Funktionale und objektorientierte Programmierkonzepte

Florian Kadner und Lukas Röhrig
(Gesamtleitung: Prof. Karsten Weihe)

Wintersemester 18/19
v1.0

Übungsblatt 13

Themen: Wrap-Up III

Relevante Folien: alle bis dahin

Abgabe der Hausübung: 15.02.2019 bis 23:55 Uhr

V Vorbereitende Übungen

V1 Bildverarbeitung

In dieser Aufgabe wollen wir uns mit dem Filtern von Bildern beschäftigen. Dies ist in der digitalen Bildverarbeitung eine häufig verwendete Technik um beispielsweise Bilder zu schärfen oder zu blurren[1]. Zur Bearbeitung dieser Aufgabe finden Sie eine Vorlage namens `UE13_V1` in moodle.

Portable Graymaps

Als Datenformat für die verwendeten Bildern benutzen wir hier Portable Graymaps. Das PGM-Format dient zur Speicherung von Rastergrafiken und kann Bilder in Graustufen darstellen. Eine PGM Datei ist dabei immer folgendermaßen aufgebaut:

```
Magischer Wert (Steht fuer das Format der Bilddaten)
# Kommentare
Breite Hoehe
Maximalwert der Helligkeit

Bilddaten in Form von Pixel
```

Als Beispiel sehen Sie sich Abbildung 1 an. Dort ist das Kürzel FOP durch folgende PGM-Datei kodiert:

```
P2
# Das Kuerzel FOP kodiert als PGM
```

[1] Blurring = Weichzeichnen

```
18 7
15
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  3  3  3  3  0  0  7  7  7  7  0  0 15 15 15 15  0
0  3  0  0  0  0  0  7  0  0  7  0  0 15  0  0 15  0
0  3  3  3  0  0  0  7  0  0  7  0  0 15 15 15 15  0
0  3  0  0  0  0  0  7  0  0  7  0  0 15  0  0  0  0
0  3  0  0  0  0  0  7  7  7  7  0  0 15  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
```

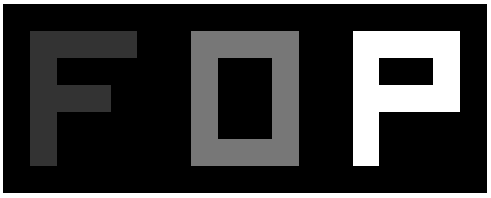

Abbildung 1: FOP als PGM-Datei

PGM-Dateien können mit einem Bildeditor wie beispielsweise GIMP visualisiert werden.

Um das Einlesen der Bilder brauchen Sie sich in dieser Übung nicht zu kümmern, das übernimmt die von uns vordefinierte Klasse `Image` für Sie. Diese enthält folgende Attribute:

- `String magicNumber`: Die magische Zahl, welche das Format der Bilddatei angibt (kann von Ihnen ignoriert werden)
- `int width`: Die Breite des Bildes (Anzahl an Pixeln)
- `int height`: Die Höhe des Bildes (Anzahl an Pixeln)
- `float max`: Der maximale Helligkeitswert des Bildes
- `float[][] data`: Die Bilddaten als Pixel in einem 2-dimensionalen Array gespeichert (erste Dimension Höhe, zweite Breite)

Außerdem liefert Sie folgende Methoden:

- Die Getter `String getMagicNumber()`, `float getData()` `float getMax()`, `int getWidth()` und `int getHeight()`, welche die jeweiligen Attribute zurückliefern
- Die Methode `float getPixel(int height, int width)`, welche den Pixelwert an der gegebenen Stelle zurückgibt
- Die Methode `float setPixel(float value, int height, int width)`, welche den Pixelwert an der gegebenen Stelle überschreibt
- Der Konstruktor `Image(Path p)`, welche den Pfad zur Bilddatei entgegennimmt und einliest
- Die Methode `save(Path p)`, welche das Bild am angegebenen Pfad abspeichert

Alles andere in der Klasse ist für Sie nicht von Bedeutung, ändern Sie in dieser Klasse auch in keinem Fall etwas ab! Wir liefern Ihnen in dieser Übung mehrere Beispielbilder mit, mit denen auch unsere Tests laufen.

V1.1 Lineares Filtern

Beim linearen Filter wird die sogenannte diskrete Faltung verwendet. Dies funktioniert folgendermaßen:

Das Eingabebild wird durch ein zweidimensionales Array repräsentiert, welches wir als Bildmatrix B bezeichnen. Für das Beispiel in Abbildung 1 sieht der Anfang der Bildmatrix folgendermaßen aus:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & \cdots \\ 0 & 3 & 3 & 3 & \cdots \\ 0 & 3 & 0 & 0 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

Betrachten Sie zur besseren Verständlichkeit Abbildung 2, in der die Berechnung eines Pixels exemplarisch vorgemacht ist.

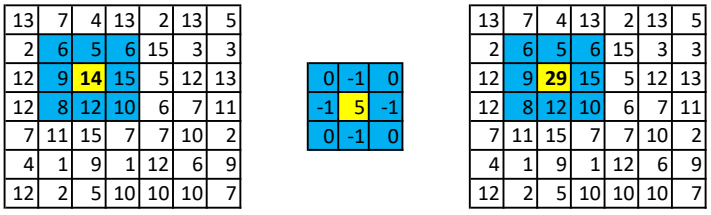

Abbildung 2

Auf die Bildmatrix B wird eine Kernelmatrix K angewendet. Dabei wird jeder Pixel des Ursprungsbildes durch eine Linearkombination seiner Nachbarschaft und sich selbst ersetzt. Die Einträge der Kernelmatrix K stellen dabei die Gewichte dieser Linearkombination dar. Bezeichnen wir A als Ausgangsbild und mit $A(i, j)$ den Pixel in der i-ten Zeile und in der j-ten Spalte, dann ist die diskrete Faltung definiert als: $A := B * K$. Die neuen Pixelwerte des Ausgangsbildes berechnen sich durch:

$$A(i,j) = \sum_{x=1}^{n} \sum_{y=1}^{n} B(i + x - a_1, j + y - a_2) \cdot K(x, y)$$

In der Formel oben steht dabei n für die Filtergröße, a_1 für die x-Koordinate und a_2 für die y-Koordinate des Kernelmittelpunktes.

Für Pixelwerte am Rand ist der Kernel eventuell zu groß und greift ins Leere. Dafür wird eine sogenannte Randbehandlungsstrategie benötigt. Die Werte außerhalb des Bildrandes werden dann einfach dem Wert des örtlich nächsten Nachbarns auf dem Bildrand gleichgesetzt. Betrachten Sie Abbildung 3 für diesen Spezialfall.

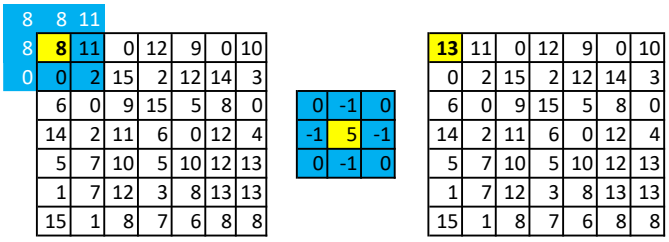

Abbildung 3

Hinweis: Sollte durch das Filtern keine natürliche Zahl herauskommen, wird diese auf die nächst kleinere ganze Zahl abgerundet. Weiterhin müssen Werte kleiner als 0 auf 0 und Werte größer als das Maximum auf das Maximum gesetzt werden.

In Abbildung 4 sehen Sie ein Beispielbild, welches mittels diskreter Faltung geschärft und weichgezeichnet wurde.

Ausgangsbild Schärfung Weichzeichner

Abbildung 4

Beispiele für solche Kernel sind:

$$\text{Schaerfen: } \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix} \qquad \text{Weichzeichnen: } \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

V1.2 Implementieren des Linearen Filters

Ergänzen Sie die Methode `linearFilter(Image image, float[][] kernel)` in der Klasse `Filter`, die das lineare Filtern mithilfe der diskreten Faltung implementiert. Beachten Sie hierbei auch die Behandlung des Bildrandes wie oben beschrieben.

V1.3 Nichtlineares Filtern

Das nichtlineare Filtern stellt eine Besonderheit des Filterns dar. Anstelle einer Linearkombination betrachten wir hier nur die Nachbarschaft des aktuellen Pixels und verwenden hier einen sogenannten Rangordnungsfilter. In dieser Hausübung beschäftigen Sie sich mit dem Medianfilter. Ein $n \times n$- Medianfilter sucht sich alle Pixel in einer Umgebung der Größe $n \times n$ und bringt diese in eine Rangordnung. Anschließend wird der Median dieser Rangordnung gebildet und der Pixel durch diesen Medianwert ersetzt. Als Beispiel betrachten Sie Abbildung 5 für einen 3×3 Medianfilter.

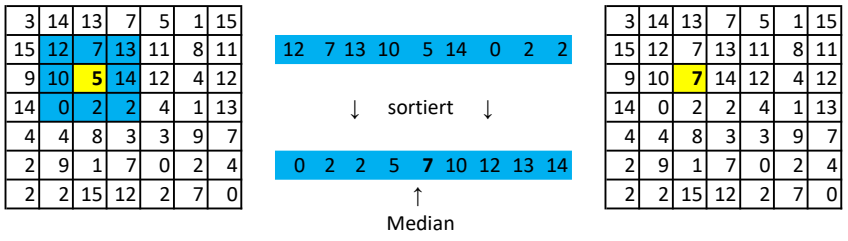

Abbildung 5

Hier wird die identische Randbehandlungsstrategie wie beim Linearen Filtern verwendet. Sie ersetzen also Pixelwerte außerhalb des Bildrandes mit den örtlich lokalen Nachbarn des Bildrandes. In Abbildung 6 sehen Sie ein Beispiel für ein Bild welches mittels eines 5×5 und eines 9×9 Medianfilters weichgezeichnet wurde.

Ausgangsbild 5×5 Medianfilter 9×9 Medianfilter

Abbildung 6

V1.4 Lokale Nachbarschaft

Zur einfacheren Implementierung des Medianfilters steht Ihnen die Klasse `LocalNeigbours` zur Verfügung. Diese dient der Speicherung der aktuellen Nachbarschaftspixel. Folgende Methoden stehen Ihnen hier zur Verfügung:

- Die Methode `void addPixel(float Pixel)` fügt der lokalen Nachbarschaft einen neuen Pixelwert hinzu.
- Die Methode `int getSize()` gibt die Anzahl der abgespeicherten Pixel in der lokalen Nachbarschaft zurück.
- Die Methode `Float[] getPixels()` gibt ein Array mit den gespeicherten Pixelwerten zurück. Die Reihenfolge dieser Pixelwerte entspricht dabei genau Reihenfolge ihrer Speicherung.

V1.5 Sortieren

Ergänzen Sie die Methode `sort(float[] Array)` der Klasse `LocalNeigbours`. Diese nimmt die aktuellen Pixel, welche in der lokalen Nachbarschaft abgespeichert wurden und sortiert diese. Zum Sortieren setzen Sie den unten angegebenen Pseudo-Code um. Es genügt also nicht das Array zu sortieren, Sie sollen auch den angegebenen Sortieralgorithmus umsetzen.

```
sort(A)
  beginn := -1
  ende := Laenge(A)
  solange beginn kleiner als ende wiederhole
    beginn := beginn + 1
    ende := ende - 1
    fuer (i = beginn, solange i < ende, i++) wiederhole
      falls A[i] > A[i + 1] dann
        vertausche(A[i], A[i+1])
      ende falls
    ende fuer
    fuer (i = ende, solange --i >= beginn) wiederhole
      falls A[i] > A[i + 1] dann
        vertausche(A[i], A[i+1])
      ende falls
    ende fuer
  ende wiederhole solange
```

V1.6 Median

Ergänzen Sie die Methode `getMedian(float[] Array)` der Klasse `LocalNeigbours`. Diese soll den Median der aktuell gespeicherten Pixel der lokalen Nachbarschaft zurückgeben. Achten Sie bei Ihrer Implementierung darauf, dass der Median sowohl bei einer geraden als auch einer ungeraden Anzahl an Pixeln funktionieren soll.

V1.7 Medianfilter

Ergänzen Sie die Methode `nonLinearFilterMedian(Image image, int filtersize)`, die das nichtlineare Filtern mithilfe eines Medianfilters implementiert. Beachten Sie hierbei auch wieder die passende Randbehandlung! Verwenden Sie die Klasse `LocalNeighbours` und die von Ihnen ergänzten Methoden um den Medianfilter korrekt umzusetzen.

V2 Ticketverkauf

Implementieren Sie eine `public`-Methode

```
sellTickets(Collection<? extends TicketRequest> requests, int capacity)
```

mit Rückgabetyp `List<TicketRequest>`. Diese Methode erhält eine nach Eingangszeitpunkt sortierte Liste von Ticketanfragen. Die Ticketanfragen erhalten neben für diese Aufgabe irrelevanten Daten wie Kundenname etc. insbesondere den Status der Kundenanfrage, modelliert als `enum ReqType`. Aufgrund der sehr hohen Nachfrage werden drei Statuswerte unterschieden: `MEMBER` für Mitglieder, `SUB` (kurz für subscription) für Nicht-Mitglieder mit Dauerkarte sowie `REGULAR` für normale Kunden (weder Mitglieder noch Dauerkarteninhaber).

Für den Verkauf steht eine Anzahl `capacity` für die Tickets zur Verfügung. Der Verkauf soll wie folgt erfolgen:

1. Als erstes erhalten alle Mitglieder in der Reihenfolge der Liste ein Ticket, solange die Ticketanzahl noch nicht erreicht wurde.
2. Sind noch Tickets verfügbar, werden diese im zweiten Schritt an alle Dauerkarteninhaber vergeben, solange die Kapazität ausreicht.
3. Sind immer noch Tickets verfügbar, werden auch die normalen Kunden bis zur Kapazitätsgrenze bedient.

Als Ergebnis ist eine Liste mit allen verkauften Tickets zu liefern. Bitte beachten Sie, dass die Einträge der Liste nicht nach Status sortiert sind. Zusätzlich ist davon auszugehen, dass die Nachfrage das Ticketangebot (deutlich) übersteigt.

Verbindliche Anforderung: Die Eingabeliste darf nicht verändert werden! Insbesondere ist das Einfügen, Löschen einzelner Elemente oder das Sortieren der Liste verboten.

V3 Klassen und Interfaces

V3.1

Schreiben Sie ein `public`-Interface `X`, das eine Objektmethode `m1` mit Rückgabetyp `java.lang.Exception` hat, wobei `m1` einen `double`-Parameter `n` und einen `String`-Parameter `str` hat.

V3.2

Schreiben Sie ein `public`-Interface `Y`, das von `X` erbt und zusätzlich eine Klassenmethode `m2` ohne Parameter hat, die `double` zurückliefert.

V3.3

Schreiben Sie eine `public`-Klasse `XY`, die `X` und Methode `m1` implementiert. Konkret soll Methode `m1` eine Referenz auf ein Objekt vom Typ `NullPointerException` zurückliefern, das mit Konstruktor ohne Parameter eingerichtet wird. Klasse `XY` soll ein `public`-Attribut `p` vom Typ `char` haben sowie einen `public`-Konstruktor ohne Parameter. Der Konstruktor soll `p` auf den Wert 42 setzen. Weiter soll `XY` eine `protected`-Objektmethode `m3` mit Rückgabe `true` oder `false` und Parameter `xy` vom Typ `XY` haben, aber nicht implementieren.

V3.4

Schreiben Sie eine `public`-Klasse `YZ`, die von `XY` erbt und sowohl `Y` als auch das Interface `java.util.Comparator<Integer>` implementiert. Die Methoden `m1` und `m2` sollen nicht in `YZ` überschrieben bzw. implementiert sein, und `m3` soll genau dann `true` zurückliefern, wenn der Wert `p` von `xy` gleich dem Wert `p` des eigenen Objektes ist. Der Konstruktor von `YZ` ist `public`, hat einen `long`-Parameter `r` und ruft den Konstruktor von `XY` auf. Methode `compare` von `Comparator<Integer>` hat bekanntlich zwei Parameter vom Typ `Integer` und liefert `int` zurück; in YZ soll sie +1 zurückliefern, wenn der `int`-Wert im ersten Parameter um mindestens 5 höher als der `int`-Wert im zweiten Parameter ist, +1, wenn es genau umgekehrt ist, und 0 sonst.

V4 Mittelwert

Gegeben sei eine Klasse `X`, die generisch mit `T` parametrisiert ist.

Die `public`-Objektmethode `m1` hat ein Array `a` mit Komponententyp Array von `int` als Parameter und gibt ein Array von `double` zurück, das dieselbe Länge wie `a` haben soll. An jedem Index `i` von `a` soll das zurückgelieferte Array das arithmetische Mittel der Werte in `a[i]` enthalten.

Ihre Methode kann ohne Überprüfung davon ausgehen, dass der Parameter `a` nicht gleich `null` ist und dass `a` positive Länge hat.

Verbindliche Anforderung:

Rekursion darf nicht verwendet werden, das heißt, die Aufgaben müssen durch Schleifen realisiert werden. Dies müssen `for`-Schleifen sein. Zudem dürfen keine Klassen aus der Java-Standardbibliothek oder aus anderen Bibliotheken verwendet werden (Arrays fallen natürlich nicht darunter).

H Dreizehnte Hausübung — Gesamt 8 Punkte
Fraktale

H1 Visualisierung komplexer Zahlen — 5 Punkte

In der Mathematik haben Sie die komplexen Zahlen kennengelernt. Diese bilden einen Körper, auf dem wir Operationen wie Addition und Multiplikation definieren können. Eine komplexe Zahl $z \in \mathbb{C}$ lässt sich dabei ausdrücken als:

$$z = a + bi \quad \text{mit } a, b \in \mathbb{R}$$

Dabei nennen wir a den realen und b den imaginären Anteil von z. In dieser Aufgabe betrachten wir nun die Folge:

$$z_{n+1} = z_n^2 + c \quad \text{mit } z, c \in \mathbb{C} \quad \text{und } z_0 = 0$$

Die Mandelbrot-Menge ist nun definiert als Menge aller $c \in \mathbb{C}$, für welche die Menge z_n beschränkt ist. Formal ausgedrückt ist die Mandelbrot-Menge also definiert als:

$$\mathcal{MB} = \{c \in \mathbb{C} | \exists b \in \mathbb{R} \forall n \in \mathbb{N} : |z_n| < b\}$$

Wir betrachten nun einen kleinen Ausschnitt der komplexen Zahlenebene und ordnen dabei jedem Pixel eine komplexe Zahl zu. In Abbildung 7 sehen Sie ein Beispiel für eine Mandelbrot-Menge.

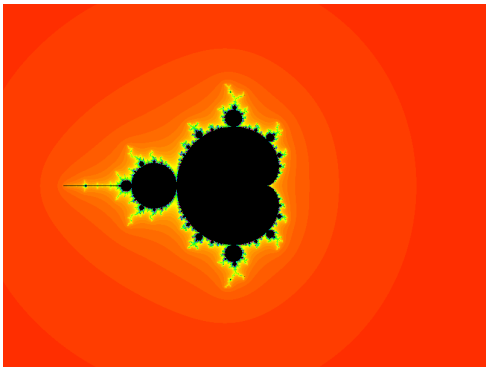

Abbildung 7: Visualisierung einer Mandelbrot-Menge

Zur Visualisierung dieser Zahlenebene iterieren wir über alle Bildpunkte. Dafür legen wir eine maximale Anzahl an Iterationen $n_{\max}$, sowie einen Radius $r_{\max} = b$ fest. Für jeden Pixel wird dann die oben genannte Folge maximal $n_{\max}$-mal iteriert. Die Folge ist genau dann beschränkt, wenn gilt:

$$\forall n \leq n_{\max} : |z_n| \leq r_{\max}$$

Besitzt ein Pixel eine beschränkte Folge, so wird dieser als schwarz eingefärbt. Für alle anderen Pixel wird das größte n gespeichert, für welches $|z_n| \leq r_{\max}$, um daraus später einen Farbwert zu bestimmen. Bildlich ist dies die Anzahl an Iterationen, in denen der Kreis mit Radius $r_{\max}$ nicht verlassen wurde.

Zum Schluss wird diese Information noch in ein Farbschema überführt und einer Farbe zugeordnet. Im Pseudo-Code sieht dieses Vorgehen **pro Pixel** (x, y) folgendermaßen aus:

Algorithm 1 Generate Mandelbrot set

```
1:  procedure GENERATE
2:      z ← 0 + 0i
3:      c ← getComplexValueForPixel(x,y)
4:      i ← 0
5:  loop
6:      if |z| ≥ rMax or i > maxIter then break
7:      z ← z · z + c
8:      i + +
9:  end loop
10:     color ← colorMap(i)
11:     setPixelColor(x, y, color)
```

Um die Werte für c zu bestimmen, beziehen Sie einen Zoomfaktor mit ein. Für den realen Teil der komplexen Zahl bilden wir die Differenz der aktuellen Pixelhöhe zur Hälfte der Bildhöhe und teilen diese Differenz durch den Zoomfaktor. Für den imaginären Teil machen wir das Äquivalente, nur mit der Breite.

Die Farbe bestimmen wir über den HSV-Farbraum. Dabei wählen wir:

$$\text{color} = \begin{cases} i/100 & = \text{H} \\ 1 & = \text{S} \\ i < n_{\max} \ ? \ 1 \ : \ 0 & = \text{V} \end{cases}$$

Zur Bearbeitung sind bereits folgende (teils unvollständige) Klassen im Package H1 gegeben:

- `ComplexNumber` dient zur Modellierung der komplexen Zahlen.
- `Set` ist die (fertig implementierte) Oberklasse der Klasse `MandelbrotSet`. Sie besitzt eine abstrakte Methode `generate`, sowie einige Attribute zur späteren Berechnung der Mandelbrot-Menge.
- `Visualizer` stellt die Menge später graphisch da. Diese Klasse ist bereits fertig und muss nicht von Ihnen bearbeitet werden.
- `Main` zum Aufruf der Methoden.

Haben Sie später alles implementiert, sollte der folgende Aufruf, welcher bereits in der Mainklasse zu finden ist, das Beispielbild in Abbildung 7 zurückliefern:

```
Set mb = new Mandelbrot();
new Visualizer(mb, 570, 150, 2 << 4).setVisible(true);
```

In den folgenden Teilaufgaben erhalten Sie weitere Informationen zur Implementierung.

H1.1 Komplexe Zahlen modellieren — 1 Punkt

Ergänzen Sie in der Klasse `ComplexNumber` folgende drei Methoden:

- `ComplexNumber add(ComplexNumber cn)`: addiert eine zweite komplexe Zahl mit der aktuellen und liefert das Ergebnis zurück
- `ComplexNumber mult(ComplexNumber cn)`: multipliziert eine zweite komplexe Zahl mit der aktuellen und liefert das Ergebnis zurück
- `double abs()`: liefert den Betrag der komplexen Zahl zurück

Die Rechenregeln finden Sie beispielsweise in der deutschsprachigen Wikipedia hier.

H1.2 Mandelbrot-Menge bestimmen — 2 Punkte

Ergänzen Sie nun in der Klasse `MandelbrotSet` die Methode `generate`. Diese bekommt die Höhe und Breite des zu erstellenden Bildes übergeben, sowie den Zoomfaktor, den Radius und die maximale Anzahl an Iterationen. Wenden Sie zur Berechnung den vorgestellten Algorithmus pro Pixel an und liefern Sie das fertige Bild zurück.

H1.3 Julia-Menge bestimmen — 1 Punkt

Zum Abschluss wollen wir uns eine zweite Menge, die Julia-Menge, anschauen. Diese steht in Beziehung zur Mandelbrot-Menge, da die Mandelbrot-Menge eine Beschreibungsmenge der Julia-Menge quadratischer Polynome ist. Sei nun wieder $z \in \mathbb{C}$ und sei p ein Polynom, dann ist die Julia-Menge dieses Polynoms definiert als:

$$z_{n+1} = p(z_n) \quad \text{mit } z \in \mathbb{C}$$

Wir sehen also schnell, dass wir die Mandelbrot-Menge genau dann erhalten, wenn wir für p ein quadratisches Polynom mit linearem Glied gleich 0 wählen. In Abbildung 8 sehen Sie das Beispiel für die Visualisierung einer Julia-Menge.

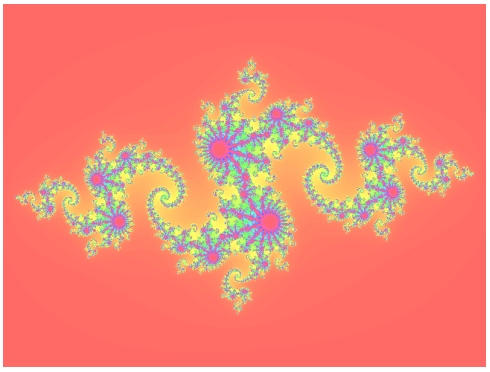

Abbildung 8: Visualisierung einer Julia-Menge

Der Unterschied der Generierung zwischen Julia- und Mandelbrotmenge liegt in der Iteration. Ihre Aufgabe ist es diesmal die Methode `generate` in der Klasse `JuliaSet` zu ergänzen. Dafür ist wieder ein Algorithmus angegeben.

Algorithm 2 Generate Julia set

```
1:  procedure GENERATE
2:      z ← chooseStartValue()
3:      c ← getComplexValueForPixel(x,y)
4:      i ← 0
5:  loop
6:      if |c| ≥ rMax or i > maxIter then break
7:      newReal ← Re(c)² − Im(c)²
8:      Im(c) ← 2 * Re(c) * Im(c)
9:      Re(c) ← newReal
10:     c ← c + z
11:     i + +
12: end loop
13:     color ← colorMap(i)
14:     setPixelColor(x, y, color)
```

Die Julia-Menge ist dabei abhängig vom gewählten Startwert von z. Um das Beispielbild zu erhalten wählen Sie $z_0 = 0.156i - 0.8$.

Das Farbschema für den HSV-Farbraum ändern wir hier minimal ab:

$$\text{color} = \begin{cases} (i \mod 256)/255 & = \text{H} \\ 0.6 & = \text{S} \\ i < n_{\max} \ ? \ 1 \ : \ 0 & = \text{V} \end{cases}$$

Haben Sie alles implementiert, sollte der folgende Aufruf, welcher bereits in der Mainklasse zu finden ist, das Beispielbild in Abbildung 8 zurückliefern:

```
Set j = new Julia();
new Visualizer(j, 250, 250, 2).setVisible(true);
```

H1.4 Erkunden der Menge

Diese Aufgabe wird nicht bepunktet und stellt eine freiwillige Möglichkeit dar. Nach erfolgreicher Implementierung können Sie einmal am Zoomfaktor und den Werten für c herumspielen und so näher in die Mengen hereinzoomen. So erhalten Sie ganz neue Einblicke.

Funktioniert Ihre Implementierung der Julia-Menge, sodass Sie das Beispielbild erhalten, können Sie einmal verschiedene Startwerte z_0 ausprobieren. Hier einige Startwerte, welche einige kreative Mengen hervorrufen:

- $z_0 = -0.81i$
- $z_0 = 0.6i - 0.4$
- $z_0 = -0.3842i - 0.70176$

H2 Drachenkurve 2 Punkte

Die sogenannte Drachenkurve ist eine fraktale Kurve, die aus wenigen Schritten konstruiert werden kann. Zur Erstellung dieser Kurve erstellen wir den Drachencode. Dieser Code gibt an, wie wir die Linien für die Drachenkurve zu zeichnen haben. Im Code selbst kodieren wir dabei durch einen String `"R"` eine Drehung um 90 Grad nach rechts und mit `"L"` entsprechend eine Drehung um 90 Grad nach links. Zwischen diesen Drehungen wird später eine Linie gezogen. Es gilt folgende Vorschrift:

- Der Drachencode 0-ter Ordnung ist leer, also `""`.
- Der Drachencode 1-ter Ordnung ist `"R"`.
- Der Drachencode n-ter Ordnung ist der Drachencode (n-1)-ter Ordnung mit einem zusätzlichen `"R"` am Ende. An diesen Code wird dann nochmals der Drachencode (n-1)-ter Ordnung gehängt, wobei jedoch das mittlere Element durch ein `"L"` ersetzt wird.

Der gesamte Drachencode wird dann als einziger String, bestehend aus einer Sequenz von `"L"` und `"R"`, kodiert.

H2.1 Drachencode erzeugen 1 Punkt

Ergänzen Sie die Funktion `String getSequence(int n)` in der Klasse `DragonCurve`, welche den Drachencode n-ter Ordnung generiert und zurückliefert.

Verbindliche Anforderung: Ihre Implementierung muss rekursiv arbeiten, Sie dürfen also keine Schleifen verwenden.

H2.2 Drachenkurve zeichnen 1 Punkt

Als nächstes wollen wir die Drachenkurve anhand des Drachencodes zeichnen. Für den Drachencode 14-ter Ordnung finden Sie die Drachenkurve in Abbildung 9.

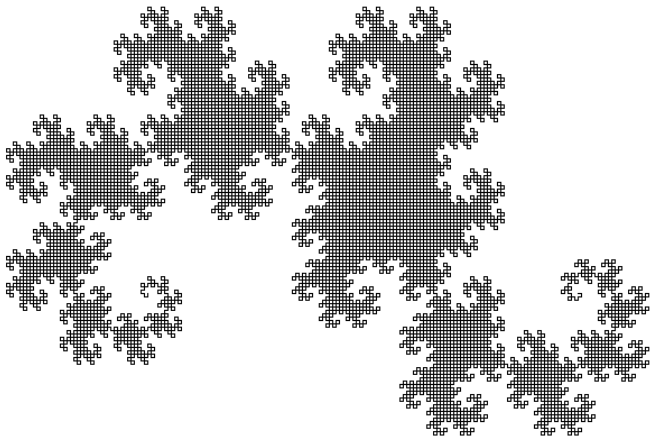

Abbildung 9: Drachenkurve für n=14

Ergänzen Sie die Methode `paint()` in der Klasse `Visualizer`. Die Klasse besitzt die `double`-Attribute `startingAngle` (der Winkel in den wir die erste Linie ziehen) und `len` (die Länge der einzelnen Linien), sowie das Attribut `turn` vom Typ `String`, welches den Drachencode aus der vorherigen Aufgabe darstellt.

Sie starten am Punkt $(250, 375)$. Zeichnen Sie von diesem Startpunkt ausgehend eine Linie der Länge `len` unter dem Winkel `startingAngle`. Von dort an aktualisieren Sie den Winkel immer gemäß des Drachencodes, um 90-Grad Drehungen nach links oder rechts. Nach dieser Drehung zeichnen Sie wieder eine Linie der Länge `len` und wiederholen dies solange, bis Sie den gesamten Drachencode abgearbeitet haben.

Hinweis:
Eine Linie von (x_1, y_1) zu (x_2, y_2) realisieren Sie durch `g.drawLine(x1, y1, x2, y2)`.

H3 Sierpinski-Teppich 2 Punkte

Als letztes Fraktal werfen wir einen Blick auf den Sierpinski-Teppich. Gegeben sei ein Quadrat, von diesem entfernen wir in der Mitte genau ein Neuntel der Fläche (hier: schwarz färben). Nun verbleiben wieder 8 quadratische Flächen. In jeder dieser Flächen entfernen wir wieder ein Neuntel der Fläche in der Mitte und immer so weiter. Sei der Flächeninhalt zu Beginn $A_0 = 1$, dann beträgt der Flächeninhalt bzw. der Anteil an weißer Flächen nach $n + 1$ Iterationen:

$$A_{n+1} = A_n - \frac{8^n}{9^{n+1}}$$

Ein Beispiel für einen Sierpinski-Teppich nach 3 Iterationen finden Sie in Abbildung 10.

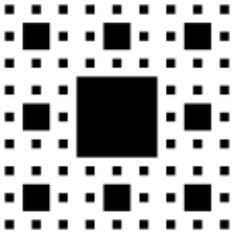

Abbildung 10: Sierpinski Teppich nach 3 Iterationen

Ergänzen Sie die Methode `paint` in der Klasse `SierpinskiCarpet`. In der Klasse wird bereits ein quadratisches Fenster der Länge `len` (gespeichert im gleichnamigen `private`-Attribut) erzeugt. Die Methode `paint` soll nun den Sierpinski-Teppich nach `n` (ebenfalls als `private`-Attribut verfügbar) Iterationen in dieses Fenster zeichnen.

Hinweis:

Über `g.fillRect(x, y, width, height)` erzeugen Sie ein gefülltes Rechteck, dessen oberer linker Punkt bei (x, y) liegt und die spezifizierte Länge und Breite besitzt.

Verbindliche Anforderung: Ihre Lösung muss rekursiv arbeiten, es sind also keine Schleifen erlaubt!