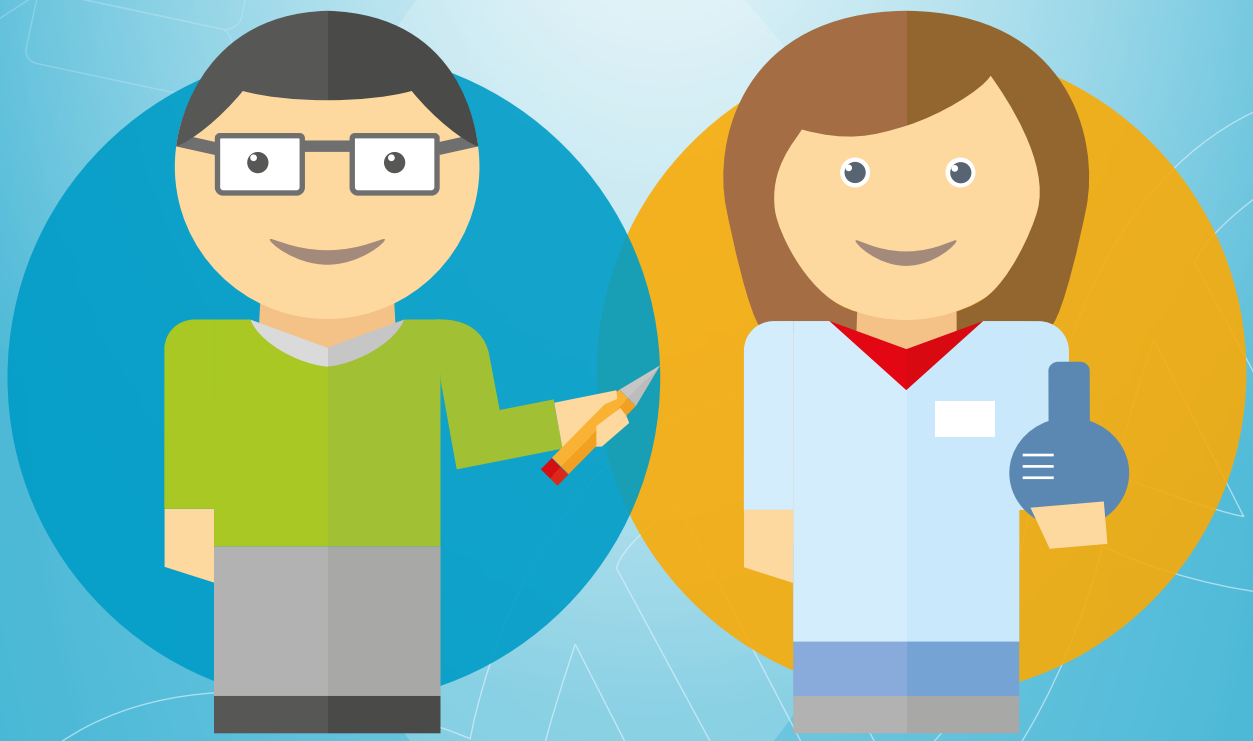
teachers + scientists

Für Wissenschaft begeistern



Kooperation Osnabrück

Fressen und gefressen werden – Vertiefung und Erweiterung von grundlegenden objektorientierten Konzepten in der Informatik anhand von Räuber-Beute-Populationen

Ein Projekt von Science on Stage Deutschland e. V. in Kooperation mit der Stiftung Jugend forscht e. V.

TEACHERS + SCIENTISTS: FÜR WISSENSCHAFT BEGEISTERN

Materialien und Konzepte für den MINT-Unterricht

28. Februar – 1. März 2013
Brainstorming zur Projektidee
Berlin

13. – 14. Juni 2014
1. überregionales Projekttreffen
Max-Delbrück-Centrum für Molekulare Medizin Berlin

23. – 24. Januar 2015
2. überregionales Projekttreffen
Universität Bielefeld

25. – 26. September 2015
3. überregionales Projekttreffen
Universitätsklinikum der RWTH Aachen

22. – 23. April 2016
4. überregionales Projekttreffen
Hochschule Osnabrück

5. Mai 2017
Abschlusspräsentation
Berlin

2017 – 2018
Lehrerfortbildungen und Teilnahme an Tagungen zur Verbreitung der Ergebnisse

über die Jahre
individuelle Treffen und Projektpräsentationen der regionalen Kooperationen

Als im Sommer 2014 das Pilotprojekt Teachers + Scientists startete, war dies für alle beteiligten Lehrkräfte und Forschenden der Beginn einer neuen Art der Zusammenarbeit – so etwas gab es bisher noch nicht!

Auch wenn bereits einzelne Kontakte bestanden, hatten sich diese bislang auf die Förderung der Schülerinnen und Schüler konzentriert. Nun sollten erstmals Lehrkräfte vom intensiven dreijährigen Austausch mit Forschenden und von Einblicken in deren aktuelle Forschung profitieren.

Mit dem Ziel, die Gelingensfaktoren und Herausforderungen solcher Kooperationen in einem Leitfaden und die Ergebnisse der gemeinsamen Zusammenarbeit in Form von Unterrichtskonzepten zu veröffentlichen, nahmen die fünf regionalen Kooperationen in Aachen, Berlin, Bielefeld, Heidelberg und Osnabrück ihre Arbeit auf.

Was den Prozess auszeichnete, war die individuelle Umsetzung: von der theoretischen Ausarbeitung über mehrtägige Laborpraktika bis zum Langzeitexperiment. Die Resultate sind demzufolge unterschiedlich aufbereitet und spiegeln die verschiedenen regionalen Kooperationsformen wider.

Die nachfolgenden Materialien sollen Ihnen nun Anregungen für den eigenen Unterricht geben und Sie ermutigen, den Kontakt zu Forschenden zu suchen. Dadurch lassen sich aktuelle wissenschaftliche Inhalte in der Schule aufgreifen, die wiederum Schülerinnen und Schüler für das Forschen begeistern!

Sollten Sie Fragen haben, melden Sie sich über info@science-on-stage.de bei Science on Stage Deutschland e. V. Wir stellen gerne den direkten Kontakt zu den teilnehmenden Forschenden und Lehrkräften her. Die jeweiligen Kontaktdaten finden Sie auch am Ende jeder Einheit.

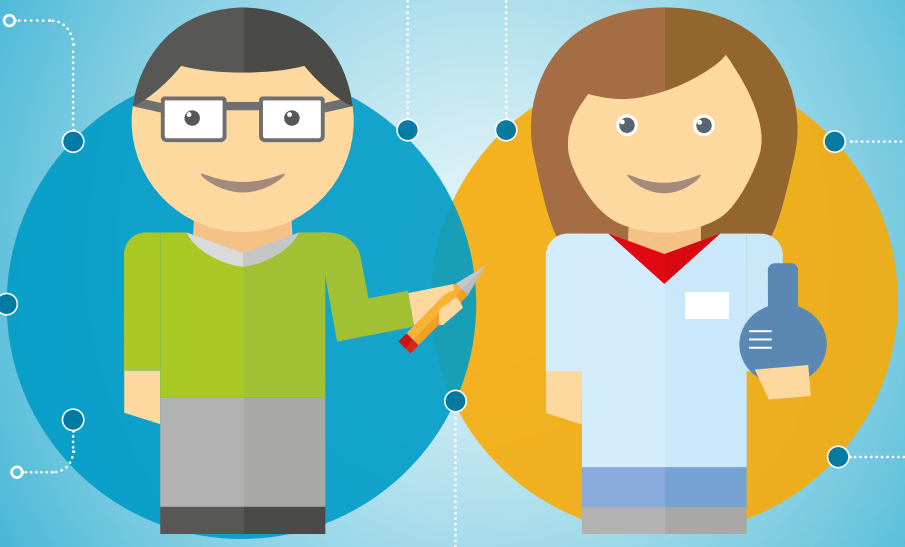
Viel Freude und Inspiration für Ihre eigene Arbeit wünschen Ihnen Science on Stage Deutschland e. V. und die Stiftung Jugend forscht e. V.!

Teachers + Scientists: Auf einen Blick

10
Schulen
Einhard-Gymnasium Aachen, Andreas-Gymnasium Berlin, Robert-Havemann-Gymnasium Berlin, Georg-Büchner-Gymnasium Berlin, OSZ Gesundheit I Berlin, Ursulaschule Osnabrück, Widukind-Gymnasium Enger, Gymnasium Heepen, Gesamtschule Hüllhorst, HBLA Ursprung/Österreich

6
Hochschulen/Forschungseinrichtungen
Universität Bielefeld, Hochschule Bielefeld, Hochschule Osnabrück, Deutsches Krebsforschungszentrum Heidelberg, Max-Delbrück-Centrum für Molekulare Medizin Berlin, Universitätsklinikum der RWTH Aachen

7
regionale Kooperationen
1× Aachen, 1× Berlin, 3× Bielefeld, 1× Heidelberg, 1× Osnabrück



5
Städte
Aachen, Berlin, Bielefeld, Heidelberg, Osnabrück

14
Lehrkräfte

12
Wissenschaftlerinnen und Wissenschaftler

4
Bundesländer
Baden-Württemberg, Berlin, Niedersachsen, Nordrhein-Westfalen

Projektinhalt und Gewinn (2014–2017)
→ Förderung langfristiger Kooperationen zwischen Lehrkräften und Forschenden
→ Lehrkräfte stehen im Mittelpunkt, sind an aktueller Forschung beteiligt und können somit Inhalte für ihren Unterricht ableiten
→ Ziel: Förderung der Unterrichtsqualität, damit sich mehr junge Menschen für MINT-Fächer begeistern

Ergebnisse
→ Leitfaden zum Aufbau von Kooperationen zwischen Lehrkräften und Forschenden
→ Unterrichtsmaterial zu den Themen: Humangenetik, Krebsforschung, Experimentelle Ökologie und Ökosystembiologie, Elementarteilchenphysik, Epidemiologische Studien, Objektorientierte Programmierung, Mechanik und Sensorik

Verbreitung
→ Bundesweite Lehrerfortbildungen
→ Präsentationen auf Fachkonferenzen
→ Fortsetzung der Kooperationen nach Projektende

Kooperation Osnabrück

STECKBRIEF

→ **Schule:**
Ursulaschule

GYMNASIUM Ursulaschule

→ **Lehrkraft:**
Dr. Andreas Degenhard

→ **Forschungseinrichtung:**
Hochschule Osnabrück, Fakultät für Ingenieurwissenschaften und Informatik

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

→ **Forschende:**
Prof. Dr. Heinz-Josef Eikerling,
Prof. Dr. Michael Uelschen

→ **Themen:**
Objektorientierte Programmierung

→ **Involvierte Unterrichtsfächer:**
Informatik (mit Bezug zur Biologie)

INTERVIEW

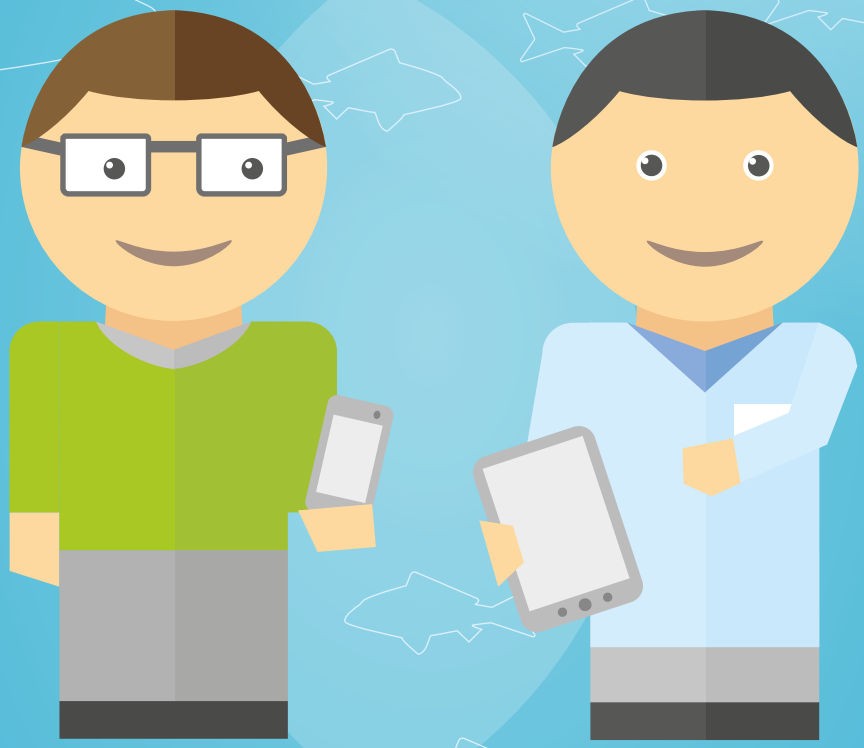
→ **Teachers + Scientists ist für uns …**
… die Möglichkeit, unsere Zusammenarbeit mit der Hochschule in einen größeren Rahmen einzubetten, um interessierte Lehrkräfte zu erreichen.

→ **Ich mache bei Teachers + Scientists mit, weil …**
… der Unterricht dadurch wichtige Impulse hinsichtlich Interdisziplinarität und Aktualität erhält.

→ **Planen Sie eine Fortsetzung der Kooperation nach Projektende?**
Auf jeden Fall.

Fressen und gefressen werden – Vertiefung und Erweiterung von grundlegenden objektorientierten Konzepten in der Informatik anhand von Räuber-Beute-Populationen

Dr. Andreas Degenhard · Prof. Dr. Heinz-Josef Eikerling · Prof. Dr. Michael Uelschen



SCHLAGWÖRTER: Objektorientierte Programmierung, Räuber-Beute-Population, Lotka-Volterra, Computersimulation

UNTERRICHTSFACH: Informatik (mit Bezug zur Biologie)

ALTERSGRUPPE DER SCHÜLERINNEN UND SCHÜLER: Ab Klasse 11/12

VERWENDETE MATERIALIEN: Softwareentwicklungsumgebung Greenfoot, Open-Source-Softwarebibliotheken

ERFORDERLICHE VORKENNTNISSE: Grundkenntnisse in der imperativen Programmierung, wie z. B. Wiederholungs- und Auswahlanweisungen; grundlegende Vertrautheit mit der Lernumgebung Greenfoot und den grundlegenden Konzepten der OOP, wie beispielsweise Klassen, Objekte, Attribute, Methoden und Vererbung, sowie die Listenstruktur als abstrakter Datentyp

1 | Einführung

Zur Einführung der objektorientierten Programmierung (OOP) und eines damit verbundenen objektorientierten Denkens stehen eine Vielzahl an Hilfsmitteln für den schulischen Unterricht zur Verfügung, beispielsweise im Bereich der Lehr- und Lernsoftware. Sollen die erlernten Konzepte der OOP dann konkret an Anwendungsbeispielen eingeübt und darüber hinaus erweitert werden, sind besonders solche Szenarien geeignet, die die Verwendung einer objektorientierten Vorgehensweise/Modellierung sinnvoll erscheinen lassen. Zudem sollte die Aufgabenstellung hinsichtlich der Komplexität überschaubar sein, sodass sich Schülerinnen und Schüler durch Aussicht auf Erfolgserlebnisse bei dem Vorhaben angesprochen fühlen. Eine Möglichkeit dies zu realisieren, stellen Problem- und Aufgabenstellungen dar, die eine schrittweise Hinzunahme weiterer Klassen/Objekte oder deren Erweiterung durch geeignete Methoden ermöglichen.

Die hier beschriebene Implementierung einer Simulation kann für eine Vertiefung gelernter Konzepte der OOP verwendet werden. Die Simulation orientiert sich an grundlegenden biologischen Prinzipien von Populationen, die sinnvoll und einsichtig sind und einen anschaulichen Zugang zur OOP ermöglichen. Darüber hinaus kann das Verhalten der Objekte, in diesem Fall also der einzelnen Tiere in den Populationen, auch spielerisch betrachtet werden, wobei das Aussterben einer Population durch das Einstellen der unterschiedlichen Parameter der Simulation vermieden werden soll. Konkret wird die schrittweise Implementation einer Räuber-Beute-Population vorgestellt, wobei die Schülerinnen und Schüler möglicherweise überwiegend selbstständig, unterstützt durch das beigefügte Material[1], die Simulation programmtechnisch umsetzen können. Der Grundaufbau der Simulation lässt sich auf vielschichtige Weise ergänzen und erweitern und bietet damit verschiedene Differenzierungsmöglichkeiten. Dabei wird zunächst eine Umsetzung mithilfe der Lernsoftware Greenfoot[2] beschrieben. Anschließend wird erläutert, wie sich das Projekt als Applikation auf mobilen Geräten realisieren lässt.

2 | Motivation und Information für Lehrkräfte

In den letzten Jahren hat Objektorientierung, nicht zuletzt mit dem verstärkten Aufkommen objektorientierter Sprachen wie JAVA, C++ oder C#, an Bedeutung gewonnen.[3] Die Konzepte der objektorientierten Programmierung sind in vielen Bereichen der professionellen Softwareentwicklung fest verankert. Auch im Unterricht für das Fach Informatik werden neben Grundlagen der imperativen Programmierung häufig auch Konzepte der OOP erarbeitet. Die Idee der Objektorientierung ist es, die Architektur der Software an der Wirklichkeit auszurichten, indem diese durch das Zusammenspiel kooperierender Objekte beschrieben wird.

Biologische Algorithmen und Simulationen zeigen einerseits interessante Zusammenhänge zwischen biologischen Prinzipien und algorithmischen Implementierungen auf, andererseits kann sich hier die Gelegenheit bieten, auf bereits vorhandenes Vorwissen der Schülerinnen und Schüler fachübergreifend zurückzugreifen. Beispielsweise werden im Fach Biologie Phänomene sowohl auf der Ebene eines einzelnen Organismus als auch in Bezug auf ganze Populationen (Evolutionsprozesse) betrachtet. Zu den zuletzt genannten gehören insbesondere auch verschiedene Räuber-Beute-Beziehungen.

Eine mögliche Beschreibung einer Räuber-Beute-Beziehung stellt das Modell von Lotka-Volterra dar, wobei ein Einstieg über beobachtete Populationsverläufe mit Oszillationen erfolgen kann.[4] Entscheidend ist dabei die Beobachtung, dass der annähernd periodische Verlauf der Anzahlen von Räubern zeitlich versetzt zu denen der Beute ist. Das Maximum bezüglich der Anzahl der Räuber wird daher erst erreicht, nachdem sich die Anzahl der Beutetiere wieder reduziert. Eine mathematische Beschreibung der daraus resultierenden Erkenntnisse basiert auf einem Satz von Regeln, welche die Grundlage für weitere Berechnungen und Simulationen darstellen. Die mathematischen Lösungen lassen sich sowohl mit den realen Daten vergleichen, als auch mit den Ergebnissen geeigneter Computersimulationen.

Eine Computersimulation, welche die Räuber-Beute-Beziehung anschaulich macht, ist die Simulation Wator von Alexander K. Dewdney und David Wiseman.[5] Dieses Modell basiert auf sehr anschaulichen und einfachen Regeln und dient hier als Grundlage für die Vertiefung der OOP mithilfe von biologisch motivierten Computersimulationen. Obgleich die Beschreibung des Wator-Modells auf nur wenigen wesentlichen Regeln basiert, finden sich in der Literatur gelegentlich geringfügige Abweichungen bei den genauen Formulierungen.[6]

Neben dem Wator-Modell wurden bereits andere Räuber-Beute-Szenarien in Greenfoot implementiert.[7] Unterschiede bestehen dabei in der Wahl eines begrenzten Territoriums, den vorgegebenen Regeln oder auch dem konkreten Vorgehen beim Erstellen der Simulation.

3 | Schrittweise Umsetzung mithilfe der Lernsoftware Greenfoot

Eine mögliche Umsetzung des Wator-Modells als Computersimulation bietet sich mithilfe der Lernsoftware Greenfoot an.[8] Diese stellt eine Entwicklungsumgebung zusammen mit einer grafischen Oberfläche bereit, welche sowohl den Simulationsverlauf als auch die Beziehung der beitragenden Klassen/Objekte darstellt.

Die schrittweise Implementierung des Wator-Modells als Simulation in Greenfoot beginnt mit einem Grundaufbau, welcher die zwei wesentlichen (Ober-)Klassen *Ocean* und *Animal* enthält. Diese Oberklassen leiten sich direkt von den in Greenfoot automatisch implementierten Klassen *World* und *Actor* ab. In der Abb. 1 ist dieser Zusammenhang am rechten Bildrand grafisch dargestellt.

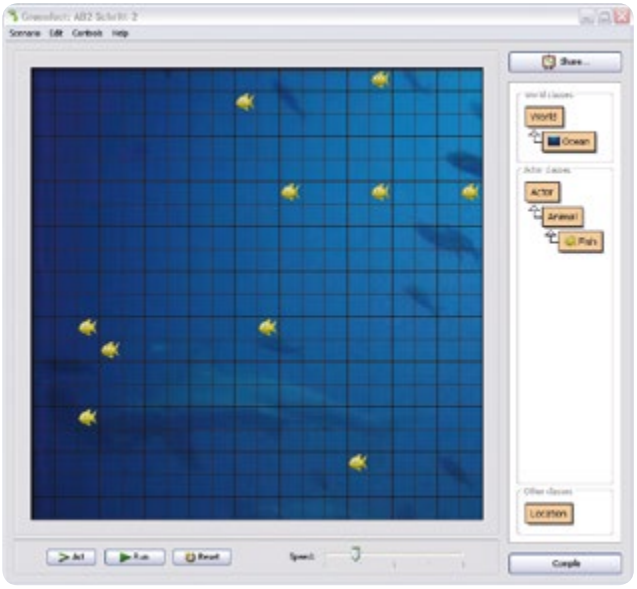


ABB. 1 Der Einstieg in die schrittweise Entwicklung der Simulation erfolgt über die Erstellung eines Ozeans, der zunächst lediglich Objekte vom Typ ‚Fisch' enthält.

Im Fall der hier zu implementierenden Simulation ist die Simulationswelt ein Ozean und die Akteure, die Objekte in dieser Welt, sind Tiere. Ein erstes Etappenziel auf dem Weg zu der Implementierung einer sinnvollen Simulation von Wator besteht darin, dass sich eine feste Anzahl von Beutefischen, im Weiteren Fische genannt, zufällig weiterbewegt (siehe Abb. 1). Um den Ozean als unbegrenztes Territorium zu simulieren, bietet sich die Verwendung toroidaler Randbedingungen an den Kanten der zweidimensionalen Simulationsfläche an. Dabei wird diese so gekrümmt, dass sich die gegenüberliegenden Kanten berühren, wie dies in Abb. 2 dargestellt ist.

Anschließend soll sichergestellt werden, dass sich immer nur ein Fisch auf einem Simulationsplatz (kurz Zelle) aufhalten kann. Daran schließt sich die Erzeugung von Jägerfischen (kurz Haie) an, die ebenfalls von der Klasse *Animal* abgeleitet werden.

Obwohl die Haie ähnliche Eigenschaften wie die (Beute-)Fische besitzen, soll die Klasse der Haie später so erweitert werden, dass Fische von Haien gefressen werden können. Während für die Fische immer ausreichend Nahrung im Ozean zur Verfügung steht, benötigen die Haie andere Fische, um zu überleben. Mithilfe geeigneter Simulationsparameter kann abschließend das Langzeitverhalten der Populationen grafisch ausgewertet und mit dem Modell von Lotka-Volterra verglichen werden.

3.1 Arbeitsmaterial I

Das online verfügbare 'Arbeitsmaterial I'[1] dient als konkrete Hilfestellung bei der Erstellung der beschriebenen Simulation. Die Arbeitsblätter enthalten Aufgabenstellungen, die aufeinander aufbauen und die Simulation schrittweise erweitern, sodass nach jeder Aufgabe ein neues lauffähiges Programm entsteht. Zu den Arbeitsblättern sind mögliche Lösungen vorhanden.[1] Diese können Schülerinnen und Schülern zur Verfügung gestellt werden, denen es nicht gelungen ist, eigene Lösungen zu erstellen. Dies ist didaktisch sinnvoll, um im Unterricht gegebenenfalls zu demselben Ausgangsniveau zurückzufinden, sodass alle mit der gleichen Aufgabenstellung fortfahren können.

Die Arbeitsblätter 1 und 2 enthalten sehr detaillierte Arbeitsaufträge, die i. d. R. ohne Hilfe der Lehrkraft selbstständig bearbeitet werden können. Durch diese schrittweise Implementation der Simulation sollen sich die Schülerinnen und Schüler mit deren Klassen vertraut machen und die erste grundlegende Entwicklungsstufe selbstständig umsetzen. Die Arbeitsblätter 3 und 4 umfassen ebenfalls Arbeitsaufträge in Form konkreter Aufgabenstellungen, wobei jedoch keine schrittweise Implementation beschrieben wird. Daher dienen diese Arbeitsblätter zwar als Anleitung für die Weiterentwicklung der Simulation, die konkrete Umsetzung im Unterricht ist damit aber nicht vorgegeben. Bei dem Arbeitsblatt 5 wird davon ausgegangen, dass den Schülerinnen und Schülern das Programm als Lösung zur Verfügung gestellt wird und sie die Möglichkeit haben, verschiedene Simulationsabläufe zu analysieren. Dabei können besonders interessierte Schülerinnen und Schüler auch Einblick in die Erweiterungen der verschiedenen Klassen bekommen, da die Lösung auf den bisherigen Implementierungen basiert.

Die vorhandenen Lösungen stellen lediglich eine mögliche Umsetzung dar, da das Wator-Modell programmtechnisch unterschiedlich implementiert werden kann. Zudem erhebt das Lösungsmaterial keinen Anspruch auf eine idealisiert didaktisch-methodische Umsetzung. Zumeist wird versucht, mit grundlegenden Kenntnissen in der Programmiersprache Java auszukommen. Erweiterte Konstrukte von Java werden verwendet, wenn dadurch das algorithmische Denken unterstützt und einfach gehalten werden soll. Dazu gehört z. B. die Verwendung eines Iterators im Umgang mit dem abstrakten Datentyp einer Liste.

Um lange Namen wie Beutefisch und Räuberfisch zu vermeiden, wird auf eine Oberklasse mit dem Bezeichner ‚Fisch' verzichtet. Somit stammen die Klassen ‚Fisch' und ‚Hai' direkt von einer Klasse *Animal* ab. Obgleich diese beiden Klassen deutsche Bezeichnungen tragen, wurden sämtliche Methoden in allen Klassen in Englisch benannt, um weitere Durchmischungen zu vermeiden. Greenfoot stellt viele Methoden wie beispielsweise `'addObject()'` und `'getObjectsAt()'` bereit, auf die in der Erstellung der Simulation zurückgegriffen wird. Dabei wird vorausgesetzt, dass die Schülerinnen und Schüler deren Beschreibung in den zahlreich vorhandenen Online-Dokumentationen nachlesen können.

4 | Wator als Android-App

Smartphones und Tablets sind aus dem Alltag der Schülerinnen und Schüler nicht mehr wegzudenken. Applikationen (kurz Apps) sind Programme, die unter den verschiedenen Betriebssystemen unterschiedliche Dienste zur Verfügung stellen.

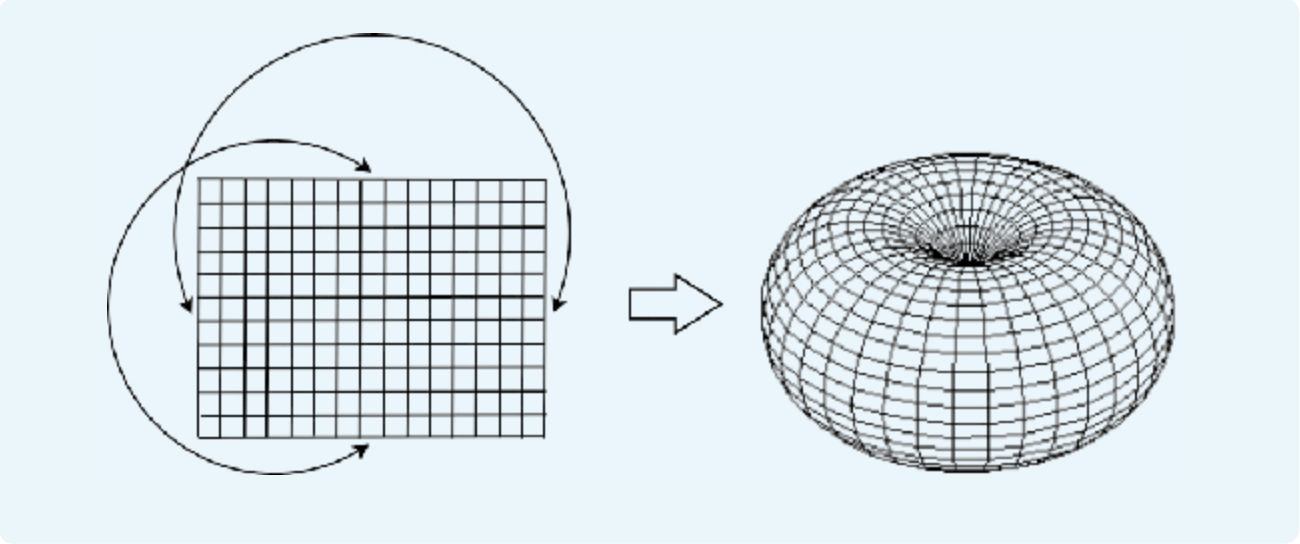


ABB. 2 Visualisierung toroidaler Randbedingungen für eine zweidimensionale Simulationsfläche

Mittlerweile gibt es auch Apps, die eine komplette Entwicklungsumgebung zum Programmieren anderer Apps anbieten. Es ist somit möglich, den grundlegenden Begriff des Objekts in der Informatik allein durch die mobile Kommunikationstechnologie als Unterrichtsmedium zu vermitteln. Dabei werden Anwendungen direkt am Smartphone oder Tablet entwickelt und als App ausgeführt, sodass Schülerinnen und Schüler das Programmieren als Teil ihrer Lebenswelt verstehen können. Dies wird durch die Kombination von den zuvor schrittweise gegliederten Entwicklungsstufen und den mittlerweile sehr leistungsfähigen mobilen Kommunikationsgeräten ermöglicht.

Im Hinblick auf die Überführung der Wator-Simulation in eine App gibt es verschiedene Möglichkeiten. Nachfolgend werden daher zwei unterschiedliche Herangehensweisen vorgestellt.

4.1 Überführung von Greenfoot in eine Android-App

Zuvor wurde bereits die Möglichkeit beschrieben, mit der Lernsoftware Greenfoot entsprechende objektorientierte Programme zu erstellen. Diese können jedoch nicht am Smartphone oder Tablet selbst entwickelt bzw. verändert werden.

Die daher hier eingesetzte Softwarebibliothek ‚JDroidLib'[9] basiert aber auf den grundlegenden Konzepten und Programmiertechniken von Greenfoot, sodass sich das Wissen von einem System zum anderen transferieren lässt.

4.1.1 Voraussetzungen zur Erstellung einer Wator-App

Auf einem Smartphone oder Tablet müssen die Anwendungen (Apps) ‚AIDE – Android IDE'[10] und ‚ProjectBuilder for AIDE' installiert werden, was nur wenige Minuten dauert. Die ‚AIDE – Android IDE' ist eine Entwicklungsumgebung, die es ermöglicht, Android-Apps direkt auf dem Smartphone oder Tablet zu installieren und zu bearbeiten. Der ‚ProjectBuilder for AIDE' ist ein Werkzeug, das bei der Erstellung von Android-Apps hilft. Die Verwendung eines solchen Werkzeugs ist sinnvoll, da der Ausgangspunkt einer Android-App nicht durch eine übliche main-Methode gebildet wird, sondern durch spezielle Activity- oder Manifest-Dateien bereitgestellt werden muss. Da diese Dateien aber eigentlich Aspekte des Layouts betreffen, ist es im Hinblick auf einen sinnvollen didaktischen Zugang angemessen, diese mit dem ‚ProjectBuilder for AIDE' automatisch generieren zu lassen, sodass sich die Erstellung der Android-App nur auf die Implementierung der Simulation beschränkt.

Um die Simulation grafisch darzustellen, steht die didaktisch konzipierte und für Schulen frei verfügbare Java-Klassenbibliothek ‚JDroidLib' zur Verfügung. Diese ermöglicht aufgrund der vorhandenen Methoden eine besonders intuitive Programmierung und wird ebenfalls vom ‚ProjectBuilder for AIDE' automatisch mit in das Android Projekt integriert. Für die Java-Klassenbibliothek ‚JDroidLib' gibt es zudem eine nahezu analoge Java-Klassenbibliothek ‚JGameGrid'[11] für die Verwendung an PCs. Dadurch können Programme auch zunächst am PC in einer Vorstufe entwickelt und später auf das Smartphone oder Tablet übertragen und dort weiterentwickelt werden.

Um die Erstellung einer App didaktisch sinnvoll umzusetzen, wird empfohlen, dass sich die Schülerinnen und Schüler eine entsprechende Vorlage der (Haupt-)Klasse ‚WasserWelt' auf das Smartphone bzw. Tablet downloaden, die sehr eindrucksvoll demonstriert, wie einfach sich die Wasserwelt auf dem Android-Gerät grafisch darstellen lässt. Diese Klasse ist der entsprechenden Klasse in der Greenfoot-Umgebung sehr ähnlich und enthält die Methode `main()` mit dem Aufruf des Standardkonstruktors, der in sehr übersichtlicher Weise das Aussehen der Oberfläche zur grafischen Visualisierung festlegt.

4.1.2 Erzeugen von Objekten

Nachdem die grafische Oberfläche der Wasserwelt auf dem Android-Gerät angezeigt wurde, kann die erste konkrete Aufgabe darin bestehen, einen einzelnen Fisch als Objekt zu er-

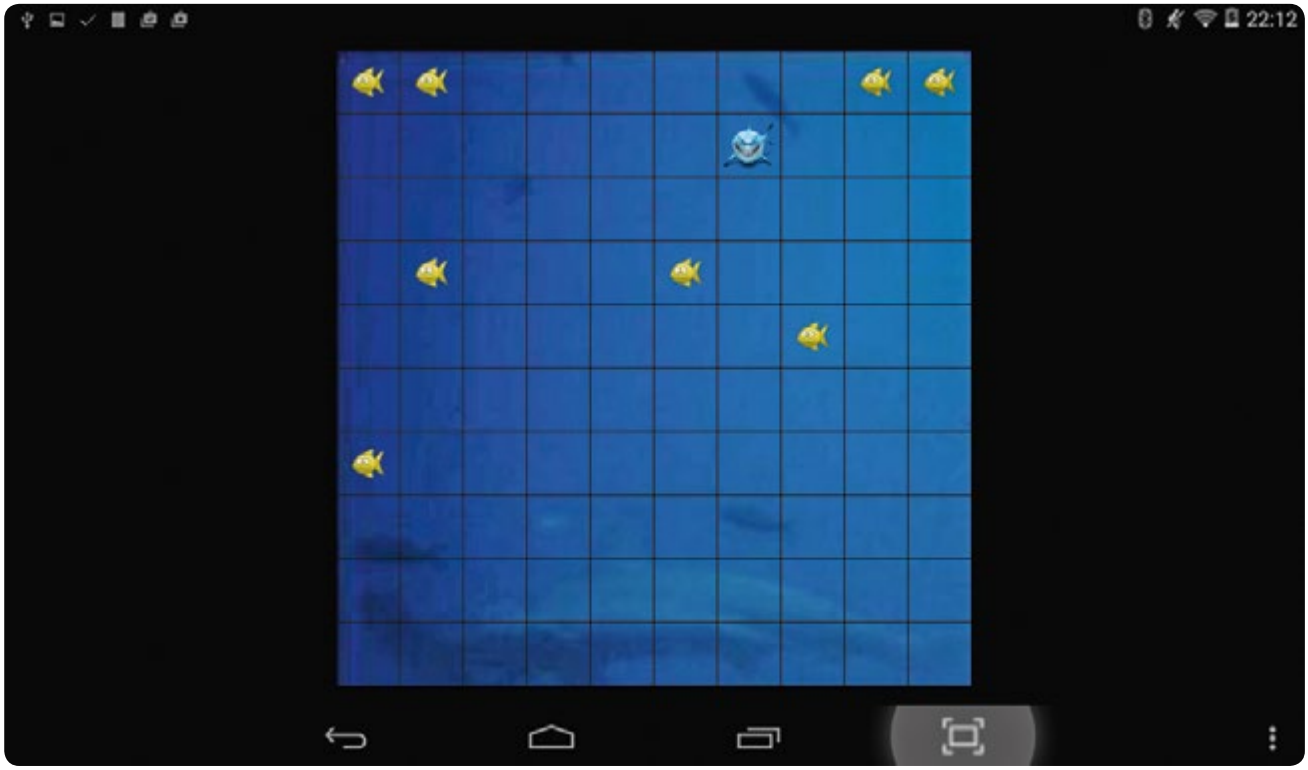


ABB. 3 Wasserwelt Android

zeugen und diesen in der Wasserwelt als zweidimensionale Simulationsebene darzustellen. Dazu müssen die Schülerinnen und Schüler zunächst eine weitere Klasse für die Fische (Fisch.java) selbstständig implementieren. Diese besteht nur aus zwei Methoden, dem Konstruktor und der Methode `act()`. Analog kann anschließend eine entsprechende Klasse für die Haie (Hai.java) implementiert werden. Während der jeweilige Konstruktor auf ein Bild, das ein entsprechendes Objekt darstellt, verweist, ist die Methode `act()` am Anfang vollständig leer, da (im Sinne der OOP) noch kein Verhalten für die Fische und die Haie festgelegt wurde. Konkret kann dann mit einer einzigen Programmzeile im Konstruktor der Klasse ‚WasserWelt' ein Objekt vom Typ ‚Fisch' oder ‚Hai' erzeugt und an einem zufälligen Ort in der Wasserwelt abgebildet werden (siehe Abb. 3).

```
addActor(new Fisch(), getRandomEmptyLocation());
```

oder entsprechend der Erzeugung eines Hais:

```
addActor(new Hai(), getRandomEmptyLocation());
```

Dabei ist es auch möglich, dass die Schülerinnen und Schüler selbstständig, ohne Vorgabe dieser Programmzeilen, vorgehen und verschiedene Möglichkeiten zum Erzeugen von Objekten mithilfe einer Übersicht vorhandener Methoden testen.

So erfahren die Schülerinnen und Schüler den Umgang mit Bibliotheken und begreifen die sinnvolle Nutzung bereits zur Verfügung stehender Methoden für die Entwicklung von Anwendungen.

4.1.3 Bewegung von Objekten

Im nächsten Schritt wird eine vorgegebene Anzahl solcher Objekte mit einer Wiederholungsanweisung erzeugt. Eine erste Bewegung der Objekte besteht im Hinblick auf die Simulation in einem Wechsel auf ein zufällig ausgewähltes Nachbarfeld, das ebenfalls in einfacher Weise in der Methode `act()` implementiert werden kann:

```
// Hai schwimmt zufaellig ein Feld weiter
r = Math.random();
if (r>0.75) turn(90);
else if (r>0.5) turn(180);
else if (r>0.25) turn(270);
move(1);
```

Spätestens hier sollte klar werden, dass dem Rand der Wasserwelt eine besondere Bedeutung zukommt. Dabei muss ein Randfeld im Programm zunächst erkannt und anschließend gesondert behandelt werden. Ein einfacher Umgang mit Randfeldern besteht darin, einen Schritt auf ein nicht vorhandenes Feld einfach nicht auszuführen und daraufhin den nächsten zufällig ausgewählten Schritt zu betrachten. Im eigentlichen Modell soll der Ozean jedoch über einen Torus als „unendlich ausgedehnt" simuliert werden, sodass das Objekt wieder passend an der gegenüberliegenden Seite in die Wasserwelt aufgenommen werden kann (siehe Abb. 2). Auch diese Art von Randbedingung lässt sich sehr einfach implementieren, indem bei einem Wechsel auf ein äußeres Randfeld sofort auf die gegenüberliegende Seite gewechselt wird. Dies stellt insofern keine Einschränkung dar, da die dargestellte Fläche beliebig dimensioniert werden kann.

4.1.4 Interaktion zwischen Objekten

Nachdem ermöglicht wurde, dass sich sowohl Fische als auch Haie beliebig über die Fläche bewegen können, sollen sich die Schülerinnen und Schüler der Umsetzung erster Interaktionen zuwenden, wobei zunächst der Unterschied zwischen Objekten vom Typ ‚Fisch' und ‚Hai' vernachlässigt werden kann und nur eine Doppelbesetzung eines Feldes vermieden werden soll. Sinnvollerweise wird dieses Verhalten wiederum in der Methode `act()` implementiert. Im Anschluss hieran kann die Klasse ‚Fisch' so bestehen bleiben, bei der Klasse ‚Hai' muss jedoch unterschieden werden, ob ein Nachbarfeld von einem normalen Fisch oder von einem Hai besetzt ist. Falls das Nachbarfeld eines Hais von einem normalen Fisch besetzt ist, kann der Hai dieses Feld besetzen. Dabei müssen die Schülerinnen und Schüler erkennen, dass die geforderte Überprüfung davon abhängt, ob das auf dem Nachbarfeld befindliche Objekt vom Typ ‚Hai' ist. Da zuvor auf den allgemeinen Objekttyp ‚Actor' geprüft werden konnte, wird auf eindrucksvolle Weise klar, dass die Klasse ‚Hai' den Typ ‚Actor' im Sinn einer Vererbung in der OOP weiter spezifiziert.

Nach der Umsetzung verschiedener Aktionen in der Methode `act()` für die unterschiedlichen Objekte, besteht das weitere Vorgehen darin, über selbst zu definierende Attribute und Methoden die zuvor festgelegten Anforderungen an die Simulation umzusetzen. In einem ersten Schritt kann geprüft werden, ob sich auf einem Feld mit einem Hai auch ein Objekt vom Typ ‚Fisch' befindet. Die möglicherweise damit verbundene Vernichtung dieses Objekts vom Typ ‚Fisch' kann durch eine Methode `tryToEat()` erfolgen, die im Wesentlichen aus den beiden folgenden Programmzeilen besteht:

```
Actor actor = gameGrid.getOneActorAt(getLocation(),
Fisch.class);
if (actor != null) actor.removeSelf();
```

Falls sich also an der aktuellen Position ein Fisch befindet, wird dieses Objekt über die Variable `actor` referenziert und kann dann in der zweiten Zeile entsprechend gelöscht werden.

4.1.5 Automatisches Generieren von neuen Objekten

An dieser Stelle muss deutlich werden, dass der Ausgang der Simulation darin besteht, dass am Ende keine Fische mehr existieren. Durch den Rückgriff auf die Vorüberlegungen zur Simulation wird klar, dass diesem Problem durch die Geburt von neuen Fischen entgegengewirkt werden kann. Um eine Geburt zu implementieren, müssen sowohl eine Konstante für die Laichzeit als auch eine Variable für das Lebensalter berücksichtigt werden, wodurch insbesondere das Voranschreiten der Zeit bei der Simulation verdeutlicht wird. Die Geburt eines normalen Fisches kann besonders einfach umgesetzt werden, wenn für einen Fisch beim Verlassen des aktuellen Feldes geprüft wird, ob das Alter die Laichzeit bereits erreicht hat. In diesem Fall kann auf dem aktuellen Feld ein neues Objekt vom Typ ‚Fisch' erzeugt werden:

```
if (this.alter >= WasserWelt.LaichZeitFisch)
{
   gameGrid.addActor(new Fisch(),
   new Location(this.getX(), this.getY()));
   this.alter =0;
}
```

Nachdem auf dem aktuellen Feld ein Nachkomme erzeugt wurde, beginnt für beide Objekte das Alter bei null, um den Zeitraum der Laichzeit wieder abzuwarten.

4.1.6 Simulation

Die Simulation ist durch das beschriebene Vorgehen noch nicht vollständig implementiert, denn Haie sollen sich auch vermehren oder sterben können. Dennoch reicht es zunächst aus, um die Situation anhand einer nicht ganz vollständigen Simulation schon einmal zu analysieren. Startet die Simulation auf einer Oberfläche mit 10 x 10 = 100 Zellen, nur wenigen Fischen und einer deutlich größeren Anzahl an Haien, so sterben die Fische mit einer hohen Wahrscheinlichkeit aus, da sie sich nicht vermehren können, bevor sie von einem Hai entdeckt werden. Dies verhält sich jedoch anders, wenn die Oberfläche vergrößert oder die Laichzeit verkürzt wird. Entscheidend ist hier, dass die Schülerinnen und Schüler verstehen, dass eine Simulation von der geeigneten Wahl der Parameter abhängt. Das Erkunden geeigneter Parameter kann somit als ein Spiel verstanden werden, wobei es darauf ankommt, dass die Population der Haie und Fische möglichst stabil gehalten wird. Dabei gibt es verschiedene Einstellungsmöglichkeiten. Eine hier mögliche Wahl für geeignete Startparameter ist z. B. die Kombination aus zwölf Fischen, drei Haien und einer Laichzeit von 18 bei einer 10 x 10 Oberfläche. Oft kann man während einer Simulation auch beobachten, dass es zu periodischen Schwankungen der Population der Fische kommt, womit sich der Kreis zu dem anfangs vorgestellten Lotka-Volterra-Modell schließt. Dadurch wird den Schülerinnen und Schülern an dieser Stelle deutlich, dass ein Computerprogramm zu demselben Resultat gelangt wie eine mathematische Analyse.

4.2 Generierung einer Android-App aus einer JGameGrid-Anwendung

In dem zuvor dargestellten Ansatz wird die Entwicklung einer Anwendung und anschließend einer mobilen App basierend auf der Software Greenfoot vorgenommen. Im Folgenden soll ein alternativer Ansatz beschrieben werden, der in die Programmierung unter Java einführt. Dabei wird Wator zunächst als Desktop-Anwendung erstellt und in einem zweiten Schritt auf mobilen Geräten mit dem Android-Betriebssystem lauffähig gemacht.

Voraussetzungen sind grundlegende Kenntnisse in:

- → Programmierung in C++
- → Objektorientierter Programmierung
- → Umgang mit Entwicklungsumgebungen, idealerweise Netbeans[12]

Der Kurs gliedert sich in zwei Lerneinheiten mit je vier Unterrichtsstunden:

- → In der ersten Einheit wird in den Umgang mit der Programmiersprache Java unter dem Netbeans-IDE anhand des Wator-Beispiels eingeführt. Für die Implementierung wird die Bibliothek ‚JGameGrid' verwendet.
- → Im zweiten Teil wird die Desktop-Anwendung durch Einsatz eines Generators in eine lauffähige App konvertiert. Das resultierende Projekt verwendet die Bibliothek ‚JDroidLib'[9] und kann in die ‚AIDE' App auf einem mobilen Gerät geladen und dort editiert werden.

Beide Teile sollten den Schülerinnen und Schülern in Form einer Arbeitsmappe zur Verfügung gestellt werden und sind in Schritte unterteilt, die einfach nachvollzogen werden können. Sämtliches Material hierzu steht zum Download als ‚Arbeitsmaterial II' bereit.[1]

5 | Fazit und Ausblick

Dieses Projekt zeigt, dass mithilfe der objektorientierten Programmierung die Simulation biologischer Populationsdynamiken im Informatikunterricht an Schulen möglich ist. Dazu wurden insgesamt drei Varianten einer möglichen Implementation näher beschrieben: eine schrittweise Umsetzung innerhalb der Lernumgebung Greenfoot, ein Vergleich mit dem Modell von Lotka-Volterra und die Erstellung der Wator-Simulation als mobile Applikation.

Durch entsprechende Materialien, die Arbeitsaufträge bzw. Anleitungen zu einer konkreten Umsetzung enthalten, gepaart mit gezielten Informationen durch die Lehrkraft, können Schülerinnen und Schüler objektorientiertes Denken sinnvoll vertiefen. Darüber hinaus zeigen Zwischenergebnisse bei dem Aufbau der vollständigen Simulation eine immer detailliertere Annäherung an das bekannte biologische Verhalten.

Die beschriebene Simulation beinhaltet überwiegend grundlegende Konzepte der objektorientierten Programmierung. Zudem ist der Grad der Komplexität so gewählt, dass die Implementierung für Schülerinnen und Schüler durchaus überschaubar bleibt. In vertiefenden Projektkursen oder Seminararbeiten können gegebenenfalls anspruchsvollere Simulationen implementiert werden, wie beispielsweise eine kollektive Intelligenz (Schwarmverhalten) oder ein Ameisenalgorithmus.

Quellen

[1] Alle Zusatzmaterialien finden Sie unter www.science-on-stage.de/teachers-scientists_materialien

[2] Greenfoot: www.greenfoot.org (abgerufen am 12.01.2017)

[3] Bernhard Lahres, Gregor Rayman, Stefan Strich, Objektorientierte Programmierung, Rheinwerk Computing (2015)

[4] Christoph Ableitinger, Biomathematische Modelle im Unterricht – Fachwissenschaftliche und didaktische Grundlagen mit Unterrichtsmaterialien, S. 143, Vieweg und Teubner (2010)

[5] A.K. Dewdney, Computer Recreations, Scientific American 251, S. 14-22 (1984)

[6] www.beltoforion.de (abgerufen am 12.01.2017)

[7] www.ralphhenne.de/informatik/greenfoot/05HaseFuchs.pdf (abgerufen am 12.01.2017)

[8] Michael Kölling, Einführung in Java mit Greenfoot, Pearson Studium (2010)

[9] JDroid: http://www.jdroid.ch (abgerufen am 12.01.2017)

[10] AIDE: http://www.android-ide.com

[11] Klassenbibliothek JGameGrid: http://www.java-online.ch/gamegrid/uebersichtMethoden.inc.php (abgerufen am 12.01.2017)

[12] Netbeans IDE: https://netbeans.org/ (abgerufen am 12.01. 2017)

Kontakt

Bei Fragen zur Simulation in Greenfoot:

- → **Dr. Andreas Degenhard**
  adegenha@t-online.de

Bei Fragen zur Überführung in eine Android-App:

- → **Prof. Dr. Heinz-Josef Eikerling**
  H.Eikerling@hs-osnabrueck.de
- → **Prof. Dr. Michael Uelschen**
  m.uelschen@hs-osnabrueck.de

Arbeitsblatt 1

Aufgabe 1: In einem ersten Schritt wird die bereits vorgefertigte Simulationsumgebung der Wator-Wasserwelt durch das Ausführen der Datei *project* im Ordner *AB1-Schritt-1* erstellt und als Teil der Greenfoot-Oberfläche dargestellt. (Abhängig von der verwendeten Greenfoot-Version ist gegebenenfalls noch das Klicken einer *Compile-Schaltfläche* im unteren Teil der Greenfoot-Oberfläche erforderlich.)

In der Objektorientierten Programmierung (OOP) gibt es in der Regel mehrere Programme, die zusammengehören und gemeinsam ein Projekt bilden. Die einzelnen Programme werden in der OOP als Klassen bezeichnet. In Greenfoot müssen immer mindestens zwei Klassen vorhanden sein, die Klasse *World* und die Klasse *Actor* . Zusätzlich zu diesen beiden Basisklassen können weiteren Klassen mit in das Projekt integriert werden, die dann an der rechten Seite der Greenfoot-Oberfläche zusammen aufgelistet sind. Öffne die Klassen *Ocean*, *Animal* und *Fisch* im Greenfoot-Editor (über Klicken der rechten Maustaste, wenn entsprechende Klasse angewählt ist) und mache Dich mit den Methoden dieser Klassen vertraut. **(AB1-Schritt-1)**

Aufgabe 2: Bei Klassen unterscheidet man zwischen Ober- und Unterklassen. Dabei erbt die Unterklasse alle Attribute und Methoden der Oberklasse und kann darüber hinaus um neue Attribute und Methoden ergänzt werden. Graphisch wird dies durch *Vererbungspfeile* von der Unterklasse zur Oberklasse angezeigt, siehe auch rechte Seite der Greenfoot-Oberfläche. Beispielsweise erbt die Klasse *Fisch* von der Klasse *Animal*, da Fische spezielle Tiere sind. Klassen sind wie Schablonen, aus denen dann Objekte erzeugt (instanziiert) werden können. Erzeuge einzelne Objekte der Klasse *Fisch* und setze diese in die Ozean-Welt (über rechte Maustaste, wenn die entsprechende Klasse angewählt ist). Dabei können die Objekte über die Schaltflächen *Act* und *Run* bewegt werden. **(AB1-Schritt-1)**

Aufgabe 3: Bei der Bewegung der Fische fällt auf, dass die Bewegung nur in eine Richtung stattfindet. Da sich alle Bewohner der Ozean-Welt ähnlich bewegen sollen, gibt es bereits in der Oberklasse *Animal* die Methode *moveRandomly().* Ändere diese Methode so ab, dass die Bewegung

1) sowohl nach links als auch nach rechts zufällig stattfindet. Dazu sollen die nachfolgenden Programmzeilen verwendet werden, deren Bedeutung zuvor von Dir geklärt werden soll:

```
int zufall = Greenfoot.getRandomNumber(2);

if(getX()<(getWorld().getWidth())-1)

setLocation((getX()+1),getY());
```

(AB1-Schritt-2)

2) in alle vier Richtungen zufällig stattfindet. **(AB1-Schritt-3)**

3) über den Rand der dargestellten Simulationsumgebung hinaus erfolgen kann, indem die Fische an der gegenüberliegenden Seite wieder erscheinen, um dadurch den Ozean als unbegrenzt zu simulieren. Mache Dich hierzu zunächst mit dem Modulo-Operator '%' vertraut und implementiere dann den unendlichen Ozean (*toroidale Randbedingungen*). **(AB1-Schritt-4)**

Aufgabe 4: Erstelle eine Methode 'populate()', welche am Anfang den Ozean mit einer zuvor festgelegten Anzahl von Objekten vom Typ Fisch initialisiert. Hierzu sollen die beiden Methoden 'addObject()' und 'getObjectsAt()' benutzt werden. Dabei soll die zuletzt genannte Methode dazu verwendet werden, eine mehrfache Besetzung einer Zelle zu vermeiden. **(AB1-Schritt-5)**

Arbeitsblatt 2

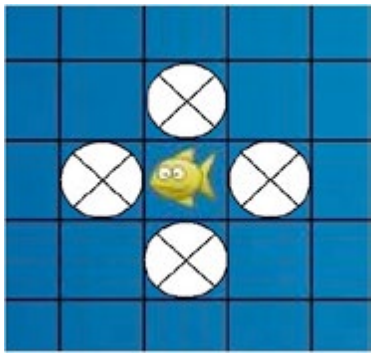
Um die Population der Fische realistischer zu simulieren, ist es erforderlich, dass sich Fische auch vermehren können. In der Natur sind Lebewesen zumeist ab einem bestimmten Alter zeugungsfähig.

Aufgabe 1: Die Klasse *Animal* soll um ein Attribut *age* (Alter) ergänzt werden. In jedem Simulationsschritt soll dann das Alter jedes bereits instanziierten Objekts vom Typ Fisch um Eins erhöht werden, wobei ein Simulationsschritt dem Klicken der Schaltfläche *Act* entspricht. Wird ein neues Objekt vom Typ *Fisch* erzeugt, erhält dieses zunächst das Alter Null. Nach jedem Simulationsschritt kann man sich dann über die I*nspect*-Möglichkeit (über Klicken der rechten Maustaste, wenn das entsprechende Objekt angewählt ist) das Alter jedes Objektes anzeigen lassen.

Aufgabe 2: Erreicht ein Fisch ein bestimmtes Alter soll ein neuer Fisch geboren werden. Um die Regeln einfach zu halten, soll der gebärende Fisch in diesem Alter auch sterben, bzw. dieses Objekt soll wieder das Alter Null erhalten, sodass ein zeugungsfähiger Fisch durch zwei Fische mit dem Alter Null ersetzt wird. Dabei soll ein zusätzliches Attribut *FBREED* in der Klasse *Fisch* implementiert werden, welches die Zeit angibt, nach der ein Fisch Nachkommen erzeugen kann.

Aufgabe 3: Die Fische erzeugen alle im gleichen Simulationsschritt ihre Nachkommen, da alle Objekte vom Typ *Fisch* in der Methode *populate()* mit dem Alter Null instanziiert wurden. Um eine realistische Durchmischung des Lebensalters innerhalb einer Simulation zu erhalten sollen die Fische der Anfangspopulation ein unterschiedliches Alter haben können. Dazu sollen die Objekte vom Typ *Fisch* in der Methode *populate()* ein zufälliges Alter erhalten, wobei das Klassenattribut 'FBREED' der Klasse *Fisch*, welches in der vorausgegangenen Aufgabe eingeführt wurde, eine Obergrenze für das mögliche Alter angibt.

Aufgabe 4: Um zu vermeiden, dass sich mehrere Fische auf demselben Simulationsplatz (kurz Zelle) befinden, sollen die nachfolgenden Regeln (entsprechend den Vorgaben für das Wator-Modell) schrittweise umgesetzt werden:



a) Ein Fisch soll sich nur dann bewegen können, wenn es mindestens eine freie Zelle in seiner direkten Nachbarschaft (weiß markierte Zellen in der nebenstehenden Abbildung) gibt, auf die er schwimmen kann.

b) Soll ein neuer Fisch geboren werden, muss mindestens eine freie Zelle (Zelle ohne Fisch) in seiner direkten Nachbarschaft vorhanden sein, auf der ein neues Objekt vom Typ *Fisch* erzeugt werden kann.

Eine besonders elegante Umsetzung der hiermit verbundenen Überprüfung freier Zellen ist das *Iterator*-Konzept. Dazu wurden die beiden Methoden *getAdjacentLocations(int x, int y)* und *getAdjacentFreeLocation(int x, int y)* in der Klasse *Animal* implementiert, da eine Überprüfung von Zellen in direkter Nachbarschaft möglicherweise auch für weitere Meeresbewohner sinnvoll erscheint. Während die erste Methode alle vier Zellen in der direkten Nachbarschaft in einer Liste abspeichert, untersucht die zweite Methode, ob es unter diesen Zellen eine freie Zelle gibt. Für die Implementierung der Methoden sind folgende Import-Dateien erforderlich:

```
import java.util.Collections;   import java.util.Iterator;   import java.util.LinkedList;   import java.util.List;
```

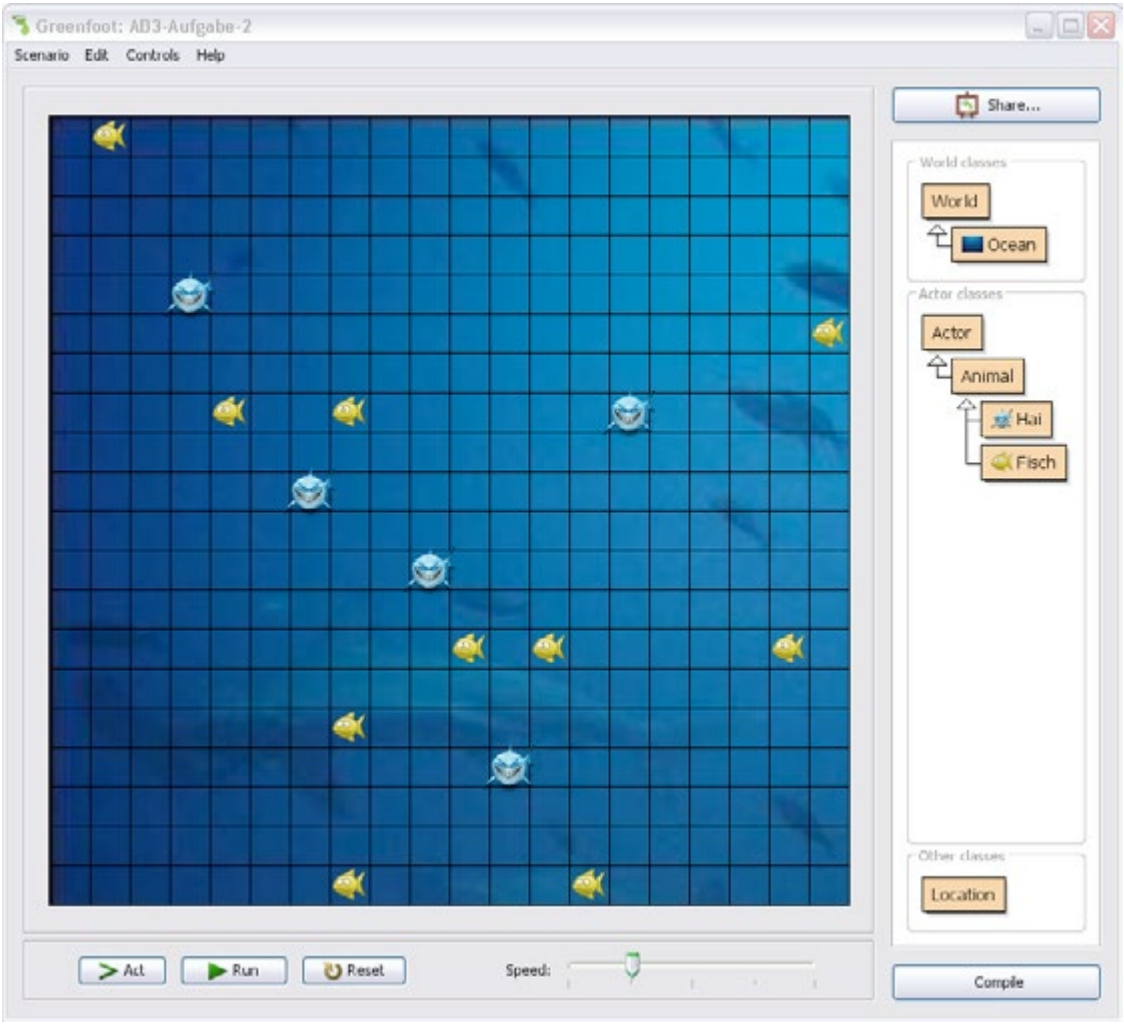
Deine Aufgabe ist es, die Funktionsweise der beiden Methoden *getAdjacentLocations(int x, int y)* und *getAdjacentFreeLocation(int x, int y) unter Benutzung des Iterator*-Konzepts zum Durchsuchen einer Listenstruktur von Zellen (*Locations*) nachzuvollziehen, wobei Literatur hinzugezogen werden kann.

Arbeitsblatt 3

Die bislang simulierte Entwicklung der Population der Fische ist in der Natur nicht realistisch, denn die Fische vermehren sich stetig. Zwar wird in dem Wator-Modell angenommen, dass ein 'unendlicher' Ozean genug Nahrung (z. B. in Form von Plankton) bereitstellt, die Population der Fische soll jedoch durch natürliche Feinde dezimiert werden können.

Aufgabe 1: Recherchiere die Begriffe 'Räuber-Beute-Beziehung' und 'Lotka-Volterra-Modell'. Notiere wesentliche Grundregeln und dokumentiere (beispielsweise grafisch), welches Verhalten sich im Allgemeinen zwischen der Population von Beutetieren und der Population der Räuber ausbilden wird.

Aufgabe 2: Implementiere eine Klasse *Hai*, deren Objekte zunächst dieselben Eigenschaften besitzen sollen wie die bisher betrachteten Objekte der Klasse *Fisch*. Dies wird besonders einfach dadurch umgesetzt, dass die Klasse *Hai* genau wie zuvor die Klasse *Fisch* grundlegende Eigenschaften von der Klasse *Animal* erben, wie dies in der nachfolgenden Abbildung dargestellt ist.



Zudem soll die Klasse *Hai* den Aufbau der Klasse *Fisch* übernehmen, wobei jedoch die Haie derzeit noch keine Nachkommen erzeugen sollen (siehe Arbeitsblatt 4). Argumentiere zunächst, welcher Zustand sich nach mehreren Simulationsschritten einstellen wird und überprüfe dies mit der Simulation.

Arbeitsblatt 4

Die bislang betrachtete friedliche Koexistenz der Fische und Haie in der Simulation widerspricht dem biologischen Verhalten in den Weltmeeren. Im Wator-Modell haben daher die Haie die Möglichkeit, die normalen (Beute-)Fische zu fressen.

Aufgabe 1: Zur Modellierung des Fressverhaltens sollen die folgenden Regeln implementiert werden:

1) Ein Hai kann einen Fisch fressen, der sich auf einer Zelle in der direkten Nachbarschaft (siehe Darstellung auf Arbeitsblatt 2) von dem Hai befindet.
2) Findet ein Hai keinen (Beute)-Fisch auf einer Zelle in der direkten Nachbarschaft zu seiner eigenen Zelle, so schwimmt er zufällig auf eines der freien, angrenzenden Felder, falls ein solches vorhanden ist.

Zur Implementierung der oben angegebenen Regeln soll die Methode *act()* in der Klasse *Hai* entsprechend angepasst werden. Hierzu soll die Klasse Animal zunächst um die Methode *getAdjacentFishLocation(int x, int y)* erweitert werden. Diese besitzt einen nahezu identischen Aufbau wie die Methode *getAdjacentFreeLocation(int x, int y),* die auf Arbeitsblatt 2 eingeführt wurde, mit dem Unterschied, dass dieses Mal ermittelt werden soll, ob sich ein Fisch auf einer direkten Nachbarzelle befindet. Die Methode *act()* in der Klasse *Hai* überprüft dann ob es eine Zelle mit einem Fisch in der direkten Nachbarschaft von einem Hai gibt. In diesem Fall wird der Hai dann auf eine solche Zelle schwimmen und den Fisch fressen. Andernfalls sucht der Hai nach einer freien Zelle auf die er schwimmen kann. Wenn ein Fisch gefressen wurde, soll dieser als Objekt vom Typ Fisch gelöscht werden. Greenfoot stellt hierfür den Befehl *removeObject()* zur Verfügung.

Erläutere, warum es nach den oben beschriebenen Veränderungen der Klassen *Animal* und *Hai* bei der Simulation zu einer Zu- und Abnahme bei der Population der Fische kommen kann.

Aufgabe 2: Bislang bleibt die Population der Haie konstant, d. h. diese können sich nicht vermehren, aber sie sterben auch nicht. Im Wator-Modell verhalten sich die Haie aber genau wie andere Lebewesen, indem sie geboren werden und auch sterben. Im Gegensatz zu den Fischen haben die Haie im Wator-Mdell jedoch keine natürlichen Feinde. Eine mögliche Dezimierung ihrer Population erfolgt dann entsprechend dadurch, dass die Haie nicht mehr genug Fische finden, um sich ernähren zu können.

Entsprechend dem Wator-Modell sollen daher die folgenden Regeln implementiert werden:

- Findet ein Hai innerhalb einer bestimmten Anzahl von Simulationsschritten, der “Shark Starve Time” (Zeitspanne, die ein Hai ohne Nahrung überleben kann), keinen Fisch, so stirbt er.
- Haie pflanzen sich genauso fort wie Fische, d. h. nach einer “Shark Breed Time” wird ein neuer Hai auf einem Nachbarfeld geboren.

Um diese Regeln in der Simulation umzusetzen, soll in der Klasse *Hai* eine Methode *breed()* ergänzt werden, die bereits in der Klasse *Fisch* vorhanden ist, wobei jedoch der Name des benötigten Klassenattributs für das gebärfähige Alter auf *Sbreed* geändert werden soll. Darüber hinaus soll in der Methode *act()* in der Klasse *Hai* die *Shark Starve Time* berücksichtigt werden, nach der ein Objekt vom Typ *Hai* möglicherweise gelöscht wird.

Nach der Implementation aller oben beschriebenen Regeln können die Zu- und Abnahme der Fisch- und der Hai-Population in der Simulation beobachtet werden. Argumentiere warum sich im Verlauf der Simulation größere Gruppen (*Cluster*) von Fischen, Haien und auch leeren Zellen herausbilden.

Arbeitsblatt 5

Die Simulation im Greenfoot-Projekt zu diesem Arbeitsblatt enthält alle bislang implementierten Regeln und wurde um ein weiteres Ausgabefenster ergänzt. Dieses zeigt den zeitlichen Verlauf der Populationsgrößen während einer Simulation, wobei die Anzahl der Fische und Haie sowie der Simulationsschritt (*Step*) mit protokolliert werden.

Bei einer Simulation können drei mögliche Verlaufs-Szenarien beobachtet werden:

1) Die Fische sterben aus, anschließend sterben auch die Haie aus.
2) Die Haie sterben aus, die Fische bevölkern den Ozean vollständig.
3) Es stellt sich eine langfristig stabile Simulation mit Haien und Fischen ein.

Ob sich ein stabiler Simulationsverlauf einstellt hängt unter anderem von der Einstellung der verschiedenen, wählbaren Parameter ab. Hier ein Beispiel für eine mögliche Parameterwahl:

FBREED = 3, SBREED = 3, und STARVE = 3;

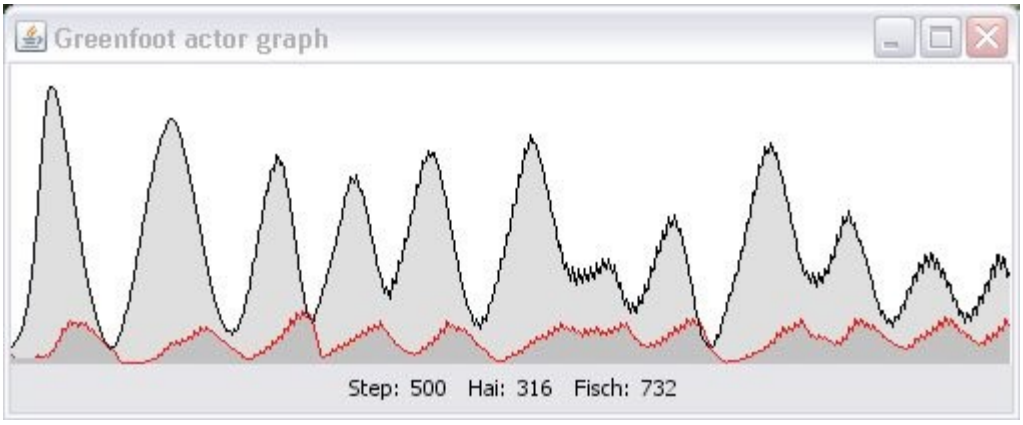
Auch die Anzahl der vorhandenen Zellen, mit der der (aufgrund der toroidalen Randbedingungen eigentlich unendliche) Ozean simuliert wird und die zu Beginn vorhandenen Anzahlen an Fischen und Haien haben einen Einfluss auf den Simulationsverlauf. Hier ein Beispiel für eine mögliche Wahl:

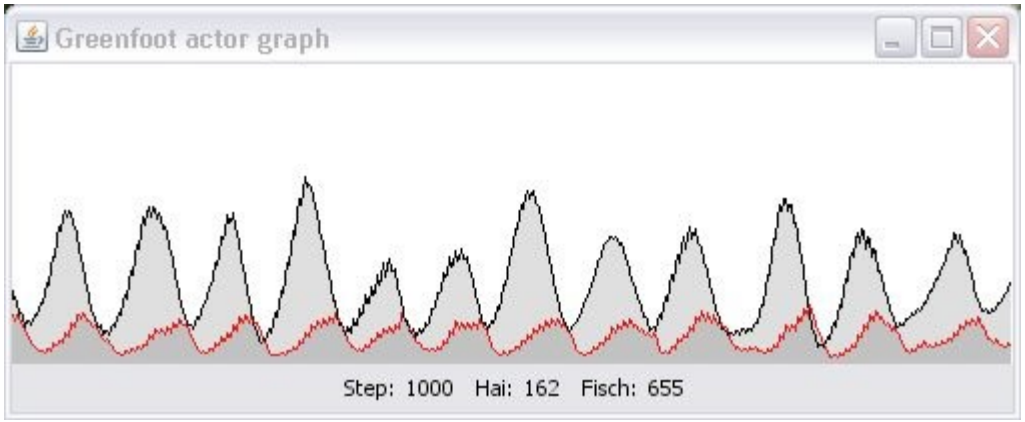
BREITE = 50, LAENGE = 50 und AnzahlFische = 60, AnzahlHaie = 20;

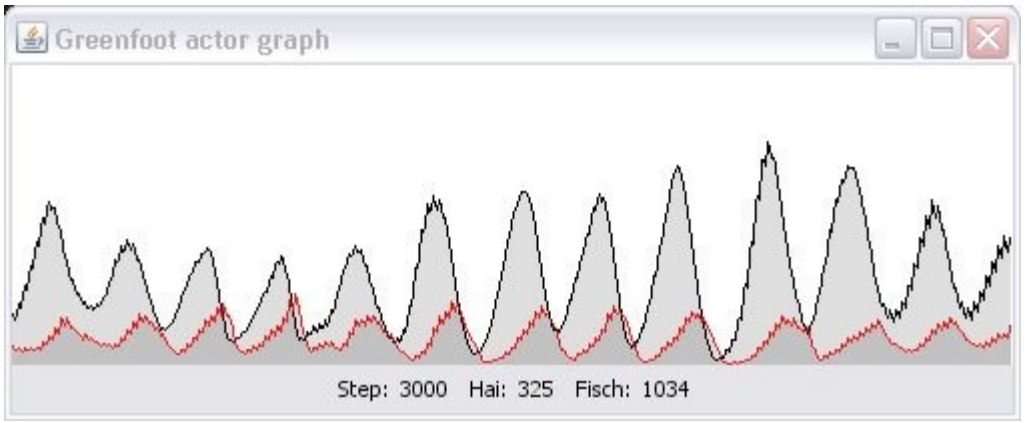
Mitentscheidend für einen stabilen Simulationsverlauf ist auch die Startkonfiguration, also die anfängliche Aufteilung der Fische und Haie, denn nicht jede Initialisierung mit Objekten vom Typ *Fisch* und *Hai* begünstigt einen stabilen Simulationsverlauf unter den oben angegebenen Parametern.

Aufgabe 1: Versuche im Zusammenspiel einer geeigneten Parameterwahl und einer möglicherweise bewusst und nicht zufällig gewählten Startkonfiguration einen stabilen Simulationsverlauf zu erzeugen.

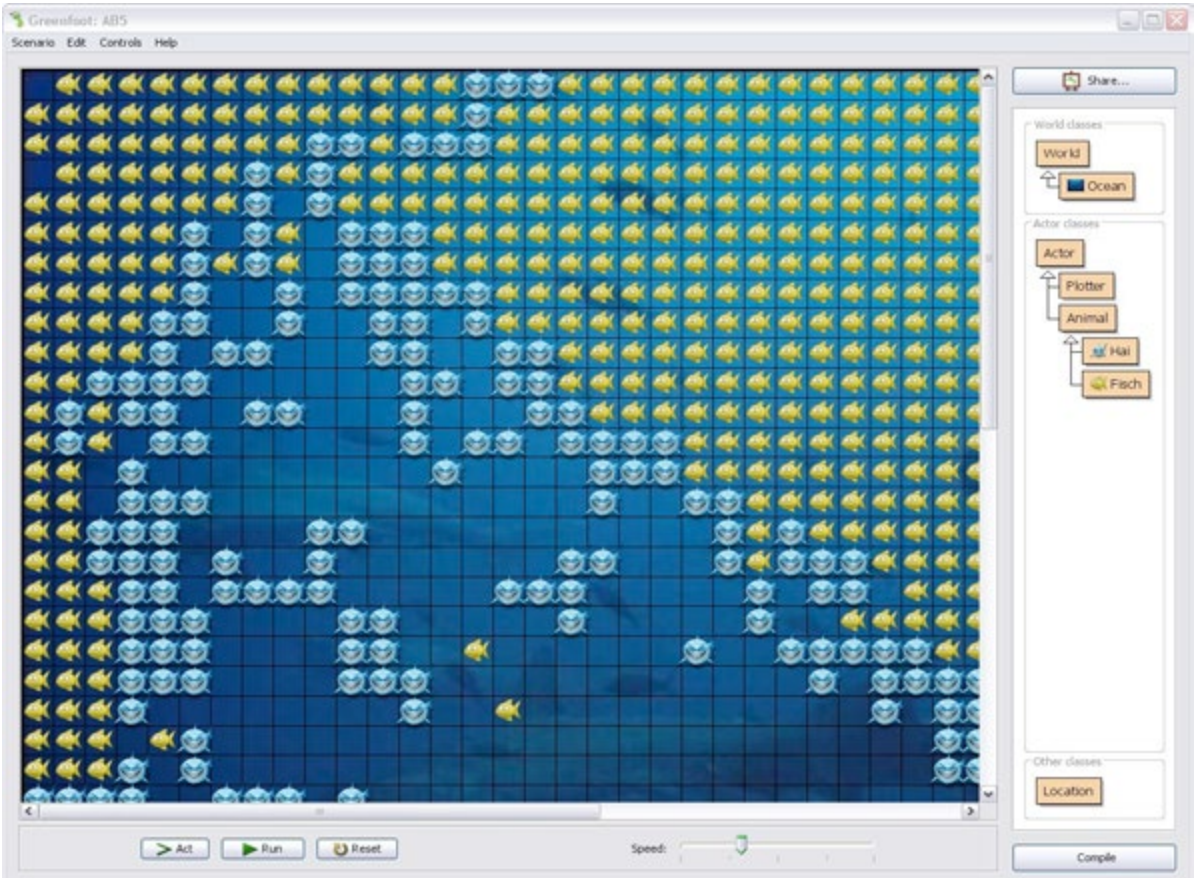
Aufgabe 2: Für die oben angegebenen Werte der Parameter für die Simulation sind nachfolgend Ausschnitte der zeitlichen Verläufe bis zu 500, 1000 und 3000 Simulationsschritten (Steps) dokumentiert. Argumentiere, welche Kurvenverläufe den (Beute-)'Fischen und den Haien zuzuordnen sind. Interpretiere die hier dargestellten oder auch eigene Simulationsverläufe im Hinblick auf die auf Arbeitsblatt 3 recherchierten Grundlagen zu den Begriffen 'Räuber-Beute-Beziehung' und 'Lotka-Volterra-Modell'.







Nachfolgend ist eine typische Konfiguration (mit *Cluster*) während einer stabilen Simulation dargestellt:



Arbeitsmaterial II

Das Material gliedert sich in drei Unterrichtseinheiten (jeweils eine Doppelstunde) und wendet sich an Schülerinnen und Schüler, die bereits in die objektorientierte Programmierung unter C++ eingeführt wurden. Der Schwerpunkt liegt dabei weniger auf der Vermittlung der mathematischen oder biologischen Hintergründe (Populationsdynamik) der WATOR-Anwendung, sondern auf der Erlernung der Werkzeuge, mit denen eine solche Anwendung auf verschiedenen Plattformen erstellt werden kann.

Die im Text genannten Materialien sollten über einen Ressourcen-Ordner zentral bereitgestellt werden.

Unterrichtseinheit 1: Java-Einführung

Java für C++ Programmierer

Den Schülerinnen und Schülern wird ein Dokument[1] zur Verfügung gestellt, welches C++ Programmierer in die wesentlichen Aspekte der Java-Programmierung einführt. Dieses arbeiten sie zunächst durch.

Netbeans IDE

Im Prinzip kann die Entwicklung mit verschiedenen Entwicklungsumgebungen vorgenommen werden. Es wird empfohlen, die Entwicklungen mit Netbeans[2] vorzunehmen. Alternativ kann auch ein anderes Werkzeug (Greenfoot, Eclipse oder IntelliJ) verwendet werden.

Die folgende Grafik gibt eine Übersicht über die wesentlichen Bedienelemente der Netbeans-Benutzeroberfläche.

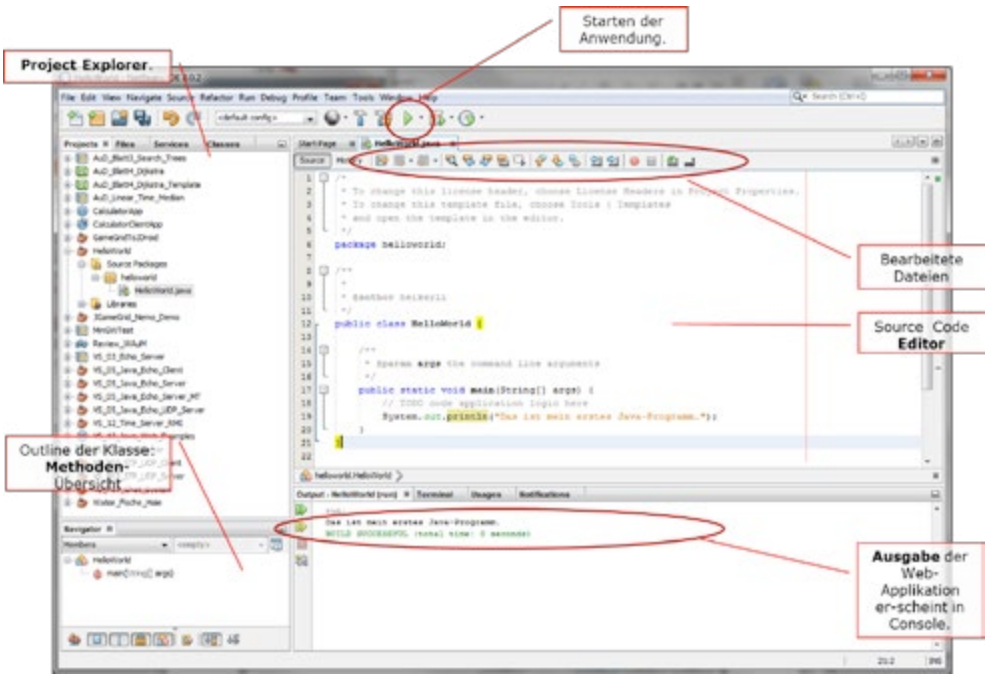


Abbildung 1: Oberfläche von Netbeans

[1] Johannes Mayer: Kurzeinführung in Java für C++-Programmierer, Uni Ulm.
[2] https://netbeans.org/

Um die wichtigsten Funktionen zu erklären, wollen wir eine sehr einfache Anwendung erstellen. Wir legen zunächst per Auswahl **File -> New Project…** in der Menüleiste ein neues Projekt an.

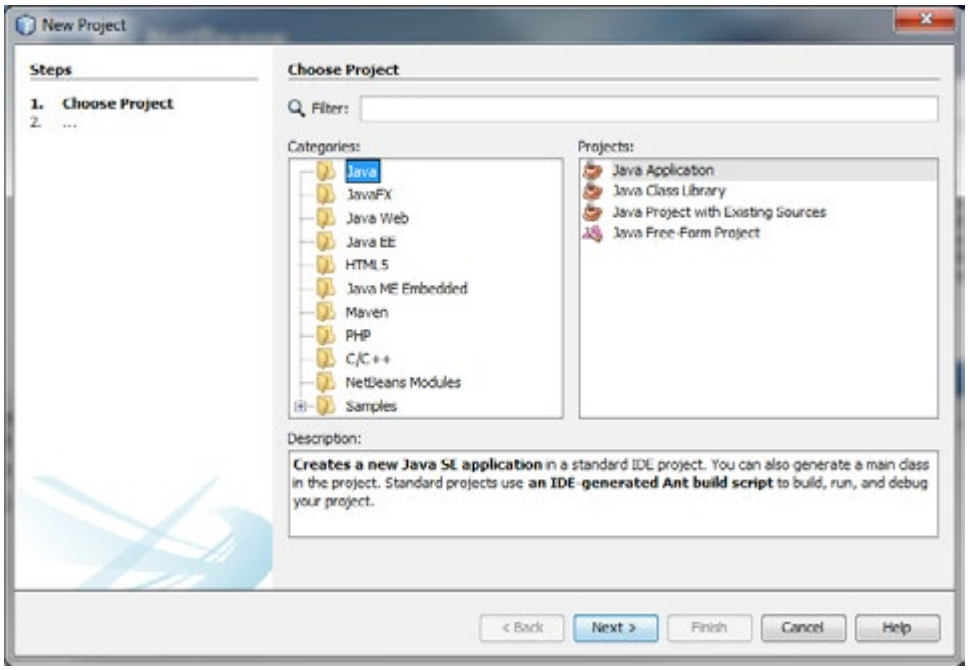


Abbildung 2: Wizard zur Projekterstellung für eine Java-Anwendung

Es erscheint ein sog. *Wizard*, in dem wir als Projektart **Java Application** auswählen. Wie man sieht, können noch viele andere Projektarten mit Netbeans realisiert werden.

Wir gehen weiter (**Next >**), um zum nächsten Dialog zu gelangen.

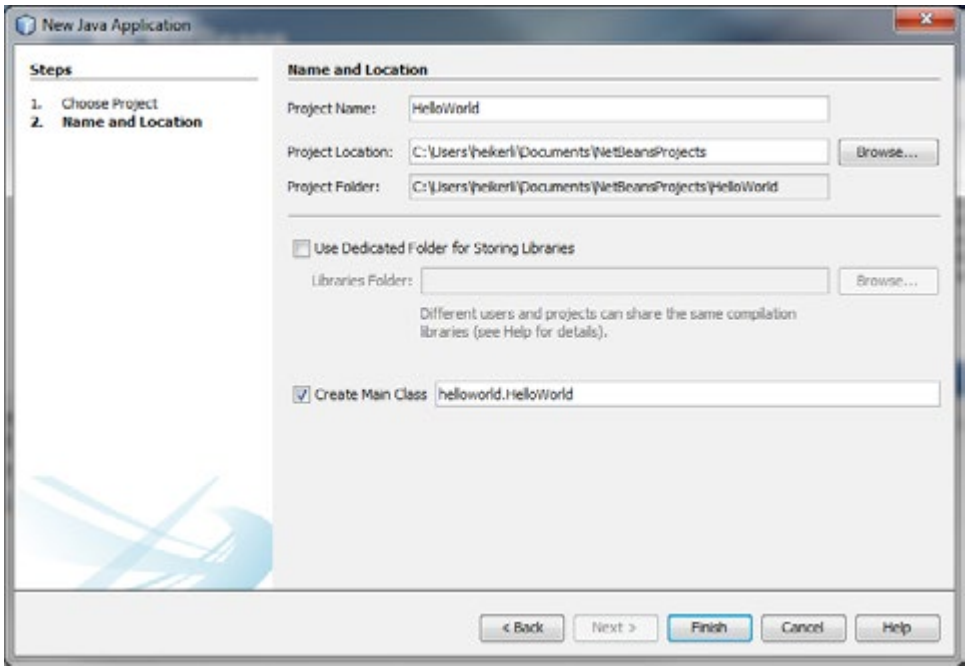


Abbildung 3: Eingabe Projektname und andere Daten zum Projekt

Unter **Project Name** geben wir **HelloWorld** ein und drücken den Button **Finish**. Anschließend wird das Projekt angelegt.

- Im linken oberen Bereich (im sog. **Project Explorer**) erscheinen die Projekte, u. a. das soeben angelegte Projekt. Wie bei einem Datei-Browser kann man sich hier die Struktur des Projektes ansehen.
- Der linke untere Bereich (**Navigator**) zeigt die Struktur der aktuell ausgewählten Datei an. Handelt es sich um eine Java-Datei, so werden hier die Klassen und Methoden der Datei angezeigt.
- Im rechten oberen Bereich wird das **Editor**-Fenster angezeigt. Hier kann der Source-Code bearbeitet werden. So kann z. B. der Code der **HelloWorld**-Klasse editiert werden.
- Im rechten unteren Bereich werden Ausgaben bei der Ausführung der Datei angezeigt.

Das Projekt ist zwar sofort lauffähig, wir ändern aber die **main()**-Methode der angelegten Klasse sofort wie folgt ab:

```
public static void main(String[] args) {
    // TODO code application logic here
    System.out.println("Das ist mein erstes Java-Programm.");
}
```

Wir können nun das Projekt starten. Dazu kann man entweder

- im Project Explorer die Klasse auswählen und dann über das Kontext-Menü (rechte Maustaste) **Run File** aufrufen.

  oder

- alternativ befindet sich in der Menü-Leiste der Run-Button (siehe Abbildung 1).

Nach dem Start der Anwendung erscheint im Ausgabefenster:

```
run:
Das ist mein erstes Java-Programm.
BUILD SUCCESSFUL (total time: 0 seconds)
```

Gratulation! Sie haben die erste Java-Anwendung erfolgreich erstellt. Aus Platzgründen können hier nicht alle Funktionen von Netbeans gezeigt werden. Es gibt aber zahlreiche Hilfen (leider zumeist in englischer Sprache) unter *www.netbeans.org*.

Unterrichtseinheit 2: Implementierung WATOR-Anwendung

Im Rahmen dieser Unterrichtseinheit soll eine kleine Animation als Java-Anwendung umgesetzt werden. Die Animation simuliert und visualisiert die Populationen von Räubern (Haie) und deren Beute (Fische) auf einem Gitter. Dazu sind in der Anwendung Regeln für die Aktoren (Haie, Fische) zu definieren, die bei Eintreten von Ereignissen auszuführen sind und damit weitere Ereignisse auslösen.

Hintergrund

Die hier verfolgte Simulation WATOR geht dabei auf *A. K. Dewdney* und *D. Wiseman* zurück und ist recht einfach.

Machen Sie sich zunächst mit den Grundgedanken der Simulation vertraut. Eine ausreichende Erklärung lässt sich bspw. im Internet finden[3].

[3] Siehe: https://de.wikipedia.org/wiki/Wator (Abruf am 21.5.2016)

Aufgabe 1: Projekt anlegen

Legen Sie zunächst ein neues Java-Projekt an, das Sie idealerweise **Wator_Fische_Haie** nennen. Eine Hauptklasse legen wir später an.

Für die eigentliche Visualisierung verwenden wir eine Bibliothek mit dem Namen **JGameGrid.jar**[4]. Sie finden diese im Ressourcen-Ordner, ebenso wie die anderen hier genannten Ressourcen für die Programmierung des Projektes. Laden Sie diese Dateien herunter und speichern Sie diese in einem lokalen Ordner.

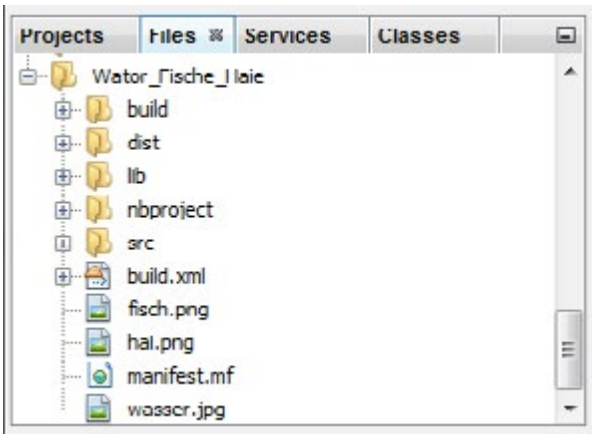


Abbildung 4: Struktur des Projektes Wator_Fische_Haie

Legen Sie im Project Explorer in dem Tab **Files** per Kontext-Menü zunächst einen Unterordner **lib** an. Per Drag-and-Drop können Sie das Archiv **JGameGrid.jar** in diesen Ordner verschieben. Wir fügen diese Datei nun als Bibliothek bei, indem wir im Project Explorer im Tab Project das Projekt auswählen und per Kontext-Menü (rechte Maustaste) **Properties** auswählen.

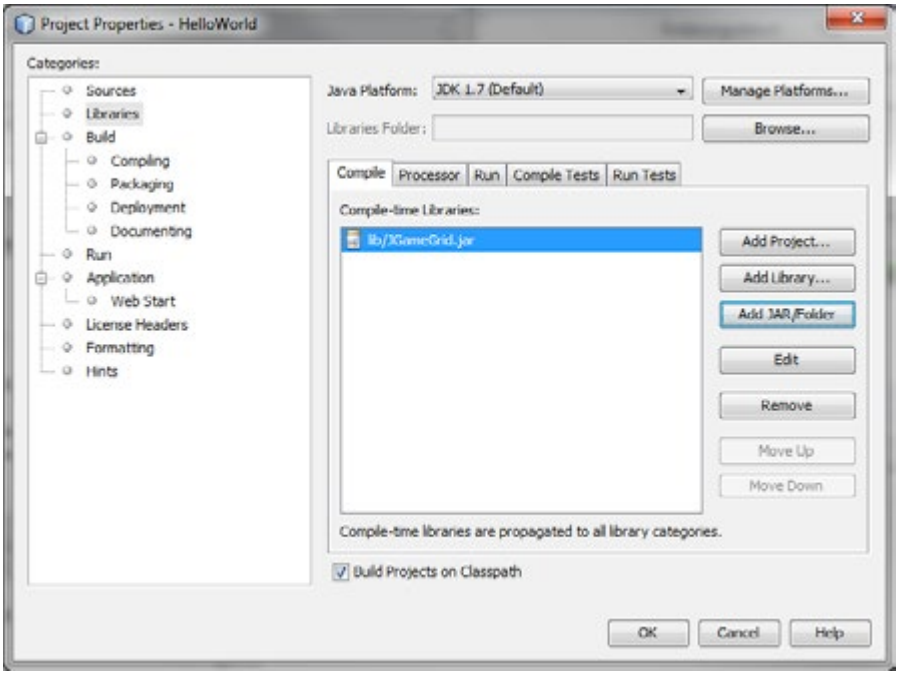


Abbildung 5: Konfiguration der Libraries des Projektes

[4] Siehe: http://www.aplu.ch/home/apluhomex.jsp?site=45 (Abruf am 21.5.2016)

Unter *Libraries* können wir nun per **Add JAR/Files** die Bibliothek in das Projekt einfügen. Die Bibliothek kann nun im Projekt genutzt werden (also **import ch.aplu.jgamegrid.*;** wird möglich).

Aufgabe 2: Hauptklasse und Klasse WasserWelt

Die Hauptklasse **Wator_Fische_Haie** soll eine Klasse **WasserWelt** verwenden. Wir erzeugen diese per Kontextmenü durch Anwahl der Hauptklasse und Auswahl von **New -> Java Class…**

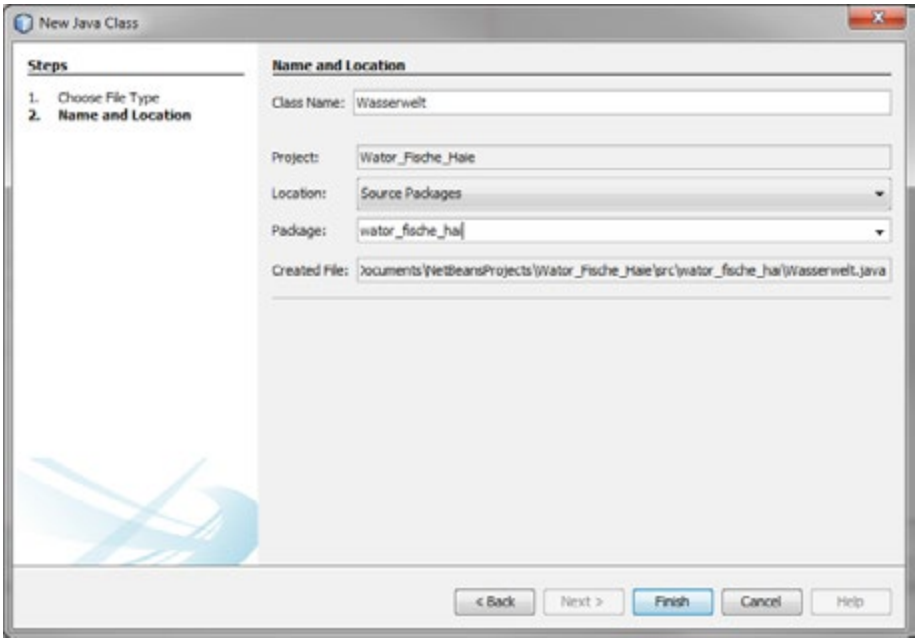


Abbildung 6: Klasse **WasserWelt** anlegen

Die Klasse soll im Paket **wator_haie_fische** angelegt werden.

Die Klasse wird nun in der **main()**-Methode der Hauptklasse aufgerufen:

```
public static void main(String[] args) {
   new WasserWelt().run();
}
```

Aufgabe 3: Anlegen der Klasse Hai und Fisch

Legen Sie nun analog zur **WasserWelt** die Klasse **Hai** an. Diese erbt von der Basisklasse **Actor**, die in der Bibliothek realisiert wird.

```
package wator_fische_haie;

import ch.aplu.jgamegrid.*;

public class Hai extends Actor
{
 public Hai()
 {
  super("hai.png");
 }

 public void act()
 { }
```

```
  private void tryToEat()
  { }
}
```

Die Methoden **act()** und **tryEat()** werden aber zunächst nicht ausimplementiert. Sie müssen die Image-Datei **hai.png** dazu in den Projektordner kopieren (siehe Abbildung 4).

Die Implementierung der Klasse **Fisch** erfolgt analog, wobei für diese die Methode **tryEat()** nicht vorgesehen ist.

Aufgabe 4: Klasse WasserWelt ausprogrammieren

Wir kümmern uns zunächst um die Klasse **WasserWelt**. In dieser Klasse wird die Umwelt der Haie und Fische als Gitter (Ableitung von **GameGrid**) angelegt. Das Gitter wird zunächst dimensioniert und im Konstruktor angelegt. In der **run()**-Methode werden dann Mengen von Haien und Fischen erzeugt und zufällig auf dem Gitter platziert.

```
package wator_fische_haie;

import ch.aplu.jgamegrid.*;

public class WasserWelt extends GameGrid
{
  // Parameter des Programms
  public static final int LeichZeitFisch = 18;

  // Dimensionierung des Wasserplaneten
  public static final int DIMX = 10;
  public static final int DIMY = 10;

  // Anzahl der Fische und Haie
  // (Dimensionierung des Wasserplaneten beachten!)
  public static final int AnzFische = 12;
  public static final int AnzHaie = 3;

  public WasserWelt()
  {
    // Gitter anlegen
    super(DIMX, DIMY, 50 /* cellZoom */, java.awt.Color.BLACK,
          "wasser.jpg", true);
  }

  public void run()
  {
    // Fische und Haie in der WasserWelt verteilen
    for (int i = 0; i < AnzHaie; i++)
      addActor(new Hai(), getRandomEmptyLocation());
    // Zuletzt werden die Actors der Fische hinzugefuegt,
    // da diese Actors dann zuerst ausgefuehrt werden.
    for (int i = 0; i < AnzFische; i++)
      addActor(new Fisch(), getRandomEmptyLocation());
    // doRun();
    show();
  }
}
```

Aufgabe 5: Implementierung der Methoden von Fisch und Hai

Die Methode **tryEat()** des Hais sei an dieser Stelle vorgegeben:

```
private void tryToEat()
{
  show(0);
  Actor actor = gameGrid.getOneActorAt(getLocation(), Fisch.class);
  if (actor != null)
  {
    actor.removeSelf();
    show(1);
  }
}
```

In der Methode wird überprüft, ob sich an der Position des Hais ein Fisch befindet. Falls dies der Fall ist, wird der Fisch entfernt (er wird gefressen). Die Methode **act()** implementiert die Bewegung des Aktors, hier also des Hais.

```
public void act()
{
  // Hai schwimmt zufaellig ein Feld weiter
  double r = Math.random();   // Wert fuer Zufallszahl
  if (r > 0.75)
    turn(90);
  else if (r > 0.5)
    turn(180);
  else if (r > 0.25)
    turn(270);

  // Hai darf nur auf ein freies Feld schwimmen,
  // sodass er ggf. auf ein anderes Feld schwimmt.
  // (Randfelder haben demnach immer ein freies Feld)
  int umlauf = 0;
  Actor actor = null;       // Referenz auf Hilfsobjekt
  do
  {
    actor =gameGrid.getOneActorAt(this.getNextMoveLocation(), Hai.class);
    turn(90);
    umlauf++;
  }
  while ((actor != null) && (umlauf < 4));

  // Wenn ein freies Feld vorhanden ist, schwimmt der Fisch weiter.
  if (umlauf < 4)
  {
    turn(270);
    move(1);

    // Torus beachten und Randkoordinaten anpassen
        // -> Aufgabe: wie wird verhindert, dass der Actor
        // das Grid links / rechts / unten / oben verlaesst?

    // Hai versucht zu fressen
    tryToEat();
  }
}
```

In der Methode fehlt die Behandlung der toroidalen Randbedingungen des Gitters. Ein Hai, der beispielsweise das Gitter nach einer Iteration rechts verlässt, sollte das Feld links wieder betreten. Dies ist an geeigneter Stelle zu ergänzen. Beachten Sie, dass die x- und y-Koordinaten jeweils die Werte von 0 … DIMX – 1 bzw. 0 … DIMY – 1 haben.

Die entsprechende Methode in der Klasse **Fisch** können Sie nun selbst implementieren. Allerdings werden die Fische nicht nur durch die Haie konsumiert, sondern sie können auch neu entstehen. Dazu wird die Member-Variable **LeichZeitFisch** in der Klasse **WasserWelt** verwendet. Überlegen Sie sich, wo das folgende Fragment am sinnvollsten platziert werden kann!

```
this.alter++;
if (this.alter == WasserWelt.LeichZeitFisch)
{
 gameGrid.addActor(new Fisch(), new Location(this.getX(), this.getY()));
 this.alter = 0;
}
```

Wie man sieht, wird der Regenerationszyklus über eine Variable **alter** (initial 0) gesteuert.

Aufgabe 6: Start der Anwendung

Die Anwendung ist nun fertig und kann gestartet werden. Mit *Step* kann ein Einzelschritt der Simulation durchgeführt werden, mit *Run* läuft die Simulation durch, wobei die Geschwindigkeit über den Schieberegler beeinflusst werden kann. Sie können die im Programm definierten Parameter variieren und schauen, was passiert.

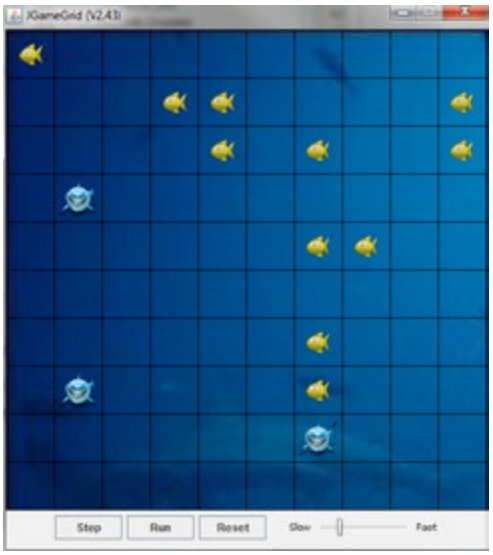


Abbildung 7: WATOR Desktop-Anwendung

Unterrichtseinheit 3: Konvertierung in eine Android-Anwendung

Wir konvertieren das Projekt nun in eine unter Android lauffähige App. Dazu steht ein Konverter (siehe **GameGridToJDroid.jar** im Ressourcen-Ordner) bereit, der selbst in Java implementiert ist.

Aufgabe 1: Generierung einer mobilen App

Der Konverter kann aufgerufen werden mit:

java –jar GameGridToJDroid.jar

Es erscheint das folgende User-Interface:

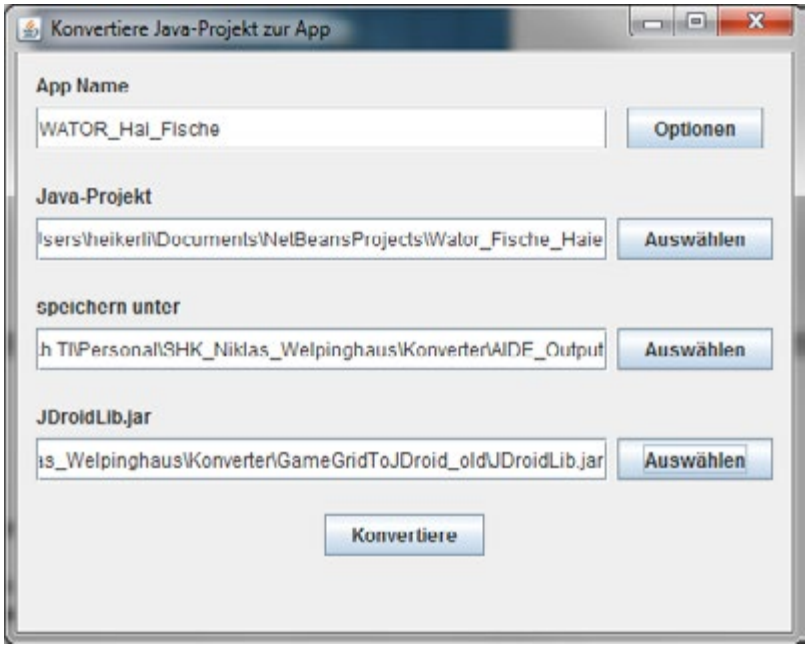


Abbildung 8: User-Interface des Konverters

Sie können hier

- den Namen der App,
- das zu konvertierende GameGrid-Projekt (aus 3.),
- das Zielverzeichnis zur Speicherung der App (kann temporär angelegt werden),
- die von der App verwendete **JDroidLib.jar** (zu finden im Ressourcen-Ordner; bitte lokal speichern)

konfigurieren.

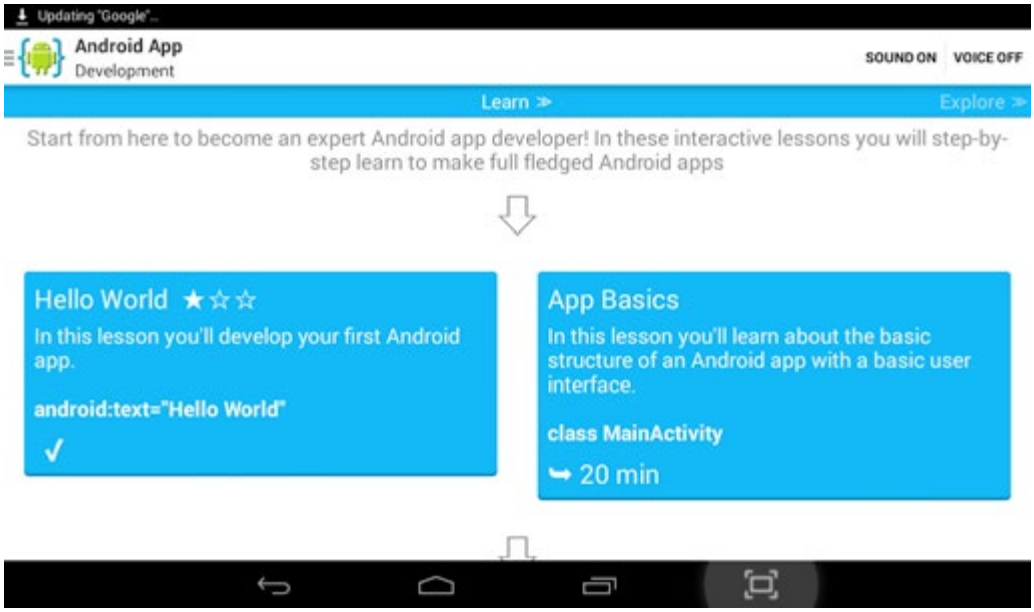
Im Ausgabe-Verzeichnis des Konverters wird neben einer Log-Datei ein Verzeichnis für das Android-Projekt angelegt. Dieses Projekt kann nun auf mobilen Geräten installiert werden, indem dort die **AIDE-Anwendung** aufgerufen und das Projekt importiert wird.

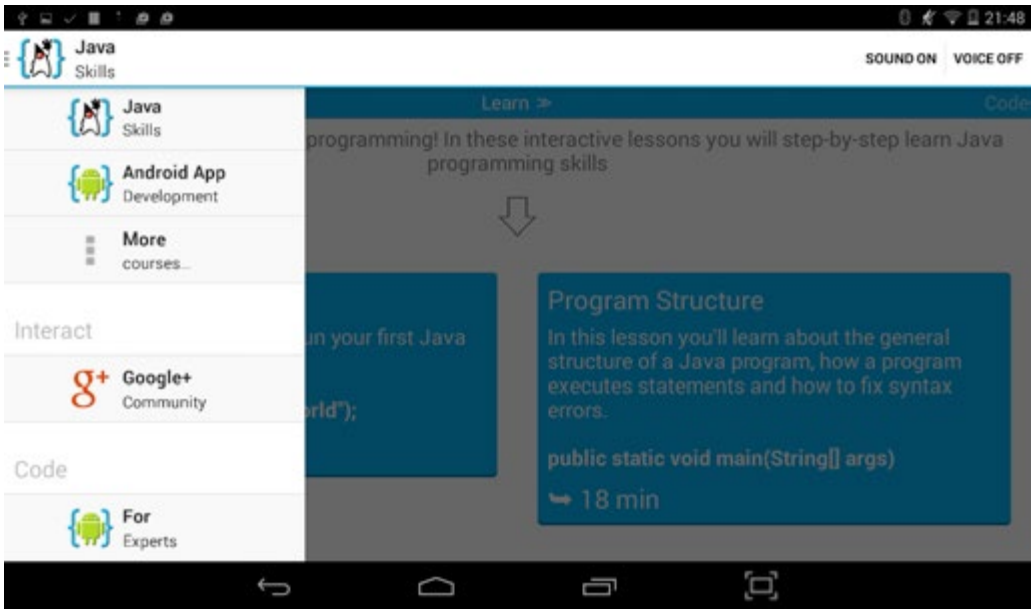
Aufgabe 2: Installation, Test und Modifikation der App

Es sollten hierzu einige Android-Tablets zur Verfügung stehen, die für das Deployment der generierten App verwendet werden können.

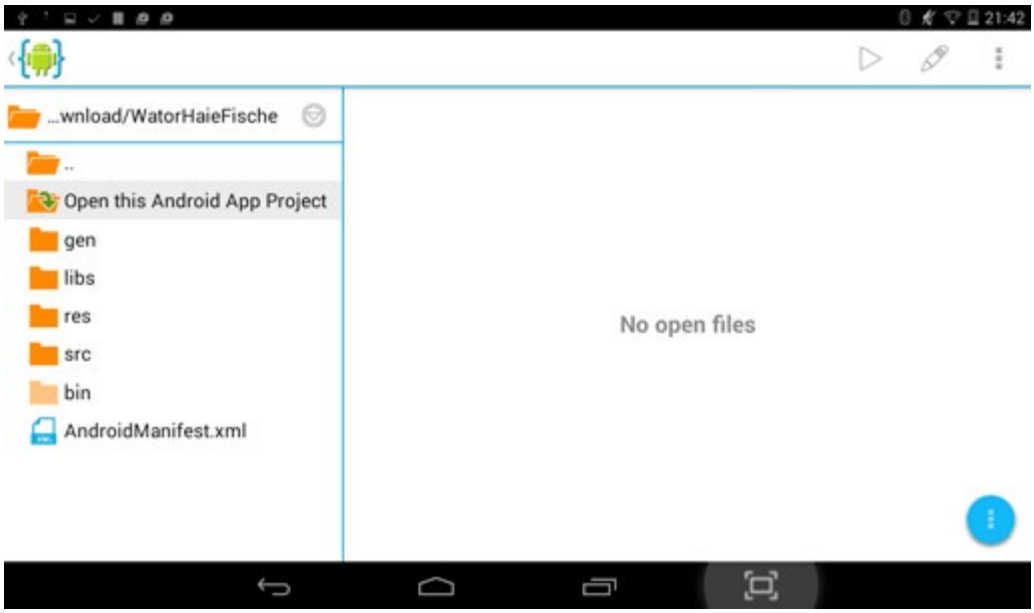
Das konkrete Vorgehen dazu ist wie folgt:

- Verbinden Sie das Tablet via USB-Kabel mit dem Entwicklungsrechner. Sie können nun (evtl. wird noch ein Geräte-Treiber installiert) über den Datei-Explorer auf das Dateisystem des Tablets zugreifen.
- Platzieren Sie das konvertierte Projekt im Ordner **Download** auf dem Tablet.
- Starten Sie dann auf dem Tablet die Applikation *AIDE*.
- AIDE zeigt nach dem Start die integrierte Lernumgebung. Wir wählen den Punkt **Explore >>**. Dann scrollen wir auf der linken Seite bis zum Punkt **For Experts** und wählen diesen aus.

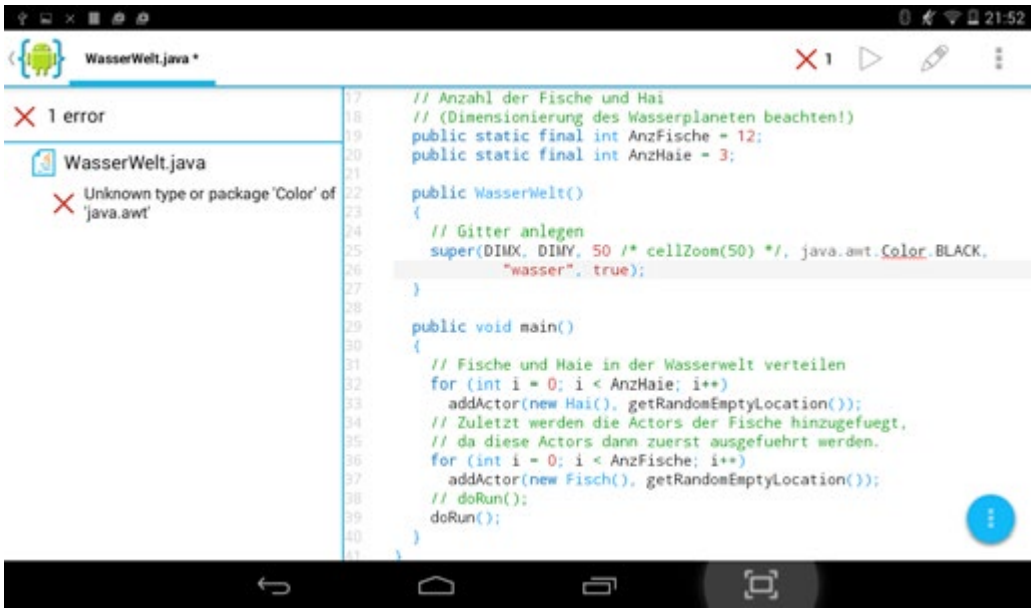




- Ein Browser erscheint und wir können das Projekt im Ordner Download in AIDE einladen. Wir wählen den Punkt **Open this Android App Project**.

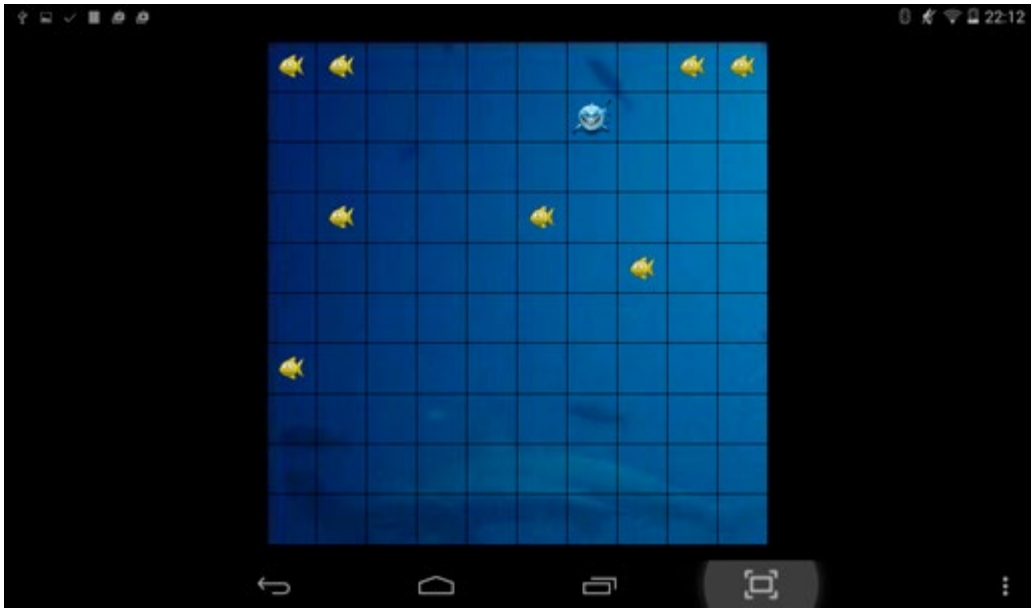


- Das Projekt wird geladen und kompiliert, was durchaus einige Zeit in Anspruch nehmen kann. Etwaige Fehler werden angezeigt und können im integrierten Editor korrigiert werden. Sie erkennen jetzt vielleicht, warum die eigentliche Programmierung – wenngleich möglich – nicht auf dem mobilen Endgerät vorgenommen werden sollte. Im vorliegenden Fall muss z. B. die Farbauswahl von einer AWT-Farbe auf eine bei Android zur Verfügung stehenden Farbpalette umgestellt werden.



- Nach Korrektur des Fehlers kann das Projekt über den Play-Button gestartet werden.

- Der Code wird kompiliert, ein Paket (sog. *APK-Datei*) mit der Anwendung erstellt und dieses auf dem Gerät installiert und automatisch gestartet.



AIDE beinhaltet auch eine integrierte Lernumgebung.

Sofern Zeit zur Verfügung steht, können Sie noch einige der Tutorials durcharbeiten.

Impressum

Entnommen aus
Teachers + Scientists: Für Wissenschaft begeistern

Herausgeber
Science on Stage Deutschland e. V. (SonSD)
Poststraße 4/5
10178 Berlin

Koordinatoren-Team
Helga Fenz, Robert-Havemann-Gymnasium Berlin, Vorstand SonSD
Christian Karus, Andreas-Vesalius-Gymnasium Wesel
Dr. Tom Steinlein, Universität Bielefeld, Fakultät für Biologie

Gesamtkoordination und Redaktion
Karoline Kirschner, Projektmanagerin SonSD
Stefanie Schlunk, Geschäftsführerin SonSD

In Kooperation mit
Stiftung Jugend forscht e. V.
jugend forscht

Hauptförderer von Science on Stage Deutschland e. V.
think ING.
Die Initiative für Ingenieurnachwuchs

Text- und Bildnachweise
Die Autorinnen und Autoren haben die Bildrechte für die Verwendung in dieser Publikation nach bestem Wissen geprüft und sind für den Inhalt ihrer Texte verantwortlich.

Gestaltung
WEBERSUPIRAN.berlin

Illustrationen
Heike Kreye

Bestellungen
www.science-on-stage.de
info@science-on-stage.de

1. Auflage 2017

Sie haben auch Interesse an einer Kooperation zwischen Lehrkräften und Forschenden? In unserem Leitfaden finden Sie praktische Tipps und Hinweise zur Umsetzung: www.teachers-and-scientists.de.