Prof. Dr. Uwe Schmidt

fhwedel
UNIVERSITY OF APPLIED SCIENCES

20. August 2004

Aufgaben zur Klausur **C** und **Objektorientierte Programmierung** im SS 2004 (WI h103, II h105, MI h353)

Zeit: 150 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.

Viel Erfolg !

Diese Klausur besteht einschließlich dieses Deckblattes aus 16 Seiten

---

Aufgabe 1:

Gegeben sei der folgende Ausschnitt eines C–Moduls für die Implementierung von binären Suchbäumen.

```
typedef long int Element;

#define compare(x,y) (((x) > (y)) - ((x) < (y)))

typedef struct Node *Set;

struct Node
{
  Element info;
  Set l;
  Set r;
};

Set mkEmptySet(void);
int isEmptySet(Set s);

Set mkOneElemSet(Element e);
Set insertElem(Element e, Set s);
Set removeElem(Element e, Set s);


Set
insertElem(Element e, Set s)
{
  if (isEmptySet(s))
    return mkOneElemSet(e);

  switch (compare(e, s->info))
    {
    case -1:
      s->l = insertElem(e, s->l);
      break;
    case 0:
      break;
    case +1:
      s->r = insertElem(e, s->r);
      break;
    }

  return s;
}
```

```
static
Set removeRoot(Set s) {
  ...
}

Set
removeElem(Element e, Set s)
{
  if (isEmptySet(s))
    return s;

  switch (compare(e, s->info))
    {
    case -1:
      s->l = removeElem(e, s->l);
      break;
    case 0:
      s = removeRoot(s);
      break;
    case +1:
      s->r = removeElem(e, s->r);
      break;
    }

  return s;
}


Set changeElem(Element e, Set s, ...) {
  ...
}

Set
removeElem1(Element e, Set s) {
  return
    changeElem(e, s, ...);
}

Set
insertElem1(Element e, Set s) {
  return
    changeElem(e, s, ...);
}
```

Man erkennt, dass die Funktionen für das Einfügen und Löschen sehr ähnlich sind. Diese Funktionen kann man zusammenfassen zu einer Funktion *changeElem*, die die Unterschiede der beiden Funktionen durch zusätzliche Parameter geliefert bekommt.

Entwickeln Sie die Funktion *changeElem*.
Definieren sie zuerst für die Typen der zusätzlichen Parameter Typdefinitionen:

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Der Programmcode für die Funktion *changeElem*.

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Der Programmcode für die neue Funktion zum Einfügen *insertElem1* und erforderliche Hilfkonstrukte:

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Der Programmcode für die neue Funktion zum Einfügen *removeElem1* und erforderliche Hilfkonstrukte:

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Aufgabe 2:

Gegeben seien die folgenden Variablen und Funktionen:

```
typedef int (*Fct1)(int);
typedef int (*Fct2)(int, int);
typedef int (*Fct3)(double);
int x;
long int s;
float f;
char *p1;
int *p2;
void *p3;
Fct1 fp1, fp2;
Fct3 fp3;
int (*pf)(int);
int (*pf2)(double);
int f1(int x1);
int f2(int x1,int x2);
int f3(double x1);
```

Bestimmen Sie für die folgenden Ausdrücke den Typ gemäß ANSI-C. Vorsicht: Es kommen fehlerhafte Ausdrücke von. Kennzeichnen Sie diese entsprechend

p2[x] .....

p3[x] .....

p1 ? x : f .....

∼p2 .....

p1 && p1 .....

p1 & p1 .....

∼s .....

s || x .....

fp1(25) .....

pf=f1 .....

pf=f1(x) .....

pf2=pf .....

fp1==f1 .....

fp1==fp2 .....

f=f3(f) .....

f1==f2 .....

∗p3 .....

Aufgabe 3:

Bilder können auf sehr unterschiedliche Arten in einem System implementiert werden. Eine sehr allgemeine und in manchen Anwendungen sehr speicherplatzeffiziente Art ist die, Bilder durch Funktionen zu repräsentieren, z.B. durch folgende Schnittstelle in Java:

```
interface Image {
    public Color at(Point p);
}
```

Die so repräsentierten Bilder sind beliebig groß und können mit beliebiger Auflösung in Rasterbilder umgerechnet werden. Die Umrechnung, und also auch das Zeichnen, erfolgt, indem man ein zu zeichnendes Rechteck bestimmt und die Auflösung in Form von Höhe mal Breite. Dieses wir hier aber nicht weiter verfolgt.

Einige häufig verwendete Bilder sind in einer Schnittstelle gesammelt:

```
interface SimpleImages {

    // useful simple images


    // black image


    public static final Image black
        = new Image() {
                public Color at(Point p) {
                    return
                        Color.black;
                }
            };

    // white image


    public static final Image white
        = new Image() {
                public Color at(Point p) {
                    return
                        Color.white;
                }
            };
```

```
    // color gradient form black to white between 0.0 <= x <= 1.0


    public static final Image black2white
        = new Image() {
                public Color at(Point p) {
                    double c = Math.max(0.0,Math.min(1.0,p.x));
                    return
                        new Color(c,c,c);
                }
            };
}
```

In der Schnittstelle sind drei Bilder vordefiniert, ein konstant weißes, ein konstant schwarzes und eines, das einen Helligkeitsverlauf entlang der x-Koordinate von Schwarz bei 0.0 bis Weiß bei 1.0 repräsentiert.

Die Koordinaten, 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);
}
```

In der Schnittstelle sind einige häufig verwendete Punkte vordefiniert.

Die Farben werden hier als RGB-Werte im Intervall 0.0 bis 1.0 dargestellt mit 0.0 als dunkelstem Wert und 1.0 als hellstem.

```
final
public class Color {
    final public double r; // red

    final public double g; // green

    final public double b; // blue


    public Color(double r1, double g1, double b1) {
        r = r1; g = g1; b = b1;
    }

    static final public Color black = new Color(0.0, 0.0, 0.0);
    static final public Color white = new Color(1.0, 1.0, 1.0);
    static final public Color red = new Color(1.0, 0.0, 0.0);
    static final public Color green = new Color(0.0, 1.0, 0.0);
    static final public Color blue = new Color(0.0, 0.0, 1.0);
}
```

In der Schnittstelle sind einige häufig genutzte Farben vordefiniert.

Erweitern Sie die Schnittstelle *SimpleImages* um ein neues Bild, *gradient*, in dem der rote Farbwert einen Verlauf entlang der x-Achse nimmt, aber im Intervall 0.0 bis 2.0, analog zu *black2white*, der grüne Kanal einen Verlauf der y-Achse im Intervall 0.0 bis 1.0 nimmt, und in dem der blaue Kanal einen Farbverlauf entlang der Diagonalen vom Punkt (0,0) bis (1,1) nimmt.

public static final Image gradient =

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Bilder können auf viele unterschiedliche Arten manipuliert werden. Ein Transformationstyp ist der, dass eine punktweise Transformation, z.B. eine Verschiebung oder Drehung, vorgenommen wird, bevor die Farbe berechnet 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) {
                    return
                        new Point(−p.y, p.x);
                }
            };

    // ...

}
```

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

Entwickeln Sie eine Klasse für die Erzeugung von Verschiebungstransformationen, also von Transformationen, die alle Punkte um einen festen Betrag in x– und y–Richtung verschieben.

```
public class Move implements Transform {

        ..........

        ..........

    public Move(..........) {

        ..........

        ..........
    }

    public Point move(Point p1) {

        ..........

        ..........

        ..........

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

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

```
public class ImageTransform implements Image {

        ..........

        ..........

    public ImageTransform(..........) {

        ..........

        ..........
    }

    public Color at(Point p) {

        ..........

        ..........

        ..........

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

Zwei Bilder oder mehrere Bilder können auf unterschiedliche Weise kombiniert werden, zum Beispiel punktweise den Mittelwert der Farben berechnen oder die hellste oder dunkelste Farbe wählen, um neue Bilder zu konstruieren. Die zweistellige Kombination wird wieder mit Hilfe einer Klasse beschrieben:

```
abstract public class CombinedImage implements Image {
    protected Image img1;
    protected Image img2;

    protected CombinedImage(Image i1, Image i2) {
        img1 = i1; img2 = i2;
    }
}
```

Entwickeln Sie eine abgeleitete Klasse für die Kombination zweier Bilder, bei denen punktweise die Farbwerte gemittelt werden.

```
public class MergedImage ..... {

    public MergedImage(.....) {

        .....

        .....
    }

    public Color at(Point p) {

        .....

        .....

        .....

        .....

        .....

        .....
    }
}
```

Gegeben sei die folgende Mini–Anwendung:

```
class Main {
    public static void main(String [] args) {
        Image img1 = SimpleImages.black2white;
        Image img2 = new ImageTransform(img1, Transform.rotate90);
        Image img3 = new MergedImage(img1,img2);

        paint(img1,10,10);
        paint(img2,10,10);
        paint(img3,10,10);
    }
    static void paint(Image img, int w, int h) {
        for (int i = 0; i < w; ++i)
            for (int j = 0; j < h; ++j) {
                paint(img.at(new Point((double)i/w, (double)j/h)), i, j);
    } }
    static void paint(Color c, int x, int y) { /* ... */ }
}
```

Wieviel Objekte der Klasse *Point* werden bei der Ausführung von *paint(img1,10,10);* erzeugt?

. . . . . .

Wieviel Objekte der Klasse *Point* werden bei der Ausführung von *paint(img2,10,10);* erzeugt?

. . . . . .

Wieviel Objekte der Klasse *Point* werden bei der Ausführung von *paint(img3,10,10);* erzeugt?

. . . . . .

Wieviel Objekte der Klasse *Color* werden bei der Ausführung von *paint(img3,10,10);* erzeugt?

. . . . . .