Prof. Dr. Uwe Schmidt

fhwedel
UNIVERSITY OF APPLIED SCIENCES

21. August 2012

Aufgaben zur Klausur **Objektorientierte Programmierung** im SS 2012 (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.

Verwenden Sie in den zu entwickelnden Java Programmteilen keine impliziten Konversionen zwischen einfachen Datentypen und kein Autoboxing und Autounboxing.

Viel Erfolg!

Diese Klausur besteht einschließlich dieses Deckblattes aus 10 Seiten.

Aufgabe 1:

Generische ADTs werden in C++ durch sogenannte *templates* unterstützt, in Java ab 1.5 durch *generics*. Überprüfen Sie die folgenden Aussagen über generische ADTs.

1. Typparameter für Klassen sind ein Konzept, das die Flexibilität von Java– und C++–Programmen einschränkt.

   ja ☐ nein ☐

   Begründung:

   . . . . . .

2. Typparameter für Klasen werden zur Laufzeit des Programms in zusätzlichen Feldern in den Objekten gespeichert und führen so zu größerem Speicherbedarf zur Laufzeit.

   ja ☐ nein ☐

   Begründung:

   . . . . . .

3. In Programmen mit Typparametern können Laufzeit–Überprüfungen auf legale Feldzugriffe oder das Dereferenzieren von null-Referenzen teilweise eliminiert werden.

   ja ☐ nein ☐

   Begründung:

   . . . . . .

4. In Programmen mit Typparametern können Fehler in der Verwendung von Methoden und Variablen teilweise schon zur Übersetzungszeit entdeckt werden.

   ja ☐ nein ☐

   Begründung:

   . . . . . .

5. In Java Programmen mit Typparametern lassen sich up– und down–casts zur Laufzeit vermeiden.

   ja ☐ nein ☐

   Begründung:

   . . . . . .

6. In Java Programmen mit Typparametern können zur Laufzeit noch mehr Überprüfungen gemacht werden als ohne.

   ja ☐ nein ☐

   Begründung:

   . . . . . .

Aufgabe 2:

Graustufenbilder mit einer festen Pixelbreite und -höhe können durch Objekte der folgenden Klasse repräsentiert werden:

```
public abstract class Image {
        public static final double black = 0.0;
        public static final double white = 1.0;

        public final int w;
        public final int h;

        protected Image(int w, int h){
                this.w = w; this.h = h;
        }
        protected abstract double at(int i, int j);

        public double pixelAt(int i,int j){
                return
                Math.max(Math.min(at(i,j),1.0),0.0);
        }
}
```

Die Grauwerte werden hier als **double** Werte im abgeschlossenen Intervall von 0.0 bis 1.0 dargestellt. 0.0 steht für Schwarz, 1.0 für Weiß. Der Zugriff auf einen Bildwert an einer Stelle (i, j) wird durch die Methode $pixelAt$ implementiert. Für die Koordinaten wird im folgenden angenommen, dass $(0, 0)$ auf die linke obere Ecke verweist, $(w-1, 0)$ auf die rechte obere Ecke und $(0, h-1)$ auf die linke untere Ecke.

Hinweis: $Math.max$ und $Math.min$ sind überladene statische Funktionen, die sowohl für **int** als auch für **double** zur Verfügung stehen.

Für einfarbige Bilder ermöglicht die Klasse Uni eine sehr Speicherplatz effiziente Darstellung.

```
public class Uni extends Image {
        private final double color;

        public Uni(int w, int h, double color){
                super(w,h);
                this.color = color;
        }
        protected double at(int i, int j) {
                return color;
        }
}
```

Entwickeln Sie analog zu dieser Klasse eine Klasse $Gradient$ für ein Bild mit einem gleichmäßigen Grauwerteverlauf von oben links (Weiß) nach unten rechts (Schwarz). Sollte ein Verlauf nicht immer berechenbar sein, so soll Schwarz das Resultat sein.

```
public class Gradient extends Image {
        public Gradient(int w, int h){
                super(w,h);
        }

        protected double at(int i, int j) {

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

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

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

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

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

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

            ..........................................
        }

}
```

Die Werte eines beliebigen Bildes können in einer Matrix abgespeichert werden. Dieses wird mit der Klasse $GreyMap$ realisiert.

```
public class GreyMap extends Image {

        private final double [][] pixels;

        public GreyMap(double [][] pixels){
                super(pixels.length,pixels[0].length);
                this.pixels = pixels;
        }
        protected double at(int i, int j) {
                return pixels[i][j];
        }
}
```

Aus diesen einfachen Bildern sollen neue Bilder kombiniert werden können. Eine erste Art des Kombinierens ist das nebeneinander Packen von zwei Bildern. Dieses soll mit der Klasse $SideBySide$ implementiert werden. Das resultierende Bild hat als Breite die Summe der Breiten der beiden Ausgangsbilder, als Höhe das Maximum der Höhen. Stimmen die Höhen nicht überein, so soll das resultierende Bild unten mit Schwarz aufgefüllt werden.

```
public class SideBySide extends Image {
        protected final Image left, right;

        public SideBySide(Image left, Image right){

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

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

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

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

            ..........................................
        }

        protected double at(int i, int j) {

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

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

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

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

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

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

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

            ..........................................
        }

}
```

Die Klasse $HalfWidth$ dient dazu, aus einem Bild ein neues zu konstruieren, das nur noch halb so breit ist. Dabei sollen die Grauwerte gemittelt werden. Bei Bildern mit ungerader Weite ist das Halbieren nicht möglich. Hier soll die rechte Spalte (konzeptionell) verdoppelt werden, so dass die Anzahl der Spalten gerade wird.

Die Implementierung:

```
public class HalfWidth extends Image {
        protected final Image img;

        public HalfWidth(Image img){
                super((img.w +1)/2,img.h);
                this.img = img;
        }
        protected double at(int i, int j) {
                return
                (img.at(2*i,j) + img.at(2*i+1,j))/2.0;
        }
}
```

Diese Implementierung enthält einen Fehler in der Methode at. Entwickeln Sie den Methodenrumpf der at–Methode neu.

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Entwickeln Sie eine analoge Klasse $HalfHeight$ zur Halbierung der Höhe.

```
public class HalfHeight extends Image {
        protected final Image img;

        public HalfHeight(Image img){

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

            ..........................................
        }
        protected double at(int i, int j) {

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

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

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

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

Eine weitere Möglichkeit der Bildkombination ist das Übereinanderlegen (engl.: superimpose) zweier Bilder. Die folgende Klasse $Superimpose$ definiert so eine Kombinierung. Das resultierende Bild hat die kleinste Geometrie, in die beide Bilder passen.

```
public class Superimpose extends Image {
        protected final Image top,bottom;

        public Superimpose(Image top, Image bottom){
                super(Math.max(top.w,bottom.w),Math.max(top.h,bottom.h));
                this.top = top;
                this.bottom = bottom;
        }

        protected double at(int i, int j) {
                if (i<top.w && j<top.h)
                        return top.at(i,j);
                if (i<bottom.w && j<bottom.h)
                        return bottom.at(i,j);
                return black;
        }
}
```

Eine mächtigere Variante dieser Operation ist das Übereinanderlegen mit der Angabe einer Transparenz oder hier einer Deckkraft (engl.: opacity) für das obere Bild. Ist die Deckkraft 1.0, so erhält man die gleiche Funktionalität wie bei $Superimpose$, ist die Deckkraft 0.0, so ist das obere Bild vollständig transparent, bei 0.5 werden beide Werte gemittelt. Diese Funktionalität soll für die überlappenden Teile der Bilder gelten, sonst soll die Funktionalität wie die aus $Superimpose$ sein.

```
public class SuperimposeWithTransparency extends

    ..........................................
    {
        protected final double opacity;

        public SuperimposeWithTransparency(
                                Image top,
                                Image bottom,
                                double opacity){

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

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

        ..........................................
        }

        protected double at(int i, int j) {

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

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

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

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

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

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

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

Die Klasse $ImageCache$ dient dazu, ein Bild vollständig zu berechnen und die Werte in einem Feld abzuspeichern. Dieses kann dann von Nutzen sein, wenn das Bild aus vielen anderen Bildern auf komplexe Weise zusammengesetzt worden ist und dieses Bild mehrfach weiterverwendet werden soll.

```
public class ImageCache extends Image {
        protected final Image img;

        public ImageCache(Image img){
                super(img.w,img.h);
                this.img = new GreyMap(eval(img));
        }

        private
        double [][] eval(Image img) {
                double [][] a = new double[img.w][img.h];
                for (int i=0; i<img.w;++i)
                        for(int j=0; j<img.h;++j)
                                a[i][j] = img.pixelAt(i,j);
                return a;
        }

        protected double at(int i, int j) {
                return img.at(i,j);
        }
}
```

Die Nutzung dieser Klasse kann aber auch zu Ineffizienzen führen, wenn das Ursprungsbild einfach zu berechnen ist. Optimieren Sie die Klasse, um diese Ineffizienzen zu vermeiden.

Die optimierten Klassenteile:

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Für die erzeugten Bilder muss die Konsistenzbedingung gelten, dass sowohl Weite als auch Höhe größer 0 sind. In welche Klasse(n) und an welche Stelle(n) würde man diese Überprüfung einbauen:

. . . . . .

. . . . . .

. . . . . .

Für die Zugriffe auf die einzelnen Bildpunkte wird keine explizite Laufzeitüberprüfung gemacht. In welche Klasse(n) und an welchen Stelle(n) würde man diese Überprüfung einbauen:

. . . . . .

. . . . . .

. . . . . .