Übungsblatt 04

4 Aufgaben, 20 Punkte

Objektorientierte Modellierung und Programmierung (inf031) Sommersemester 2020
Carl von Ossietzky Universität Oldenburg, Fakultät II, Department für Informatik

Dr. C. Schönberg

Ausgabe: 2020-05-12 14:00 **Abgabe:** 2020-05-19 12:00

Aufgabe 1: *Funktionsketten* **(4 Punkte)**

In dieser Aufgabe geht es um den Aufbau von Funktionenketten. Funktionenketten sind Funktionen, die eine geordnete Menge von mathematischen Funktionen enthalten, die der Reihe nach auf den berechneten Werten der Vorgängerfunktion ausgeführt werden, startend mit einem gegebenen Wert.

Als Funktionen werden dabei in dieser Aufgabe beliebige Funktionen betrachtet, die einen **`double`**-Wert auf einen anderen **`double`**-Wert abbilden. Die Reihenfolge der Ausführung der Funktionen soll beliebig angeordnet werden können. Die Funktionenkette soll prinzipiell beliebig lang werden können.

Beispiel: Es gebe die Funktionen f, g, h und p.
Die Funktionskette k_1 bestehe aus drei Funktionen in der Reihenfolge f, g und h, also $k_1 = f \circ g \circ h$. Dann gilt: $k_1(x) = f\left(g\left(h\left(x\right)\right)\right)$.
Die Funktionskette k_2 bestehe aus vier Funktionen in der Reihenfolge g, f, g und p, also $k_2 = g \circ f \circ g \circ p$. Dann gilt: $k_2(x) = g\left(f\left(g\left(p\left(x\right)\right)\right)\right)$.

Sie sollen sich in dieser Aufgabe Gedanken darüber machen, wie solche Funktionenketten implementiert werden können. Letztendlich soll es möglich sein, mit den von Ihnen zu entwickelnden Klassen Programme folgender Gestalt zu schreiben:

```
1 Function chain = new CosineFunction(new SquareRootFunction(new SineFunction()))
      ;
2 double x = IO.readDouble("Value: ");
3 System.out.println(chain.calculate(x));
```

In diesem Beispiel wird (bei der Definition entsprechender Klassen `Function`, `CosineFunction`, `SquareRootFuction` und `SineFunction`) eine Funktionenkette aufgebaut und durch den Aufruf von `chain.calculate` aktiviert. Berechnet wird der Wert $\cos\left(\sqrt{\sin\left(x\right)}\right)$ für den vom Benutzer eingegebenen **`double`**-Wert x.

Andere mögliche Funktionenketten sind (bei entsprechenden Klassen):

```
1 Function chain1 = new SquareRootFunction(new SquareRootFunction(new
      TangentFunction()));
2 Function chain2 = new SineFunction(new SquareFunction(new IdentityFunction()));
```

Aufgabe: Überlegen Sie aufbauend auf den Konzepten der Polymorphie und des dynamischen Bindens ein Konzept zur Realisierung obiger Funktionenketten. Implementieren Sie neben der notwendigen Klasse `Function` zwei konkrete Klassen `SquareFunction` und `SineFunction`.

Hinweis: Sie dürfen die Methoden der Klasse `java.lang.Math` verwenden.

Aufgabe 2: *Zahlenfolgen* **(1 + 1 + 1 + 2 + 3 Punkte)**

Quellenangabe: Diese Aufgabe stammt aus dem Buch „Programmieren mit Java“ von Reinhard Schiedermeier, erschienen 2010 im Verlag Pearson Studium.

In dieser Aufgabe geht es um die Modellierung von unendlichen Zahlenfolgen. Als Zahlenfolge wird in der Mathematik eine Auflistung von endlich oder unendlich vielen fortlaufend nummerierten Objekten bezeichnet. Die Objekte sind in dieser Aufgabe **`int`**-Werte.

a) Definieren Sie ein Interface `Sequence` für allgemeine unendliche Zahlenfolgen mit **`int`**-Werten. `Sequence` hat eine Methode `getNext()`, welche die nächste Zahl der Folge liefert.

b) Die natürlichen Zahlen $(1, 2, 3, \ldots)$ sind eine konkrete unendliche Zahlenfolge. Definieren Sie eine Klasse `Naturals`, die das Interface `Sequence` implementiert. Der erste Aufruf von `getNext` für ein `Naturals`-Objekt liefert 1, der nächste 2, dann 3 und so weiter.

c) Ein Filter ist eine Zahlenfolge, die keine eigene Zahlenquelle enthält. Ein Filter „ernährt“ sich stattdessen von einer anderen Zahlenfolge, deren Elemente er entweder durchlässt oder absorbiert. Definieren Sie eine abstrakte Klasse `Filter` (**`abstract class`** `Filter` **`implements`** `Sequence`) mit einem Konstruktor, dem als Parameter ein `Sequence`-Objekt – die Zahlenquelle – übergeben wird. Die Zahlenquelle wird in einem Attribut gespeichert.

d) Leiten Sie von der abstrakten Klasse `Filter` eine konkrete `Filter`-Klasse `ZapMultiples` ab. Deren Konstruktor erwartet neben der Zahlenquelle als weiteren Parameter eine Basiszahl. Der Filter absorbiert die Basiszahl und alle ganzzahligen Vielfachen der Basiszahl und gibt den Rest weiter.
Beispiel 1: Der mehrfache Aufruf der Methode `getNext` von
```
new ZapMultiples(3, new Naturals())
```
liefert folgende Zahlen: $1, 2, 4, 5, 7, 8, 10, \ldots$.
Beispiel 2: Der mehrfache Aufruf der Methode `getNext` von
```
new ZapMultiples(2, new ZapMultiples(3, new Naturals()))
```
liefert folgende Zahlen: $1, 5, 7, 11, 13, \ldots$ (Absorption aller Vielfachen von 2 und 3).

e) Leiten Sie von der abstrakten Klasse `Filter` eine weitere konkrete `Filter`-Klasse `Primes` ab, die Primzahlen in aufsteigender Reihenfolge berechnet. Der mehrfache Aufruf der Methode `getNext` von **`new`** `Primes()` soll also folgende Zahlen liefern: $2, 3, 5, 7, 11, 13, 17, \ldots$
Nutzen Sie dabei zur Ermittlung der Primzahlen den folgenden Algorithmus („Sieb des Eratosthenes“):

i) Nimm die Folge der natürlichen Zahlen und entferne die 1.

ii) Die erste Zahl p des Restes der Folge ist eine Primzahl.

iii) Entferne p und alle Vielfachen von p aus der Folge.

iv) Zurück zu Schritt ii).

Implementieren Sie diesen Algorithmus mit den in den Teilaufgaben b) und d) implementierten Klassen!

Aufgabe 3: *Vergleichbarkeit* **(4 Punkte)**

Schauen Sie sich das folgende Interface sowie die beiden folgenden Klassen an:

```
1  interface Comparable {
2    /**
3     * Vergleicht das aufrufende Objekt mit dem als Parameter uebergebenen
4     * Objekt; liefert: -1 falls das aufrufende Objekt kleiner ist als das
5     * Parameterobjekt, 0 falls beide Objekte gleich gross sind, 1 falls das
6     * aufrufende Objekt groesser ist als das Parameterobjekt.
7     */
8    public int compareTo(Comparable obj);
9  }
10
11 class Utils {
12
13   /**
14    * Liefert das ”kleinste” (auf der Basis der Comparable-Implementierung!)
15    * Element des Parameter-Arrays.
16    * Achtung: Man kann davon ausgehen, dass das Parameter-Array
17    * mindestens ein Element enthaelt (also weder null noch leer ist)
18    */
19   public static Comparable getMinimum(Comparable[] elements) {
20     //todo: implement this
21   }
22 }
23
24 class Integer {
25   protected int value;
26
27   public Integer(int value) {
28     this.value = value;
29   }
30
31   public int getValue() {
32     return value;
33   }
34 }
```

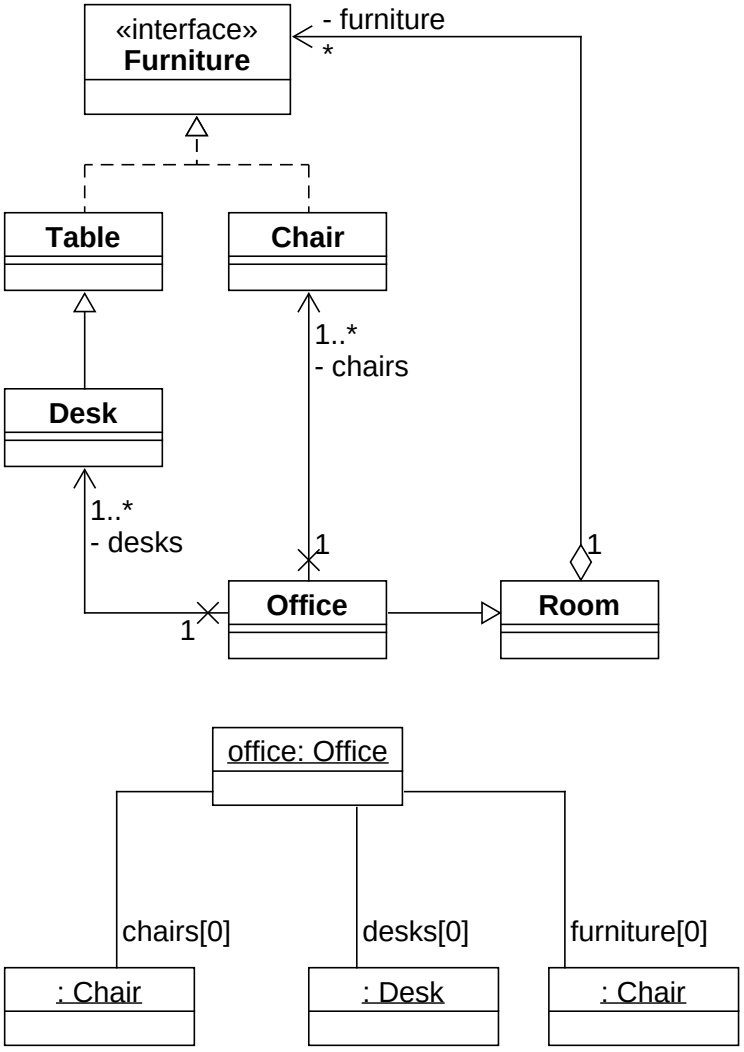
a) Implementieren Sie die Methode `getMinimum()` der Klasse `Utils`.

b) Leiten Sie von der Klasse `Integer` eine Klasse `ComparableInteger` ab, die das Interface `Comparable` implementiert.

c) Schreiben Sie ein kleines Testprogramm, das zunächst ein Array mit `ComparableInteger`-Objekten erzeugt und initialisiert, anschließend die Funktion `getMinimum()` mit diesem Array aufruft und den Wert des ermittelten kleinsten Elements auf den Bildschirm ausgibt.

Aufgabe 4: *Modellierung von Möbeln* **(4 Punkte)**

Gegeben sei folgendes UML-Klassendiagramm und das dazu passende UML-Objektdiagramm:



Implementieren Sie die modellierten Klassen und Interfaces in Java. Erstellen Sie außerdem ein kleines Testprogramm, das den im Objektdiagramm modellierten Zustand herstellt.