Funktionale und objektorientierte Programmierkonzepte

TECHNISCHE UNIVERSITÄT DARMSTADT

Alexander Mainka, Kim Berninger,
Dustin Glaser, Jan Bormet und
Julius Hardt
(Gesamtleitung: Prof. Dr. Karsten Weihe)

Wintersemester 20/21
v1.0.2

Übungsblatt 4

Themen:	Klassen, Interfaces, Schleifen, Arrays, Methoden
Relevante Foliensätze:	01d-g, 03a-03c
Abgabe der Hausübung:	11.12.2020 bis 23:50 Uhr

V Vorbereitende Übungen

V1 Grundbegriffe ☆ ☆ ☆

Erklären Sie kurz in eigenen Worten die Unterschiede der folgenden Konzepte zueinander:

1. Klasse vs. Objekt
2. Objekt- vs. Klassenmethoden
3. Abstrakte Klassen vs. Interfaces
4. Überladen von Methoden vs. Überschreiben von Methoden

V2 Brumm, Brumm, Brumm ★ ☆ ☆

Schreiben Sie eine Klasse `Car` zur Repräsentation von Autos, die folgende Anforderungen erfüllen soll:

- Ein Auto hat einen Namen vom Typ `String` und einen Kilometerstand `mileage` vom Typ `double`. Beide Attribute sollen `private`, nicht `public`, sein.
- Der Konstruktor soll einen String als Parameter erhalten, der den Namen des Autos angibt. Der Konstruktor soll den Namen des Autos setzen und den Kilometerstand auf `0.0` setzen.
- Schreiben Sie die Methoden `public double getMileage()` und `public String getName()`. Diese liefern die entsprechenden Attribute der Klasse `Car` zurück.
- Schreiben Sie die Methode `public void drive(double distance)`, die eine Distanz in Kilometern als Argument erhält und auf den alten Kilometerstand addiert.

V3 Interfaces ★ ☆ ☆

Gegeben sei folgendes Inferface:

```java
public interface I1 {
    void m1();
    int[] m2(double[] param1);
}
```

Schreiben Sie nun eine Klasse `C1`, die das Interface `I1` implementiert. Sofern im Body einer Methode eine Rückgabe erwartet wird, geben sie `null` zurück.

V4 Vererbung ★ ☆ ☆

Gegeben seien folgende zwei Klassen:

```java
class B1 {
    public float f;
    private boolean b;
    protected byte by;

    int m1() {
        return -1;
    }

    private int m2() {
         return -2;
    }
}

public class B2 extends B1 {
    int i;
    public double d;

    protected int m2() {
        return 2;
    }

    public static void main(String[] args) {
        B2 obj = new B2();
    }
}
```

Betrachten Sie die `main`-Methode der Klasse `B2`. Auf welche Attribute können Sie mit dem Objekt `obj` zugreifen? Welche Methoden können Sie aufrufen und welchen Wert geben die Methoden zurück?

V5 Gleicher Abstand ★ ★ ☆

Schreiben Sie eine Methode `public static boolean evenlySpaced(int a, int b, int c)`, welche genau dann `true` zurückliefert, wenn der Abstand zwischen dem kleinsten und dem mittleren Element genauso groß ist wie der Abstand zwischen dem mittleren und dem größten Element. Dabei kann jeder der Parameter `a`, `b` oder `c` das kleinste, mittlere oder größte Element sein. Die Klasse, zu der die Methode gehört, muss nicht implementiert werden.

V6 Zahlen aneinanderreihen ★ ★ ☆

Schreiben Sie eine Methode `public static int appendIntegers(int[] a)`. Die Methode bekommt ein `int`-Array übergeben und liefert eine Zahl zurück, die entsteht wenn man alle Zahlen des übergebenen Arrays aneinanderreiht. Der Aufruf `appendIntegers(new int[] {1,2,3})` liefert `123` zurück. Der Aufruf `appendIntegers(new int[] {43,2,7777})` liefert `4327777` zurück. Sie dürfen nur Variablen von Typ `int` in ihrer Implementation verwenden, keine `Strings` oder Ähnliches. Schleifen sind natürlich erlaubt.

V7 Zahlen einsortieren ★ ★ ☆

Gegeben sei folgende Klasse:

```
1 public class ArrayTuple {
2     public int[] iArr;
3     public double[] dArr;
4 }
```

Erweitern Sie diese Klasse um eine `public`-Klassenmethode `ArrayTuple split(double[] a)`. Die Methode liefert ein neues Objekt von Typ `ArrayTuple` zurück, in dessen `int`-Array sich alle ganzen Zahlen aus dem übergebenen Array `a` befinden. Im `double`-Array befinden sich die restlichen Zahlen aus dem übergebenen Array.

V8 Statischer und dynamischer Typ ★ ★ ☆

Betrachten Sie das folgende Programm und beantworten Sie die nachfolgenden Fragen über die Zustände beim Ausführen der `main`-Methode der Klasse `Gamma`:

```
1  class Alpha {
2      protected int v;
3
4      public Alpha(int a) {
5          v = a;
6      }
7  }
8
9  class Beta extends Alpha {
10     public Beta(int b, int c) {
11         super(b);
12         v = c;
13     }
14
15     public Alpha x1() {
16         super.v++;
17         return new Beta(0, v);
18     }
19
20     public int x2(int x) {
21         return x + ++v + v++;
22     }
23 }
24
25 public class Gamma extends Beta {
26     private short y;
27
28     public Gamma(int d, int e) {
29         super(d, e);
30         y = (short) d;
31     }
32
33     public int x2(int x) {
34         return x - y;
35     }
36
37     public static void main(String[] args) {
38         Alpha a = new Alpha(7);
39         Beta b = new Beta(0, 1);
40         Gamma g = new Gamma(9, 2);
41         a = b.x1();
42         int t = b.x2(5);
43         a = new Beta(10, 12).x1();
44         b = g;
45         int r = g.x2(50);
46     }
47 }
```

Hinweis: Nach Zeile X heißt unmittelbar nach X, noch vor Zeile X+1.

1) Welchen statischen und dynamischen Typ haben `a`, `b` und `g` nach Zeile 40?

2) Welchen statischen und dynamischen Typ hat `a` und welchen Wert hat `a.v` nach Zeile 41?

3) Welchen Wert haben `t` und `b.v` nach Zeile 42?

4) Welchen statischen und dynamischen Typ haben `a`, `b` und welchen Wert hat `a.v` nach Zeile 44?

5) Welchen Wert haben `r` und `b.v` nach Zeile 45?

V9 Klassen, Interfaces und Methoden ★ ★ ★

V9.1

Schreiben Sie ein `public`-Interface `A` mit einer Objektmethode `m1`, die Rückgabetyp `double`, einen `int`-Parameter `n` und einen `char`-Parameter `c` hat.

V9.2

Schreiben Sie ein `public`-Interface `B`, das von `A` erbt und zusätzlich eine Objektmethode `m2` hat, die keine Parameter hat und einen `String` zurückliefert.

V9.3

Schreiben Sie eine `public`-Klasse `XY`, die `A` implementiert, aber `m1` nicht. Klasse `XY` soll ein `protected`-Attribut `p` vom Typ `long` haben sowie einen `public`-Konstruktor mit Parameter `q` vom Typ `long`. Der Konstruktor soll `p` auf den Wert von `q` setzen. Weiter soll `XY` eine `public`-Objektmethode `m3` mit Rückgabetyp `void` und Parameter `xy` vom Typ `XY` haben, aber nicht implementieren.

V9.4

Schreiben Sie eine public-Klasse `YZ`, die von `XY` erbt und `B` implementiert. Die Methode `m1` soll `n+c+p` zurückliefern und `m2` den String `"Hallo"`. `m3` soll den Wert `p` von `xy` auf den Wert `p` des eigenen Objektes addieren. Der Konstruktor von `YZ` ist `public`, hat einen `long`-Parameter `r` und ruft damit den Konstruktor von `XY` auf.

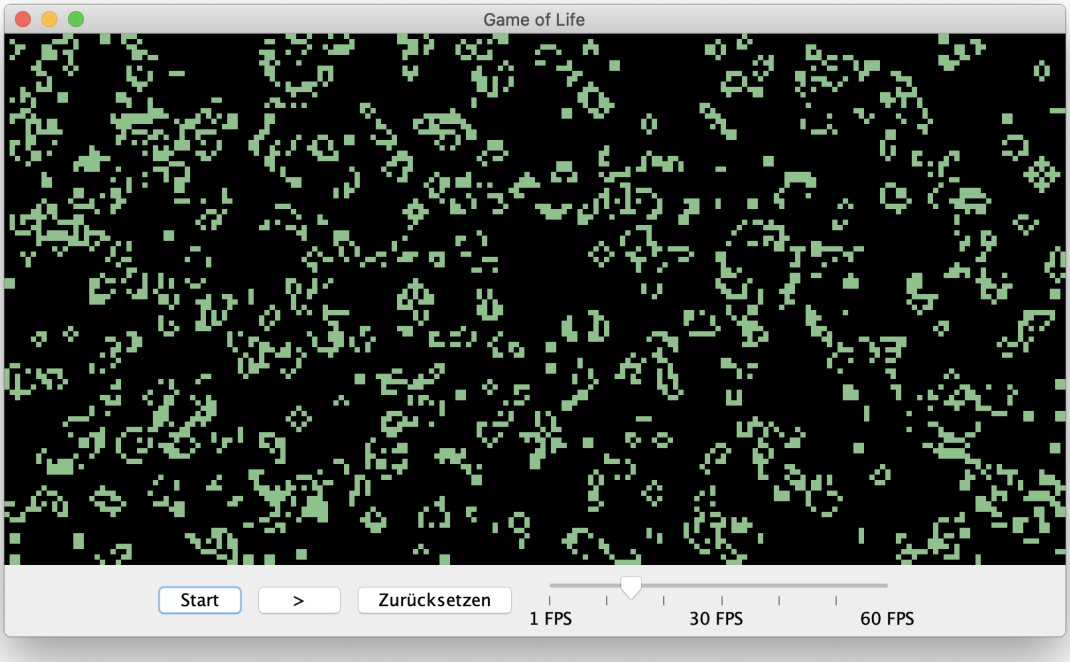
V10 Jedes dritte Element ★ ★ ★

Gegeben sei eine Klasse `X`. Schreiben Sie für diese Klasse die `public`-Objektmethode `foo`. Diese hat ein Array `a` von Typ `int` als formalen Parameter und liefert ein anderes Array b vom Typ `int` zurück, das aus a entsteht, indem jedes dritte Element gelöscht wird. Das heißt, die Elemente von `a` an den Indizes $0, 3, 6, 9, \ldots$ werden nicht nach `b` kopiert, alle anderen Elemente von `a` werden in derselben Reihenfolge, wie sie in `a` stehen, nach `b` kopiert. Weitere Elemente hat `b` nicht. Sie dürfen voraussetzen, dass a mindestens Länge 2 hat und ungleich `null` ist. Sie dürfen den Operator `=` für das Kopieren von Elementen verwenden.

Hinweis: Überlegen Sie sich die Gesetzmäßigkeit, nach der die Indizes $1, 2, 4, 5, 7, 8, \ldots$ in `a` auf die Indizes $0, 1, 2, 3, 4, 5, \ldots$ in `b` abzubilden sind. Für die Länge von `b` werden Sie eine Fallunterscheidung benötigen, je nachdem, welchen Rest `a.length` dividiert durch 3 ergibt. Denken Sie auch an die letzten beiden Elemente von `a`.

H Vierte Hausübung — Gesamt 25 Punkte

Game of Life



Hinweise

In dieser Aufgabe stellen wir Ihnen ein recht umfangreiches Framework zur Erstellung eines graphischen und interaktiven Programmes zu Verfügung. Ihre Aufgabe wird sein, dieses um gewisse Funktionen zu erweitern, die letztendlich zu einem großen Paket zusammen geschnürt werden, das ohne seine Einzelteile selbstverständlich nicht vollständig funktionieren kann. Dennoch haben wir uns bemüht, die Aufgaben möglichst modular aufzubauen. Sollten Sie also an einer Stelle hängen bleiben, zögern Sie nicht, zuerst die nächste Aufgabe zu bearbeiten.

Mit Ausnahme der letzten stellen wir zu jeder der bepunkteten Aufgaben außerdem Unit-Tests zur Verfügung, sodass sie Ihren Zwischenstand jederzeit überprüfen können. Allerdings ist ein korrektes Testergebnis noch keine Garantie dafür, dass Sie die Aufgabe vollständig richtig bearbeitet haben! Gehen Sie also sicher, dass sie auch die verbindlichen Anforderungen der jeweiligen Teilaufgaben erfüllt haben und dass ihr Code auch allen gegebenen Spezifikationen genügt. Schauen Sie hierfür auch in die JavaDoc-Dokumentation des Frameworks. Sie finden es im Ordner H04/doc/.

Sie selbst müssen für diese Aufgabe weder Unit-Tests noch JavaDoc-Kommentare schreiben! Zudem müssen Sie keine der gegebenen Dateien bearbeiten. Bewertet werden lediglich die folgenden Dateien

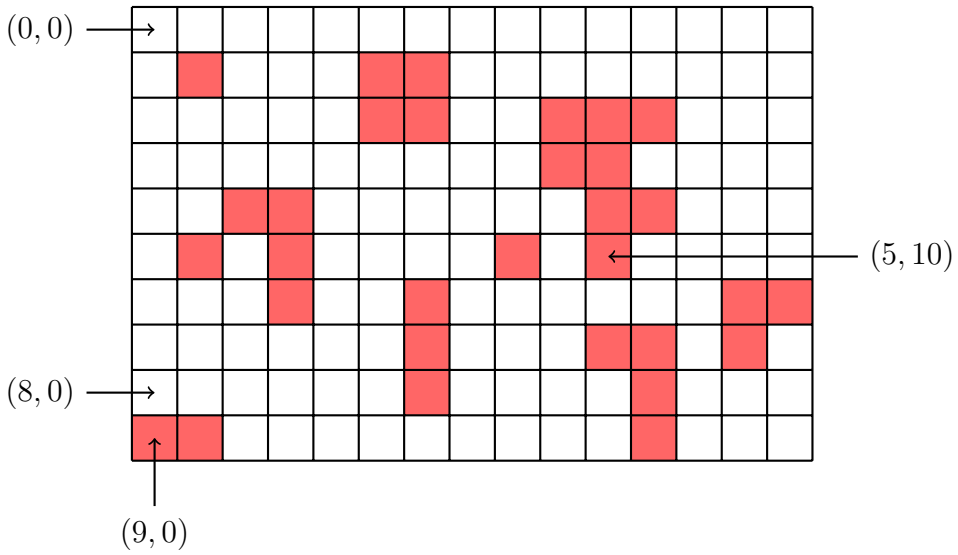
- `SamePadding.java`

- `DonutPadding.java`
- `MooreNeighborhood.java`
- `GameOfLifeRules.java`
- `HighlifeRules.java`
- `RandomInitializer.java`
- `ModularAutomaton.java`
- `ModularAutomatonBuilder.java`

Einführung

Das *Game of Life* spielt sich auf einer speziellen Form eines sogenannten *zellulären Automaten* ab und soll eine stark vereinfachte Form einer Zivilisation im Laufe der Zeit simulieren. Diese setzt sich zusammen aus Lebewesen, die in einem rechteckigen Gitter der Höhe h und Breite w angeordnet sind. Wir bezeichnen diese Lebewesen von nun an als *Zellen*. Jede einzelne dieser Zellen kann entweder lebendig oder tot sein.

Ein möglicher Zustand eines solchen zellulären Automaten mit den Maßen $h \times w = 10 \times 15$ könnte also wie folgt aussehen:



Die rot eingefärbten Kästchen repräsentieren hier lebendige Zellen, die restlichen sind tot.

Um eine einheitliche Indizierung der Zellen festzulegen, werden wir für den Rest der Aufgabe mit $(0, 0)$ immer die Zelle bezeichnen, die sich oben links im Gitter befindet, und (i, j) sei die Zelle in der i-ten Zeile und j-ten Spalte mit $i \in \{0, 1, \ldots, h-1\}$ und $j \in \{0, 1, \ldots, w-1\}$.

Im obigen Beispiel wäre also $(9, 0)$ die Zelle in der linken unteren Ecke und $(8, 0)$ die Zelle darüber.

Zusätzlich zu der Menge an Zuständen, definiert ein zellulärer Automat noch eine Übergangsfunktion, die den gegenwärtigen Zustand in einen fest definierten Nachfolgezustand

überführt. Um diesen zu bestimmen, betrachten wir für jede einzelne Zelle, egal ob lebendig oder tot, eine fest definierte Nachbarschaft und zählen, wie viele ihrer Nachbarn am Leben sind.

In den Standardregeln, die durch den Erfinder des Game of Life John Conway (siehe Seite 18) definiert wurden, wird hierfür eine sogenannte *Moore-Nachbarschaft* benutzt. Diese beinhaltet diejenigen Zellen, die jeweils genau ein Feld in horizontaler, vertikaler oder diagonaler Richtung um die betrachtete Zelle liegen. Mit Ausnahme der Zellen am Rand hat hier also jede Zelle genau acht Nachbarn.

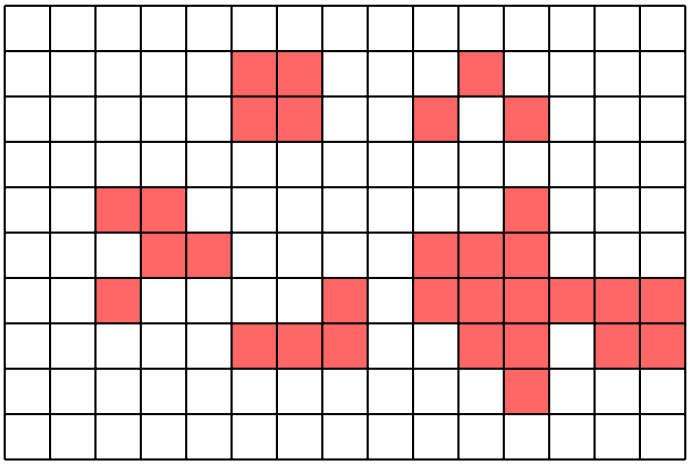
1	1	1	0	1	2	2	1	0	0	0	0	0	0	0
1	0	1	0	2	3	3	2	1	2	3	2	1	0	0
1	1	1	0	2	3	3	2	2	3	4	2	1	0	0
0	1	2	2	2	2	2	1	2	4	6	5	2	0	0
1	2	3	2	2	0	0	1	2	5	4	3	1	0	0
1	1	5	3	3	1	1	2	0	3	2	3	2	2	2
1	1	3	1	2	2	1	3	1	3	3	3	3	2	2
0	0	1	1	1	3	2	3	0	1	2	2	4	2	3
2	2	1	0	0	2	1	2	0	1	4	3	4	1	1
1	1	1	0	0	1	1	1	0	0	2	1	2	0	0

In der obigen Grafik ist jede Zelle mit der Anzahl ihrer lebendigen Nachbarn beschriftet. Zusätzlich ist die Nachbarschaft der Zelle $(7, 10)$ blau schraffiert. In diesem Fall haben wir auch für die außenliegenden Zellen nur diejenigen Nachbarn gewertet, die innerhalb des Rasters liegen. Wir werden im späteren Verlauf dieser Übung noch alternative Möglichkeiten der Randbetrachtung kennenlernen und miteinander vergleichen.

Der Fortbestand einer Zelle im Folgezustand hängt nun zum einen von der Anzahl ihrer lebendigen Nachbarn ab und zum anderen davon, ob sie im gegenwärtigen Zustand selbst am Leben ist. Das Game of Life definiert die folgenden fünf Regeln:

- hat eine tote Zelle genau drei lebendige Nachbarn, dann wird sie im Folgezustand zum Leben erweckt,
- hat eine tote Zelle weniger oder mehr als drei lebendige Nachbarn, dann bleibt sie im Folgezustand tot,
- hat eine lebendige Zelle genau zwei oder genau drei lebendige Nachbarn, dann bleibt sie auch im Folgezustand am Leben,
- hat eine lebendige Zelle mehr als drei lebendige Nachbarn, dann stirbt sie an Überbevölkerung und ist im Folgezustand tot,
- hat eine lebendige Zelle weniger als zwei lebendige Nachbarn, dann stirbt sie an Einsamkeit und ist im Folgezustand tot.

Anhand dieser Regeln können wir für die obige Ausgangssituation nun also den nächsten Zustand des zellulären Automaten bestimmen:

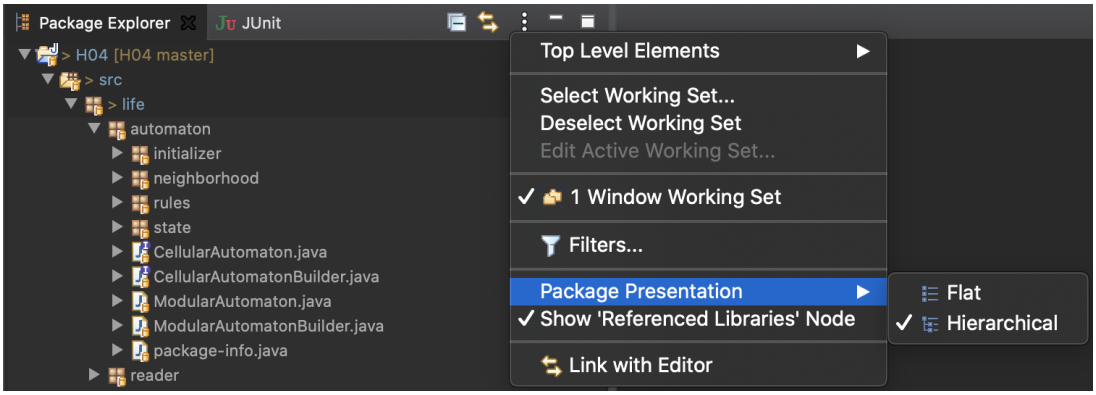


Wiederholen wir diese Schritte nun, um fortwährend einen Zustand in den nächsten zu überführen, erhalten wir eine anschauliche Simulation des zellulären Automaten. Hierbei gibt es zwei mögliche Ausgänge, da die Anzahl der denkbaren Zustände eines solchen Automaten mit 2^{hw} nach oben beschränkt und somit endlich ist: entweder er erreicht irgendwann einen stationären Zustand, der sich nicht weiter verändert, oder beginnt zu oszillieren, also periodisch eine gleichbleibende Abfolge von Zuständen zu durchlaufen.

Das Ziel dieser Übung ist, eine objektorientierte Version des Game of Life in Java umzusetzen. Außerdem soll diese möglichst modular gehalten werden, damit Sie schließlich verschiedene Konfigurationen miteinander vergleichen und das Programm auch jederzeit um neue Optionen erweitern können. Hierfür stellen wir Ihnen ein umfangreiches Framework zur Verfügung, dass im Folgenden kurz betrachten werden soll, bevor Sie sich an die Arbeit machen.

Framework

Machen Sie sich zunächst mit dem Codegerüst etwas vertraut. Das Projekt befindet sich in einem Modul mit dem Namen `h04` und exportiert das Package `life` und die meisten seiner Unterpackages. Sollten Sie Eclipse benutzen, empfehlen wir Ihnen, zur hierarchischen Package-Ansicht zu wechseln, indem Sie im Package Explorer oben auf die drei vertikalen Punkte klicken und **Package Presentation** ▷ **Hierarchical** auswählen



Grundsätzlich können Sie die Packages `life.ui` und `life.reader` ignorieren. Hierin befinden sich die Klassen und Interfaces, die sich um die grafische Benutzeroberfläche und das Einlesen von für die Repräsentation von zellulären Automaten gängigen Dateiformaten

kümmern.

Für Sie interessant sind zunächst insbesondere die Interfaces `CellularAutomaton` im Package `life.automaton` und `AutomatonState` im Package `life.automaton.state`. Ersteres definiert die notwendigen Methoden für das Nutzerinterface um mit einem zellulären Automaten zu interagieren. Und letzteres repräsentiert einen allgemeinen Zustand für einen zellulären Automaten.

Für diesen stellen wir Ihnen bereits eine konkrete Implementation `life.automaton.state.ArrayAutomatonState` zur Verfügung, die auf einem zweidimensionalen Array vom Komponententyp `boolean` basiert. Wie Sie sehen können, stellt diese Klasse einen Konstruktur mit der Signatur `ArrayAutomatonState(int, int)` zur Verfügung, die einen Zustand mit den Maßen `height`×`width` initialisiert, in dem zunächst alle Zellen tot sind. Während die Methode `void giveBirth(int, int)` die Zelle mit den Koordinaten (`col`, `row`) zum Leben erweckt, sorgt die Methode `void kill(int, int)` dafür, dass die entsprechende Zelle nach dem Aufruf tot ist. Besonders wichtig ist die Methode `boolean isAlive(int, int)`, die genau dann `true` liefert, wenn die entsprechende Zelle am Leben ist.

Damit Sie möglichst unabhängig von dieser konkreten Implementation arbeiten können, werden Sie nicht den Konstruktor von `ArrayAutomatonState` verwenden, sondern stattdessen auf das Interface `Initializer` im Package `life.automaton.initializer` zurückgreifen, das mit seiner Methode `AutomatonState createState()` imstande ist, einen allgemeinen Zustand zu erzeugen. Dies ermöglicht Ihnen und anderen Entwicklerinnen und Entwicklern, auch im Nachhinein noch weitere Implementationen von `AutomatonState` zu verwirklichen und diese jederzeit auszutauschen, ohne dass Code, der auf diesem Framework basiert, aufhört zu funktionieren.

H1 Randbehandlung 4 Punkte

Bevor Sie sich mit der Implementation der Nachbarschaften beschäftigen, wollen wir zunächst verschiedene Varianten betrachten, mit denen sich die Nachbarn von Zellen am Rand des Spielfeldes ermitteln lassen. Dies geschieht, indem das Spielfeld um einen virtuellen Rahmen mit unendlicher Größe eingebettet wird, sodass der Abruf von Nachbarn außerhalb des Spielfeldes weiterhin valide Ergebnisse liefert. Eine solche Einbettung wird häufig als *Padding* bezeichnet.

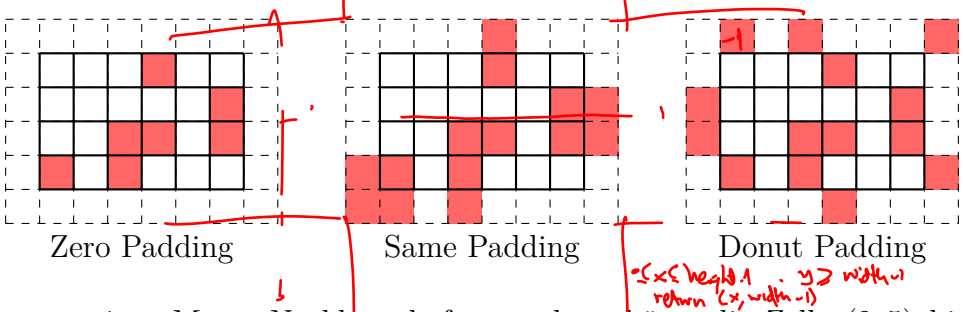
Eine Möglichkeit, die wir im obigen Beispiel bereits in Aktion gesehen haben, wäre die des sogenannten *Zero Paddings*. Hier wird das Spielfeld effektiv durch einen Rahmen von toten Zellen erweitert und somit spielen für die Anzahl der lebenden Nachbarn einer Zelle immer nur diejenigen innerhalb des Spielfeldes eine Rolle.

Andere Varianten wäre das *Same Padding*, bei dem das Spielfeld von Zellen eingerahmt wird, deren Zustand jeweils demjenigen der nächsten Zelle innerhalb des Spielfeldes entspricht. Effektiv wird der Rand des Spielfeldes also in den Rahmen „ausgestreckt“.

Bei einer dritten Art des Paddings, die wir hier noch betrachten wollen, handelt es sich um das sogenannte *Donut Padding*. Dieses Verfahren kommt Ihnen vielleicht durch das Spiel Pac-Man bekannt vor. Hierbei wird das Spielfeld periodisch fortgesetzt, das heißt wenn Sie es beispielsweise nach rechts verlassen, betreten Sie es wieder von der linken Seite.

Bildlich können Sie es sich vorstellen, als würde man die obere Kante des Spielfeldes mit der unteren und gleichzeitig die linke mit der rechten verkleben, wodurch man eine Donut-förmige Oberfläche erhält.

In der folgenden Grafik sehen Sie alle drei vorgestellten Varianten in Aktion. Wir betrachten jeweils denselben Zustand eines zellulären Automaten mit den Maßen 4×6. Die roten Zellen sind am Leben und die gestrichelt umrandeten Zellen stellen den imaginären Rahmen dar, den wir jeweils verwenden, um eine komplette Nachbarschaft für die Zellen innerhalb des Spielfeldes bilden zu können.



Wenn wir von einer Moore-Nachbarschaft ausgehen, hätte die Zelle $(2, 5)$ hier also je nach Variante 1, 3 oder 2 lebendige Nachbarn. Wir sehen also, dass das Überleben einer Zelle durchaus von der Wahl des Paddings abhängen kann, weshalb wir diese miteinander vergleichen wollen.

Ihre erste Aufgabe ist nun, die vorgestellten Padding-Strategien in Java umzusetzen. Diese sollen hierfür das bereits vorgegebene Interface `Padding` implementieren.

```
public interface Padding {
    boolean isAlive(AutomatonState state, int row, int col);
}
```

Dieses definiert eine Methode `isAlive`, die genau dann `true` liefern soll, wenn die Zeile mit den Koordinaten (`row`, `col`) im Zustand `state` am Leben ist.

Anders als bei der Methode `isAlive(int row, int col)` des Interfaces AutomatonState, die lediglich dann valide Ergebnisse liefern muss, wenn `row` und `col` gültige Indizes innerhalb des Spielfeldes sind, soll die Methode des Paddings für jede denkbare Eingabe einen gültigen Wert liefern, der entsprechend der jeweiligen Padding-Strategie berechnet wird.

Um Ihnen die Arbeit ein wenig zu erleichtern, haben wir bereits die Klasse `ZeroPadding` für Sie implementiert, es fehlen also nur noch `SamePadding` und `DonutPadding`.

Legen Sie also hierfür im Package `life.automaton.neighborhood` die public-Klassen `SamePadding` und `DonutPadding` an, indem sie die Dateien `SamePadding.java` und `DonutPadding.java` erstellen. In Eclipse erreichen Sie das ganz einfach mit einem Rechtsklick auf das Package `neighborhood`, gefolgt von **New** ▷ **Class**. In dem Dialogfeld, dass sich hierauf öffnet, können Sie auch direkt das entsprechende Interface hinzufügen, dass die Klassen implementieren sollen.

Tipp: werfen Sie einen Blick auf die statischen Methoden `min`, `max` und `floorMod` der Klasse `java.lang.Math`. Diese könnten Ihnen bei dieser Aufgabe wohlmöglich von Nutzen sein.

Wenn Sie die beiden Klassen vollständig implementiert haben, prüfen Sie diese mit Hilfe des Unit-Tests `PaddingTest` im package `life.test` auf Korrektheit. Öffnen Sie hierfür die

Datei `PaddingTest.java` und wählen Sie dann **Run ▷ Run as ... ▷ JUnit Test**.

H2 Nachbarschaften 3 Punkte

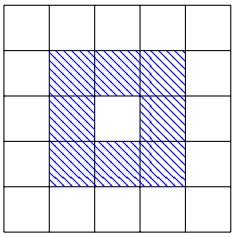
Nachdem Sie nun die Gefahr beseitigt haben, uns versehentlich außerhalb des Spielfeldes zu bewegen, können Sie uns nun endlich um die Betrachtung der Nachbarschaft einer Zelle kümmern. Hierfür haben wir für Sie das Interface `Neighborhood` zur Verfügung gestellt.

```
public interface Neighborhood {
    int getNumberOfAliveNeighbors(
        AutomatonState state, int row, int col, Padding padding);
}
```

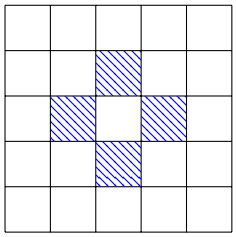
Dieses definiert eine Methode `getNumberOfAliveNeighbors`, die uns die Anzahl der lebendigen Nachbarn für die Zelle mit den Koordinaten (`row`, `col`) unter Berücksichtigung des Paddings `padding` liefert.

Ihre Aufgabe hier ist es, die bereits bekannte Moore-Nachbarschaft zu modellieren, indem sie im Package `life.automaton.neighborhood` die public-Klasse `MooreNeighborhood` erstellen und damit das Interface `Neighborhood` implementieren.

Als Vorlage haben wir für Sie bereits eine andere Art von Nachbarschaft, die sogenannte *Von-Neumann-Nachbarschaft*, implementiert. Diese betrachtet nur die vier Zellen in unmittelbarer horizontaler und vertikaler Umgebung einer Zelle.



Moore-Nachbarschaft



Von-Neumann-Nachbarschaft

Verbindliche Anforderung Verwenden Sie für Ihre Umsetzung eine oder mehrere `for`-Schleifen, sodass Sie den Aufruf von `isAlive(AutomatonState, int, int)` lediglich einmal in Ihrem Code ausschreiben müssen!

Wenn Sie mit der Implementierung fertig sind, können Sie die Unit-Tests der Klasse `NeighborhoodTest` im Package `life.test` nutzen, um sicher zu gehen, dass Sie korrekt gearbeitet haben. Gehen Sie hierbei wie auch schon mit `PaddingTest` vor.

H3 Spielregeln 4 Punkte

Die Spielregeln des Game of Life haben Sie im Einführungsabschnitt bereits kennen gelernt. Wir fassen nochmal kurz zusammen:

- hat eine tote Zelle genau drei lebendige Nachbarn, dann wird sie im Folgezustand zum Leben erweckt,
- hat eine tote Zelle weniger oder mehr als drei lebendige Nachbarn, dann bleibt sie im Folgezustand tot,

- hat eine lebendige Zelle genau zwei oder genau drei lebendige Nachbarn, dann bleibt sie auch im Folgezustand am Leben,
- hat eine lebendige Zelle mehr als drei lebendige Nachbarn, dann stirbt sie an Überbevölkerung und ist im Folgezustand tot,
- hat eine lebendige Zelle weniger als zwei lebendige Nachbarn, dann stirbt sie an Einsamkeit und ist im Folgezustand tot.

Oft wird dieser Satz an Regeln kurz mit *B3/S23* notiert. B3 steht dafür, dass eine tote Zelle neu geboren wird, wenn sie genau drei lebendige Nachbarn besitzt, und S23 bedeutet, dass eine lebendige Zelle überlebt, wenn sich genau 2 oder 3 lebendige Zellen in ihrer Nachbarschaft befinden. Machen Sie sich am besten kurz bewusst, warum die beiden Angaben vollkommen ausreichen, um die obigen Regeln zu beschreiben.

Sie werden ein solches Regelwerk umsetzen, indem Sie das Interface `UpdateRules` im Package `life.automaton.rules` implementieren:

```
public interface UpdateRules {
    boolean getNextCellState(boolean isAlive, int aliveNeighbors);
}
```

Die Funktionsweise hier ist recht simpel: die Methode `getNextCellState` erhält den momentanen Zustand `isAlive` einer Zelle sowie die Anzahl derer lebendigen Nachbarn `aliveNeighbors` und soll genau dann `true` liefern, wenn die entsprechende Zelle im nächsten Zeitschritt am Leben sein soll.

Erstellen Sie hierfür die public-Klasse `GameOfLifeRules` im package `life.automaton.rules`, die `UpdateRules` so implementiert, dass sie die Regeln des Game of Life, also B3/S23, korrekt widerspiegelt.

> **Verbindliche Anforderung** Sie dürfen in dieser Teilaufgabe keine `if`- oder `switch`-Abfragen verwenden. Arbeiten Sie ausschließlich mit booleschen Operatoren!

Zusätzlich sollen Sie zum Vergleich noch ein zweites berühmtes Regelwerk mit dem Namen *Highlife* mit der Kodierung B36/S23 umsetzen. Legen Sie hierfür die public-Klasse `HighlifeRules` im package `life.automaton.rules` an und lassen Sie diese auch `UpdateRules` implementieren. Halten Sie sich auch hier an die oben genannte Anforderung!

Ob Sie korrekt gearbeitet haben, können Sie im Anschluss an diese Aufgabe mit den Tests in `life.test.RulesTest` überprüfen.

H4 Zufällige Startzustände 4 Punkte

Bisher haben wir uns noch keine Gedanken darüber gemacht, wie wir eigentlich Start-Zustände für unsere zellulären Automaten generieren wollen. Sie kennen bereits die Klasse `ArrayAutomatonState`. Diese stellt zwar einen Konstruktor zur Verfügung, jedoch werden Sie diesen nicht explizit nutzen. Somit ist gewährleistet, dass Code, der auf unserem Framework basiert, auch dann noch weiterhin funktioniert, sollte irgendwann in der Zukunft beschlossen werden, die konkrete Implementation von `AutomatonState` auszutauschen.

Stattdessen nutzen Sie eine allgemeine Fabrik-Klasse, die imstande ist, einen `AutomatonState` zu generieren. Erst die konkreten Implementationen dieser Fabrik müssen dann über den konkreten Aufbau des von uns genutzten `AutomatonState` Bescheid wissen. Sie finden diese im Package `life.automaton.initializer` in Form des Interfaces `StateInitializer`:

```
public interface StateInitializer {
    AutomatonState createState();
}
```

Hier finden Sie zudem zwei konkrete Implementationen `EmptyInitializer` und `FileInitializer`, die beide auf der Erzeugung eines `ArrayAutomatonState` basieren. Mit diesen ist es möglich, ohne die Kenntnis von `ArrayAutomatonState` einen leeren Zustand zu erstellen oder einen aus einer Datei zu lesen.

Zusätzlich wollen wir noch die Möglichkeit haben, einen zufälligen Zustand zu erzeugen. Erstellen Sie hierfür im Package `life.automaton.initializer` die `public`-Klasse `RandomInitializer`, die das Interface `StateInitializer` implementiert. Ein `RandomInitializer` soll die `private`-Attribute `height` und `width` jeweils vom Typen `int` und `rate` vom Typen `double` besitzen.

Schreiben Sie außerdem einen Konstruktor

```
public RandomInitializer(int height, int width, double rate) {
    ...
}
```

mit dem die oben genannten Attribute gesetzt werden sollen. Beim Aufruf von `createState()` soll ein neuer `ArrayAutomatonState` mit der Höhe `height` und der Breite `width` erzeugt werden, in dem jeder Zelle ein zufälliger Zustand zugewiesen wird. Dieser soll so bestimmt werden, dass `rate` die Wahrscheinlichkeit festlegt, mit der eine Zelle im resultierenden Zustand am Leben ist. Wird also beispielsweise ein Wert von 0.3 für `rate` übergeben, dann sollten im Durchschnitt etwa 30 Prozent der Zellen im resultierenden Zustand lebendig und die restlichen tot sein.

Überschreiben Sie die Methode `createState()` in `RandomInitializer` so, dass diese Bedingung erfüllt ist. Hierfür bietet es sich an, die Methode `nextDouble()` der Klasse `java.util.Random` zu verwenden. Sie können Ihre Implementierung mit `life.test.InitializerTest` überprüfen.

H5 Zusammenbauen des Automaten 10 Punkte

Nun haben Sie fast alles, was benötigt wird, um das Programm endlich zum Laufen zu bringen! Es fehlt lediglich eine konkrete Implementation von `CellularAutomaton`, die Sie mit den bereits implementierten Komponenten bestücken können.

Erstellen Sie hierfür im Package `life.automaton` eine public-Klasse mit dem Namen `ModularAutomaton`, die das Interface `CellularAutomaton` implementiert. Erweitern sie diese um folgende `private`-Attribute:

- `initialState` und `currentState` jeweils vom Typ `AutomatonState`,

- `rules` vom Typ `UpdateRules`,
- `neighborhood` vom Typ `Neighborhood` und
- `padding` vom Typ `Padding`.

Legen Sie zusätzlich einen public-Konstruktor mit dem Kopf

```
public ModularAutomaton(
    AutomatonState initialState,
    UpdateRules rules,
    Neighborhood neighborhood,
    Padding padding) {
    ...
}
```

an, der die jeweiligen Attribute entsprechend initialisiert. Zu Beginn soll `currentState` auf eine Kopie von `initialState` verweisen. Dieser Zustand soll auch bei jedem Aufruf von `reset()` wiederhergestellt werden.

Die Methoden `isAlive(int, int)`, `getHeight()` und `getWidth()` überschreiben Sie so, dass sie die entsprechenden Eigenschaften von `currentState` widerspiegeln.

Zuletzt implementieren Sie die Methode `update()`. Hierfür erzeugen Sie lokal einen neuen `AutomatonState` als Kopie von `currentState`. Beleben oder töten Sie dessen Zellen zunächst dementsprechend, dass er dem Folgezustand von `currentState` entspricht, indem Sie über dessen Zeilen und Spalten iterieren. Verwenden Sie für die Ermittlung der lebendigen Nachbarn und des Folgezustandes jeder einzelnen Zelle unbedingt die jeweiligen Strategien, die in `neighborhood`, `padding` und `rules` gespeichert sind! Zuletzt weißen Sie `currentState` diesen neuen Zustand zu.

Schließlich soll noch erreicht werden, dass sich ein zellulärer Automat ohne die Kenntnis von `ModularAutomaton` konstruieren lässt. Hierfür verwenden Sie das Interface `CellularAutomatonBuilder` im Package `life.automaton`. Dieses definiert Methoden, mit denen Sie die Komponenten eines Automaten sukzessive kombinieren können, um schließlich ein Objekt daraus zu erzeugen.

Legen Sie im Package `life.automaton` eine neue Klasse `ModularAutomatonBuilder` an, die das Interface `CellularAutomatonBuilder` implementiert und über folgende `private`-Attribute verfügt:

- `state` vom Typ `AutomatonState`,
- `rules` vom Typ `UpdateRules`,
- `neighborhood` vom Typ `Neighborhood` und
- `padding` vom Typ `Padding`.

Zu Beginn sollen diese Attribute alle mit dem Wert `null` initialisiert werden. Implementieren Sie dann die Methoden `addState(AutomatonState)`, `addRules(UpdateRules)`, `addNeighborhood(Neighborhood)` und `addPadding(Padding)` so, dass im entsprechenden Attribut nach dem Aufruf eine Referenz auf den jeweiligen Parameter gespeichert ist. Der Rückgabewert all dieser Methoden soll das Objekt vom Typen `ModularAutomatonBuilder` sein,

auf die die jeweilige Methode aufgerufen wurde. Implementieren Sie schließlich die Methode `CellularAutomaton buildAutomaton()` so, dass sie den Konstruktor von `ModularAutomaton` nutzt, um ein Objekt aus den gespeicherten Komponenten zu erzeugen.

H6 Testlauf 0 Punkte

Geschafft! Damit wären Sie nun eigentlich fertig. Aber selbstverständlich wollen Sie die Früchte Ihrer Arbeit noch in voller Pracht erleben. Kommentieren Sie hierfür den mit `TODO`-Kommentar versehenen Abschnitt in der `main`-Methode der Klasse `life.GameOfLife` wieder ein und führen Sie diese aus.

Zunächst sehen Sie einen Dialog, indem Sie ihren zellulären Automaten konfigurieren und den Zustand entweder aus einer Datei laden oder zufällig generieren können. Haben Sie die Simulation gestartet, öffnet sich ein weiteres Fenster, in dem der zelluläre Automat dargestellt wird. Hier können Sie die Simulation starten, anhalten, zurücksetzen und mit dem Schieberegler ihre Geschwindigkeit anpassen.

Sie finden im Ordner `src/life/states` zudem einige Dateien mit interessanten Startzuständen, die Sie über das User Interface laden können. Die unterstützten Dateiformate sind `.rle` und `.cells`. Weitere Zustände in diesen Formaten können Sie zum Beispiel auf *LifeWiki*[1] herunterladen. Sie sind herzlich eingeladen, interessante Funde mit Ihren Kommilitoninnen und Kommilitonen zu teilen.

Wir wünschen Ihnen viel Spaß beim Ausprobieren der zahlreichen Optionen, die Sie in dieser Aufgabe erstellt haben! Mit den Methoden `register*` der Klasse `GameOfLifeLauncher` können Sie jederzeit neue Regeln, Farbschemata, etc. zum Programm hinzufügen. Zögern Sie daher nicht, neue Optionen zu erstellen und auszuprobieren.

[1] `https://www.conwaylife.com/wiki`

John Horton Conway – der Erfinder des Game of Life

Abbildung 1: John Horton Conway (*26.12.1937; †11.4.2020) im Jahre 2005. Foto von Thane Plambeck (`https://www.flickr.com/photos/thane/20366806/`)

.

Am 11. April des Jahres 2020 ist der britische Mathematiker John Horton Conway im Alter von 82 Jahren von uns gegangen, nachdem er sich mit SARS-CoV-2 infiziert hatte. Zu Lebzeiten beschäftigte er sich zunächst bis ins Jahr 1986 an der University of Cambridge insbesondere mit Zahlen- und Gruppentheorie, nachdem er dort im Jahre 1964 promovierte. Später verschlug es ihn als Professor in die USA an die Princeton University in New Jersey, wo er im Bereich Computermathematik tätig war und auch weiterhin intensiv mit dem Gebiet der Symmetriegruppen forschte.[2]

Zu weltweiter Prominenz, auch außerhalb von Mathematikerrängen, gelangte der „world's most charismatic mathematician", wie ihn der Guardian einst betitelte[3], insbesondere im Jahre 1970 durch seine Erfindung des *Game of Life* und der damit einhergehenden Arbeit an sogenannten *zellulären Automaten*, die die Basis hierfür bilden. Nachdem sein Name quasi synonym für das Game of Life wurde, entwickelte Conway selbst eine gewisse Hassliebe zu diesem Thema, das er persönlich als nicht sonderlich interessant einschätzte.[4]

[2] `https://www.nytimes.com/2020/04/15/technology/john-horton-conway-dead-coronavirus.html`

[3] `https://www.theguardian.com/science/2015/jul/23/john-horton-conway-the-most-charismatic-mathematician-in-the-world`

[4] `https://youtu.be/E8kUJL04ELA`