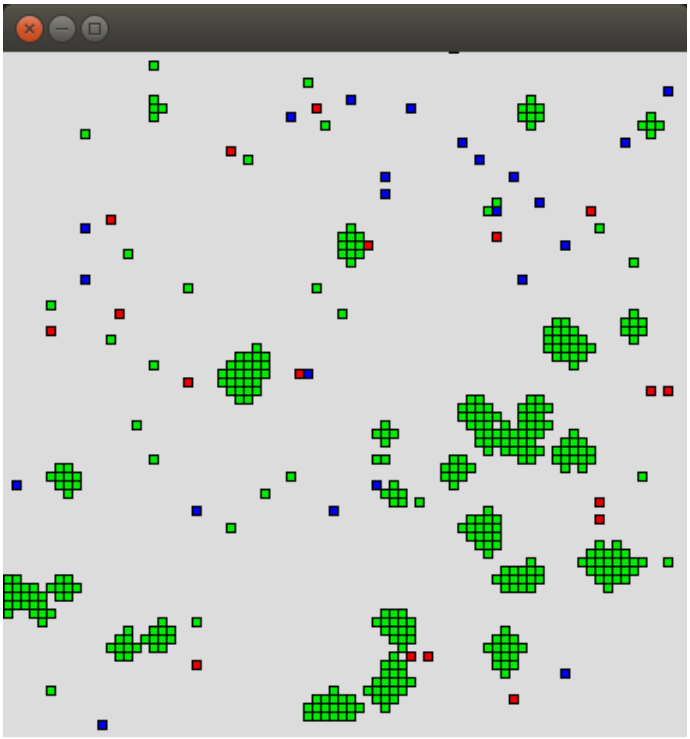
Abstrakte Klassen und Schnittstellen

1. Life



In diesem Praktikum schreiben Sie eine Life-Simulation nach einer bestimmten Struktur. In einem GUI-Fenster werden einzelne Figuren in Form von Vierecken dargestellt. Diese Figuren repräsentieren jeweils ein Lebewesen, welche sich im Takt bewegen und je nach Eigenschaft sich vermehren oder ein darunterliegendes Feld „fressen“. Die jeweilige Farbe der Figur stellt dessen Art dar. In diesem Praktikum werden folgende Lebewesen verwendet:

- Grün: Grass
- Rot: Fuchs
- Blau: Hase

Eigenschaften

Fuchs

Der Fuchs kann sich in alle Richtungen (links, rechst, oben und unten) bewegen. Dies wird nach jedem Takt durchgeführt. Die Auswahl der Bewegungsrichtung wird durch eine Zufallsberechnung ausgewählt, wobei jede Richtung eine Chance von 25% hat ausgeführt zu werden.

Der Fuchs soll beim Auftreten eines Hasen den Hasen fressen (entfernen).

Hase

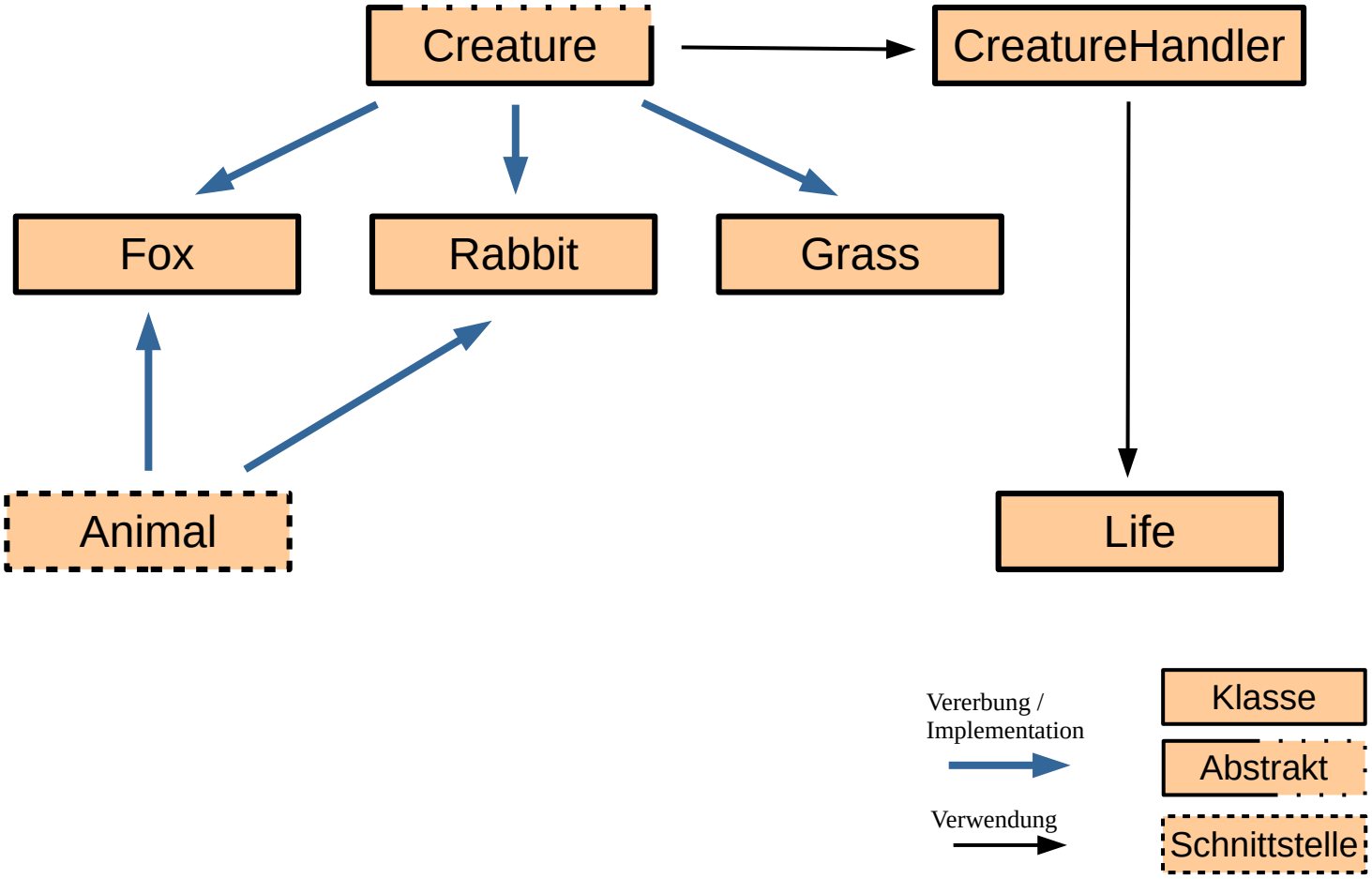
Der Hase Bewegt sich genau gleich wie der Fuchs.

Er frisst jedoch ein Grass-Feld sobald er auf dieses auftrifft. Dabei besteht beim Hasen die Chance (10%), dass er sich nach dem Fressen vermehrt. Der neu erzeugte Hase wird auf die selbe Stelle wie das Eltern-Tier platziert.

Grass

Das Grass kann nach jedem Schritt (Takt) mit einer Chance (1%) sich vermehren. Dabei werden vier neue Pflanzen jeweils neben der Eltern-Pflanze platziert (links, rechts, oben und unten).

Struktur



Life

Hauptklasse Mit Main-Methode welches das Fensters erstellt. Besitzt auch eine endlose while-Schleife mit einem Delay (SimplefiedDelay) welcher den Takt der Simulation vorgibt und die Figuren entsprechend Zeichnet. Diese Klasse besitzt alle Grundwerte für die Simulation als statische Variablen:

- WINDOW_WIDTH
- WINDOW_HEIGHT
- BLOCK_SIZE
- GRID_WIDTH
- GRID_HEIGHT
- NUMBER_OF_FOXES
- NUMBER_OF_RABBITS
- NUMBER_OF_GRASS
- STEP_TIME

Animal

Schnittstelle welche die Methode *eat* besitzt. Diese Methode soll über einen Parameter eine Kreatur aufnehmen.

Creature

Abstrakte Klasse welche die grundlegende Eigenschaften und Methoden der Lebewesen besitzt:

- Eigenschaft: x, y Raster-Position des Lebewesens
- Eigenschaft: Farbe
- Eigenschaft: Merker, ob dieses Objekt gelöscht / gefressen wurde
- Konstruktor welcher als Parameter die Farbe der Figur aufnimmt
- Zeichnungsmethode mit *Graphics*-Parameter
- Löschmethode welche den Merker entsprechend setzt (keine Parameter)
- Methode um die Position des Lebewesens zu setzten (x, y als Parameter)
- Abstrakte Methode für den Schritt (keine Parameter)

Fox

Klasse welche von *Creature* erbt und die Schnittstelle *Animal* implementiert. Ruft über den Konstruktor den Super-Konstruktor von *Creature* auf und setzt damit die Farbe des Fuchses. Zudem wird eine zufällige Position im Konstruktor gewählt. Die Klasse überschreibt die offenen Methoden der abstrakten Klasse und der Schnittstelle. Die Schnittstellen-Methode *eat* soll überprüfen, ob die übergebene Kreatur „essbar“ ist und diese in diesem Fall entfernen („fresses“).

Hase

Gleicher Aufbau wie bei Fox.

Grass

Gleicher Aufbau wie bei Fox, jedoch implementiert nicht die *Animal*-Schnittstelle.

CreatureHandler

Verwaltet und behält alle Kreaturen:

- Eigenschaft: Array mit allen Kreaturen (Grösse z.B. 10000)
- Konstruktor welcher das Array erstellt (keine Parameter)
- Methode um Kreaturen hinzuzufügen (*Creature* als Parameter, welches dem Array hinzugefügt werden soll). Dabei wird schrittweise die Elemente des Arrays überprüft, ob kein Lebewesen an dieser Stelle hinterlegt ist oder ob hier ein gelöschtes Lebewesen besteht. Trifft dies zu, wird an dieser Stelle das übergebene Lebewesen platziert.
    - Hinweis: um zu überprüfen, ob ein Objekt an einer Stelle in einem Array vorhanden ist, kann das Schlüsselwort ***null*** verwendet werden: if(creatures[i] != null) → true, wenn ein Objekt vorhanden ist.
- Schritt-Methode welche alle Schritt-Methoden der Kreaturen aufruft. (keine Parameter)
- Zeichnungsmethode welche alle Zeichnungsmethoden der Kreaturen aufruft. (*Graphics* als Parameter)
- Methode um alle Kreaturen auf einem Punkt zu erhalten. Nimmt als Parameter x und y auf und gibt ein Array mit allen Kreaturen, welche sich auf diesen Punkt befinden.
- Methode um das Fressen zu überprüfen (keine Parameter). Geht alle Kreaturen durch und führ bei allen *Animal*-Kreaturen Kollisionstest durch. Dabei wird dessen Position mit der vorherigen Funktion verwendet und alle Kreaturen auf diesem Feld ermittelt. All diese Kreaturen werden der Schnittstellen-Methode *eat* der entsprechenden Kreatur übergeben.

Aufbau

Raster

Die Simulation soll in einem Raster aufgebaut werden. Legen Sie dazu die Blockgrösse fest (im Beispiel 5px). Die Position in *Creature* soll die Raster-Position sein. Beispiel: Kreatur soll auf Bildschirm x: 200px und y: 200px positioniert sein → Kreature x: 40 und y: 40

Bei der Darstellung kann dann auf die Pixel-Werte umgewandelt werden.

Takt und Darstellung

In der Simulation soll der Ablauf und der Darstellungs-Aufruf durch einen Takt bestimmt sein. Der Takt soll in der *while*-Schleife durchgeführt werden. Dabei werden alle Takt-Funktionen der Kreaturen aufgerufen (*CreatureHandler*) und durch einen Delay bis zur nächsten Ausführung pausiert. Für den Delay können Sie die vorgegebene *SimplifiedDelay*-Klasse verwenden. Um den Zeichnungsprozess von *paint* direkt auszuführen, können Sie die Methode *repaint()* verwenden.

Random

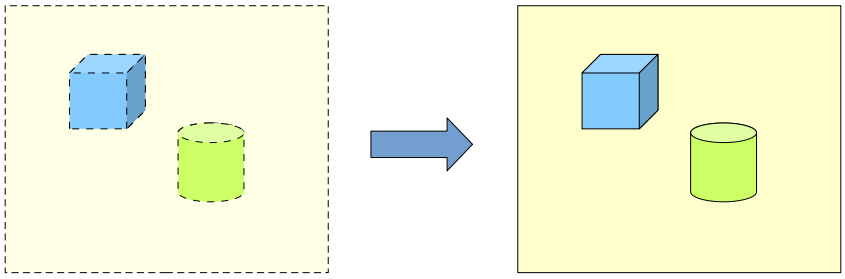
Für zufällige Prozesse können Sie die vorgegebene Klasse *SimplifiedRandom* verweden.

2. Buffered Life

In dieser Übung erweitern Sie Ihre Life-Simulation so, dass alle zu zeichnenden Elemente zuerst auf ein Buffered-Image aufgebaut werden und das Image dann im letzten Schritt dargestellt wird. Dank der Klassenhierarchie und dem Überschreiben von Methoden ist diese Implementierung mit wenigen Schritten möglich. Zudem implementieren Sie einen Sortieralgorithmus um die Zeichnungsreihenfolge zu bestimmen.

Buffered Image

Ihnen ist wahrscheinlich aufgefallen, dass bei der letzten Übung die Darstellung der grafischen Elemente flackernd erscheinen (flicker). Dies geschieht meistens, wenn für die Darstellung die Elemente nicht einheitlich, sondern zur unterschiedlichen Zeit erscheinen, also mitten im Aufbauprozess. Daher können gewisse Elemente noch nicht gezeichnet sein, was so aussieht, als würden diese kurzzeitig verschwinden und wieder auftauchen. Dabei bietet es sich an diese Elemente zuerst auf einer „virtuellen Fläche“ zu zeichnen und erst am Schluss diese Fläche darzustellen.



Elemente werden verdeckt auf einer Fläche aufgebaut

Verdeckte Fläche mit den gezeichneten Elementen wird in einem Schritt dargestellt.

Im weiteren Sinne nennt man dieses Prinzip auch Doppelpufferung.

Implementierung

Für Ihre Anwendung können Sie die Klasse *BufferedImage* verwenden. Lesen Sie sich dazu in die Java-Docs ein. Mit der Methode *createGraphics()* können Sie ein *Graphics*-Objekt erstellen. Alle *Graphics*-Methoden, die in diesem Objekt aufgerufen werden, wirken sich auf das *BufferedImage*-Objekt aus. Wenn Sie also mit diesem *Graphics*-Objekt etwas zeichnen, wird dies auf diesem *BufferedImage*-Objekt gezeichnet und nicht direkt auf das Fenster. Um das *BufferedImage*-Objekt darzustellen, können Sie die Methode *drawImage()* aus der *Graphics*-Klasse verwenden. Rufen Sie

dabei die Methode mit folgenden Parametern auf *drawImage(buffered_image, 0, 0, null);* , wobei *buffered_image* Ihr *BufferedImage*-Objekt ist.

Implementieren Sie diese Strutkur in die Haupt-*paint*-Methode.

Sortierung

Die Kreaturen werden bei der bisherigen Version in der Reihenfolge, wie sie erstellt wurden, dargestellt. Sind zwei Kreatur auf dem selben Feld, wird die Kreatur, welches als letztes erstellt wurde angezeigt. Dies kann zu unerwünschten Effekten führen. Damit eine bestimmte Reihenfolge verwendet werden kann, sollen die Kreaturen sortiert werden.

Implementierung

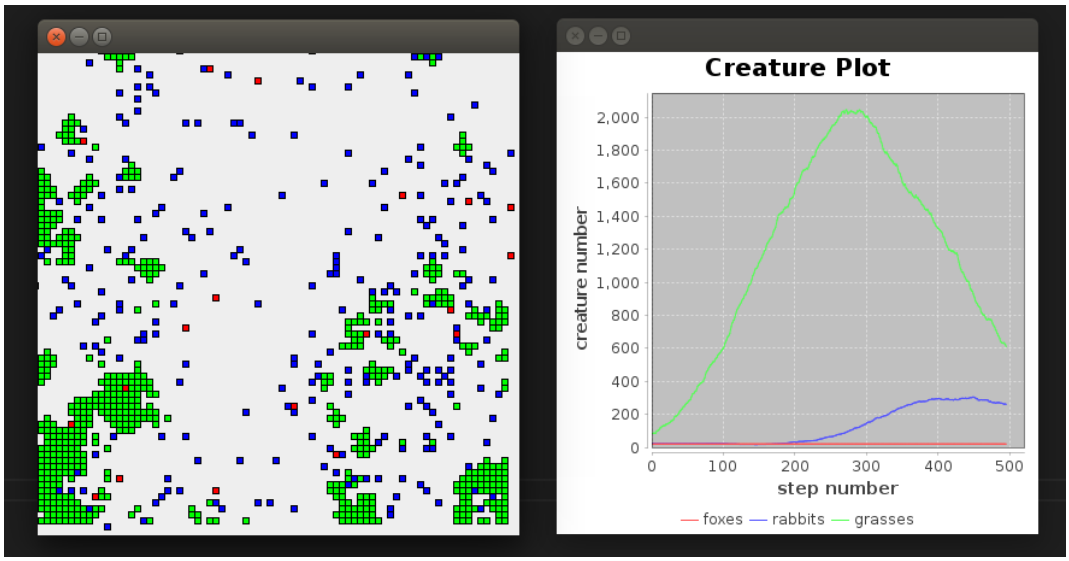
Erweitern Sie die Klasse *Creatur* mit der Eigenschaft (Variable) *int layer*. Setzten Sie durch die Konstruktoraufruf der entsprechenden Tiere diese Variable wie folgt:

- Fox → 3
- Rabbit → 2
- Grass → 1

Bauen Sie nun eine Methode in der Klasse *CreatureHandler* ein, welche das *Creature*-Array nach diesem Wert sortiert. Nutzen Sie den Sortieralgorithmus aus dem letzten Semester (Praktikum Programmstrukturen → Bubblesort). Dabei soll der Fuchs zuoberst sein.

Sortieren Sie nach jedem Takt das Array.

3. GraphLife (Zusatz)



In diesem Abschnitt erweitern Sie Ihre Live-Simulation mit einem Liniendiagramm, welche die Anzahl der Kreaturen der jeweiligen Gruppen darstellt. Dazu sollen Sie die externe Programmbibliothek **JFreeChart** verwenden.

Implementierung JFreeChart

JAR-Datei

JFreeChart ist nicht in der allgemeinen Java-Bibliothek vorhanden und muss zusätzlich bezogen werden. Für Java existieren dazu verschiedene Möglichkeiten. Eine Variante ist der Bezug einer JAR-Datei (**J**ava **Ar**chive). Diese Datei besitzt in sich die gesamte Programmstruktur einer Bibliothek und kann daher kompakt dem Projekt hinzugefügt werden. Die JAR-Datei von JFreeChart finden Sie im Praktikum-Ordner. Alternativ können Sie auch die JAR-Datei von der Projektwebseite beziehen.

Schnittstelle

Auf folgender Seite finden Sie die Java-Doc-Seite von JFreeChart:
http://www.jfree.org/jfreechart/api/javadoc/index.html

Verschaffen Sie sich einen groben Überblick über folgende Klassen:

- JFreeChart
- ChartPanel
- XYPlot
- XYSeries
- XYDataset
- XYSeriesCollection

Schauen Sie sich folgendes Beispiel (Abschnitt ***JFreeChart line chart***) an und versuchen Sie mit dem gewonnen Wissen Ihre Life-Simulation so zu erweitern, dass durch ein weiteres Fenster die Anzahl an Kreaturen der entsprechenden Gruppen dargestellt werden. Das Diagramm soll nach jedem Takt automatisch aktualisiert werden:

http://zetcode.com/java/jfreechart/