OPERATIONS RESEARCH
KACKERTSTR. 7
52072 AACHEN

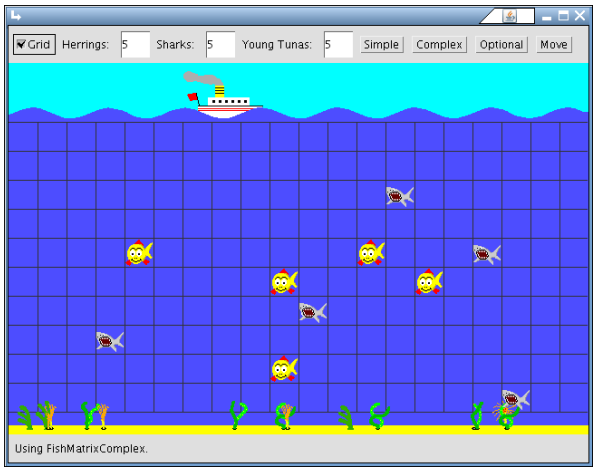
Prof. Dr. Marco Lübbecke
Michael Bastubbe

Johannes Gocke
Florian Thimm

Programmieren, Algorithmen, Datenstrukturen
6. Programmieraufgabe

Abnahme bis spätestens 10.12.2014

In dieser Aufgabe sollt ihr eine kleine Simulation eines Ozeans schreiben, in dem sich mehrere Fische tummeln. Der Ozean besteht aus einem zweidimensionalen Array und jeder Eintrag dieses Arrays kann einen Fisch enthalten. Jede Runde kann jeder Fisch sich auf ein Nachbarfeld im Array bewegen oder stehen bleiben. Das Fischbecken ist am linken und rechten Rand verbunden (die x-Koordinate wird also modulo gerechnet). Ein solches Array, das sich rundenbasiert verändert, heißt *Zellulärer Automat*.



Auf der Homepage findet ihr die Datei `PA06_ocean.zip`, die alle benötigten Dateien enthält. Da das Programm aus über 20 Klassen besteht, wurden diese in Pakete (sogenannte *Packages*) gruppiert. Das gesamte Programm gehört zum Paket `ocean`. Darin gibt es die Pakete `ocean.core` für die Programmlogik und Datenstrukturen, `ocean.fish` für alle Fisch-Klassen, `ocean.gui` für die Oberfläche und darin `ocean.gui.animatedobjects` für die Animation. Um alle beteiligten Dateien zu kompilieren, müsst ihr die Dateien so entpacken, dass das Verzeichnis `ocean` innerhalb eines weiteren Verzeichnisses steht, z.B. `prog06`. Die `main`-Methode befindet sich in `ocean/gui/FishWindow.java`. Ihr könnt aus dem Verzeichnis, in dem `ocean` liegt, mit `javac ocean/*/*.java ocean/*/*/*.java` kompilieren. Danach kann man mit `java ocean/gui/FishWindow` das Programm aufrufen. In `eclipse` legt ihr ein neues Java Projekt und importiert über die Option `General/Archive File` das Zip-Archiv in euren `src`-Ordner. Ruft danach `run` auf die Klasse `gui/FishWindow` auf. Es sollte sich ein Fenster ähnlich zu dem obigen öffnen (die Fische werden fehlen). Falls das Seegras fehlt und ihr **KEIN** `eclipse` benutzt, müsst ihr ggf. im Paket `ocean.gui` in der Klasse `ImageData` die Konstante `baseDirectory` auf "`ocean/gifs/`" ändern.

Eure Aufgabe wird es sein, mehrere neue Klassen im Paket `ocean.fish` zu schreiben, die unterschiedliche Fische modellieren. Diese müssen alle von der Klasse `ocean.fish.Fish` abgeleitet sein und die abstrakte Methode `getDirection` enthalten. Die Methode `doSomething` wird ausgeführt, wenn man auf einen Fisch mit der Maus klickt. Im Paket `ocean.fish` sind außerdem noch die folgenden drei Schnittstellen (sogenannte *Interfaces*) enthalten:

`Eatable` definiert keinerlei Methoden und macht somit nichts, außer der Klasse die Eigenschaft "hat das Interface Eatable" zu geben. So etwas nennt man *Marker Interface*.

`Feedable` bedeutet, dass der Fisch andere Fische, die `Eatable` implementieren, essen kann. Darin gibt es eine Methode `feed`, um den fressenden Fisch mit einem `Eatable` Fisch zu füttern.

`Changing` ist für Fische gedacht, die sich nach jedem Schritt verändern können. Das Interface definiert eine Methode `roundPassed`, die nach jeder Runde für solche Fische aufgerufen werden soll.

Die Klasse `ocean.core.FishMatrix` verwaltet das zentrale Array mit Fischen. Davon ist die Klasse `FishMatrixSimple` abgeleitet. Davon ist `FishMatrixComplex` und davon wiederum `FishMatrixOptional` (für die optionale Aufgabe) abgeleitet!

In diesen abgeleiteten Klassen sollt ihr die fehlende Programmlogik implementieren. `FishMatrix` stellt aber schon jetzt jede Menge nützliche Methoden zur Verfügung und muss nicht verändert werden. Schaut euch diese Klasse gut an, z.B. per JavaDoc. Die Klassen `Coordinate` und `Direction` werden eigentlich überall verwendet, damit man wenig Probleme mit dem Koordinatensystem hat. Alle anderen Klassen sollt ihr auf keinen Fall verändern.

Die simple Simulation. Zunächst sollt ihr eine relativ einfache Simulation eines Ozeans erstellen. Implementiert dafür die Fischklassen `SimpleHerring` und `SimpleShark`.

(a) Die Klasse `SimpleHerring` erbt von `Fish`. Schreibt einen parameterlosen Konstruktor, der den Superkonstruktor mit dem Bild `ImageData.HERRING` aufruft, und außerdem einen Konstruktor, der ein als `ImageData` gegebenes Bild an den Superkonstruktor weiterleitet. Implementiert die Bewegung des Fisches in `getDirection(FishMatrix matrix)`. Ein Fisch vom Typ `SimpleHerring` bewegt sich stets ein Feld nach rechts (`Direction.RIGHT`), sofern das Feld rechts von ihm frei ist. Sonst bleibt er stehen. Überschreibt noch die Methode `toString` durch eine passende.

(b) Die Klasse `SimpleShark` erweitert auch `Fish`. Schreibt die selben Konstruktoren und verwendet das Bild `ImageData.SHARK`. Ein simpler Hai bewegt sich nach oben, bis das nicht mehr geht. Dann bewegt er sich einmal nicht. Ab nächster Runde will er sich nach unten bewegen bis dies nicht mehr geht. Diese Bewegung wiederholt sich. Dabei kehrt er auch vor einem anderen `SimpleShark` um, ein anderer Fisch behindert die Bewegung des Hais aber nicht.

(c) Füllt nun die mit TODO gekennzeichnete Stellen in `FishMatrixSimple` aus. Schreibt die Methode `moveFish`, die den übergebenen Fisch von der einen übergebenen Koordinate auf die andere setzt, und das Zielfeld überschreibt. Die neue Position soll zurückgegeben werden, auch wenn diese (noch) nicht weiterverarbeitet wird. Schreibt die Methode

`moveAllFish`, die alle Fische (links oben angefangen, zeilenweise) mittels `getDirection` nach einer Richtung fragt und mittels der `moveFish` dorthin bewegt. Sorgt dafür, dass in einem solchen Durchgang kein Fisch doppelt bewegt wird! Schreibt außerdem die Methode `fillMatrix`, so dass sie eine `LinkedList<Fish>` erzeugt und an die entsprechende Methode in `FishMatrix` übergibt. (Die Klasse `LinkedList` gehört zum Java-Sprachstandard und ist im Paket `java.util` enthalten. Ihr benötigt eigentlich nur die `add`-Methode, die ein Objekt zur Liste hinzufügt.)

Wenn ihr das gemacht habt, könnt ihr die Simulation in der GUI ausprobieren. Gebt die Anzahl der Heringe und Haie ein und drückt auf "Simple", um die simple Fisch-Matrix zu erzeugen und zu füllen. Der Knopf "Move" führt dann jeweils `moveAllFish` aus.

Die komplexe Simulation. Nun sollt ihr eine komplexere Simulation eines Ozeans erstellen. Diesmal gibt es die Fischklassen `Herring` und `Shark`. Außerdem sollt ihr die mit TODO gekennzeichneten Stellen in `FishMatrixComplex` ergänzen.

(i) Die Klasse `Herring` muss ebenfalls `Fish` erweitern und außerdem das (leere) Interface `Eatable` implementieren. Ein Hering bewegt sich immer zufällig auf ein freies Nachbarfeld. Nur wenn es kein freies Nachbarfeld gibt, bleibt er stehen.

(ii) Ein `Shark` erweitert `Fish` und implementiert `Feedable` und `Changing`. Falls auf einem Nachbarfeld ein essbarer Fisch steht, bewegt sich der Hai dorthin. Ansonsten wählt er gleichverteilt unter den möglichen freien Nachbarn und dem aktuellen Feld.

Ein Hai hat einen internen Zähler, der angibt wieviel Fettpolster er noch hat. Es beginnt bei 10 und kann nie mehr als 20 sein. Fällt es unter 0, stirbt der Hai. Frisst der Hai einen essbaren Fisch, wird das Fettpolster auf 20 aufgefüllt. Diese Funktionalität spiegelt sich in `Feedable` und `Changing` wieder. Wenn die Methode `feed` aufgerufen wird, wird die Fettreserve auf 20 gesetzt. Nach jeder Runde, in der sich der Hai tatsächlich bewegt hat, verliert er eine Einheit der Futterreserve. Das wird in `roundPassed` imlementiert. (Dafür muss die vorherige Position des Hais gespeichert werden, z.B. in der `getDirection`.) Falls nötig löscht sich der Hai selbstständig in dieser Methode aus der Matrix. (Die eigene Position lässt sich mit der Methode `whereIsMyFish` in `FishMatrix` herausfinden.) In der `doSomething` passiert zwar nichts, aber es soll ein String zurückgegeben werden, wieviel Fettpolster der Hai noch hat.

(iii) Die Klasse `FishMatrixComplex` erbt von `FishMatrixSimple`. Sie soll sich zum einen darin unterscheiden, dass `moveFish` überprüft, ob das Zielfeld frei ist. Ist es nicht frei, kann der Fisch nur dorthin ziehen, falls er `Feedable` ist und auf ein Feld mit einem `Eatable` Fisch zieht. In dem Fall wird die `feed`-Methode des fressenden Fischs aufgerufen und der gefressene Fisch überschrieben. Ist ein Zug in die gewünschte Richtung nicht möglich, soll der Fisch nicht bewegt werden. Wiederum soll die neue Koordinate des Fisches zurückgegeben werden, was nun nicht mehr zwingend dem Ziel entsprechen muss.

Zum anderen soll, nachdem alle Fische bewegt wurden, in der Methode `moveAllFish` für jeden Fisch, der `Changing` implementiert, die Methode `roundPassed` aufgerufen werden. Um das mit möglichst wenig Schreibarbeit zu erreichen, könnt ihr die Bewegung der `FishMatrixSimple` zuerst mit `super.moveAllFish()` aufrufen. Schließlich soll noch die

Methode `fillMatrix(int, int)` so überschrieben werden, dass sie nun `Herring` und `Shark` statt der simplen Fische erzeugt.

(Optional) **Die optionale Simulation.** Optional könnt ihr eure Simulation noch um Thunfische erweitern. Implementiert dafür zwei weitere Fischklassen, `YoungTuna` und `OldTuna`. Nur der junge Thunfisch kann gegessen werden und verändert sich nach 7 Runden zu einem alten Thunfisch. Ein alter Thunfisch ist nicht essbar und verändert sich von alleine nicht mehr. Allerdings kann man auf einen alten Thunfisch klicken und erzeugt damit zwei kleine Thunfische: Einen an der Stelle des alten und einen auf einem freien Nachbarfeld (falls möglich). Alle Thunfische bewegen sich zufällig auf freie Nachbarfelder, genau wie ein `Hering`. Das kann man ausnutzen: Schreibt man "zwischen" `Fish` und `Herring` eine Klasse `RandomMovingFish`, die genau diese Sorte Bewegung implementiert (aber kein weiteres Interface hat), kann man davon alle drei Fischsorten ableiten und muss die Bewegung nur einmal implementieren. Falls ihr das macht, schreibt zumindest die neue Methode `fillMatrix` der Klasse `FishMatrixOptional`, um auch Tunfische erzeugen zu können.

Hinweise

- Denkt daran, dass ihr eine Klasse aus einem anderen Paket importieren müsst, bevor ihr sie verwenden könnt.
- Man kann über alle Werte eines Enum-Typs iterieren. Beispielsweise gibt

  ```
  for (Direction dir : Direction.values()) { System.out.println(dir); }
  ```

  alle möglichen Werte in `Direction` aus. Die Reihenfolge ist dabei nicht festgelegt. Dies ist aber hilfreich, um alle Nachbarfelder zu überprüfen.
- Die `FishMatrix` ist so geschrieben, dass sie die x-Koordinate nach rechts und die y-Koordinate nach unten *interpretiert*. Die Klasse `Coordinate` hat die öffentlichen Felder `x` und `y`. Solange also mittels den vorgegebenen `get/setFish(Coordinate coord)` gearbeitet wird, ist das Koordinatensystem intuitiv. Die Methode `getCoordinate (from, dir)` gibt zu einer Koordinate und einer Richtung die resultierende Koordinate zurück. `getFish` kann auch direkt mit einer Koordinate und einer Richtung aufgerufen werden, um benachbarte Fische abzufragen. Intern ist das Array aber wie eine Matrix indziert: Erst y, dann x.

Viel Spaß und Erfolg!