Prof. Dr. Uwe Schmidt

fhwedel
UNIVERSITY OF APPLIED SCIENCES

30. Januar 2012

Aufgaben zur Klausur **Objektorientierte Programmierung** im WS 2011/12 (BInf 211, BTInf 211, BMInf 211, BWInf 211)

Zeit: 90 Minuten erlaubte Hilfsmittel: keine

Bitte tragen Sie Ihre Antworten und fertigen Lösungen ausschließlich an den gekennzeichneten Stellen in das Aufgabenblatt ein. Ist ihre Lösung wesentlich umfangreicher, so überprüfen Sie bitte nochmals Ihren Lösungsweg.

Sollten Unklarheiten oder Mehrdeutigkeiten bei der Aufgabenstellung auftreten, so notieren Sie bitte, wie Sie die Aufgabe interpretiert haben.

Tipp: Bei der Entwicklung der Lösung können kleine Skizzen manchmal hilfreich sein.

Viel Erfolg!

Diese Klausur besteht einschließlich dieses Deckblattes aus 12 Seiten.

Aufgabe 1:

Lesen Sie bitte vor der Bearbeitung die gesamte Aufgabe durch. Es sind in der Aufgabenstellung Vorwärtsreferenzen enthalten.

Zweidimensionale geometrische Figuren (Kreise, Rechtecke, ...) können auf sehr unterschiedliche Arten in einem System implementiert werden. Eine sehr allgemeine und in manchen Anwendungen sehr speicherplatzeffiziente Art ist die, Figuren durch Funktionen zu repräsentieren, z.B. durch folgende Schnittstelle in Java:

```
interface Figure {
    public boolean contains(Point p);
}
```

Die so repräsentierten Figuren können beliebig groß sein, zum Beispiel einen ganzen Quadranten darstellen, im Extremfall sogar die gesamte Ebene.

Wenn eine reellwertige Funktion gegeben ist, so kann aus dieser auf einfache Art eine Figur erzeugt werden, die den Graphen der Funktion als Bild repräsentiert. Funktionen werden durch folgende Schnittstelle repräsentiert:

```
interface Function {
    public double at(double x);
}
```

Das Visualisieren von Figuren kann implementiert werden, indem der gesamte darzustellende Bereich *abgetastet* wird, d.h. es wird punktweise getestet, ob eine Koordinate in einer Figur liegt oder außerhalb. Dieses wird beispielhaft in dem Testprogramm am Ende der Aufgabe gemacht.

Einige Figuren und Figuren erzeugende Funktionen sind in einer Klasse *FigureFactory* gesammelt. Tipp: Vor dem Entwickeln der fehlenden Teile bitte erst weiter lesen.

```
abstract class FigureFactory {
    public final static Figure unitCircle = new UnitCircle();

    public static Figure circle(double r) {
        return
            new TransformedFigure(unitCircle,
                                  new Scale(new Point(r, r)));
    }
    public final static Figure unitSquare = new UnitSquare();

    public static final Figure halfPlaneX =
        new Figure() {
            public boolean contains(Point p) {
                return
                    p.x >= 0;
            }
        };
    public final static Figure sineCurve
        = new FunctionFigure(new Function() {
                public double at(double x) {
                    return Math.sin(x);
                }
            });

    public static final Figure halfPlaneN =

        ..........................................

        ..........................................

        ..........................................

        ..........................................

    public static Figure rectangle( ..........................................) {

        ..........................................

        ..........................................

        ..........................................

        ..........................................
    }
}
```

In der Klasse *FigureFactory* sind vier Figuren vordefiniert, das Einheitsquadrat mit dem Ursprung als unterer linker Ecke, ein Kreis um den Ursprung mit dem Radius 1.0, die Halbebene rechts der y–Achse (alle Werte mit positiver x–Koordinate) und die Figur, die den Graphen der Sinusfunktion repräsentiert.

Erweitern Sie die Klasse *FigureFactory* um eine neue Figur, *halfPlaneN*, die die abgeschlossene Halbebene unterhalb der Geraden durch den Ursprung mit einer Steigung von -1 repräsentiert. Diese Methode soll analog zu *halfPlaneX* implementiert werden.

Erweitern Sie die Klasse *FigureFactory* weiterhin um eine neue statische Methode, *rectangle*, die analog zu *circle* Rechtecke beliebiger Größe mit unterer linker Ecke im Ursprung erzeugt.

Die Klasse *UnitCircle* ist wie folgt definiert:

```
public class UnitCircle implements Figure {

   public boolean contains(Point p) {
      return
            p.len() <= 1.0;
   }
}
```

Entwickeln Sie analog dazu eine Klasse *UnitSquare*

```
public class UnitSquare implements Figure {

   public boolean contains(Point p) {

            ..........................................

            ..........................................

            ..........................................

            ..........................................

            ..........................................
   }
}
```

Die Punkte werden in diesem Beispiel durch folgende Klasse realisiert:

```
final
public class Point {
    final public double x;
    final public double y;

    public Point(double x1, double y1) {
        x = x1; y = y1;
    }
    static final public Point org = new Point(0.0, 0.0);
    static final public Point x1 = new Point(1.0, 0.0);
    static final public Point y1 = new Point(0.0, 1.0);
    static final public Point xy = new Point(1.0, 1.0);

    public Point add(Point p2) {
        return
            new Point(x + p2.x, y + p2.y);
    }
    public Point sub(Point p2) {
        return
            new Point(x − p2.x, y − p2.y);
    }
    public Point mul(Point p2) {
        return
            new Point(x ∗ p2.x, y ∗ p2.y);
    }
    public Point div(Point p2) {
        return
            new Point(x / p2.x, y / p2.y);
    }
    public double len() {
        return
            Math.sqrt(x∗x + y∗y);
    }

}
```

In der Klasse sind einige Punkt-Operationen und einige häufig verwendete Punkte vordefiniert. Nutzen Sie diese, wenn möglich, beim Entwickeln der fehlenden Codestücke.

Figuren können auf viele unterschiedliche Arten manipuliert werden. Ein Transformationstyp ist der, dass eine punktweise Transformation, z.B. eine Verschiebung oder Drehung, vorgenommen wird. Transformationen können ebenfalls durch Objekte repräsentiert werden. Die folgende Klasse wird in diesem Beispiel verwendet:

```
interface Transform {

   public Point move(Point p);

   // useful elementary transformations

   static final public Transform mirror =
      new Transform() {
         public Point move(Point p) {
            return
               new Point(−p.x, −p.y);
         }
      };

   static final public Transform rotate90 =
      new Transform() {
         public Point move(Point p) {

            ..........................................

            ..........................................
         }
      };
}
```

In dieser Schnittstelle sind zwei Transformationen durch die Verwendung von anonymen Klassen realisiert, eine punktweise Spiegelung am Ursprung und eine 90–Grad–Drehung im Uhrzeigersinn. Entwickeln Sie die fehlenden Teile.

Mit den Methoden aus der Translationsklasse können Figuren verschoben werden.

```
public class Translation implements Transform {

    private Point p;

    public Translation(Point p1) {
        p = p1;
    }

    public Point move(Point p1) {
        return
            new Point(p1.x − p.x, p1.y − p.y);
    }
}
```

Entwickeln Sie analog zur Translationsklasse eine Klasse für die Erzeugung von Skalierungen, also von Transformationen, die alle Koordinaten mit einem festen Wert in x–Richtung skalieren, und mit einem zweiten Wert in y–Richtung.

```
public class Scale implements Transform {

        ..........

        ..........

    public Scale(..........) {

        ..........
    }

    public Point move(Point p1) {

        ..........

        ..........
    }
}
```

Eine Transformation kann auf eine Figur angewendet werde. Das Resultat ist wieder eine Figur. Dieser Prozess wird ebenfalls durch eine Klasse implementiert. Vervollständigen sie diese Klasse:

```
public class TransformedFigure implements Figure {

        ..........

        ..........

    public TransformedFigure( ..........) {

        ..........

        ..........
    }

    public boolean contains(Point p) {

        ..........

        ..........

        ..........
    }
}
```

Zwei oder mehrere Figuren können auf unterschiedliche Weise kombiniert werden, zum Beispiel sind alle Mengenoperationen (Vereinigung, Durchschnitt, Subtraktion, ...) möglich. Die zweistellige Kombination wird wieder mit Hilfe einer Klasse beschrieben:

```
abstract public class CombinedFigure implements Figure {
    protected Figure fig1;
    protected Figure fig2;

    protected CombinedFigure(Figure f1, Figure f2) {
        fig1 = f1; fig2 = f2;
    }
}
```

Entwickeln Sie eine abgeleitete Klasse für die Vereinigung zweier Figuren.

```
public class UnionFigure ..........{

        ..........

    public UnionFigure(..........) {

        ..........
    }

    public boolean contains(Point p) {

        ..........

        ..........

        ..........
    }
}
```

Welche Zeile(n) aus dieser Klasse müssen modifiziert werden für eine analoge Klasse *IntersectFigure* für den Durchschnitt zweier Figuren?

. . . . . .

. . . . . .

Um aus einer Funktion eine Figur zu erzeugen, wird die Klasse *FunctionFigure* genutzt.

```
public class FunctionFigure implements Figure {
    protected Function f;

    public FunctionFigure(Function f1) {
        f = f1;
    }

    public boolean contains(Point p) {
        return
            p.y == f.at(p.x);
    }
}
```

Warum wird mit dieser Implementierung beim Zeichnen (beim Abtasten) einer Figur nur in Ausnahmefällen der Graph der Funktion sichtbar?

. . . . . .

. . . . . .

Entwickeln Sie eine Klasse *AboveFunctionFigure*, mit der die Fläche oberhalb des Funktionsgraphen (einschließlich des Graphen selbst) als Figur repräsentiert wird:

```
public class AboveFunctionFigure ..... {
    public AboveFunctionFigure(Function f1) {

        .....
    }

    public boolean contains(Point p) {

        .....

        .....
    }
}
```

Gegeben sei die folgende Mini–Anwendung:

```
class Main {
   public static void main(String [] args) {
      Figure fig1 = FigureFactory.unitCircle;
      Figure fig2 = new UnionFigure(fig1, FigureFactory.unitSquare);
      Figure fig3 = FigureFactory.circle(0.5);
      Figure fig4 = new TransformedFigure(fig3, new Translation(new Point(0.5,0.5)));
      Figure fig5 = new AboveFunctionFigure(new Function() {
                  public double at(double x) {
                        return Math.sin(x);
                  }
            });

      scan(fig1,20,10);
      scan(fig2,20,10);
      scan(fig3,20,10);
      scan(fig4,20,10);
      scan(fig5,20,10);

      fig4 = null; // .1
      fig3 = null; // .2
      fig2 = null; // .3
      fig1 = null; // .4

   }
   static void scan(Figure fig, int w, int h) {
         for (int j = h-1; j >= 0; --j) {
               for (int i = 0; i < w; ++i)
                     put(fig.contains(new Point((double)i/w, (double)j/h)));
               System.out.println();
         }
         System.out.println();

   }
   static void put(boolean c) { System.out.print(c ? 'X' : '.'); }
}
```

Wie viele Objekte der Klasse *Point* werden bei der Ausführung von *scan(fig1,20,10);* erzeugt?

. . . . . .

Wie viele Objekte der Klasse *Point* werden bei der Ausführung von *scan(fig3,20,10);* erzeugt?

. . . . . .

Wie viele Objekte werden nicht mehr referenziert und könnten vom Garbage-Collector eingesammelt werden, nachdem die Anweisung *fig4 = null;* ausgeführt worden ist *(Markierung .1)*.

. . . . . .

Wie viele Objekte werden nicht mehr referenziert und könnten vom Garbage-Collector eingesammelt werden, nachdem die Anweisung *fig3 = null;* ausgeführt worden ist *(Markierung .2)*.

. . . . . .

Wie viele Objekte werden nicht mehr referenziert und könnten vom Garbage-Collector eingesammelt werden, nachdem die Anweisung *fig2 = null;* ausgeführt worden ist *(Markierung .3)*.

. . . . . .

Wie viele Objekte werden nicht mehr referenziert und könnten vom Garbage-Collector eingesammelt werden, nachdem die Anweisung *fig1 = null;* ausgeführt worden ist *(Markierung .4)*.

. . . . . .

Ist in der Klasse *FunctionFigure* das Schlüsselword **protected** vor der Variablen *f* sinnvoll gewählt?

ja ☐ nein ☐

Begründung:

. . . . . .

. . . . . .