VU Grundlagen der Programmkonstruktion — Übungsteil

Organisatorisches

Wie sollen Ihre Lösungen aussehen

Beachten Sie, dass nicht die konkrete Lösung im Vordergrund steht, sondern der Weg, wie Sie zu dieser gelangen. Geben Sie daher bei Aufgaben immer auch den Rechenweg an. Bei jeder Aufgabe ist ein Beispiel angeführt, wie dieser aussehen soll. Ohne Angabe des Rechenwegs erhalten Sie für die Aufgabe keine Punkte.

Geben Sie ihre Lösungen im TUWEL als (bevorzugt) Text-Dokument (Erweiterung `.txt`) oder als PDF-Dokument (Erweiterung `.pdf`) ab.

Gruppenarbeiten

Sie können die Aufgaben in Gruppenn zu 2-6 Personen lösen. Die Zusammenstellung und Organisation der Gruppen ist dabei Ihnen überlassen.

Wenn Sie die Aufgabe in einer Gruppe gelöst haben, so soll EIN Gruppenmitglied die Aufgabe in TUWEL abgeben. Die Namen MIT Matrikelnummern aller Gruppenmitglieder muss im abgegebenden Dokument am Anfang ihres Dokuments vermerkt sein.

Abgabe Ihrer Lösungen

Die Abgabe Ihrer Lösungen erfolgt elektronisch via TUWEL (TU Wien E-Learning Center). Sie finden einen Link dorthin am Beginn der LVA-Seite im TISS ("Zum TUWEL Online-Kurs").

Die Abgabe muss via TUWEL bis spätestens 17. November 2011, 12:00, erfolgen. Nachträgliche Abgaben werden nicht berücksichtigt.

Fragen

Bei Fragen zu TUWEL wenden Sie sich bitte zunächst an das TUWEL-Hilfe-System und Ihre Kollegen. Sollte in der Angabe etwas unklar sein, sehen Sie bitte im TUWEL nach, ob dort bereits ergänzende Anmerkungen zum Übungsteil stehen. Wichtige Ankündigungen zum Übungsteil werden auch in der Vorlesung und online bekannt gegeben.

3. Übungsaufgabe

1 Objekte und Interfaces

Gegeben ist folgende Klasse Linie:

```
public class Linie {
    // Gerade von p1 nach p2
    private Punkt p1;
    private Punkt p2;

    public Linie(Punkt initP1, Punkt initP2) {
        p1 = initP1;
        p2 = initP2;
    }
}
```

Die Klasse Punkt entspricht dabei der Klasse in den Vorlesungsfolien aus Einheit 6.

Zusätzlich sind folgende Interfaces gegeben:

```
public interface Verschiebbar {
    // verschiebe Objekt um deltaX Einheiten nach rechts
    // und um deltaY Einheiten nach oben (wenn positiv)
    // bzw. nach links und unten (wenn negativ)
    void verschiebe(double deltaX, double deltaY);
}

public interface Skalierbar {
    // Skaliert Objekt um faktor
    void skaliere(double faktor);
}
```

Aufgabe - Teil 1

Geben Sie die Klasse `Linie` an, nachdem Sie die Interfaces `Verschiebbar` und `Skalierbar` implementiert haben.

Aufgabe - Teil 2

Implementieren Sie das Interface `Skalierbar` auch für die Klassen `Punkt` und `Scheibe` aus den Folien aus Einheit 6 und geben Sie die neuen Methoden an.

Beispiel

Ihre Lösung sollte für folgende Objekte folgende Ergebnisse liefern:

```
Linie l = new Linie( new Punkt( 1.0, 1.0 ), new Punkt( 2.0, 3.0 ) );

l.verschiebe(-1.0, 1.0); // ==> l == Linie ( (0.0, 2.0), (1.0, 4.0) )
l.skaliere(2.0);       // ==> l == Linie ( (0.0, 4.0), (2.0, 8.0) )

Punkt p = new Punkt( -1.0, -2.0 );
p.skaliere(3.0);       // ==> Punkt ( -3.0, -6.0 );

Scheibe s = new Scheibe ( 2.0, 1.0, 17.0 );
s.skaliere(2.0);       // ==> Scheibe ( 4.0, 2.0, 34.0 );
```

Lösung

Klasse Linie:

```
public class Linie implements Verschiebbar, Skalierbar{
    // Gerade von p1 nach p2
    private Punkt p1;
    private Punkt p2;

    public Linie(Punkt initP1, Punkt initP2) {
        p1 = initP1;
        p2 = initP2;
    }

    public void verschiebe(double deltaX, double deltaY) {
        p1.verschiebe(deltaX, deltaY);
        p2.verschiebe(deltaX, deltaY);
    }


    public void skaliere(double faktor) {
        p1.skaliere(faktor);
        p2.skaliere(faktor);

        /* Alternativ: Annahme Point mit getX/getY
         * p1 = new Point(p1.getX() * faktor, p1.getY() * faktor);
         * p2 = new Point(p2.getX() * faktor, p2.getY() * faktor);
         */
    }
}
```

Skaliere in Punkt:

```
    public void skaliere(double faktor) {
        x *= faktor;
        y *= faktor;
    }
```

Skaliere in Scheibe:

```
    public void skaliere(double faktor) {
        x *= faktor;
        y *= faktor;
        r *= faktor;
    }
```

2 Dynamisch vs. switch-case

Gegeben ist folgende Klasse:

```
public class Begruessung {
    public static final int FRAU = 0, MANN = 1, KIND = 2;

    public static String begruesse(int person, String name) {

        String gruss = "";
        String anrede = "";

        switch(person) {
            case KIND : gruss = "Hallo"; anrede = name; break;
            case FRAU : gruss = "Sehr geehrte"; anrede = "Frau " + name; break;
            case MANN : gruss = "Sehr geehrter"; anrede = "Herr " + name; break;
        }

        return gruss + " " + anrede;
    }
}
```

Wandeln Sie die Methode `begruesse(int person, String name)` so um, dass anstatt der Parameter und der switch-case-Anweisung eine Instanz des Interfaces `Person` verwendet wird. Erstellen Sie dazu drei Klassen `Mann`, `Frau`, `Kind`, die das folgende Interface `Person` implementieren:

```
public interface Person {
    String gruss();
    String anrede();
}
```

Aufgabe

Geben Sie in ihrer Lösung folgende Klassen und Methoden an:

1. Die Klassen `Kind`, `Mann` und `Frau`, und
2. die Methode `begruesse(Person person)`.

Beispiel

Diese Aufgabe ist ähnlich zu den Code-Beispielen des Interfaces `Beurteilung` in den Folien der siebenten Einheit.

Lösung

Eine zentrale Frage ist, was mit `String name` passieren soll. Die beste Lösung ist, eine Objektvariable einzuführen und den Namen im Konstruktor zu setzen:

```
public class Kind implements Person {
    private String name;

    public Kind(String initName) {
        name = initName;
    }

    public String gruss() {
        return "Hallo";
    }

    public String anrede() {
        return name;
    }
}

public class Frau implements Person {
    private String name;

    public Frau(String initName) {
        name = initName;
    }

    public String gruss() {
        return "Sehr geehrte";
    }

    public String anrede() {
        return "Frau " + name;
    }
```

```
}

public class Mann implements Person {
    private String name;

    public Mann(String initName) {
        name = initName;
    }

    public String gruss() {
        return "Sehr geehrter";
    }

    public String anrede() {
        return "Herr " + name;
    }
}

    public static String begruesse(Person person) {
        return person.gruss() + " " + person.anrede();
    }
```

3 Rekursive Datenstrukturen - IntStack - Teil 1

Ein Stack ist eine sehr einfache Datenstruktur, die zwei elementare Operationen kennt: `push` fügt ein Element dem Stack hinzu, `pop` nimmt ein Element vom Stack und gibt es zurück.

Gegeben ist die folgende Klasse `IntStack`, die mittels einer verketteten Liste (realisiert über die Klasse `Node`) einen Stack implementiert:

```
public class IntStack {
    private Node head = null;

    public void push(int n) {
        head = new Node(n, head);
    }

    public int pop() {
        int n = head.getValue();
        head = head.getNext();
        return n;
    }
}

class Node {
    private int value;
```

```
    private Node next;

    Node(int initValue, Node initNext) {
        value = initValue;
        next = initNext;
    }

    int getValue() { return value; }
    Node getNext() { return next; }
}
```

Sollte `head` den Wert `null` enthalten, so ist der Stack leer.

Aufgabe

Ergänzen Sie die Klasse `IntStack` um folgende Methoden:

1. `boolean isEmpty()` liefert `true`, wenn der Stack leer ist.
2. `void plus()` nimmt die beiden letzten Werte vom Stack (führt also zwei Mal `pop` aus) und legt ihre Summe auf den Stack (führt also ein Mal `push` aus).
3. `void neg()` nimmt den letzten Wert vom Stack, negiert ihn und legt ihn auf den Stack.

Sie benötigen keine Tests in `plus()` und `neg()`, ob der Stack genug Werte enthält.

Beispiel

Der Stack soll nach folgenden Befehlen folgendermaßen aussehen:

```
IntStack stack = new IntStack();

stack.push(1); // Stack = [ 1 ]
stack.push(2); // Stack = [ 2 , 1 ]
stack.push(3); // Stack = [ 3 , 2 , 1 ]

stack.plus(); // Stack = [ 5 , 1 ]
stack.neg();  // Stack = [ -5 , 1 ]
```

Lösung

```
public class IntStack {
    /* ... */
```

```
    public boolean isEmpty() {
        return head == null;
    }

    public void plus() {
        int arg0 = pop();
        int arg1 = pop();

        push(arg0 + arg1);
    }

    public void neg() {
        int arg = pop();
        push(-arg);
    }
}
```

4 Rekursive Datenstrukturen - IntStack - Teil 2

4.1 `equals(Object other)` und `toString()`

Fügen Sie der Klasse `IntStack` zwei Methoden `public String toString()` und `public boolean equals(Object other)` hinzu, wie sie in `Object` definiert sind. Verwenden Sie bei `equals` nicht `instanceof`!

`toString()` soll hierbei den Stack in folgendem Format ausgeben:

- Der leere Stack ist `[]`.
- Ein Stack mit exakt einem Wert 3 ist `[ 3 ]`.
- Ein Stack mit den Werten 1, 2 und 3 ist `[ 1 , 2 , 3 ]`.

`equals(Object other)` vergleicht zwei Stacks und gibt true zurück, wenn `other` ein `IntStack` ist und der Inhalt des Stacks in Reihenfolge und Inhalt dem aktuellen Stack entspricht. Ansonsten soll `false` zurückgegeben werden.

Einige Beispiele:

```
IntStack stack1 = new IntStack();
IntStack stack2 = new IntStack();

System.out.println(stack1.equals(stack2)); // ==> true

stack1.push(3);

System.out.println(stack1.equals(stack2)); // ==> false

stack2.push(3);

System.out.println(stack1.equals(stack2)); // ==> true

System.out.println(stack1.equals("Kein Stack")); // ==> false
```

Lösung

Lösungsmöglichkeit:

```
public class IntStack {
    /* ... */

    public boolean equals(Object other) {
     if(this == other) return true;
     if(other == null) return false;
        if(this.getClass() != other.getClass()) return false;

        // other ist von der selben Klasse wie this.
```

```
        IntStack that = (IntStack) other;

        Node n0 = this.head;
        Node n1 = that.head;

        while(n0 != null && n1 != null) {
            if(n0.getValue() != n1.getValue()) {
                // Wenn Werte nicht uebereinstimmen
                break; // beende Schleife.
            }

            n0 = n0.getNext();
            n1 = n1.getNext();
        }

        // Wenn in beiden das Ende der Liste erreicht wurde:
        return n0 == null && n1 == null;
    }

    public String toString() {
        String s = "[";

        Node n = head;

        while(n != null) {
            if(n != head) s += ",";

            s += " ";
            s += n.getValue();
            s += " ";

            n = n.getNext();
        }

        return s + "]";
    }
}

class Node {
    private int value;
    private Node next;

    Node(int initValue, Node initNext) {
        value = initValue;
        next = initNext;
    }
```

```
    int getValue() { return value; }
    Node getNext() { return next; }
}
```

4.2 Aufwandsabschätzung

Geben Sie an, wie hoch der Aufwand (konstant, logarithmisch, linear, quadratisch, exponentiell) für einen Aufruf der folgenden Methoden ist und begründen Sie ihre Antwort.

- `pop()`
- `plus()`
- `toString()`
- `equals(Object other)`

Lösung

- `pop()`: Konstant
- `plus()`: Konstant
- `toString()`: Linear - jedes Element im Stack wird 1x angeschaut.
- `equals(Object other)`: Linear - jedes Element im Stack wird 1x angeschaut.