RWTH Aachen
Lehrgebiet Theoretische Informatik
Frohn–Reidl–Rossmanith–Sánchez–Ströder

WS 2013/14
Übungsblatt 1
28.10.2013

Übung zur Vorlesung Programmierung

Aufgabe T1

Für die Entwicklung einer Verwaltungssoftware für die Vereinigung Bayerischer Gemüsebauern e.V. ist es Ihre Aufgabe, Gemüsesorten objektorientiert zu modellieren. Die Klassen auf der linken Seite sollen sinnvoll in eine Vererbungshierarchie gebracht werden, dazu finden Sie mögliche Oberklassen auf der rechten Seite. Diskutieren Sie, welche Klassen wie in die Hierarchie eingeordnet werden sollten. Bedenken Sie dabei, daß die Verwaltungssoftware in Zukunft möglicherweise durch weiteres Gemüse erweitert wird!

Karotte	Zwiebel	Schnittlauch
Radieschen	Knoblauch	Endivie
Rettich	Pastinake	Feldsalat
Bärlauch	Chicoree	Petersilie

Wurzelgemüse	Gemüse	Grünes Gemüse
Weißes Gemüse	Blattgemüse	Herbstgemüse
Gelbes Gemüse	Wintergemüse	Zwiebelgemüse
Sommergemüse	Salate	Stangengemüse

Lösungsvorschlag

Die wohl sicherste Variante ist es, als Oberklassen die Klassen *Gemuese*, *Blattgemuese*, *Wurzelgemuese* und *Zwiebelgemuese* zu verwenden: diese Einteilung ist nämlich der Landwirtschaft entnommen und daher eindeutig. Andere Kombinationen von Klassen bieten diese Eindeutigkeit nicht, so ist etwa nicht klar, welches Gemüse nun unter etwa *Sommergemuese* oder *Herbstgemuese* fällt (ähliche Probleme treten bei Unterscheidung nach etwa Farbe oder Form auf).

Es sei aber betont, daß je nach Anwendungsfall andere Hierarchien die Problemlösung vereinfachen! Die Auswahl der zu modellierenden Objekte alleine reicht eben nicht aus, um diese Entscheidung zu treffen.

Aufgabe T2

Neben den in der Vorlesung vorgestellten Schleifen exisitiert in Java noch eine dritte Variante, die **do-while**-Schleife. Die Syntax **do** $\{P;\}$**while**$(B);$ bedeutet semantisch dabei folgendes: P wird zunächst einmal ausgeführt, danach wird die Bedingung B geprüft. Solange B als *true* evaluiert wird, wird P wiederholt ausgeführt.

Die drei Schleifenarten **while**, **do-while** und **for** haben alle die gleiche Ausdrucksstärke, d.h. man kann jede Schleife der einen Art in eine Schleife einer anderen Art übersetzen, ohne die Semantik des Programms zu ändern. Beispielsweise lässt sich eine **for**-Schleife auch als **do-while** Schleife ausdrücken:

Die Schleife **for**$(Z; B; I)\{P;\}$ ist äquivalent zu

```
{
  Z;
  if(B) {
    do {
      P; I;
    } while(B);
  }
}
```

Zeigen Sie, wie man auf ähnliche Weise

1. eine **do-while**-Schleife in eine **for**-Schleife,
2. eine **for**-Schleife in eine **while**-Schleife und
3. eine **while**-Schleife in eine **do-while**-Schleife umwandeln kann.

Lösungsvorschlag

1. Die Schleife **do** $\{P;\}$ **while** $(B);$ ist äquivalent zu $\{P;$ **for**$(;B;)$ $\{P;\}\}$.
2. Die Schleife **for**$(Z;B;I)\{P;\}$ ist äquivalent zu

   ```
   {
     Z;
     while(B) {
         P; I;
     }
   }
   ```

3. Die Schleife **while** (B) $\{P;\}$ ist äquivalent zu

   ```
   if (B) {
     do {
       P;
     } while (B);
   }
   ```

Aufgabe T3

Auf der Webseite der Vorlesung finden Sie die Datei `ArrayList.java`. Diese Klasse soll auf Basis eines Arrays die Datenstruktur 'Liste', genauer gesagt eine Liste natürlicher Zahlen, implementieren. In einer Liste werden Elemente *geordnet* gespeichert, das heißt [1,3,2] ist eine andere Liste als [2,1,3]. Außerdem können Listen *Duplikate* enthalten, so daß auch $[1, 3, 1]$ eine zulässige Liste ist.
Vervollständigen Sie die Implementierung der Methoden *compare* und *merge*, so daß sie die folgenden Aufgaben erfüllen:

- *compare*: Testet, ob diese Liste und eine andere als Argument übergebene Liste die gleichen Elemente in der gleichen Reihenfolge enthalten.

- *merge*: Verschmilzt diese Liste mit einer anderen als Argument übergebenen Liste und gibt als Ergebnis eine sortierte Liste zurück, die alle Elemente der beiden Listen enthält. Beide Listen müßen dazu sortiert sein, andernfalls ist der Rückgabewert *null*.

Selbstverständlich dürfen Sie zur Implementierung dieser Methoden auch die restlichen Methoden der Klasse *ArrayList* verwenden, auch dann, wenn diese noch nicht fertig implementiert sind.

```
public class ArrayList {
    private int[ ] data;
    public ArrayList() {
     this.data = new int[0];
    }
    private ArrayList(int[ ] dataArg) {
        this.data = dataArg;
    }
    public boolean isEmpty() {...}
    public int get(int index) {...}
    public ArrayList append(int newElement) {...}
    public int head() {...}
    public ArrayList tail() {...}
    public ArrayList concat(ArrayList tail) {...}
    public boolean compare(ArrayList that) {...}
    public boolean isSorted() {
        int last = 0;
        ArrayList remainder = this;
        while (!remainder.isEmpty()) {
            if (remainder.head() < last) {
                return false;
            }
            last = remainder.head();
            remainder = remainder.tail();
        }
        return true;
    }
    public ArrayList merge(ArrayList that) {...}
}
```

Lösungsvorschlag

```
public class ArrayList {
    private int[ ] data;
    public ArrayList() {
        this.data = new int[0];
    }
    private ArrayList(int[ ] dataArg) {
        this.data = dataArg;
    }
```

```
private void error(String errorMessage) {
    System.err.println(errorMessage);
}
public int length() {
    return this.data.length;
}
public boolean isEmpty() {
    return this.length() == 0;
}
public int get(int index) {
    if (index < 0 || index >= this.length()) {
        error("Unzulaessiger Index.");
        return -1;
    }
    return this.data[index];
}
public ArrayList append(int newElement) {
    if (newElement < 0) {
        error("Unzulaessiges Element.");
        return null;
    }
    int newLength = this.length() + 1;
    int[ ] newData = new int[newLength];
    for (int i = 0; i < this.length(); i++) {
        newData[i] = this.get(i);
    }
    newData[newLength - 1] = newElement;
    return new ArrayList(newData);
}
public int head() {
    if (this.isEmpty()) {
        error("Aufruf von head auf einer leeren Liste.");
        return -1;
    }
    return this.get(0);
}
public ArrayList tail() {
    if (this.isEmpty()) {
        error("Aufruf von tail auf einer leeren Liste.");
        return null;
    }
    ArrayList result = new ArrayList();
    for (int i = 1; i < this.length(); i++) {
        result = result.append(this.get(i));
    }
    return result;
}
public ArrayList concat(ArrayList tail) {
```

```
    ArrayList result = this;
    ArrayList remainder = tail;
    while (!remainder.isEmpty()) {
        result = result.append(remainder.head());
        remainder = remainder.tail();
    }
    return result;
}
public boolean compare(ArrayList that) {
    ArrayList remainderOfThis = this;
    ArrayList remainderOfThat = that;
    while (!remainderOfThis.isEmpty() &&!remainderOfThat.isEmpty()) {
        if (remainderOfThis.head() != remainderOfThat.head()) {
            return false;
        }
        remainderOfThis = remainderOfThis.tail();
        remainderOfThat = remainderOfThat.tail();
    }
    return remainderOfThis.isEmpty() && remainderOfThat.isEmpty();
}
public boolean isSorted() {
    int last = 0;
    ArrayList remainder = this;
    while (!remainder.isEmpty()) {
        if (remainder.head() < last) {
            return false;
        }
        last = remainder.head();
        remainder = remainder.tail();
    }
    return true;
}
public ArrayList merge(ArrayList that) {
    if (!this.isSorted() ||!that.isSorted()) {
        error("Verschmelzen unsortierter Listen nicht moeglich.");
        return null;
    }
    ArrayList result = new ArrayList();
    ArrayList remainderOfThis = this;
    ArrayList remainderOfThat = that;
    while (!remainderOfThis.isEmpty() &&!remainderOfThat.isEmpty()) {
        if (remainderOfThis.head() < remainderOfThat.head()) {
            result = result.append(remainderOfThis.head());
            remainderOfThis = remainderOfThis.tail();
        } else {
            result = result.append(remainderOfThat.head());
            remainderOfThat = remainderOfThat.tail();
        }
```

```
        }
        result = result.concat(remainderOfThis);
        result = result.concat(remainderOfThat);
        return result;
    }
}
```

Aufgabe H1 (5+5 Punkte)[1]

Im Folgenden finden Sie die Klassen *Person* (eine andere als diejenige aus der Vorlesung!) und *Vehicle*.

1. Implementieren Sie die Methode *calculateWeight* der Klasse *Vehicle*: dazu soll das Gesamtgewicht der Insassen zu dem Eigengewicht des Fahrzeugs addiert und zurückgegeben werden.

2. Erstellen Sie eine Unterklasse *Car* von *Vehicle*: ein Auto hat in diesem Kontext immer fünf Sitzplätze, aber ein variables Eigengewicht. Außerdem gilt folgende Einschränkung für die Methode *placePerson*: eine Person kann nur auf dem Platz Nummer 3 (der mittlere Platz auf der Rückbank) plaziert werden, wenn Sie kleiner als 1,9 m ist. Größere Personen werden schlicht nicht plaziert.

```
class Person {
  private String name;
  private double weight; // Gewicht in Kilogramm
  private int height; // Hoehe in Zentimetern
  public Person(String n, double w, int h) {
    name = n;
    weight = w;
    height = h;
  }
  public double getWeight() {
    return weight;
  }
  public int getHeight() {
    return height;
  }
  public String getName() {
    return name;
  }
}

class Vehicle {
  private Person[ ] seats;
  private double weight; // Gewicht des Fahrzeugs in Kilogramm
  public Vehicle(int numSeats, double w) {
    seats = new Person[numSeats];
    weight = w;
  }
  public boolean isFree(int seatNumber) {
```

```
    if(seatNumber < 0 || seatNumber >= seats.length) {
      return false;
    }
    return seats[seatNumber] == null;
  }
  public void placePerson(Person person, int seatNumber) {
    if(seatNumber >= 0 && seatNumber < seats.length && isFree(seatNumber)) {
      seats[seatNumber] = person;
    }
  }
  public double calculateWeight() {
    // TODO
  }
}
```

Lösungsvorschlag

```
class Vehicle {
  ...
  public double calculateWeight() {
    double res = weight;
    for(int i = 0; i < seats.length; i++) {
      if(!isFree(i)) {
        res += seats[i].getWeight();
      }
    }
    return res;
  }
}

class Car extends Vehicle {
  public Car(double weight) {
    super(5, weight);
  }
  public void placePerson(Person person, int seatNumber) {
    if(seatNumber == 3 && person.getHeight() >= 190) {
      return;
    }
    super.placePerson(person, seatNumber);
  }
}
```

Aufgabe H2 (10 Punkte)[1]

Auf der Webseite der Vorlesung finden Sie die Datei `ArrayList.java` (siehe Aufgabe T3). Vervollständigen Sie die Implementierung der Methoden *append*, *concat*, *get*, *head* und *tail*, so daß diese die in den zugehörigen Kommentaren beschriebenen Aufgaben erfüllen.

Lösungsvorschlag
Siehe Lösung T3.

Aufgabe H3 (10 Punkte)[1]
Wir entwickelten in der Vorlesung ein Programm, welches mithilfe von Backtracking das n-Damen-Problem zu lösen vermag. In dieser Aufgabe soll dieses Programm so erweitert werden, daß das schwierigere *n-Amazon Queen*-Problem gelöst wird. Eine *Amazon Queen* kann wie die übliche Dame im Schach gerade und diagonal ziehen, zusätzlich aber auch noch alle Züge eines Springers durchführen.
Führen Sie am besten eine Unterklasse von *Brett* ein, welche die *safePosition*-Methode geeignet überschreibt. Stellen Sie sodann fest, ob es eine Möglichkeit gibt, acht *Amazon Queens* auf ein reguläres 8×8-Schachbrett zu stellen. Was ist das kleinste n, für welches es möglich ist, n *Amazon Queens* auf ein $n \times n$-Schachbrett zu stellen, ohne daß sich zwei von ihnen bedrohen?

Lösungsvorschlag

```
class AmazonBrett extends Brett {

  public AmazonBrett(int n) {
    super(n);
  }

  boolean safePosition(int y, int x) {
    return super.safePosition(y, x) &&
           obenSprungLeer(y − 2, x + 1) &&
           obenSprungLeer(y − 2, x − 1) &&
           obenSprungLeer(y − 1, x − 2) &&
           obenSprungLeer(y − 1, x + 2);
  }

  boolean obenSprungLeer(int y, int x) {
    if(0 <= y && y < n && 0 <= x && x < n) {
      return (!brett[y][x]);
    }
    return true;
  }

}
```

Es stellt sich heraus, daß erst auf einem 10×10-Schachbrett eine Lösung existiert, welche in Abbildung 1 zu finden ist.

Organisatorisches
Die Übungen werden auf der Webseite der Vorlesung jeden Montag zur Verfügung gestellt. Die Abgabe erfolgt sowohl in schriftlicher als auch in elektronischer Form (Programme) bis zum Dienstag der folgenden Woche um 15 Uhr. Der Abgabekasten befindet sich im Informatikzentrum am Eingang Halifaxstraße. Es wird in Zweiergruppen abgegeben. Namen,

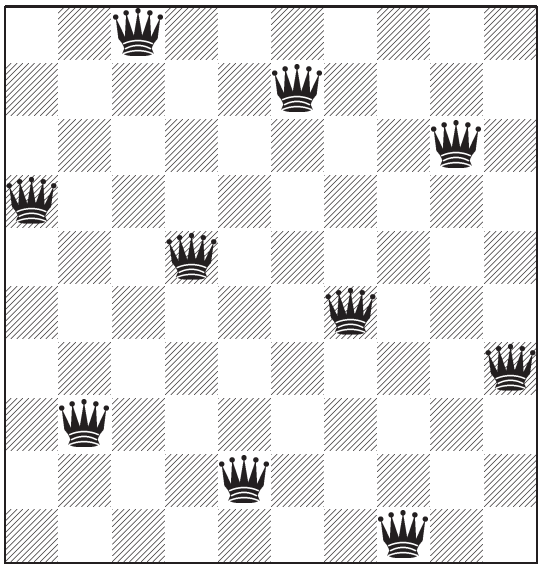

Abbildung 1: Zehn sich nicht bedrohende Amazon Queens

Matrikelnummern und Gruppennummern müssen auf jeder Abgabe angegeben werden. Die elektronische Abgabe wird direkt zum jeweiligen Tutor per E-mail gesendet.

[1]Bitte Quelltext für die Abgabe ausdrucken und zusätzlich per E-mail an den jeweiligen Tutor senden.