Ludwig-Maximilians-Universität München
Institut für Informatik
Prof. Dr. M. Wirsing, M. Hammer, A. Rauschmayer

SS 2006
Übungsblatt 7

Informatik II Musterlösung

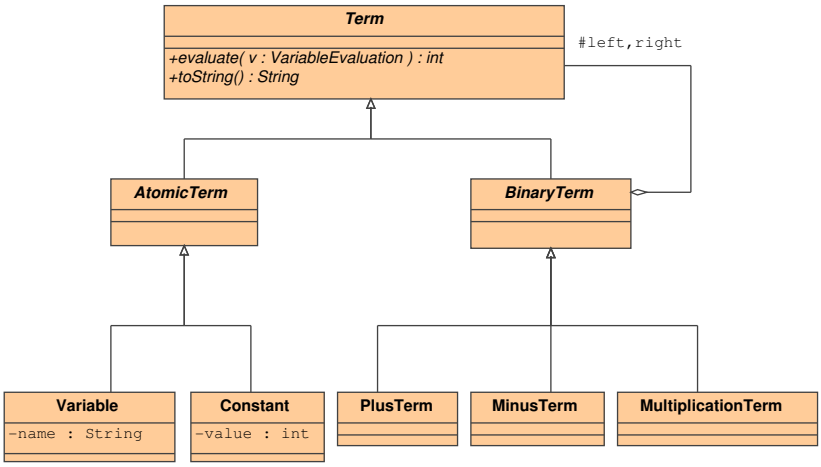
Zu jeder Aufgabe sind Dateien abzugeben, deren Namen rechts in der Aufgabenüberschrift stehen. Stellen Sie die Dateien in ein extra Verzeichnis (mit beliebigem Namen) und packen Sie dieses zu einem ZIP-Archiv. Geben Sie dieses, wie üblich, per UniWorx ab.

Aufgabe 7-1 — Vererbung (8 Punkte)

Ein *Term* über ganzen Zahlen $\mathbb{Z}$ ist wie folgt definiert: für $z \in \mathbb{Z}$ ist z ein Term, und ein Variablenname v ist ein Term. Sind A und B Terme, dann sind auch $A + B$, $A - B$ und $A * B$ Terme.

a) Geben Sie eine Klassenhierarchie, mit der Sie Terme darstellen können, als UML-Diagramm an. Beachten Sie, daß eine mehrstufige Hierarchie unter Umständen hilfreich sein kann.

Lösung: Ohne nicht-abstrakte Methoden:



b) Implementieren Sie Ihre Klassenhierarchie in Java, und geben Sie eine `toString() : String`-Methode an, die einen Term in lesbarer Form ausgibt.

c) Implementieren Sie eine Methode `evaluate(v : VariableEvaluation) : int`, die einen Term mit einer Variablenbelegung (also einer Funktion Variablenname $\rightarrow \mathbb{Z}$) auswertet. Eine Beispielimplementierung von `VariableEvaluation` finden Sie auf der Vorlesungs-Homepage.

Lösung:

`Term.java`

```
public abstract class Term {
    public abstract String toString();

    public abstract int evaluate(VariableEvaluation val);

    public static void main(String[] args) {
        Term t1 = new PlusTerm(
                new MinusTerm(new Constant(5), new Variable("a")),
                new Variable("b"));
        System.out.println(t1.toString());
        VariableEvaluation ve = new VariableEvaluation();
        ve.addValuation("a", 9);
```

```
        ve.addValuation("b", 2);
        ve.addValuation("c", 3);
        System.out.println(ve);
        System.out.println("Term 1: " + t1.evaluate(ve));
    }
}
```

AtomicTerm.java

```
/** Abstrakte Oberklasse fuer Terme, die atomar sind, also keine Unterterme
 * besitzen. Dient hier nur zur Organisation.
 */
public abstract class AtomicTerm extends Term {
}
```

BinaryTerm.java

```
/** Abstrakte Oberklasse fuer binaere Verknuepfungen (wie +). */
public abstract class BinaryTerm extends Term {
    protected Term left, right;

    public BinaryTerm(Term left, Term right) {
        this.left = left;
        this.right = right;
    }

    /**
     * Die erbenden Klassen sollen hier das Infix-Symbol bereitstellen, das in
     * der toString()-Methode verwendet wird */
    protected abstract String getInfixSymbol();

    /**
     * Da toString nur das Symbol von der Unterklasse braucht, koennen wir uns
     * hier Arbeit sparen */
    public String toString() {
        return "(" + left.toString() + ")" + getInfixSymbol() + "("
                + right.toString() + ")";
    }
}
```

Constant.java

```
public class Constant extends AtomicTerm {
    private int value;

    public Constant(int konstante) {
        this.value = konstante;
    }

    public String toString() {
        return Integer.toString(value);
    }

    public int evaluate(VariableEvaluation val) {
        return value;
    }
}
```

Variable.java

```
public class Variable extends AtomicTerm {
    private String name;

    public Variable(String name) {
        this.name = name;
    }

    public String toString() {
        return name;
    }

    /** Hier ist der (einzige) Punkt, wo die VariableEvaluation
     * tatsaechlich verwendet wird. */
    public int evaluate(VariableEvaluation val) {
        return val.getValuation(name);
    }
}
```

PlusTerm.java

```
public class PlusTerm extends BinaryTerm {
    public PlusTerm(Term left, Term right) {
        super(left, right);
    }

    protected String getInfixSymbol() {
        return "+";
    }

    public int evaluate(VariableEvaluation val) {
        return left.evaluate(val) + right.evaluate(val);
    }
}
```

MinusTerm.java

```
public class MinusTerm extends BinaryTerm {
    public MinusTerm(Term left, Term right) {
        super(left, right);
    }

    protected String getInfixSymbol() {
        return "-";
    }

    public int evaluate(VariableEvaluation val) {
        return left.evaluate(val) - right.evaluate(val);
    }
}
```

(MultiplicationTerm analog!)

Aufgabe 7-2 **Strategy-Pattern** (8 Punkte)

Eine häufig benutzte Anwendung des Strategy-Patterns in der Java-API findet sich in der Implementierung des Sortieralgorithmus für Arrays. Mit der Prozedur `Arrays.sort` der Klasse `java.util.Arrays`

wird ein Array sortiert. Für Arrays von `int` oder Objekte, die das Interface `Comparable` implementieren (wie z.B. `java.lang.String`), funktioniert dies direkt:

```
import java.util.Arrays;

public class ComparableExample {
    public static void main(String[] args) {
        Arrays.sort(args);
        System.out.println(Arrays.toString(args));
    }
}
```

Für beliebige andere Objekte wird das Strategy-Pattern eingesetzt. Dabei wird das Interface `java.util.Comparator` verwendet, um eine *totale Ordnung* zu repräsentieren:

Definition 1 ((totale) Ordnung) *Sei M eine Menge. Eine Relation $\preceq \subseteq M \times M$ heisst* Ordnung*, wenn gilt:*

- $\forall x \in M . x \preceq x$ *(Reflexivität)*
- $\forall x, y \in M . x \preceq y \wedge y \preceq x \implies x = y$ *(Antisymmetrie)*
- $\forall x, y, z \in M . x \preceq y \wedge y \preceq z \implies x \preceq z$ *(Transitivität)*

Eine Relation $\prec$ heisst total*, wenn $\forall x, y \in M . x \preceq y \vee y \preceq x$ gilt.*

Wir schreiben $a \prec b$ für $a \preceq b \wedge a \neq b$.

`java.util.Comparator` deklariert eine Methode `int compare(Object o1, Object o2)`, für deren Rückgabewert r gelten muss:

- $r < 0$ genau dann wenn `o1` $\prec$ `o2`
- $r = 0$ genau dann wenn `o1` $=$ `o2`
- $r > 0$ genau dann wenn `o2` $\prec$ `o1`

Ein Array wird dann sortiert, indem man `Arrays.sort(array,comparator)` mit einem Array `array` und einer `Comparator`-Instanz `comparator` aufruft.

a) Die Klasse `java.io.File` repräsentiert ein File im Dateisystem, das entweder ein konkretes File wie z.B. eine Textdatei oder ein Verzeichnis sein kann. Ist ein File-Objekt `f` ein Verzeichnis, bekommen Sie mit `f.listFiles()` die Liste der enthaltenen File-Objekte (als ein Array von `File`-Instanzen). Schreiben Sie drei `Comparator`-Implementierungen, die folgende totale Ordnungen repräsentieren:

   1. eine lexikographische Ordnung bzgl. des Filenamens, also $a \prec b$ gdw. der Name des Files a lexikographisch kleiner ist als der Name des Files b. Beachten Sie hierbei, dass `String` das `Comparable`-Interface implementiert.
   2. eine Ordnung, die Dateigrößen vergleicht.
   3. eine Ordnung, die das Datum der letzten Änderung vergleicht.

   Benutzen Sie die Java-API, um die entsprechenden Methoden von `java.io.File` zu finden:

   `http://java.sun.com/j2se/1.5.0/docs/api/java/io/File.html`

b) Schreiben Sie eine statische Methode, um die Files des aktuellen Verzeichnisses „.“ in den drei Varianten sortiert auszugeben.

Lösung:

`FileSorter.java`

```
import java.io.File;
import java.util.Arrays;
import java.util.Comparator;

/** Klasse, die nur die main-Methode enthaelt */
public class FileSorter {
    /** Fuer Klassen, die nur statische Methoden enthalten, kann man so
     * verhindern, dass eine (unnuetze) Instanz erzeugt wird. Der
     * Default-Konstruktor wird versteckt.
     */
    private FileSorter() {}

    public static void main(String[] args) {
        File[] td = new File(".").listFiles();

        System.out.println("Alphabetic:");
        Comparator c1 = new AlphabeticSortStrategy();
        Arrays.sort(td, c1);
        System.out.println(Arrays.toString(td));

        System.out.println("Size:");
        Comparator c2 = new SizeSortStrategy();
        Arrays.sort(td, c2);
        System.out.println(Arrays.toString(td));

        System.out.println("Change date:");
        Comparator c3 = new DateSortStrategy();
        Arrays.sort(td, c3);
        System.out.println(Arrays.toString(td));
    }
}
```

AlphabeticSortStrategy.java

```
import java.io.File;
import java.util.Comparator;

public class AlphabeticSortStrategy implements Comparator {
    public int compare(Object o1, Object o2) {
        File sf1 = (File) o1;
        File sf2 = (File) o2;
        /* Wir verwenden einfach die compareTo-Methode von String,
         * die im Comparable-Interface deklariert wird: */
        return sf1.getName().compareTo(sf2.getName());
    }
}
```

NumericValueComparator.java

```
import java.io.File;
import java.util.Comparator;

/** Abstrakte Oberklasse fuer Comparator-Implementierungen, die numerische
 * Werte vergleichen. */
public abstract class NumericValueComparator implements Comparator {
    /** Von der konkreten Subklasse zu implementieren: Soll die Differenz
     * zwischen den beiden Files bzgl. der Ordnung berechnen. */
```

```
    protected abstract long getDifference(File f1, File f2);

    public int compare(Object o1, Object o2) {
        File sf1 = (File) o1;
        File sf2 = (File) o2;
        long diff = getDifference(sf1, sf2);
        /* Wir koennen diff nicht zurueck geben, da ein long-Wert. Also
         * geben wir nur -1,0 oder 1 zurueck: */
        return (diff < 0)?-1:((diff > 0)?1:0);
    }

}
```

SizeSortStrategy.java

```
import java.io.File;

public class SizeSortStrategy extends NumericValueComparator {
    protected long getDifference(File f1, File f2) {
        return f2.length() - f1.length();
    }
}
```

DateSortStrategy.java

```
import java.io.File;

public class DateSortStrategy extends NumericValueComparator {
    protected long getDifference(File f1, File f2) {
        return f2.lastModified() - f1.lastModified();
    }
}
```

Aufgabe 7-3 **Warteschlange** (7 Punkte)

Gegeben sei die folgende Schnittstelle für Warteschlangen:

IntQueue.java

```
package queue;

/**
 * Schnittstelle einer Warteschlange, die ganzzahlige Werte aufnehmen kann.
 */
public interface IntQueue {
    /**
     * Wie viele Elemente befinden sich gerade in der Queue?
     */
    public int size();

    /**
     * Stelle einen Wert in die Queue.
     */
    public void enq(int value);

    /**
     * Hole einen Wert aus der Queue
     */
```

```
    public int deq();
}
```

a) Schreiben Sie eine Implementierung `SimpleIntQueue` dieser Schnittstelle. Hinweise:

- Eine Instanz dieser Klasse soll eine Kapazität haben, die während der gesamten Lebensdauer unverändert bleibt. Diese Kapazität soll dem Konstruktor als Argument übergeben werden.
- Überprüfen Sie, wo nötig, Verbedingungen per `assert`.
- Die Warteschlangen-Elemente sollen in einem `int`-Array abgelegt werden.

Lösung:

`SimpleIntQueue.java`

```
package queue;

import java.util.Arrays;

public class SimpleIntQueue implements IntQueue {
    protected int[] data;
    protected int tail = 0;
    protected int size = 0;

    public SimpleIntQueue(int initialCapacity) {
        assert initialCapacity >= 0;
        this.data = new int[initialCapacity];
    }

    public SimpleIntQueue() {
        this(4);
    }

    public int size() {
        return this.size;
    }
    public void enq(int value) {
        assert this.size < data.length; // schon voll?
        this.data[(this.tail + this.size) % this.data.length] = value;
        this.size++;
    }

    public int deq() {
        assert this.size > 0;
        int value = this.data[tail];
        this.tail = (this.tail + 1) % this.data.length;
        this.size--;
        return value;
    }
    @Override
    public String toString() {
        return tail+" "+Arrays.toString(this.data)+" "+size;
    }

    /**
     * Wird nur für Testzwecke eingesetzt, deshalb package-private
     */
    int getCapacity() {
        return this.data.length;
```

```
    }
}
```

b) Schreiben Sie eine Unterklasse `ResizableIntQueue` von `SimpleIntQueue`, die die Methode `enq()` so überschreibt, dass die Kapazität automatisch verdoppelt wird, wenn die Warteschlange voll ist.

Lösung:

`ResizableIntQueue.java`

```
package queue;

public class ResizableIntQueue extends SimpleIntQueue {
    public ResizableIntQueue() {
        super();
    }

    public ResizableIntQueue(int initialCapacity) {
        super(initialCapacity);
    }

    @Override
    public void enq(int value) {
        if (this.size >= this.data.length) {
            expandCapacity();
        }
        super.enq(value);
    }

    private void expandCapacity() {
        // Wir können nicht einfach nur das alte Array in das neue kopieren,
        // da das Array geteilt ist durch den "Ring-Sprung" vom Ende an den Anfang.
        int[] newData = new int[this.data.length * 2];
        for(int i=0; i < this.size; i++) {
            newData[i] = this.data[(this.tail + i) % this.data.length];
        }
        this.tail = 0; // size bleibt unverändert
        this.data = newData;
    }
}
```

c) Testen Sie Ihre Implementierungen mit der Klasse `QueueTester` (herunterzuladen von der Informatik-Homepage).

Abgabe: Per UniWorx, bis spätestens Montag, den 26.6.2006 um 9:00 Uhr.