Dr. Annette Bieniusa
Mathias Weber, M. Sc.
Peter Zeller, M. Sc.

TU Kaiserslautern
Fachbereich Informatik
AG Softwaretechnik

Lösungshinweise/-vorschläge zum Übungsblatt 8: Software-Entwicklung 1 (WS 2017/18)

Die Hinweise und Vorschläge in diesem Dokument sollen der Lösungsfindung dienen und erheben demnach weder Anspruch auf Vollständigkeit noch Korrektheit. Sollten Sie Fehler finden, würden wir uns freuen, wenn Sie uns diese mitteilen. (Kontaktdaten finden Sie auf unserer Webpräsenz.)

Aufgabe 1 Subtyping und dynamisches Binden (8 Punkte)

```
1  interface A {
2    String m();
3  }
4
5  interface B {
6    String m();
7  }
8
9  interface C extends A {
10   String p();
11 }
12
13 interface D extends A, B {
14   String q();
15 }
16
17 class E implements B {
18   public String m() {
19     return "Em";
20   }
21   public String r() {
22     return "Er";
23   }
24 }
25 class F implements C, D {
26   public String m() {
27     return "Fm";
28   }
29   public String p() {
30     return "Fp";
31   }
32   public String q() {
33     return "Fq";
34   }
35   public String r() {
36     return "Fr";
37   }
38 }
39
40
41 public class Subtyping {
42   public static void main(String[] args) {
43
44     // ...
45
46   }
47 }
```

Betrachten Sie die folgenden Code-Beispiele, welche in der `main`-Methode in Zeile 44 eingefügt werden sollen. Entscheiden Sie jeweils, ob das Beispiel vom Java akzeptiert wird oder nicht. Falls das Beispiel akzeptiert wird, geben Sie die Ausgabe des Programms an. Andernfalls erklären Sie, warum es nicht akzeptiert wird.

a)
```
A x = new A();
System.out.println(x.m());
```

Fehler beim Übersetzen: A ist ein Interface und kann nicht mit **new** erstellt werden.

b)
```
E x = new E();
System.out.println(x.r());
```

Ausgabe `"Er"`.

c)
```
A x = new F();
System.out.println(x.m());
```

Ausgabe `"Fm"`

d)
```
A x = new F();
System.out.println(x.r());
```

Übersetzungsfehler: Kann nicht auf `r` zugreifen, da der statische Typ von `x` der Interfacetyp `A` ist.

e)
```
B x = new E();
B y = new F();
System.out.println(x.r() + y.r());
```

Übersetzungsfehler: `x` und `y` haben statischen Typ `B` und Interface `B` hat keine Methode `r`.

f)
```
B x = new E();
B y = new F();
System.out.println(x.m() + y.m());
```

Ausgabe: `"EmFm"`

g)
```
B x = new F();
A y = x;
System.out.println(x.m() + y.m());
```

Übersetzungsfehler: `x` hat Typ `B`, was kein Subtyp von `A` ist. Also ist die Zuweisung nicht erlaubt.

h)
```
D x = new F();
System.out.println(x.m() + x.p());
```

Übersetzungsfehler: `x` hat Typ `D` und das Interface `D` hat keine Methode `p`.

Aufgabe 2 Figuren (6 Punkte)

Gegeben ist das Interface `Figur` und die Klassen `Rechteck` und `Kreis`:

```
interface Figur {

}

class Rechteck implements Figur {
    private final double x;
    private final double y;
    private final double width;
    private final double height;

    Rechteck(double x, double y, double width, double height) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }
}

class Kreis implements Figur {
    private final double x;
    private final double y;
    private final double radius;

    Kreis(double x, double y, double radius) {
        this.x = x;
        this.y = y;
        this.radius = radius;
    }
}
```

Schreiben Sie eine Prozedur `static double flaeche(Figur[] figuren)` in einer Klasse `Figuren`, welche ein Array von Figuren nimmt und die Gesamtfläche berechnet, die von den Figuren eingenommen wird (also die Summe der Flächen der einzelnen Figuren im Array). Erweitern Sie dazu die Klassen und das Interface um passende Methoden. Verwenden Sie keine Casts.

Freiwillige Zusatzaufgabe: Implementieren Sie die Prozedur `flaeche` so, dass Überschneidungen von Figuren bei der Berechnung nur einmal verrechnet werden und nicht (wie oben) bei jeder Figur mit aufsummiert

werden.

Diese Zusatzaufgabe können Sie selbstständig bearbeiten und abgeben. Dazu finden Sie im Exclaim eine zusätzliche Übung “SE1WS17Z”.

```
interface Figur {
    double flaeche();
}

class Rechteck implements Figur {
    private final double x;
    private final double y;
    private final double width;
    private final double height;

    Rechteck(double x, double y, double width, double height) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }

    public double flaeche() {
        return width * height;
    }
}

class Kreis implements Figur {
    private final double x;
    private final double y;
    private final double radius;

    Kreis(double x, double y, double radius) {
        this.x = x;
        this.y = y;
        this.radius = radius;
    }

    public double flaeche() {
        return Math.PI * radius * radius;
    }
}

public class Figuren {
    public static double flaeche(Figur[] figuren) {
        double sum = 0;
        for (Figur f: figuren) {
            sum += f.flaeche();
        }
        return sum;
    }

    public static void main(String[] args) {
        Figur[] figs = {
            new Rechteck(3, 4, 10, 3),
            new Kreis(7, 8, 10),
            new Rechteck(100, 10, 5, 5)
        };
        double f = Figuren.flaeche(figs);
        System.out.println(f);
    }
}
```

Aufgabe 3 Interface für Listen (4 Punkte)

Laden Sie sich die Implementierungen zu `IntList` und `IntArrayList` aus den Vorlesungs-Materialien herunter.

a) Schreiben Sie ein Interface `ListOfInt`, welches die Methoden `add`, `get`, und `size` enthält. Passen Sie die Klassen `IntArrayList` und `IntList` so an, dass sie das Interface implementieren.

b) Erstellen Sie ein Interface `IteratorOfInt`, welches die gemeinsamen Methoden der beiden Iterator-Klassen beinhaltet.

c) Erweitern Sie das Interface `ListOfInt` um die Methode `IteratorOfInt iterator()` und passen Sie die Klassen entsprechend an.

```
public interface ListOfInt {
    void add(int element);

    IteratorOfInt iterator();

    int get(int index);

    int size();

}

public interface IteratorOfInt {
    boolean hasNext();
    int next();
}
```

Die Klassen müssen dann entsprechend angepasst werden:

```
public class IntArrayList implements ListOfInt { ...
public class IntArrayListIterator implements IteratorOfInt { ...
public class IntList implements ListOfInt { ...
public class IntListIterator implements IteratorOfInt { ...
```

Für die Klasse `IntArrayList` muss außerdem noch die Methode `size` implementiert werden:

```
public int size() {
    return size;
}
```

Aufgabe 4 Monitoring (12 Punkte)

Eine Monitoring Anwendung beobachtet die Entwicklung von Daten-Strömen und reagiert bei auffälligen Änderungen. In dieser Aufgabe sollen Sie ein einfaches Monitoring-System nach der folgenden Beschreibung in Java implementieren und dazu Interfaces und Klassen verwenden.

Ein `Monitor` hat eine Methode `report`, mit der ein neuer Datensatz gemeldet werden kann. Ein Datensatz besteht dabei aus einem Zeitstempel (Typ **`long`**, Zeit in Millisekunden) und einem Wert vom Typ **`double`**. Außerdem hat der Monitor Methoden, um Trigger und Alerts hinzuzufügen.

Ein Trigger erkennt bestimmte Änderungen in den Daten und kann damit Alerts auslösen. Jedes mal, wenn mit der `report`-Methode ein Datensatz an den Monitor gemeldet wird, benachrichtigt der Monitor alle Trigger, die ihm hinzugefügt worden sind. Die Trigger prüfen jeweils den Datensatz und wenn einer oder mehrere Trigger feuern, werden alle hinzugefügten Alerts benachrichtigt. Sie können davon ausgehen, dass die Zeitstempel in den Aufrufen der `report`-Methode immer größer werden.

Das System soll erweiterbar sein, so dass später andere Trigger und Alerts hinzugefügt werden können. Für diese Aufgabe sollen nur zwei Arten von Triggern und zwei verschiedene Alerts implementiert werden:

Ein `AboveTrigger` ist ein Trigger, der über den Konstruktor `AboveTrigger(double bound)` erstellt wird. Er wird ausgelöst, wenn ein Datensatz mit einem Wert größer als `bound` gemeldet wird.

Ein anderer Trigger ist der `DeltaTrigger`, welcher über den Konstruktor `DeltaTrigger(long t, double maxChange)` erstellt wird und ausgelöst wird, wenn ein Datensatz-Wert sich in den letzten `t` Millisekunden um mehr als `maxChange` verändert hat.

Als Alert soll es einen `TextAlert` geben, welcher über den Konstruktor `TextAlert(String message)` erstellt wird. Wenn dieser ausgelöst wird soll er die Nachricht `message` auf der Konsole ausgeben. Dabei soll in `message` der String `%t` durch den zuletzt gemeldeten Zeitstempel ersetzt werden und `%v` durch den zuletzt gemeldeten Wert (gerundet auf zwei Nachkommastellen).

Ein weiterer Alert soll der `EmailAlert` sein, welcher über den Konstruktor `EmailAlert(String email, String message)` erstellt wird. Dieser soll beim auslösen zuerst die `email` in einer Zeile ausgeben und danach die `message` wie der `TextAlert`.

Implementieren Sie Ihre Klassen und Interfaces so, dass sie wie in folgendem Beispiel verwendet werden können:

```
Monitor m = new Monitor();
m.addTrigger(new AboveTrigger(39));
m.addTrigger(new DeltaTrigger(4000, 1));
m.addAlert(new TextAlert("%t ms: Value changed to %v"));

m.report(new Dataset(0, 38.4));
m.report(new Dataset(1000, 38.5));
m.report(new Dataset(2000, 38.5));
m.report(new Dataset(3000, 38.6));
m.report(new Dataset(4000, 39.1)); // AboveTrigger(39) feuert
m.report(new Dataset(5000, 38.9));
m.report(new Dataset(6000, 38.5));
m.report(new Dataset(7000, 38.0)); // DeltaTrigger(4000, 1) feuert
m.addAlert(new EmailAlert("hans@example.com", "ALERT!"));
m.report(new Dataset(8000, 38.0)); // DeltaTrigger feuert erneut
m.report(new Dataset(9000, 38.0));
```

Die Ausgabe bei diesem Beispiel soll dann wie folgt aussehen:

```
4000 ms: Value changed to 39.10
7000 ms: Value changed to 38.00
8000 ms: Value changed to 38.00
hans@example.com
ALERT!
```

```
import java.util.ArrayList;
import java.util.List;

public class Monitor {
    private List<Trigger> triggers = new ArrayList<>();
    private List<Alert> alerts = new ArrayList<>();

    void report(Dataset dataset) {
        boolean triggered = false;
        for (Trigger t : triggers) {
            if (t.check(dataset)) {
                triggered = true;
            }
        }
        if (triggered) {
            for (Alert alert : alerts) {
                alert.process(dataset);
            }
        }
    }

    void addTrigger(Trigger t) {
        triggers.add(t);
    }

    void addAlert(Alert a) {
        alerts.add(a);
    }

}

public class Dataset {
    private final long timestamp;
    private final double value;

    Dataset(long timestamp, double value) {
        this.timestamp = timestamp;
        this.value = value;
    }

    public long getTimestamp() {
        return timestamp;
    }

    public double getValue() {
        return value;
    }
}

public interface Trigger {
    boolean check(Dataset dataset);
}

public class AboveTrigger implements Trigger {
    private final double bound;

    AboveTrigger(double bound) {
        this.bound = bound;
    }

    @Override
    public boolean check(Dataset dataset) {
        return dataset.getValue() > bound;
    }
```

```
}

import java.util.*;

public class DeltaTrigger implements Trigger {

    private final long timeFrame;
    private final double maxChange;
    private final List<Dataset> changes = new LinkedList<>();

    DeltaTrigger(long t, double maxChange) {
        timeFrame = t;
        this.maxChange = maxChange;
    }

    @Override
    public boolean check(Dataset dataset) {
        Iterator<Dataset> iterator = changes.iterator();
        while (iterator.hasNext()) {
            Dataset d = iterator.next();
            if (d.getTimestamp() + timeFrame < dataset.getTimestamp()) {
                // alte Einträge löschen
                iterator.remove();
            } else if (Math.abs(d.getValue() - dataset.getValue()) > maxChange) {
                return true;
            }
        }
        changes.add(dataset);
        return false;
    }
}

public interface Alert {
    void process(Dataset dataset);
}

public class TextAlert implements Alert {

    private final String message;

    TextAlert(String message) {

        this.message = message;
    }

    @Override
    public void process(Dataset dataset) {
        String msg = message;
        msg = msg.replace("%t", "" + dataset.getTimestamp());
        msg = msg.replace("%v", String.format("%.02f", dataset.getValue()));
        System.out.println(msg);
    }
}

public class EmailAlert implements Alert {

    private final String email;
    private final String message;

    EmailAlert(String email, String message) {
        this.email = email;

        this.message = message;
    }
```

```
    @Override
    public void process(Dataset dataset) {
        System.out.println(email);
        String msg = message;
        msg = msg.replace("%t", "" + dataset.getTimestamp());
        msg = msg.replace("%v", String.format("%.02f", dataset.getValue()));
        System.out.println(msg);
    }
}
```

Aufgabe 5 Sortierte, markierte Bäume (Einreichaufgabe, 6 Punkte)

a) Schreiben Sie eine Methode `int max()`, welche die maximale Markierung im Baum zurück gibt. Wenn der Baum leer ist soll `Integer.MIN_VALUE` zurückgegeben werden.

```
    public int max() {
        return max(root);
    }

    private int max(TreeNode node) {
        if (node == null) {
            return Integer.MIN_VALUE;
        }
        // Groessere Elemente sind immer rechts, also pruefen, ob es dort
        // noch Elemente gibt
        if (node.getRight() == null) {
            return node.getMark();
        }
        return max(node.getRight());
    }
```

b) Schreiben Sie eine Methode `int size()`, welche die Anzahl der Einträge im Baum berechnet.

```
    public int size() {
        return size(root);
    }

    private int size(TreeNode node) {
        if (node == null) {
            return 0;
        }
        return 1 + size(node.getLeft()) + size(node.getRight());
    }
```

c) Schreiben Sie eine Methode `int[] toArray()`, welche ein aufsteigend sortiertes Array mit den Einträgen des Baums zurückgibt.

Bei dieser Aufgabe gibt es verschiedene Ansätze. Die unten gezeigte Lösung legt am Anfang ein Array der passenden Größe an und verwendet dann rekursive Aufrufe, um das Array zu füllen. Die rekursive Hilfsmethode erhält neben dem zu füllenden Array auch noch die aktuelle Einfüge-Position. Außerdem gibt sie die aktualisierte Einfüge-Position zurück, nachdem sie die Einträge für den aktuellen Teilbaum in das Array eingetragen hat.

Alternativ kann man auch eine Hilfsmethode `int[] toArray(TreeNode n)` schreiben, welche für den leeren Baum (`n == null`) ein leeres Array zurückgibt und ansonsten das Array für den linken und rechten Baum rekursiv berechnet und die beiden Arrays zusammen mit der Markierung dann zu einem neuen Array zusammenbaut.

```java
public int[] toArray() {
    int[] result = new int[size()];
    fillArray(root, result, 0);
    return result;
}

private int fillArray(TreeNode node, int[] res, int pos) {
    if (node == null) {
        return pos;
    }
    pos = fillArray(node.getLeft(), res, pos);
    res[pos] = node.getMark();
    pos++;
    return fillArray(node.getRight(), res, pos);
}

// Alternative Variante:
public int[] toArray2() {
    return toArrayR(root);
}

public int[] toArrayR(TreeNode node) {
    if (node == null) {
        return new int[0];
    }
    int[] left = toArrayR(node.getLeft());
    int[] right = toArrayR(node.getRight());
    int[] res = new int[left.length + 1 + right.length];
    System.arraycopy(left, 0, res, 0, left.length);
    res[left.length] = node.getMark();
    System.arraycopy(right, 0, res, left.length + 1, right.length);
    return res;
}
```