3. wie sie die Klassen miteinander kombinieren.

Wir definieren uns ein Regal wie folgt:

Ein Regal besteht aus einer festen Anzahl an Fächern auf die Gegenstände gestellt und von der Gegenstände genommen werden können. Jedes Regal kann eine unterschiedliche Anzahl an Fächern besitzen, die bei der Erstellung des Regals einmalig festgelegt wird und anschließend nicht mehr verändert werden kann. Jedes Fach kann dabei ebenfalls eine feste, für alle Fächer eines einzelnes Regal identische Anzahl an Gegenstände aufnehmen. Wir unterscheiden drei Arten von Gegenständen: Bücher, Vasen und Pflanzen. Ein Buch hat jeweils eine Anzahl an Seiten, ein Autor und einen Titel (alle drei Eigenschaften sind unveränderlich). Eine Vase hat eine Höhe, einen Durchmesser und eine Farbe (alle drei Eigenschaften sind unveränderlich). Eine Pflanze hat eine Höhe, eine Art und eine Farbe (alle drei Eigenschaften sind veränderlich). Jedes Fach kann unterschiedliche Gegenstände aufnehmen (das heißt, es lassen sich sowohl Bücher, Vasen als auch Pflanzen auf das selbe Fach stellen).

Für diese Aufgabe gibt es erneut keine Tests, bei Fragen dürfen Sie sich aber gerne jederzeit an Ihren jeweiligen Tutor wenden! Verwenden Sie Kommentare direkt im Code um Ihre Entscheidungen zu begründen — dabei müssen Sie sich aber nicht unnötig wiederholen, sollte eine Begründung zur Wahl der Sichtbarkeit beispielsweise für alle Attribute einer Klasse aus dem selben Grund `protected` sein, so reicht es dies einmal zu erwähnen und hier auf die anderen Variablen zu verweisen.

c) Alternativen ●

Java bietet verschiedene Möglichkeiten ein Programm zu strukturieren, die auf den ersten Blick vielleicht erschlagend wirken. Finden Sie für jeden der folgenden Strukturmechanismen daher mindestens ein (weiteres) Beispiel, bei dem Sie sich vorstellen könnten, dass dieser in einem realen Programm sinnvoll wäre:

1. Vererbung
2. Komposition
3. Generics
4. Interfaces

Aufgabe 2: An Enumerating Calendar

Vielleicht erinnern Sie sich noch an die Summen-/Aufzählungstypen aus Haskell, die Sie in der Veranstaltung *Grundlagen der praktischen Informatik* kennengelernt haben (sofern sie diese bereits hören durften). Hier erzeugen wir so:

```
data Color = Red | Green | Blue
```

einen Typ `Color`, der die Werte `Red`, `Green` und `Blue` annehmen kann. In Java gibt es hierfür die sogenannten Enumerationen, die wir in dieser Aufgabe kennenlernen wollen.

Auf dem Papier sind Enumerationen in Java nur besondere Klassen, deren möglichen Instanzen wir direkt angeben. Um eine Farbe mit den Werten *red*, *green* und *blue* zu definieren, schreiben wir:

```
public enum Color {
   RED, GREEN, BLUE
}
```

Dabei ist es in Java (je nach Style-Guide) üblich, aber nicht notwendig die Werte in Großbuchstaben zu schreiben. Der Zugriff auf die Werte erfolgt dann über `Color.RED`, `Color.GREEN` und `Color.BLUE`.

a) Die Tage der Woche

Erstellen Sie eine Enumeration `DayOfWeek` für die Wochentage, mit den Werten `SUNDAY`, `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, `FRIDAY`, und `SATURDAY` sein. Idealerweise erstellen Sie die Enumeration in einer separaten Datei mit dem Namen `DayOfWeek.java`.

b) Ein Datum

Erstellen Sie eine Klasse `Date`, die ein einzelnes Datum repräsentiert. Ein Datum besteht dabei aus einem Tag, einem Monat und einem Jahr.

c) Schaltjahre

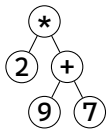
Erstellen Sie eine Methode `isLeapYear`, die überprüft, ob ein gegebenes Jahr ein Schaltjahr ist. Ein Jahr ist ein Schaltjahr, wenn es durch 4 teilbar ist, es sei denn, es ist durch 100 teilbar. In diesem Fall ist es nur ein Schaltjahr, wenn es auch durch 400 teilbar ist.

d) Der Tag der Woche

Erstellen Sie eine Methode `DayOfWeek getDayOfWeek()`, die den Wochentag eines Datums berechnet. Hierfür können Sie die Doomsday-Methode verwenden!

Tipp: Wenn Sie nicht weiterkommen, versuchen Sie sich zum Beispiel zunächst eine Hilfsfunktion `int daysBetween(Date other)` zu schreiben, die die Anzahl der Tage zwischen zwei Daten berechnet oder informieren Sie sich über die „Julian Day Number“.

Aufgabe 3: My First Expression Tree



In dieser Aufgabe möchten wir gleich mehrere Dinge kennenlernen und uns einen Ausdrucksbaum in Java basteln. Diese Struktur erlaubt es uns mathematische Ausdrücke wie beispielsweise $2 * (9 + 7)$ darzustellen und einfach zu evaluieren. So ist die Baumstruktur für diesen Ausdruck einmal rechts dargestellt.

Wir beginnen unsere Reise dabei mit dem folgenden Interface, welches Sie auch auch in Ihrem Repository unter `src/main/java/exercise3/ExpressionNode.java` finden können:

```java
public interface ExpressionNode {
  Optional<Integer> evaluate();
}
```

Der Einfachheit wegen beschränken wir uns hier auf die Auswertung von ganzen Zahlen, die aber in einem `Optional` gekapselt sind um mit dem Scheitern einer Berechnung umgehen zu können!

a) Einen Ausdrucksbaum Konstruieren — Values ★

Implementieren Sie eine Klasse `ValueNode`, die eine Zahl speichert und das Interface `ExpressionNode` implementiert. Die Methode `evaluate` soll dabei einfach den Wert des Knotens zurückgeben.

Beispiel:

```java
var tree = new ValueNode(1);
tree.evaluate(); // → Optional.of(1)
```

Dieser Baum entspricht also dem mathematischen Ausdruck 1.

b) Einen Ausdrucksbaum Konstruieren — Addition ★★■

Implementieren Sie nun eine Klasse `AdditionNode`, die bis zu zwei `ExpressionNodes` als Kindknoten besitzt, ebenfalls `ExpressionNode` implementiert und mit `evaluate()` die Summe deren Werte berechnet. Sollte der Knoten keine Kindknoten besitzen oder mindestens einer der Kindknoten ein leeres `Optional` zurückliefern, so soll `evaluate` ebenfalls ein leeres `Optional` zurückgeben. Sollte der Knoten nur einen Kindknoten besitzen, so soll die Methode `evaluate` den Wert dieses Kindknotens direkt zurückgeben.

Beispiel:

```java
var tree = new AdditionNode(
   new ValueNode(1),
   new ValueNode(2)
);
tree.evaluate(); // → Optional.of(3)
```

Dieser Baum entspricht also 1 + 2 = 3.

c) Einen Ausdrucksbaum Konstruieren — Multiplikation ★■

Wiederholen Sie die Schritte aus Teilaufgabe 3b für die Multiplikation (falls Sie bereits über anonyme Funktionen in Java Bescheid wissen, steht es Ihnen frei eine allgemeinere Implementation zu finden die nicht den gesamten Code dupliziert). Der Multiplikationsknoten soll dabei den Namen `MultiplicationNode` tragen.

Beispiel:

```java
var tree = new MultiplicationNode(
   new ValueNode(21),
   new AdditionNode(
      new ValueNode(1),
      new ValueNode(1)
   )
);
```

```
tree.evaluate(); // → Optional.of(42)
```

Dieser Baum entspricht also $21 \cdot (1 + 1) = 42$.

d) Einen Ausdrucksbaum Konstruieren — Refactoring 🟢🟢

So wie die `AdditionNode` und die `MultiplicationNode` können wir viele „Binäroperatoren“ definieren, die (teils bis-zu) zwei Kindknoten besitzen und diese auf eine ähnliche Weise verarbeiten (also beispielsweise ein leeres `Optional` direkt weitergeben). Erstellen Sie eine neue Klasse `BinaryOperatorNode`, die die Gemeinsamkeiten der beiden Klassen abstrahiert und schreiben sie nun die `AdditionNode` und `MultiplicationNode` so um, dass sie von `BinaryOperatorNode` erben.

e) Einen Ausdrucksbaum Konstruieren — Pretty-Printing 🟦🟦

Erweitern Sie nun das Interface `ExpressionNode` aus Teilaufgabe 3a um eine *neue* Methode `String prettyPrint()`, die den Ausdruck des Baumes in einer menschenlesbaren Form zurückgibt. Achten Sie dabei auf korrekte Klammerung!

So soll beispielsweise der Baum aus Teilaufgabe 3b den String `"1 + 2"` liefern, der Baum aus Teilaufgabe 3c den String `"21 * (1 + 1)"` (je nach Implementation können Sie auch mehr Klammern erzeugen).

f) Einen Ausdrucksbaum Konstruieren — Diskussion 🟢🟢

Sehen Sie bei der von uns gewählten Struktur ein Problem, wenn wir nun:

1. weitere Methoden zu `ExpressionNode`,
2. neue Operationsknoten oder
3. die Unterstützung neuer Werte (wie Zeichenketten)

hinzufügen wollen? Verwenden Sie ruhig Ihre Erfahrungen aus den vorherigen Aufgaben (z.B. Teilaufgabe 3e) um Ihre Antwort zu begründen.