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

TU Kaiserslautern
Fachbereich Informatik
AG Softwaretechnik

Lösungshinweise/-vorschläge zum Übungsblatt 9: Software-Entwicklung 1 (WS 2016/17)

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 2 Equals, Hashcode, Comparator (9 Punkte)

a) Schreiben Sie eine Klasse `Ort` mit einem Konstruktor `Ort(int postleitzahl, String name)` und entsprechenden Attributen und `get`-Methoden.

b) Schreiben Sie eine Klasse `Addresse` mit einem Konstruktor `Addresse(Ort ort, String strasse, String hausNummer)` und entsprechenden Attributen und `get`-Methoden.

c) Schreiben Sie eine geeignete `equals`-Methode für Ihre Klassen.

d) Implementieren Sie eine geeignete `hashCode`-Methode für Ihre Klassen.

```
import java.util.Objects;

public class Ort {
    private int postleitzahl;
    private String name;

    public Ort(int postleitzahl, String name) {
        this.postleitzahl = postleitzahl;
        this.name = name;
    }

    public int getPostleitzahl() {
        return postleitzahl;
    }

    public String getName() {
        return name;
    }

    @Override
    public boolean equals(Object obj) {
        if (obj instanceof Ort) {
            Ort ort = (Ort) obj;
            return postleitzahl == ort.postleitzahl
                    && Objects.equals(name, ort.name);
        }
        return false;
    }

    @Override
    public int hashCode() {
        return Objects.hash(postleitzahl, name);
    }
}
```

```java
import java.util.Objects;

public class Addresse {
    private Ort ort;
    private String strasse;
    private String hausNummer;

    public Addresse(Ort ort, String strasse, String hausNummer) {
        this.ort = ort;
        this.strasse = strasse;
        this.hausNummer = hausNummer;
    }

    public Ort getOrt() {
        return ort;
    }

    public String getStrasse() {
        return strasse;
    }

    public String getHausNummer() {
        return hausNummer;
    }

    @Override
    public boolean equals(Object obj) {
        if (obj instanceof Addresse) {
            Addresse a = (Addresse) obj;
            return Objects.equals(ort, a.ort)
                    && Objects.equals(strasse, a.strasse)
                    && Objects.equals(hausNummer, a.hausNummer);
        }
        return false;
    }

    @Override
    public int hashCode() {
        return Objects.hash(ort, strasse, hausNummer);
    }
}
```

e) Schreiben Sie geeignete Implementierungen von `Comparator<Ort>` und `Comparator<Addresse>`.

```java
import java.util.Comparator;

public class OrtVergleicher implements Comparator<Ort> {
    @Override
    public int compare(Ort o1, Ort o2) {
        if (o1.getPostleitzahl() < o2.getPostleitzahl()) {
            return -1;
        } else if (o1.getPostleitzahl() > o2.getPostleitzahl()) {
            return 1;
        } else {
            return o1.getName().compareTo(o2.getName());
        }
    }
}
```

```java
import java.util.Comparator;

public class AddressVergleicher implements Comparator<Addresse> {
    @Override
    public int compare(Addresse a1, Addresse a2) {
        OrtVergleicher ortVergleicher = new OrtVergleicher();
        int ortVergleich = ortVergleicher.compare(a1.getOrt(), a2.getOrt());
```

```
        if (ortVergleich != 0) {
            return ortVergleich;
        } else {
            int strassenVergleich = a1.getStrasse().compareTo(a2.getStrasse());
            if (strassenVergleich != 0) {
                return strassenVergleich;
            } else {
                return a1.getHausNummer().compareTo(a2.getHausNummer());
            }
        }
    }
}
```

Seit Java 8 lassen sich Vergleiche auch deutlich kompakter implementieren, zum Beispiel:

```
import java.util.Comparator;

public class Vergleicher {

    public static Comparator<Ort> ortVergleicher() {
        return Comparator
                .comparing(Ort::getPostleitzahl)
                .thenComparing(Ort::getName);
    }

    public static Comparator<Addresse> addressVergleicher() {
        return Comparator
                .comparing(Addresse::getOrt, ortVergleicher())
                .thenComparing(Addresse::getStrasse)
                .thenComparing(Addresse::getHausNummer);
    }
}
```

Aufgabe 3 Grafikeditor (Sinnvoll bearbeiten)

In dieser Aufgabe sollen Sie den Grafikeditor weiterentwickeln, den Sie als `Grafikeditor.zip` herunterladen können.

a) Übersetzen Sie die vorgegebene Klasse `Main` und führen Sie das Programm dann aus. Sie sollten in der Lage sein Linien zu zeichnen und diese zu verschieben. Falls Sie Probleme mit dem Ausführen haben, kontaktieren Sie bitte Ihren Tutor.

   Machen Sie sich mit dem Code in den Dateien `Main`, `ZeichenBlatt`, `Punkt`, `Figur`, `Linie`, `ZeichenTool`, `ZeichenToolLinie` und `ZeichenToolVerschieben` vertraut.

   Die Klasse `Gui` stellt die grafische Benutzeroberfläche bereit und muss in dieser Aufgabe nicht verstanden oder angepasst werden.

b) Erweitern Sie den Editor um eine Funktion um Kreise zu zeichnen. Dazu sind die folgenden Schritte notwendig:

   - Erstellen Sie eine neue Unterklasse von `Figur` namens `Kreis`.

```
public class Kreis extends Figur {
    private Vec2 mittelpunkt;
    private double radius;

    public Kreis(Vec2 mittelpunkt, int radius) {
        this.mittelpunkt = mittelpunkt;
        this.radius = radius;
    }

    @Override
    public void zeichnen(SEGraphics seg) {
        super.zeichnen(seg);
        seg.drawCircle(mittelpunkt.x, mittelpunkt.y, radius);
    }

    @Override
    public double abstandZu(Vec2 p) {
        return Math.max(0, mittelpunkt.distanceTo(p) - radius);
    }


    @Override
    public void verschiebenUm(Vec2 delta) {
        mittelpunkt = mittelpunkt.plus(delta);
    }

    public Vec2 getMittelpunkt() {
        return mittelpunkt;
    }

    public void setRadius(double radius) {
        this.radius = radius;
    }

}
```

   - Erstellen Sie eine neue Unterklasse von `ZeichenTool` für das Zeichnen von Kreisen.

```
public class ZeichenToolKreis extends ZeichenTool {

    @Override
    public String getName() {
        return "Kreis";
    }

    private Kreis kreis;
```

```
    @Override
    public void start(Vec2 pos) {
        kreis = new Kreis(pos, 1);
        zeichenBlatt.addFigur(kreis);
    }

    @Override
    public void ziehen(Vec2 pos) {
        kreis.setRadius(kreis.getMittelpunkt().distanceTo(pos));
    }

}
```

- Passen Sie die `main`-Methode in der Klasse `Main` an, indem sie dort ihr Tool zum Kreise zeichnen in die Liste der Tools eintragen.

```
tools.add(new ZeichenToolKreis());
```

c) Erweitern Sie den Editor noch um Rechtecke. Gehen Sie dabei analog zur vorherigen Teilaufgabe vor.

```
public class Rechteck extends Figur {
    private Vec2 start;
    private Vec2 groesse;

    public Rechteck(Vec2 start, Vec2 groesse) {
        this.start = start;
        this.groesse = groesse;
    }

    @Override
    public void zeichnen(SEGraphics seg) {
        super.zeichnen(seg);
        seg.drawRect(start.x, start.y, groesse.x, groesse.y);
    }


    @Override
    public double abstandZu(Vec2 p) {
        Vec2 untenLinks = new Vec2(start.x, start.y + groesse.y);
        Vec2 obenRechts = new Vec2(start.x+groesse.x, start.y);
        Vec2 untenRechts = start.plus(groesse);
        if (p.x < start.x) {
            // Links vom Rechteck
            return p.distanceToLine(start, untenLinks);
        } else if (p.x > start.x + groesse.x) {
            // Rechts vom Rechteck
            return p.distanceToLine(obenRechts, untenRechts);
        } else {
            // x-Koordinate im Rechteck
            if (p.y < start.y) {
                // Oberhalb des Rechtecks
                return p.distanceToLine(start, obenRechts);
            } else if (p.y > start.y + groesse.y) {
                // Unterhalb des Rechtecks
                return p.distanceToLine(untenLinks, untenRechts);
            } else {
                // Im Rechteck
                return 0;
            }
        }
    }


    @Override
```

```
    public void verschiebenUm(Vec2 delta) {
        start = start.plus(delta);
    }


    public void setStart(Vec2 start) {
        this.start = start;
    }


    public void setGroesse(Vec2 groesse) {
        this.groesse = groesse;
    }


    public Vec2 getGroesse() {
        return groesse;
    }

}

public class ZeichenToolRechteck extends ZeichenTool {

    private Rechteck reckteck;
    private Vec2 startPunkt;

    @Override
    public String getName() {
        return "Rechteck";
    }

    @Override
    public void start(Vec2 pos) {
        reckteck = new Rechteck(pos, new Vec2(1, 1));
        startPunkt = pos;
        zeichenBlatt.addFigur(reckteck);
    }

    @Override
    public void ziehen(Vec2 pos) {
        double startX = Math.min(startPunkt.x, pos.x);
        double startY = Math.min(startPunkt.y, pos.y);
        reckteck.setStart(new Vec2(startX, startY));
        double groesseX = Math.abs(startPunkt.x - pos.x);
        double groesseY = Math.abs(startPunkt.y - pos.y);
        reckteck.setGroesse(new Vec2(groesseX, groesseY));
    }

}
```

d) Die Methode `verschiebenUm` muss für jede Figur implementiert werden. Wie könnte man den Code umbauen, so dass es nur eine Implementierung der Methode gibt? Es ist ausreichend, wenn Sie Ihre Idee nur beschreiben und nicht implementieren.

Man kann der abstrakten Klasse `Figur` einen Punkt geben, der die Position der Figur festlegt. Dann muss man die Methode `verschiebenUm` nur in der Klasse `Figur` implementieren und dort diese Position anpassen. Wichtig ist dann aber, dass man keine anderen absoluten Punkte mehr verwendet, sondern nur noch relative. Zum Beispiel müsste der End-Punkt der Linie relativ zur Start-Position gegeben werden.