KAPITEL

11 Simulationen

Lernziele

Themen: Simulationen

Konzepte: Emergentes Verhalten, Experimente

In diesem Kapitel werden wir auf eine besondere Art von Softwareanwendungen etwas näher eingehen: Simulationen.

Simulationen sind ein faszinierendes Gebiet, da sie hoch experimentell sind und uns erlauben, die Zukunft vorherzusagen. Viele Simulationen werden (und wurden) für Computer entwickelt: Verkehrssimulationen, Wettervorhersagesysteme, Wirtschaftssimulationen (Simulationen der Aktienmärkte), Simulationen von chemischen Reaktionen, Atomexplosionen, Umweltsituationen und vieles mehr.

Konzept

Eine **Simulation** ist ein Computerprogramm, das ein Phänomen der realen Welt nachbildet. Wenn Simulationen genau genug sind, verraten sie interessante Dinge über die reale Welt.

In ***Kapitel 8*** haben wir bereits eine einfache Simulation kennengelernt, die einen Teil des Sonnensystems abbildete. Diese Simulation war zwar nicht präzise genug, um die Flugbahnen realer Planeten genau vorherzusagen, kann aber durchaus dazu verwendet werden, uns einige Aspekte der Astrophysik näherzubringen.

Grundsätzlich können Simulationen zwei verschiedenen Aufgaben dienen: der Analyse des simulierten Systems (um es besser zu verstehen) oder der Erstellung von Vorhersagen über das simulierte System.

Im ersten Fall kann die Modellierung eines Systems (wie das Sterne-Planeten-Modell) dazu beitragen, bestimmte Aspekte des Systemverhaltens besser zu verstehen. Im zweiten Fall – vorausgesetzt wir haben ein akkurates System – können wir „Was-wäre-wenn"-Szenarien durchspielen. Angenommen, wir würden über eine Verkehrssimulation für eine reale Stadt verfügen und würden in dieser Stadt jeden Morgen an einer bestimmten Kreuzung einen Verkehrsstau beobachten. Wie könnten wir diese Situation verbessern? Sollen wir einen neuen Verkehrskreisel bauen? Oder die Verkehrsampeln anders schalten? Vielleicht sollten wir eine Umgehungsstraße bauen?

Die Auswirkungen dieser Maßnahmen sind oft schwer vorherzusagen. Wir können nicht alle Alternativen im wahren Leben ausprobieren, um festzustellen, was am günstigsten ist, da die Veränderungen zu tief greifend und zu teuer wären. Aber wir können sie simulieren. In unserer Verkehrssimulation können wir jede Alternative durchspielen und sehen, wie sie sich auf den Verkehr auswirkt.

Wenn die Simulation präzise ist, lässt sich das Ergebnis, das wir in der Simulation beobachten, auch auf das wahre Leben übertragen. Doch die Betonung liegt auf dem „Wenn": Eine Simulation zu entwickeln, deren Genauigkeit unseren Ansprüchen genügt, ist recht schwierig und mit viel Arbeit verbunden. Doch für viele Systeme sind Simulationen machbar – mit durchaus nützlichen Ergebnissen.

Simulationen der Wettervorhersage sind inzwischen so akkurat, dass Vorhersagen für den folgenden Tag relativ zuverlässig sind. Eine Sieben-Tage-Vorhersage hingegen ist wie Würfeln: Die Simulationen sind einfach noch nicht gut genug.

Wer Simulationen einsetzt, sollte sich vor allem ihrer Grenzen bewusst sein. So weisen Simulationen stets einen gewissen Grad an Ungenauigkeit auf, da sich die Verhaltensweisen der Akteure selten absolut realistisch modellieren lassen und auch der genaue Zustand des Ausgangssystems nicht immer bekannt ist.

Außerdem bilden Simulationen gezwungenermaßen nur einen Teil der Realität ab, sodass man sich stets bewusst sein sollte, dass es nicht berücksichtigte relevante Aspekte geben könnte.

So ist vielleicht in unserer Verkehrsstau-Simulation keine der oben genannten Alternativen die beste Lösung. Man könnte zur Reduzierung des Verkehrsaufkommens ja auch den öffentlichen Nahverkehr ausbauen oder Fahrradspuren einrichten. Wenn unsere Simulation diese Aspekte nicht berücksichtigt, werden wir nie auf die besagten Lösungen stoßen, egal, wie akkurat die Simulation ist.

Trotz dieser Beschränkungen sind gute Simulationen unglaublich nützlich; ja, sogar sehr einfache Simulationen wissen zu faszinieren und laden zum Herumspielen ein. Simulationen sind in der Tat so nützlich, dass auf fast allen der weltweit schnellsten Computer vorwiegend oder zumindest häufig Simulationen ausgeführt werden.

Randbemerkung: Supercomputer

Im Internet findest du unter *http://www.top500.org* eine regelmäßig aktualisierte Liste der schnellsten Supercomputer der Welt einschließlich einiger Zusatzinformationen. Bei vielen findest du außerdem Links zu Websites, die Zweck und Einsatzbereich der Computer beschreiben.

Beim Aufrufen dieser Websites wirst du feststellen, dass viele dieser Computer von großen Forschungsinstituten oder dem Militär betrieben werden und dass auf fast allen vornehmlich physikalische Simulationen ausgeführt werden. Das Militär nutzt sie zum Beispiel, um Atomexplosionen zu simulieren, und Forschungsinstitute führen auf diese Weise die verschiedensten Forschungsexperimente durch.

Darüber hinaus kommt den Simulationen ein besonderer Platz in der Geschichte der objektorientierten Programmierung zu: denn Objektorientierung wurde mehr oder weniger explizit für die Ausführung von Simulationen erfunden.

Die allererste objektorientierte Programmiersprache, die bezeichnenderweise den Namen *Simula* erhielt, wurde in den 1960er Jahren von Kristen Nygaard und Ole-Johan Dahl am norwegischen Computerzentrum in Oslo entwickelt. Wie der

Name bereits nahelegt, diente sie vornehmlich der Erstellung von Simulationen. Alle heutigen objektorientierten Programmiersprachen gehen auf diese Sprache zurück. Dahl und Nygaard erhielten für ihre Leistungen in diesem Bereich 2001 den Turing-Award – dem Pendant der Informatik zum Nobelpreis.

Doch genug der einleitenden Worte und zurück an die Arbeit!

In diesem Kapitel wollen wir eine bereits vorhandene Simulation kurz vorstellen und uns dann selbst an einer versuchen, die etwas ausführlicher besprochen wird. Wir haben nun ein Stadium erreicht, in dem wir dir den Großteil der Arbeit überlassen: Wir werden die Grundzüge der zu implementierenden Aufgaben darstellen, doch du musst die Details selbst ausarbeiten.

11.1 Füchse und Hasen

Die erste Simulation, die wir untersuchen wollen, trägt den Namen *foxes-and-rabbits* (Abbildung 11.1). Sie ist ein typisches Beispiel für sogenannte *Räuber-Beute-Simulationen* – eine Simulationsart, in der eine Kreatur eine andere jagt (und frisst). In unserem Fall ist der Fuchs der Räuber und der Hase die Beute.

Die Idee dieser Simulation ist folgende: Das Szenario zeigt ein Feld mit Populationen von Füchsen und Hasen. Füchse werden durch blaue Quadrate und Hase durch gelbe repräsentiert.

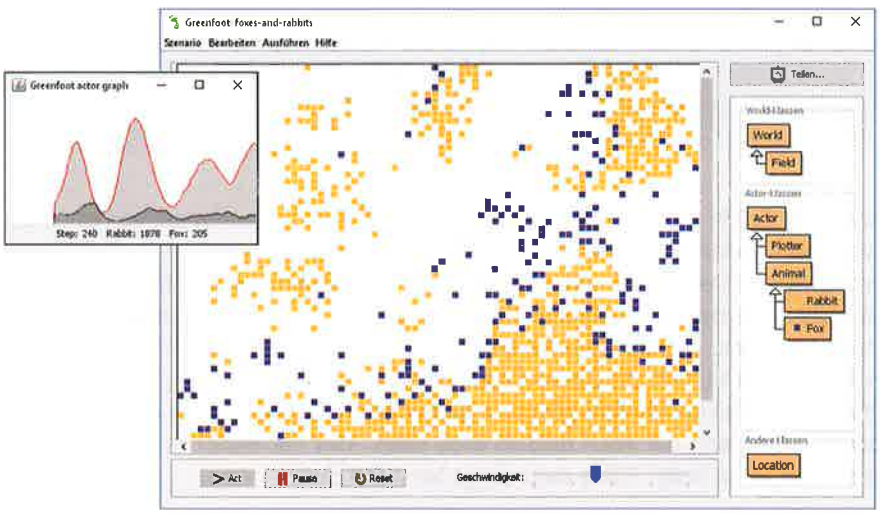


Abbildung 11.1
Die Füchse-und-Hasen-Simulation.

Beide Tierarten weisen ein ziemlich einfaches Verhalten auf: Hasen bewegen sich und zeugen – wenn sie alt genug sind – Nachkommen. Bei jedem **act**-Schritt besteht für Hasen eine festgelegte Wahrscheinlichkeit, dass sie sich vermehren. Hasen können auf zwei Arten sterben: entweder aufgrund ihres Alters oder weil sie von einem Fuchs gefressen werden.

Füchse bewegen und vermehren sich ähnlich wie die Hasen (auch wenn sie seltener und weniger Nachkommen produzieren). Im Gegensatz zu den Hasen allerdings gehen Füchse auf die Jagd. Wenn sie hungrig sind und neben ihnen ein Hase sitzt, bewegen sie sich auf die Position des Hasen, um ihn zu fressen.

Füchse können ebenfalls aus zwei Gründen sterben: Entweder sterben sie aus Altersgründen oder sie verhungern. Wenn sie über längere Zeit keinen Hasen zum Fressen finden, müssen sie sterben. (Bei Hasen gehen wir davon aus, dass sie immer genügend Futter finden.)

Übung 11.1 Öffne das Szenario *foxes-and-rabbits*. Führe es aus. Erläutere die Populationsmuster und die Bewegung, die du im Feld beobachtest.

Übung 11.2 Du wirst feststellen, dass bei der Ausführung dieses Szenarios ein zweites kleines Fenster mit den Bevölkerungskurven eingeblendet wird. Eine Kurve zeigt die Anzahl der Füchse, die andere die Anzahl der Hasen. Erläutere den Verlauf dieser Kurven.

Wie du siehst, ist die Simulation in vieler Hinsicht sehr einfach gehalten: Die Tiere müssen keinen Geschlechtspartner finden, um sich fortzupflanzen (das können sie alles allein), Nahrungsquellen für Hasen sind in der Simulation nicht vorgesehen, andere Todesursachen (wie Krankheiten) werden ignoriert und so weiter. Dennoch sind die Parameter, die wir simulieren, ziemlich interessant und wir können einige Experimente damit durchführen.

Übung 11.3 Sind die aktuellen Populationen stabil? Das heißt, läuft die Simulation kontinuierlich weiter, ohne dass eine der Arten ausstirbt? Wenn Arten aussterben, wie lange leben sie im Durchschnitt?

Spielt die Größe des Feldes eine Rolle? Stellen wir uns beispielsweise vor, es ginge um einen Nationalpark, der eine gefährdete Art beherbergt, und irgendjemand wollte eine Schnellstraße mitten durch diesen Park bauen, die von den Tieren nicht überquert werden kann, sodass der Park hinterher mehr oder weniger aus zwei Parks bestünde. Die Befürworter der Schnellstraße könnten anführen, dass dies keine Rolle spielt, weil die Gesamtgröße des Nationalparks noch ungefähr die gleiche sei. Die Parkgesellschaft könnte argumentieren, dass dies nicht gut sei, da die beiden Parks jetzt nur noch halb so groß seien. Wer hat recht? Spielt die Größe eines Parks eine Rolle? Führe dazu einige Versuche durch.

Übung 11.4 In der Klasse **`Field`** wurden oben im Quelltext Konstanten für die Breite und Höhe definiert. Ändere diese Konstanten, um die Größe des Feldes zu verändern. Hat die Größe des Feldes einen Einfluss auf die Stabilität der Populationen? Sterben die Arten schneller aus, wenn das Feld kleiner oder größer ist? Oder macht es keinen Unterschied? Probiere sehr kleine und sehr große Felder aus.

Andere Parameter, mit denen du herumspielen kannst, sind als Konstanten oben in den Klassen **Rabbit** und **Fox** definiert. So gibt es bei den Hasen Konstanten zum Höchstalter, zum Fortpflanzungsalter, zur Fortpflanzungshäufigkeit (als Wahrscheinlichkeit für jeden Schritt angegeben) und zur maximalen Anzahl der Jungtiere eines Wurfes. Füchse weisen die gleichen Parameter auf plus eines zusätzlichen Parameters: der Nährwert eines gefressenen Hasen (ausgedrückt als Anzahl an Schritten, die der Fuchs weiterlebt, nachdem er den Hasen gefressen hat). Der Futterlevel des Fuchses nimmt bei jedem Schritt um eins ab und nimmt wieder zu, wenn er einen Hasen gefressen hat. Wenn der Futterlevel null erreicht, stirbt der Fuchs.

Übung 11.5 Wähle eine Feldgröße, in der die Populationen fast stabil sind, aber ab und zu aussterben. Nimm dann Änderungen an den **Rabbit**-Parametern vor, um die Chancen zu erhöhen, dass die Population überlebt. Kannst du Einstellungen finden, die die Populationen stabiler machen? Gibt es Einstellungen, die die Stabilität gefährden? Entsprechen die beobachteten Effekte deinen Erwartungen oder weichen sie davon ab?

Übung 11.6 Wenn die Fuchspopulation gelegentlich auszusterben droht, könnten wir probieren, die Überlebenschance der Füchse zu verbessern, indem wir den Futterwert der Hasen erhöhen. Wenn Füchse nach dem Fressen eines Hasen länger leben, sollten weniger von ihnen sterben. Prüfe diese Hypothese. Verdopple den Wert der Konstanten **RABBIT_FOOD_VALUE** und teste das Szenario. Lebt die Fuchspopulation länger? Erläutere das Ergebnis.

Übung 11.7 Ändere andere Parameter der Klasse **Fox**. Schaffst du es, dass die Populationen stabiler werden?

Diese Übungen zeigen, dass wir mit dieser Simulation experimentieren können, indem wir einige Parameter ändern und ihre Auswirkungen beobachten. Wir wollen uns in diesem Kapitel aber nicht allein mit dem Herumspielen an einer bereits vorhandenen Simulation zufriedengeben, wir wollen unsere eigene Simulation entwickeln. Dies werden wir in den nächsten Abschnitten unter Verwendung eines Szenarios mit Ameisen (*ants*) machen.

11.2 Ameisen

Das Szenario *ants* (Abbildung 11.2) simuliert das Verhalten von Ameisenkolonien auf der Futtersuche. Genauer gesagt, wir würden dieses Verhalten gerne simulieren, aber noch funktioniert es nicht. Deshalb werden wir das Szenario weiterentwickeln. Für das Szenario in seinem jetzigen Zustand wurden bereits einige Vorarbeiten geleistet: Der grafische Teil existiert bereits und ein Teil der Implementierung ist auch schon vorhanden. Die Hauptfunktionalität fehlt jedoch noch; diese zu ergänzen soll unsere Aufgabe sein.

Übung 11.8 Öffne das Szenario namens *ants* aus dem Ordner zu diesem Kapitel. Erzeuge einen Ameisenhügel in der Welt und führe das Szenario aus. Was kannst du beobachten?

Übung 11.9 Studiere den Quelltext der Klasse **`Ant`**. Was kann die Ameise bereits?

Übung 11.10 **`Ant`** ist von der Klasse **`Creature`** abgeleitet. Welche Funktionalität weist die Klasse **`Creature`** auf?

Übung 11.11 Studiere den Quelltext der Klasse **`AntHill`**. Was macht diese Klasse? Wie viele Ameisen befinden sich in einem Ameisenhügel?

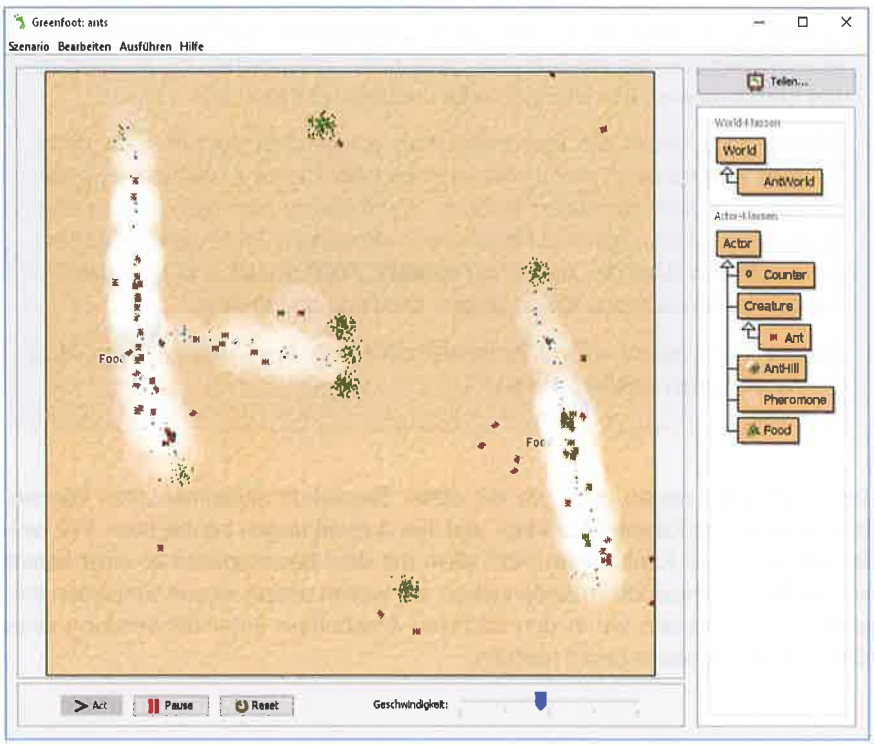


Abbildung 11.2
Die (fertige) Ameisen-Simulation.

Das Erste, was uns auffällt, ist, dass sich die Ameisen nicht bewegen. Sie werden mitten auf dem Ameisenhügel erzeugt, doch da sie sich nicht bewegen, sitzen sie nach einer kurzen Weile alle übereinander. Deshalb wollen wir als Erstes dafür sorgen, dass die Ameisen ihren Hügel verlassen.

Übung 11.12 Füge in die **`act`**-Methode von **`Ant`** eine Codezeile ein, die die Ameisen in Bewegung setzt. Verwende hierzu nicht die **`move`**-Methode von **`Actor`**, sondern geerbte Methoden der Klasse **`Creature`**. Lies die Dokumentation der Klasse **`Creature`**, um herauszufinden, welche Methoden sich hierfür anbieten.

Im Folgenden wollen wir schrittweise eine Reihe von Verbesserungen an diesem Szenario vornehmen:

- Wir werden Futter zu dem Szenario hinzufügen, damit die Ameisen etwas zum Einsammeln haben.
- Wir werden die Ameisen verbessern, sodass sie das Futter finden und nach Hause tragen können.
- Als Nächstes werden wir eine Klasse **Pheromone** hinzufügen. Pheromone sind chemische Botenstoffe, die von einigen Tieren produziert werden, um ihren Artgenossen Nachrichten zu hinterlassen.
- Dann werden wir die Ameisen so programmieren, dass sie die Pheromone einsetzen: Sie sollen Pheromonmarkierungen auf dem Boden hinterlassen, wenn sie Futter gefunden haben; andere Ameisen können dann diese Pheromone riechen und ihren Wanderpfad daran ausrichten.

All diese Schritte zusammen simulieren grob das Futtersammelverhalten von Ameisenkolonien. Wenn wir fertig sind, können wir mit der Simulation ein wenig herumspielen.

11.3 Futter sammeln

Unsere erste Aufgabe soll sein, in unserem Szenario Futter zu erzeugen, das die Ameisen dann einsammeln und zu ihrem Ameisenhaufen zurücktragen.

Übung 11.13 Erzeuge eine neue Klasse namens **Food**. Für diese Klasse benötigst du kein spezielles Bild. Wir werden das Bild im Code der Klasse erzeugen und zeichnen.

Jedes Objekt der Klasse **Food** repräsentiert einen Haufen Futterkrümel. Wir beabsichtigen, ein neues, dynamisch gezeichnetes Bild für das **Food**-Objekt zu erzeugen und dabei einen kleinen Punkt für jeden Krümel im Haufen zu zeichnen. Ein Haufen könnte am Anfang, sagen wir, aus 100 Krümeln bestehen und jedes Mal, wenn eine Ameise auf diesen Haufen stößt, nimmt sie einige Krümel fort. Das bedeutet, dass das Bild des Haufens jedes Mal, wenn eine Ameise etwas Futter aufnimmt, neu gezeichnet werden muss.

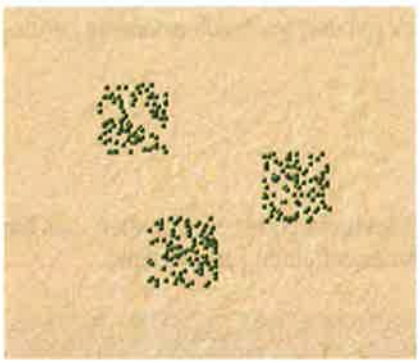
Übung 11.14 Erzeuge in der Klasse **Food** ein Zustandsfeld für die Anzahl der Krümel im Haufen und initialisiere es mit 100.

Übung 11.15 Erzeuge eine neue **`private`**-Methode namens **`updateImage`**, die ein Bild fester Größe erzeugt und auf diesem Bild für jeden Krümel im Haufen einen Punkt an einer Zufallsposition zeichnet. Wähle für das Bild des Haufens eine Größe, die deiner Meinung nach gut aussieht. Rufe diese Methode vom Konstruktor aus auf.

Wenn du die Übung oben gemacht hast und die Futterkrümel in dem Bild des **Food**-Objekts an einer Zufallsposition gezeichnet wurden (mithilfe der **Greenfoot.getRandomNumber**-Methode, um die Koordinaten zu ermitteln), wirst du festgestellt haben, dass der Krümelhaufen einen eher quadratischen Umriss hatte (Abbildung 11.3a). Dies liegt daran, dass das Bild selbst quadratisch ist und die Krümel darin gleichmäßig verteilt sind.

Um dies zu korrigieren und die Krümel anders zu platzieren, sodass das Ganze mehr wie ein Haufen aussieht (bei dem die meisten Krümel in der Mitte liegen und einige im Kreis darum herum), können wir eine andere Methode zur Erzeugung von Zufallszahlen verwenden. Wie diese Methode heißt und wie man sie einsetzt, ist Thema der Übungen 11.16 und 11.17. Beachte, dass diese Übungen etwas für Fortgeschrittene sind und rein kosmetische Zwecke verfolgen: Sie ändern lediglich das Aussehen des Futterhaufens und nicht dessen Funktionalität, sodass du sie getrost überspringen kannst.

Abbildung 11.3
Platzierung der Futterkrümel mit verschiedenen Algorithmen zur zufälligen Positionierung.



Krümel mit einer gleichmäßigen Verteilung

Krümel mit einer Gauß'schen Verteilung

Übung 11.16 Rufe die API-Dokumentation zur Java-Standardklassenbibliothek auf. Dort findest du die Dokumentation der Klasse **Random** aus dem Paket **java.util**. Objekte dieser Klasse sind *Zufallszahlengeneratoren*, die flexibler sind als die Greenfoot-Methode **getRandomNumber**. Die Methode, die uns hier besonders interessiert, ist eine, die Zufallszahlen in einer Gauß'schen Verteilung (auch „Normalverteilung" genannt) zurückliefert. Wie lautet der Name dieser Methode und was macht sie?

Übung 11.17 Ändere in deiner Klasse **Food** die Anordnung der Futterkrümel im Bild so, dass die Gauß'sche Verteilung von Zufallszahlen Anwendung findet. Dazu musst du zur Erzeugung von Zufallszahlen ein **java.util.Random**-Objekt anstelle von **Greenfoot.getRandomNumber** verwenden.

Randbemerkung: Zufallsverteilung

Bei Programmen mit Zufallsverhalten ist es manchmal wichtig darüber nachzudenken, welche Art von Zufallsverteilung wir benötigen. Viele Zufallsfunktionen, so wie **`Greenfoot.getRandomNumber`**, erzeugen eine gleichmäßige Verteilung. Bei dieser Verteilung ist die Chance eines jeden möglichen Ergebnisses gleich. Mit der Gauß'schen Zufallszahlenfunktion erhalten wir eine *Normalverteilung*. Bei dieser Verteilung treten die Durchschnittsergebnisse häufiger auf, während die extremen Ergebnisse seltener sind.

Wenn wir zum Beispiel ein Würfelspiel programmieren, benötigen wir eine gleichmäßige Verteilung. Jede Seite des Würfels hat die gleiche Wahrscheinlichkeit, angezeigt zu werden. Wenn wir es hingegen mit der Geschwindigkeit von Autos in einer Verkehrssimulation zu tun haben, wäre die Normalverteilung die bessere Wahl. Die meisten Autos fahren mehr oder weniger mit einer Durchschnittsgeschwindigkeit und nur einige wenige Autos sind sehr langsam oder sehr schnell.

Als Nächstes müssen wir die Funktionalität ergänzen, die einige Krümel von dem Futterhaufen entfernt, damit die Ameisen etwas Futter aufnehmen können.

Übung 11.18 Füge in die Klasse **Food** eine öffentliche (**`public`**) Methode hinzu, die einige Krümel vom Haufen entfernt. Stelle sicher, dass das Bild nach dem Neuzeichnen die korrekte Anzahl an übrig gebliebenen Krümeln aufweist. Wenn die Krümel alle weg sind, soll sich das **Food**-Objekt selbst von der Welt entfernen. Teste diese Methode interaktiv.

Nachdem wir einen Haufen Futterkrümel in unserem Szenario zur Verfügung haben, können unsere Ameisen das Futter auch einsammeln. Die Ameisen wechseln jetzt zwischen zwei verschiedenen Verhalten:

- Wenn sie kein Futter tragen, suchen sie nach Futter.
- Wenn sie gerade Futter tragen, gehen sie damit nach Hause.

Die Ameisen wechseln zwischen diesen beiden Verhaltensmustern, wenn sie entweder einen Futterhaufen oder ihren heimatlichen Ameisenhügel erreichen. Wenn sie nach Futter suchen und auf einen Futterhaufen stoßen, nehmen sie etwas Futter auf und wechseln vom ersten zum zweiten Verhalten. Beim Erreichen des Ameisenhügels lassen sie das Futter fallen und wechseln wieder zurück zum ersten Verhaltensmuster.

Dieses Verhalten wollen wir in unserer Klasse **Ant** implementieren. Im Pseudocode könnte das in etwa folgendermaßen aussehen:

```
if (Futter getragen wird) {
    gehe nach Hause;
    prüfe, ob wir Zuhause erreicht haben;
}
else {
    suche nach Futter;
}
```

Das wollen wir jetzt Schritt für Schritt implementieren.

Übung 11.19 Implementiere in der Klasse **Ant** eine Methode **searchForFood**. Diese Methode soll am Anfang den Akteur einfach nur Zufallswege einschlagen lassen und prüfen, ob er auf Futter gestoßen ist. Wenn er einen Futterhaufen gefunden hat, soll die Ausführung beendet werden. (Wir wollen hiermit lediglich testen, ob wir auch wirklich Futter gefunden haben.)

Übung 11.20 Füge Funktionalität hinzu, um etwas Futter aufzunehmen, wenn du auf einen Futterhaufen gestoßen bist (anstatt die Ausführung zu beenden). Wir müssen einige Krümel von dem Futterhaufen entfernen (eine Methode dafür sollte eigentlich bereits vorhanden sein), registrieren, dass wir jetzt Futter tragen (hierfür benötigen wir wahrscheinlich ein Zustandsfeld), und das Bild der Ameise entsprechend ändern. In dem Szenario findest du bereits ein Bild, das du dafür verwenden kannst: *ant-with-food.png*.

Übung 11.21 Stelle sicher, dass die Ameise nach Hause läuft, wenn sie Futter trägt.

Übung 11.22 Implementiere eine Methode, die prüft, ob eine Ameise den Heimathügel erreicht hat. Wenn sie ihr Zuhause erreicht hat, soll sie ihr Futter fallen lassen. Das Fallenlassen des Futters besteht darin, festzuhalten, dass sie kein Futter mehr trägt (wobei wieder zum alten Bild der Ameise zurückgekehrt wird), und die Methode **countFood** der Klasse **AntHill** aufzurufen, die zählt, wie viele Futterkrümel eingesammelt wurden.

Konzept

Die Verwendung von **kurzen Methoden**, die nur eine Aufgabe erfüllen, führt zu einer besseren **Codequalität**.

Achte auf die Qualität deines Codes: Verwende kurze Methoden, die jeweils nur einen bestimmten Zweck erfüllen, und stelle sicher, dass deine Methoden gut kommentiert sind. Schreibe nicht zu viel Code in eine einzelne Methode.

Eine Implementierung der bisher diskutierten Funktionalität findest du in dem Szenario *ants-2*. Nach der Beendigung deiner Implementierung (oder wenn du hängen geblieben bist), möchtest du vielleicht deine Lösung mit unserer vergleichen.

11.4 Die Welt einrichten

Bevor wir unser Szenario um Pheromone ergänzen, wollen wir zuerst ein paar Zeilen Initialisierungscode hinzufügen, der automatisch einige Ameisenhügel und Futterhaufen erzeugt, sodass wir dies nicht bei jedem Testen manuell machen müssen.

Übung 11.23 Füge in die Klasse **`AntWorld`** Code ein, der automatisch zwei Ameisenhügel und einige Futterhaufen in der Welt erzeugt. Du kannst den Code entweder manuell schreiben oder die Funktion *SaveTheWorld* in der Greenfoot-Schnittstelle verwenden.

11.5 Pheromone hinzufügen

Wir verfügen jetzt über eine gute Ausgangsbasis und können die Pheromone hinzufügen. Jedes Pheromon-Objekt ist ein kleiner Tropfen einer chemischen Substanz, den die Ameisen auf dem Boden hinterlassen. Diese Tropfen verdunsten relativ schnell und verschwinden schließlich ganz.

Ameisen senden Pheromone aus, wenn sie von der Futterquelle zu ihrem Heimathügel laufen. Andere Ameisen, die den Pheromontropfen riechen, können sich dann von ihrem Heimathügel abwenden und in Richtung des Futters laufen.

Übung 11.24 Erzeuge eine Klasse namens **Pheromone**. Für diese Klasse benötigst du kein Bild – wir werden das Bild im Quelltext zeichnen.

Übung 11.25 Implementiere in der Klasse **Pheromone** eine Methode **`updateImage`**. Am Anfang sollte diese Methode ein Bild mit einem weißen Kreis zeichnen und dies als Bild des Akteurs festlegen. Der weiße Kreis sollte halbtransparent sein. Rufe diese Methode im Konstruktor auf.

Übung 11.26 Richte für die Pheromone ein Attribut für die *Intensität* ein. (Das bedeutet, füge ein Zustandsfeld **`intensity`** hinzu.) Die Intensität eines Pheromon-Objekts sollte mit einer definierten maximalen Intensität beginnen und bei jedem **`act`**-Schritt abnehmen. Wenn die Intensität den Wert 0 erreicht, entferne das Pheromon aus der Welt. Ein Pheromontropfen sollte nach ungefähr 180 **`act`**-Zyklen verdunstet sein.

Übung 11.27 Mache in deiner Methode **`updateImage`** von der Intensität des Pheromons Gebrauch. Mit abnehmender Intensität sollte der weiße Kreis auf dem Bildschirm kleiner und transparenter werden. Stelle sicher, dass **`updateImage`** von der **`act`**-Methode aus aufgerufen wird, sodass wir auf dem Bildschirm verfolgen können, wie sich das Bild ändert.

Übung 11.28 Teste deine Pheromone, indem du sie manuell in der Welt platzierst und deine Simulation ausführst.

Damit verfügen wir über eine Klasse **Pheromone**, die unsere Ameisen verwenden können. Wir müssen jetzt nur noch dafür sorgen, dass die Ameisen diese Klasse nutzen. Diese Aufgabe ist zweigeteilt. Zuerst müssen wir das Pheromon in der Welt platzieren. Anschließend gilt es, das Pheromon zu bemerken und als Reaktion die Richtung zu ändern. Doch eins nach dem anderen.

Übung 11.29 Füge zu deiner Ameise eine Methode hinzu, die einen Tropfen Pheromon in der Welt fallen lässt. Rufe diese Methode wiederholt auf, während die Ameise nach Hause läuft.

Wenn du in der vorangehenden Übung bei jedem `act`-Schritt einen Tropfen Pheromon fallen gelassen hast, wirst du festgestellt haben, dass du damit viel zu viel Pheromon abgibst. Ameisen können nicht unendlich viele Pheromone produzieren. Nachdem sie einen Tropfen abgegeben haben, benötigen sie einige Zeit, um sich für eine weitere Pheromonabgabe zu regenerieren.

Übung 11.30 Ändere die Ameise so, dass sie höchstens bei jedem 18. Schritt einen Tropfen Pheromon abgibt. Dafür musst du ein Zustandsfeld einrichten, das den aktuellen Pheromonspiegel einer Ameise speichert. Wenn die Ameise einen Tropfen Pheromon abgibt, fällt der Pheromonspiegel (die übrig gebliebenen Pheromone im Körper der Ameise) auf 0, um dann erneut langsam anzusteigen, bis die Ameise wieder einen weiteren Tropfen abgeben kann.

Abbildung 11.4 zeigt eine Pheromonspur[1], die von der Ameise hinterlassen wurde. Die Tropfen stehen auf Abstand (die Ameise benötigt einige Zeit, um neue Pheromone zu produzieren) und die älteren Pheromontropfen sind bereits am Verdunsten – sie sind kleiner und transparenter.

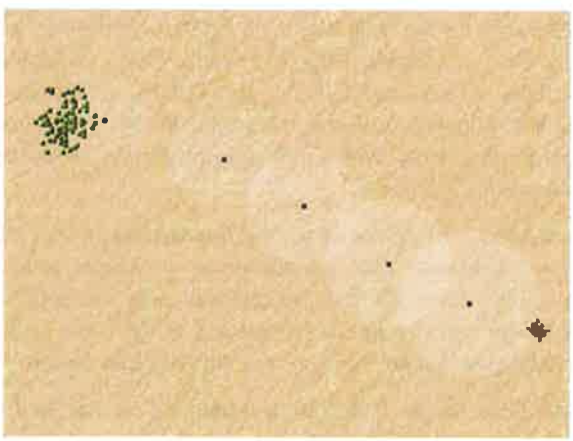


Abbildung 11.4
Eine Ameise, die eine Pheromonspur hinterlässt.

1 Wenn du genau hinsiehst, wirst du feststellen, dass ich das Pheromon-Bild ein wenig überarbeitet habe, sodass es in der Mitte einen kleinen schwarzen Punkt aufweist. Damit sollen die Pheromone nur besser zu sehen sein, vor allem wenn sie bereits ziemlich transparent sind.

Zum Abschluss müssen wir nur noch dafür sorgen, dass die Ameisen die Pheromone riechen, um anschließend ihre Bewegungsrichtung zu ändern.

Wenn eine Ameise einen Tropfen Pheromon riecht, soll sie sich von ihrem Heimathügel für eine begrenzte Zeit abwenden. Findet sie in dieser Zeit kein Futter oder einen neuen Tropfen Pheromon, nimmt sie ihre Zufallswanderung wieder auf. Unser Algorithmus für die Futtersuche könnte in etwa wie folgt aussehen:

```
if (wir kürzlich auf einen Tropfen Pheromon gestoßen sind) {
    wende dich von deinem Heimathügel ab;
}
else if (wir jetzt Pheromon riechen) {
    gehe in Richtung des Pheromontropfenzentrums;
    if (wir im Zentrum des Pheromontropfens sind) {
        registriere, dass wir Pheromon gefunden haben;
    }
}
else {
    wandere willkürlich umher;
}
prüfe auf Futter;
```

Wenn du dies in deinem eigenen Szenario implementierst, denke daran, für jede einzelne Unteraufgabe eine eigene Methode zu erzeugen. Auf diese Weise bleibt dein Code gut strukturiert, leicht zu verstehen und zu ändern.

Übung 11.31 Implementiere die oben diskutierte Funktionalität in deinem eigenen Szenario. Wenn Ameisen Pheromone riechen, wenden sie sich für die nächsten 30 Schritte von ihrem Heimathügel ab, bevor sie zu ihrem Standardverhalten zurückkehren.

Nach Abschluss dieser Übung ist deine Ameisen-Simulation mehr oder weniger fertig (sofern eine Softwareanwendung überhaupt je fertig sein kann). Wenn du jetzt das Szenario ausführst, solltest du Ameisen sehen, die einen Pfad zu den Futterquellen ausbilden.

11.6 Pfad ausbilden

Ein interessanter Aspekt dieses Szenarios ist, dass es nirgendwo in diesem Projekt Code gibt, der vorgibt, Pfade auszubilden. Das Verhalten der einzelnen Ameisen ist relativ einfach: „Wenn du Futter hast, geh nach Hause; wenn du Pheromone riechst, geh weg; andernfalls geh überall hin." Im Zusammenspiel jedoch zeigen die Ameisen ein ziemlich hoch entwickeltes Verhalten. Sie bilden stabile Pfade aus, frischen die verdunstenden Pheromone auf und tragen sehr effizient das Futter zurück zu ihrem Ameisenhügel.

Konzept

Simulationen von Systemen weisen oft **emergentes Verhalten** auf. Dieses Verhalten ist nicht in den einzelnen Akteuren programmiert, sondern ergibt sich automatisch als Summe der Verhaltensweisen von allen.

Dies wird auch als *emergentes Verhalten* bezeichnet. Das Verhalten ist nicht bei den einzelnen Akteuren programmiert, sondern ist ein Systemverhalten, das sich durch die Interaktionen vieler (relativ einfacher) Akteure ausbildet.

Die meisten komplexen Systeme weisen eine Art von emergentem Systemverhalten auf – sei es, dass es sich um Verkehrssysteme in Großstädten, Computernetzwerke oder Menschenmengen handelt. Diese Effekte vorherzusagen ist sehr schwer und Computersimulationen können helfen, solche Systeme zu verstehen.

Übung 11.32 Wie realistisch ist unsere Simulation des Einsatzes von Pheromonen bei Ameisen? Recherchiere ein wenig über die tatsächliche Verwendung von Pheromonen in Ameisenkolonien und schreibe auf, welche Aspekte unserer Simulation realistisch sind und wo wir etwas vereinfacht haben.

Übung 11.33 Angenommen durch die Umweltverschmutzung würde eine toxische Substanz in das Umfeld der Ameisen gelangen. Der Effekt wäre, dass ihre Pheromonproduktion nur noch ein Viertel der vorherigen Produktion betragen würde. (Die Zeit zwischen den einzelnen Tropfen ist viermal so lang.) Können sie dann immer noch Pfade ausbilden? Teste deinen Code.

Übung 11.34 Angenommen, ein weiterer Schadstoff hätte die Fähigkeit der Ameisen auf ein Drittel reduziert, sich daran zu erinnern, dass sie kürzlich ein Pheromon gerochen haben. Anstelle von 30 Schritten können sie sich das Pheromon nur 10 Schritte lang merken. Welche Auswirkung hat dies auf ihr Verhalten?

Es gibt noch viele weitere Experimente, die du durchführen kannst. Am nächstliegenden ist, die Positionen der Ameisenhügel und Futterquellen zu variieren und verschiedene Werte für die Attribute einzugeben, die das Verhalten der Ameisen festlegen.

Das Szenario *ants-3* im Ordner zu diesem Kapitel zeigt eine Implementierung der bisher besprochenen Aufgaben. Sie umfasst drei verschiedene Methoden zum Einrichten der Welt-Klasse, die interaktiv von dem Kontextmenü der Welt aufgerufen werden können.

Zusammenfassung der Programmiertechniken

In diesem Kapitel haben wir zwei Simulationsbeispiele kennengelernt. Dies diente einem doppelten Zweck. Zum einen bot es uns die Möglichkeit, viele der in früheren Kapiteln besprochenen Programmiertechniken zu üben, bei denen wir viele der vorgestellten Java-Konstrukte einsetzen mussten. Zum anderen sind Simulationen eine interessante Art von Softwareanwendung, mit denen man herumexperimentieren kann. Simulationen werden im realen Leben für viele Zwecke verwendet, z.B. Wettervorhersage, Verkehrsplanung, Umweltverträglichkeitsstudien, physikalische Forschung und viele weitere.

Wenn es dir gelungen ist, all die Übungen in diesem Kapitel zu lösen, hast du bereits eine ganze Menge von dem verstanden, was dir dieses Buch vermitteln möchte, und verfügst über grundlegende Programmierkenntnisse.

Zusammenfassung der Konzepte

- Eine **Simulation** ist ein Computerprogramm, das ein Phänomen der realen Welt nachbildet. Wenn Simulationen genau genug sind, verraten sie interessante Dinge über die reale Welt.
- Die Verwendung von **kurzen Methoden**, die nur eine Aufgabe erfüllen, führt zu einer besseren **Codequalität**.
- Simulationen von Systemen weisen oft **emergentes Verhalten** auf. Dieses Verhalten ist nicht in den einzelnen Akteuren programmiert, sondern ergibt sich automatisch als Summe der Verhaltensweisen von allen.