PROGRAMMING PRACTICE AND APPLICATIONS ASSIGNMENT 3 REPORT – PREDATOR-PREY SIMULATION

Christian Impollonia (K1925245) and Samiira Mohamed (K1922089)

SHORT DESCRIPTION

Our predator prey simulation portrays the relationship between 5 acting species and plants in the Savannah. It also shows the changing patterns of animal behaviour in response to time and weather.

BASE TASKS

1. **Your simulation should have at least five different kinds of acting species. At least two of these should be predators (they eat another species), and at least two of them should not be predators (they may eat plants)**

Our predator-prey simulation shows the relationship between 5 animals: lions and tigers which are predators that consume prey: antelopes, giraffes and zebras. When predators intake prey, their food level increases and similarly, when preys intake plants, their energy level also increases. We implemented this base task by creating an inheritance hierarchy. Predators and Preys are subclasses of Animal, which is also a subclass of Actor. Actor includes methods which are in common in both our plants and animals. Our Animal class is broken down into Predator and Prey. The Predator class includes the shared behaviours of our lions and tigers whilst the Prey class includes the shared behaviours of our giraffes, antelopes and zebras. The individual animal classes itself have static variables and values which are unique to each animal.

2. **At least two predators should compete for the same food source.**

Lions and tigers compete for all three preys. In our Predator class, the findFood method is implemented such that if either a lion or a tiger comes across any of the three preys, the prey dies as the predator consumes. Thus, both predators are in competition for the same food source.

3. **Some or all of the species should distinguish male and female individuals. For these, the creatures can only propagate when a male and female individual meet.**

In the Animal class, we created a boolean field called isMale which is randomly given true or false when the object is created. This is the gender of the animal. When an animal is trying to breed, there is a check to see if the other animal has the opposing isMale value. We have done this by creating the isPartner boolean method in the Animal class. Because the process is the same for all animals, there was no need to have it in each animal's class, so we avoided code duplication. The method starts by checking if an animal within a radius is the same animal (by using the getClass method and comparing the two classes) and if it is of the opposite gender (by comparing the two isMale fields). If so, it returns true and it can breed. To balance the simulation, because animals don't simply duplicate like before, we increased the range of tiles they can meet at, from adjacent tiles to 10 tiles. We did this by adding a method closeLocations in the field class. We also increased our breeding probability to increase the likelihood of mating within our simulation.

4. **You should keep track of the time of day. At least some creatures should exhibit different behaviour at some time of the day**

   We have created a time field in the Simulator class which takes count of time. It increments by an hour for every step in the simulation. A modulo operator is used which resets our clock to 00:00 after the 24th hour. We also created a boolean field isDay, which is true when time is between 6 and 18, and false otherwise. We modified the simulateOneStep method to make sure that animals don't act during the night (they sleep), and that plants grow much slower. To balance the simulation, we preferred having plants and animals not age, therefore not die, during the night. We added a time stamp to the top left corner by modifying the SimulatorView class. Due to inactivity in the night, the program ran much faster compared to daytime, so we used the delay method. This method is run only when isDay is false (so when it's night) and sleeps the thread for 100ms.

CHALLENGE TASKS

1. **Add plants. Plants grow at a given rate, but they do not move. Some creatures eat plants. They will die if they do not find their food plant.**

Plant was created as a subclass of our Actor superclass. Since plants are consumed by prey, we have created a findFood method for preys too. Differently from the Predator one, this one checks if a plant is located closely to the prey (using instanceof) and if so, consumes the plant which increases the prey's food level. In each individual prey, the act method was updated so that as the animal moves around, their food level decreases using the incrementHunger method called from the animal superclass. If the foodLevel value reaches 0, they die. Plants are randomly added to our simulation in the simulateOneStep method in the Simulator

class. Every step, the simulator checks for free spaces in the field, and plants are created in accordance to their creation probability. The growth rate of the plant also depends on weather.

2. **Add weather. Weather can change, and it influences the behaviour of some simulated aspects.**

We created a Weather class, that is the super class of three different types of weather being snow, rain and fog . Weather is then a subclass of Event, that is a random event happening in the Simulation, that can't move or be seen. In the Simulator class we created an Array of Weather, and added the different weather types after creating the objects. The current event is randomly chosen from that array every 12 steps (twice a day). Each type of weather has a different event probability. Based on that probability, the event in the currentWeatherEvent field can happen (calls the happen method overridden in each subclass of Weather), if not the weather remains clear.
Rain increases the growth of plants whilst Snow stops it. We implemented this by creating a modifier field which is initially set to 1, the depower method makes the modifier 0 and is called when Snow happens, while the boost method makes the modifier 2 and is called when Rain happens. The modifier influences how many times the loop that creates plants is run.
Fog influences the sight of predators, making them less likely to find food if foggy. In the fog class, there is a fogDepower field that is initially set to 1 but changes to 0.5 when it is foggy. We modified the findFood method in the Predator class so that it works as based on the chance of fogDepower. So when there is fog the chances to find prey are halved. You can see the current weather at the top of the screen. We added the label the same way we added the time label, by modifying the SimulatorView class.

3. **Add disease. Some animals are occasionally infected. Infection can spread to other animals when they meet.**

The disease we thought of is a disease that every animal has a small chance of getting at birth. It is extremely contagious, it is spread sexually between two animals that breed, but the particular aspect of it is that only males can be infected during breeding.
We created a boolean field called isDiseased which has a very small probability to be true when an animal is created. To spread the disease, we modified the breed method in all individual animal classes, so that the female (if diseased) will call the setDisease method of the partner, turning isDiseased to true. If an animal is diseased, they die after 10 steps. We implemented this using a field called death_count in the animal class which is set to 10. If an animal is diseased, the value decreases per step and the animal is setDead when the value reaches 0.

4. **The challenge task that we came up with is to have Earthquakes that will kill most animals and plants. They are very rare.**

To implement earthquakes we extended our inheritance hierarchy, so that Earthquake could be a subclass of NaturalDisaster, which is a subclass of Event. Earthquakes have an event probability, that is extremely low, and a killing probability, that is very high. The Simulator class creates an earthquake object, and every step there is a small chance for the Earthquake method to call the method "happen". When the earthquake is happening, there is a big chance for each actor to be set dead inside the simulateOneStep method

CODE QUALITY CONSIDERATIONS

COUPLING: All classes and fields are kept private in order to ensure that they cannot be accessed by other classes and without the use of accessor methods. This leads to loose coupling, avoiding the risk of breaking the program by making a small change.

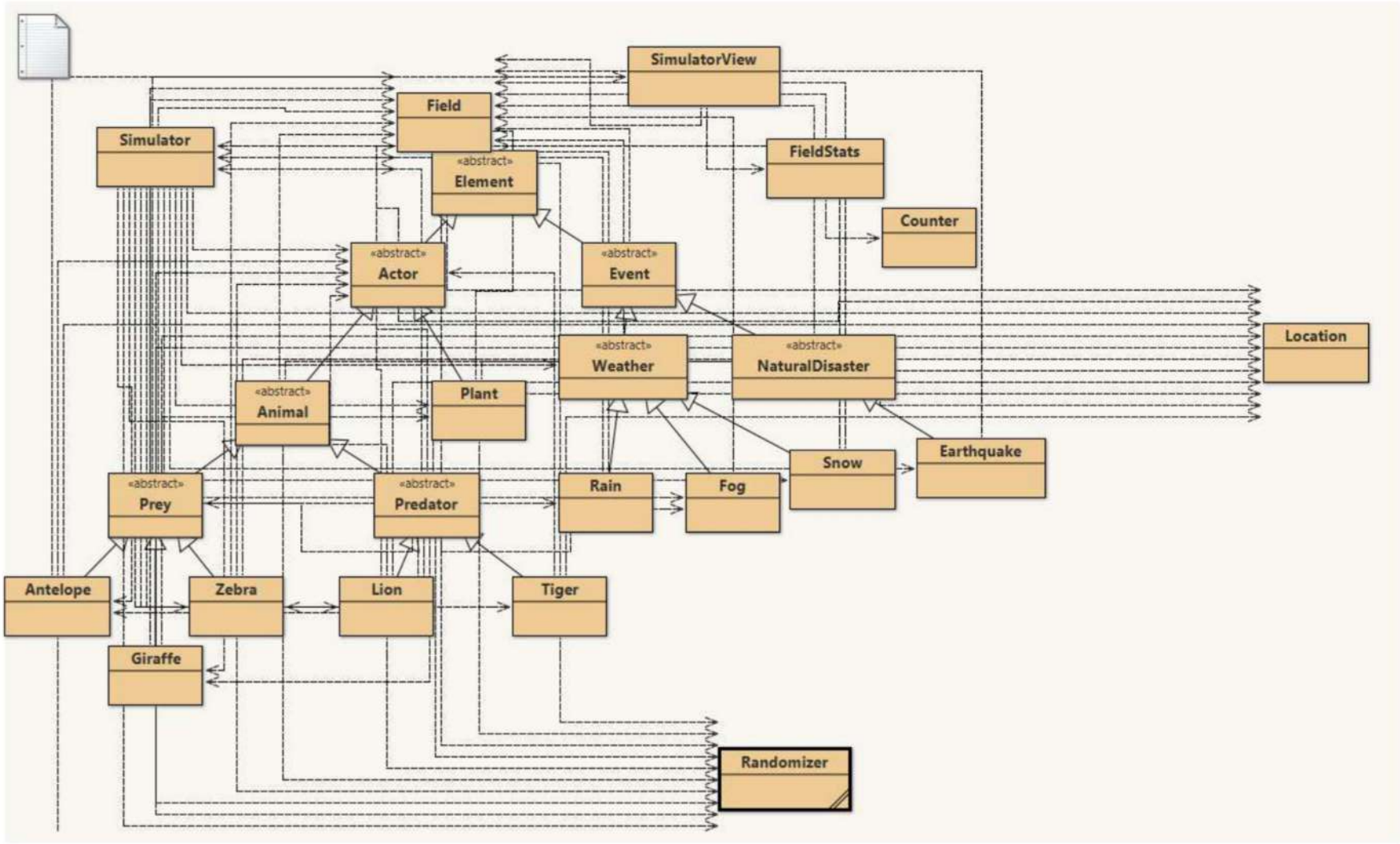
COHESION: All classes must represent one entity, thus the creation probability fields, which were initially located in the simulator class, were relocated to the individual animal classes. Also, the Fox class stored the food value of the rabbit, and, for separation of concerns, it shouldn't. Therefore, each prey stores its own food value. The Simulator class stores the sum of the food values of all preys, to be accessed as a max value for foodLevel when a predator is created.

RESPONSIBILITY-DRIVEN DESIGN: each class Is responsible for the modification of its own fields. For example, in the individual animal classes, the food level and age fields increase in each animal's class and not in the simulator, that should only be accessing them through accessor methods.

MAINTAINABILITY: to increase maintainability, we tried to avoid code duplication where possible. To do this, we created an inheritance hierarchy. For example, the findFood method was moved to the Predator class rather than having the method in the individual animal classes. Or for instance, the incrementHunger method was moved to the Animal class, given that Preys can get hungry as well, with the addition of plants. Overriding was also used to avoid code duplication in the Simulator class. An example is the different types of weather that are stored in a Weather array, and they have a single happen method in common that is called once, despite doing different things.

PROBLEMS AND LIMITATIONS

1. The main problem that we encountered was that due to the presence of static final variables in our individual animal classes, there was some code duplication. We decided that it was better to have some duplicated code instead of having easily modifiable variables that could potentially break the program.
2. Another limitation was that we had to include a protected field. In our Animal superclass, our foodLevel field is protected so that it can be accessed by the subclasses to avoid code duplication.


SimulatorView
Field
Simulator
FieldStats
«abstract»
Element
Counter
«abstract»
Actor
«abstract»
Event
Location
«abstract»
Weather
«abstract»
NaturalDisaster
«abstract»
Animal
Plant
Earthquake
Snow
«abstract»
Prey
«abstract»
Predator
Rain
Fog
Antelope
Zebra
Lion
Tiger
Giraffe
Randomizer

```java
import java.util.Random;

/**
 * Provide control over the randomization of the simulation. By using the shared, fixed-seed
 * randomizer, repeated runs will perform exactly the same (which helps with testing). Set
 * 'useShared' to false to get different random behaviour every time.
 *
 * @author David J. Barnes and Michael Kölling
 * @version 2016.02.29
 */
public class Randomizer
{
    // The default seed for control of randomization.
    private static final int SEED = 1111;
    // A shared Random object, if required.
    private static final Random rand = new Random(SEED);
    // Determine whether a shared random generator is to be provided.
    private static final boolean useShared = true;

    /**
     * Constructor for objects of class Randomizer
     */
    public Randomizer()
    {
    }

    /**
     * Provide a random generator.
     * @return A random object.
     */
    public static Random getRandom()
    {
        if(useShared) {
            return rand;
        }
        else {
            return new Random();
        }
    }

    /**
     * Reset the randomization.
     * This will have no effect if randomization is not through
     * a shared Random generator.
     */
    public static void reset()
    {
        if(useShared) {
            rand.setSeed(SEED);
        }
    }
}

```

```java

/**
 * Snow is a weather event that stops the growth of plants
 *
 * @author Christian Impollonia and Samiira Mohamed
 * @version 2020.02.17
 */
public class Snow extends Weather
{
    // the probabilty that snow will occur
    private static final double EVENT_PROBABILITY = 0.18;
    //the string displayed on screen when snowing
    private String eventString = "Snowing";

     /**
     * Constructor for objects of class Rain. Calls to the weather superclass
     */
    public Snow(Field field)
    {
       super(field);
    }

    /**
     * returns the probabilty that snow will occur
     */
    public double getEventProbability()
    {
        return EVENT_PROBABILITY;
    }

    /**
     * stops the growth of plants
     */
    public void happen()
    {
        Simulator.depower();
    }

    /**
     * stops the effects of snow
     */
    public void stopHappening()
    {
        Simulator.modifierReset();
    }

   /**
     * returns the string displayed when snowing
     */
    public String getEventString()
    {
        return eventString;
    }

```

```
55
56 }
57
```

```java
import java.util.List;
import java.util.Iterator;
import java.util.Random;

/**
 * A simple model of a zebra.
 * Zebras age, move, breed, and die.
 *
 * @author David J. Barnes, Michael Kölling, Christian Impollonia and Samiira Mohamed
 * @version 2020.02.17
 */
public class Zebra extends Prey
{
    // Characteristics shared by all zebras (class variables).

    // The age at which a zebra can start to breed.
    private static final int BREEDING_AGE = 5;
    // The age to which a zebra can live.
    private static final int MAX_AGE = 40;
    // The likelihood of a zebra breeding.
    private static final double BREEDING_PROBABILITY = 0.32;
    // The maximum number of births.
    private static final int MAX_LITTER_SIZE = 4;
    //The food value of a zebra
    private static final int FOOD_VALUE = 9;
    //The probability that a zebra is created
    private static final double CREATION_PROBABILITY = 0.08;


    // A shared random number generator to control breeding.
    private static final Random rand = Randomizer.getRandom();

    // Individual characteristics (instance fields).

    // The zebra's age.
    private int age;
    private int count;
    /**
     * Create a new zebra. A zebra may be created with age
     * zero (a new born) or with a random age.
     *
     * @param randomAge If true, the zebra will have a random age.
     * @param field The field currently occupied.
     * @param location The location within the field.
     */
    public Zebra(boolean randomAge, Field field, Location location)
    {
        super(field, location);
        age = 0;
        if(randomAge) {
            age = rand.nextInt(MAX_AGE);
            setRandomFoodLevel();
        }
```

```
        else {
            age = 0;
            setMaxFoodLevel();
        }
        count = getDeathCount();
    }

    /**
     * This is what the zebra does most of the time - it runs
     * around and eats plants. Sometimes it will breed or die of old age.
     * If diseased, it will last very little.
     * @param newZebras A list to return newly born zebras.
     */
    public void act(List<Actor> newZebras)
    {
        incrementAge();
        incrementHunger();
        if(isAlive()) {
            giveBirth(newZebras);
            // Try to move into a free location.
            Location newLocation = findFood();
            if(newLocation == null)    {
                    newLocation = getField().freeAdjacentLocation(getLocation());
                }
            if(newLocation != null) {
                setLocation(newLocation);
                if(getIsDiseased()) count--;
            }
            else {
                // Overcrowding.
                setDead();
            }
        }
        if(count==0) setDead();
    }

    /**
     * Increase the age.
     * This could result in the zebra's death.
     */
    private void incrementAge()
    {
        age++;
        if(age > MAX_AGE) {
            setDead();
        }
    }

    /**
     * Check whether or not this zebra is to give birth at this step.
     * New births will be made into free adjacent locations.
     * @param newZebras A list to return newly born zebras.
     */
    private void giveBirth(List<Actor> newZebras)
```

```
108     {
109         // New zebras are born into adjacent locations.
110         // Get a list of adjacent free locations.
111         Field field = getField();
112         List<Location> free = field.getFreeAdjacentLocations(getLocation());
113         int births = breed();
114         for(int b = 0; b < births && free.size() > 0; b++) {
115             Location loc = free.remove(0);
116             Zebra young = new Zebra(false, field, loc);
117             newZebras.add(young);
118         }
119     }
120
121     /**
122      * Generate a number representing the number of births,
123      * if it can breed. Only females will breed. If diseased, it will give its
    disease to the animal it's breeding with.
124      * @return The number of births (may be zero).
125      */
126     private int breed()
127     {
128         int births = 0;
129         if(canBreed() && getGender() == false && rand.nextDouble() <=
    BREEDING_PROBABILITY) {
130             getPartner().giveDisease();
131             births = rand.nextInt(MAX_LITTER_SIZE) + 1;
132         }
133         return births;
134     }
135
136
137     /**
138      * A zebra can breed if it has reached the breeding age and it has a suitable
    partner near it.
139      * @return true if the zebra can breed, false otherwise.
140      */
141     private boolean canBreed()
142     {
143         if((age >= BREEDING_AGE) && isPartner())
144             return true;
145             else
146                 return false;
147     }
148
149
150     /**
151      * Returns the food value of the zebra
152      */
153     public static final int getFoodValue()
154     {
155         return FOOD_VALUE;
156     }
157
158     /**
```

```
159         * Returns the probability this animal is created
160         */
161        public static final double getCreationProbability()
162        {
163            return CREATION_PROBABILITY;
164        }
165    }
166
```

```java

/**
 * This class represents an element that is part of the simulation which can be either an actor or an event
 *
 * @author Christian Impollonia and Samiira Mohamed
 * @version 2020.02.17
 */
public abstract class Element
{
    //The state of the field the element should be in
    private Field field;

    /**
     * Constructor for objects of class Element
     */
    public Element(Field field)
    {
        this.field = field;
    }

     /**
     * Return the element's field.
     */
    protected Field getField()
    {
        return field;
    }

    /**
     * Sets the field of the element to null
     */
    protected void setFieldNull()
    {
        field = null;
    }

}

```

```
1
2  /**
3   * A Natural Disaster is as event that can randomly occur. It occurs very rarely
   and has
4   * devastating effects on the animals or plants.
5   *
6   * @author Christian Impollonia and Samiira Mohamed
7   * @version 2020.02.17
8   */
9  public abstract class NaturalDisaster extends Event
10 {
11
12     /**
13      * Constructor for objects of class NaturalDisaster. It calls to the Event
       superclass
14      */
15     public NaturalDisaster(Field field)
16     {
17         super(field);
18
19     }
20
21 }
22
```

```
1  import java.awt.*;
2  import java.awt.event.*;
3  import javax.swing.*;
4  import java.util.LinkedHashMap;
5  import java.util.Map;
6
7  /**
8   * A graphical view of the simulation grid.
9   * The view displays a colored rectangle for each location
10  * representing its contents. It uses a default background color.
11  * Colors for each type of species can be defined using the
12  * setColor method.
13  *
14  * @author David J. Barnes and Michael Kölling, Christian Impollonia and Samiira
    Mohamed
15  * @version 2020.02.17
16  */
17 public class SimulatorView extends JFrame
18 {
19     // Colors used for empty locations.
20     private static final Color EMPTY_COLOR = Color.white;
21
22     // Color used for objects that have no defined color.
23     private static final Color UNKNOWN_COLOR = Color.gray;
24
25     private final String STEP_PREFIX = "Step: ";
26     private final String POPULATION_PREFIX = "Population: ";
27     private final String TIME_PREFIX = "Time: ";
28     private final String TIME_POSTFIX = ":00";
29     private final String WEATHER_PREFIX = "Weather: ";
30     private JLabel stepLabel, population, infoLabel, timeLabel, weatherLabel;
31     private FieldView fieldView;
32
33     // A map for storing colors for participants in the simulation
34     private Map<Class, Color> colors;
35     // A statistics object computing and storing simulation information
36     private FieldStats stats;
37
38     /**
39      * Create a view of the given width and height.
40      * @param height The simulation's height.
41      * @param width  The simulation's width.
42      */
43     public SimulatorView(int height, int width)
44     {
45         stats = new FieldStats();
46         colors = new LinkedHashMap<>();
47
48         setTitle("Predator-Prey Simulation");
49         stepLabel = new JLabel(STEP_PREFIX, JLabel.CENTER);
50         infoLabel = new JLabel("  ", JLabel.CENTER);
51         population = new JLabel(POPULATION_PREFIX, JLabel.CENTER);
52         timeLabel = new JLabel(TIME_PREFIX, JLabel.CENTER);
53         weatherLabel = new JLabel(WEATHER_PREFIX, JLabel.CENTER);
```

```
54
55          setLocation(100, 50);
56
57          fieldView = new FieldView(height, width);
58
59          Container contents = getContentPane();
60
61          JPanel infoPane = new JPanel(new BorderLayout());
62              infoPane.add(stepLabel, BorderLayout.EAST);
63              infoPane.add(timeLabel, BorderLayout.WEST);
64              infoPane.add(infoLabel, BorderLayout.CENTER);
65              infoPane.add(weatherLabel, BorderLayout.NORTH);
66          contents.add(infoPane, BorderLayout.NORTH);
67          contents.add(fieldView, BorderLayout.CENTER);
68          contents.add(population, BorderLayout.SOUTH);
69          pack();
70          setVisible(true);
71      }
72
73      /**
74       * Define a color to be used for a given class of animal.
75       * @param animalClass The animal's Class object.
76       * @param color The color to be used for the given class.
77       */
78      public void setColor(Class animalClass, Color color)
79      {
80          colors.put(animalClass, color);
81      }
82
83      /**
84       * Display a short information label at the top of the window.
85       */
86      public void setInfoText(String text)
87      {
88          infoLabel.setText(text);
89      }
90
91      /**
92       * @return The color to be used for a given class of animal.
93       */
94      private Color getColor(Class animalClass)
95      {
96          Color col = colors.get(animalClass);
97          if(col == null) {
98              // no color defined for this class
99              return UNKNOWN_COLOR;
100         }
101         else {
102             return col;
103         }
104     }
105
106     /**
107      * Show the current status of the field.
```

```
108     * @param step Which iteration step it is.
109     * @param field The field whose status is to be displayed.
110     */
111    public void showStatus(int step, Field field, int time, String weather)
112    {
113        if(!isVisible()) {
114            setVisible(true);
115        }
116
117        stepLabel.setText(STEP_PREFIX + step);
118        timeLabel.setText(TIME_PREFIX + time + TIME_POSTFIX);
119        weatherLabel.setText(WEATHER_PREFIX + weather);
120        stats.reset();
121
122        fieldView.preparePaint();
123
124        for(int row = 0; row < field.getDepth(); row++) {
125            for(int col = 0; col < field.getWidth(); col++) {
126                Object animal = field.getObjectAt(row, col);
127                if(animal != null) {
128                    stats.incrementCount(animal.getClass());
129                    fieldView.drawMark(col, row, getColor(animal.getClass()));
130                }
131                else {
132                    fieldView.drawMark(col, row, EMPTY_COLOR);
133                }
134            }
135        }
136        stats.countFinished();
137
138        population.setText(POPULATION_PREFIX +
stats.getPopulationDetails(field));
139        fieldView.repaint();
140    }
141
142    /**
143     * Determine whether the simulation should continue to run.
144     * @return true If there is more than one species alive.
145     */
146    public boolean isViable(Field field)
147    {
148        return stats.isViable(field);
149    }
150
151    /**
152     * Provide a graphical view of a rectangular field. This is
153     * a nested class (a class defined inside a class) which
154     * defines a custom component for the user interface. This
155     * component displays the field.
156     * This is rather advanced GUI stuff - you can ignore this
157     * for your project if you like.
158     */
159    private class FieldView extends JPanel
160    {
```

```
161     private final int GRID_VIEW_SCALING_FACTOR = 6;
162
163     private int gridWidth, gridHeight;
164     private int xScale, yScale;
165     Dimension size;
166     private Graphics g;
167     private Image fieldImage;
168
169     /**
170      * Create a new FieldView component.
171      */
172     public FieldView(int height, int width)
173     {
174         gridHeight = height;
175         gridWidth = width;
176         size = new Dimension(0, 0);
177     }
178
179     /**
180      * Tell the GUI manager how big we would like to be.
181      */
182     public Dimension getPreferredSize()
183     {
184         return new Dimension(gridWidth * GRID_VIEW_SCALING_FACTOR,
185                              gridHeight * GRID_VIEW_SCALING_FACTOR);
186     }
187
188     /**
189      * Prepare for a new round of painting. Since the component
190      * may be resized, compute the scaling factor again.
191      */
192     public void preparePaint()
193     {
194         if(! size.equals(getSize())) {  // if the size has changed...
195             size = getSize();
196             fieldImage = fieldView.createImage(size.width, size.height);
197             g = fieldImage.getGraphics();
198
199             xScale = size.width / gridWidth;
200             if(xScale < 1) {
201                 xScale = GRID_VIEW_SCALING_FACTOR;
202             }
203             yScale = size.height / gridHeight;
204             if(yScale < 1) {
205                 yScale = GRID_VIEW_SCALING_FACTOR;
206             }
207         }
208     }
209
210     /**
211      * Paint on grid location on this field in a given color.
212      */
213     public void drawMark(int x, int y, Color color)
214     {
```

```
215             g.setColor(color);
216             g.fillRect(x * xScale, y * yScale, xScale-1, yScale-1);
217         }
218
219         /**
220          * The field view component needs to be redisplayed. Copy the
221          * internal image to screen.
222          */
223         public void paintComponent(Graphics g)
224         {
225             if(fieldImage != null) {
226                 Dimension currentSize = getSize();
227                 if(size.equals(currentSize)) {
228                     g.drawImage(fieldImage, 0, 0, null);
229                 }
230                 else {
231                     // Rescale the previous image.
232                     g.drawImage(fieldImage, 0, 0, currentSize.width,
    currentSize.height, null);
233                 }
234             }
235         }
236     }
237 }
238
```

```java
import java.util.List;
import java.util.Iterator;
import java.util.Random;

/**
 * Write a description of class Tiger here.
 *
 * @author Christian Impollonia and Samiira Mohamed
 * @version 2020.02.17
 */
public class Tiger extends Predator
{
    // Characteristics shared by all tigers (class variables).

    // The age at which a tiger can start to breed.
    private static final int BREEDING_AGE = 20;
    // The age to which a tiger can live.
    private static final int MAX_AGE = 166;
    // The likelihood of a tiger breeding.
    private static final double BREEDING_PROBABILITY = 0.01;
    // The maximum number of births.
    private static final int MAX_LITTER_SIZE = 4;
    // The probability of a tiger being created
    private static final double CREATION_PROBABILITY = 0.04;
    // A shared random number generator to control breeding.
    private static final Random rand = Randomizer.getRandom();

    // Individual characteristics (instance fields).
    // The tiger's age.
    private int age;
    // The tiger's food level, which is increased by eating zebras.
    private int count;

    /**
     * Create a tiger. A tiger can be created as a new born (age zero
     * and not hungry) or with a random age and food level.
     *
     * @param randomAge If true, the tiger will have random age and hunger level.
     * @param field The field currently occupied.
     * @param location The location within the field.
     */

    /**
     * Constructor for objects of class Tiger
     */
    public Tiger(boolean randomAge, Field field, Location location)
    {
    super(field, location);
        if(randomAge) {
            age = rand.nextInt(MAX_AGE);
            setRandomFoodLevel();
        }
        else {
            age = 0;
```

```
            setMaxFoodLevel();
        }
        count = getDeathCount();
    }

    /**
     * This is what the tiger does most of the time: it hunts for
     * preys. In the process, it might breed, die of hunger,
     * or die of old age. If diseased, it will last very little.
     * @param field The field currently occupied.
     * @param newtigers A list to return newly born tigers.
     */
    public void act(List<Actor> newTigers)
    {
        incrementAge();
        incrementHunger();
        if(isAlive()) {
            giveBirth(newTigers);
            // Move towards a source of food if found.
            Location newLocation = findFood();
            if(newLocation == null) {
                // No food found - try to move to a free location.
                newLocation = getField().freeAdjacentLocation(getLocation());
            }
            // See if it was possible to move.
            if(newLocation != null) {
                setLocation(newLocation);
                if(getIsDiseased()) count--;
            }
            else {
                // Overcrowding.
                setDead();
            }
        }
        if(count==0) setDead();

    }

    /**
     * Increase the age. This could result in the tiger's death.
     */
    private void incrementAge()
    {
        age++;
        if(age > MAX_AGE) {
            setDead();
        }
    }


    /**
     * Check whether or not this tiger is to give birth at this step.
     * New births will be made into free adjacent locations.
```

```
109     * @param newtigers A list to return newly born tigers.
110     */
111    private void giveBirth(List<Actor> newTigers)
112    {
113        // New tigers are born into adjacent locations.
114        // Get a list of adjacent free locations.
115        Field field = getField();
116        List<Location> free = field.getFreeAdjacentLocations(getLocation());
117        int births = breed();
118        for(int b = 0; b < births && free.size() > 0; b++) {
119            Location loc = free.remove(0);
120            Tiger young = new Tiger(false, field, loc);
121            newTigers.add(young);
122        }
123    }
124
125    /**
126     * Generate a number representing the number of births,
127     * if it can breed. Only females will breed. If diseased, it will give its
    disease to the animal it's breeding with.
128     * @return The number of births (may be zero).
129     */
130    private int breed()
131    {
132        int births = 0;
133        if(canBreed() && getGender() == false && rand.nextDouble() <=
    BREEDING_PROBABILITY) {
134            getPartner().giveDisease();
135            births = rand.nextInt(MAX_LITTER_SIZE) + 1;
136        }
137        return births;
138    }
139
140      /**
141     * A tiger can breed if it has reached the breeding age and if a suitable
    partner is close.
142     * @return true if the tiger can breed, false otherwise.
143     */
144    private boolean canBreed()
145    {
146        if((age >= BREEDING_AGE) && isPartner())
147            return true;
148            else
149               return false;
150    }
151
152
153    /**
154     * Returns the probability this animal is created
155     */
156    public static final double getCreationProbability()
157    {
158        return CREATION_PROBABILITY;
159    }
```

```
160 }
161
```

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

/**
 * Represent a rectangular grid of field positions.
 * Each position is able to store a single animal.
 *
 * @author David J. Barnes and Michael Kölling, Christian Impollonia and Samiira Mohamed
 * @version 2020.02.17
 */
public class Field
{
    // A random number generator for providing random locations.
    private static final Random rand = Randomizer.getRandom();

    // The depth and width of the field.
    private int depth, width;
    // Storage for the animals.
    private Object[][] field;

    /**
     * Represent a field of the given dimensions.
     * @param depth The depth of the field.
     * @param width The width of the field.
     */
    public Field(int depth, int width)
    {
        this.depth = depth;
        this.width = width;
        field = new Object[depth][width];
    }

    /**
     * Empty the field.
     */
    public void clear()
    {
        for(int row = 0; row < depth; row++) {
            for(int col = 0; col < width; col++) {
                field[row][col] = null;
            }
        }
    }

    /**
     * Clear the given location.
     * @param location The location to clear.
     */
    public void clear(Location location)
    {
```

```
54          field[location.getRow()][location.getCol()] = null;
55      }
56
57      /**
58       * Place an animal at the given location.
59       * If there is already an animal at the location it will
60       * be lost.
61       * @param animal The animal to be placed.
62       * @param row Row coordinate of the location.
63       * @param col Column coordinate of the location.
64       */
65      public void place(Object animal, int row, int col)
66      {
67          place(animal, new Location(row, col));
68      }
69
70      /**
71       * Place an animal at the given location.
72       * If there is already an animal at the location it will
73       * be lost.
74       * @param animal The animal to be placed.
75       * @param location Where to place the animal.
76       */
77      public void place(Object animal, Location location)
78      {
79          field[location.getRow()][location.getCol()] = animal;
80      }
81
82      /**
83       * Return the animal at the given location, if any.
84       * @param location Where in the field.
85       * @return The animal at the given location, or null if there is none.
86       */
87      public Object getObjectAt(Location location)
88      {
89          return getObjectAt(location.getRow(), location.getCol());
90      }
91
92      /**
93       * Return the animal at the given location, if any.
94       * @param row The desired row.
95       * @param col The desired column.
96       * @return The animal at the given location, or null if there is none.
97       */
98      public Object getObjectAt(int row, int col)
99      {
100         return field[row][col];
101     }
102
103     /**
104      * Generate a random location that is adjacent to the
105      * given location, or is the same location.
106      * The returned location will be within the valid bounds
107      * of the field.
```

```
108     * @param location The location from which to generate an adjacency.
109     * @return A valid location within the grid area.
110     */
111    public Location randomAdjacentLocation(Location location)
112    {
113        List<Location> adjacent = adjacentLocations(location);
114        return adjacent.get(0);
115    }
116
117    /**
118     * Get a shuffled list of the free adjacent locations.
119     * @param location Get locations adjacent to this.
120     * @return A list of free adjacent locations.
121     */
122    public List<Location> getFreeAdjacentLocations(Location location)
123    {
124        List<Location> free = new LinkedList<>();
125        List<Location> adjacent = adjacentLocations(location);
126        for(Location next : adjacent) {
127            if(getObjectAt(next) == null) {
128                free.add(next);
129            }
130        }
131        return free;
132    }
133
134    /**
135     * Try to find a free location that is adjacent to the
136     * given location. If there is none, return null.
137     * The returned location will be within the valid bounds
138     * of the field.
139     * @param location The location from which to generate an adjacency.
140     * @return A valid location within the grid area.
141     */
142    public Location freeAdjacentLocation(Location location)
143    {
144        // The available free ones.
145        List<Location> free = getFreeAdjacentLocations(location);
146        if(free.size() > 0) {
147            return free.get(0);
148        }
149        else {
150            return null;
151        }
152    }
153
154    /**
155     * Return a shuffled list of locations adjacent to the given one.
156     * The list will not include the location itself.
157     * All locations will lie within the grid.
158     * @param location The location from which to generate adjacencies.
159     * @return A list of locations adjacent to that given.
160     */
161    public List<Location> adjacentLocations(Location location)
```

```
162     {
163         assert location != null : "Null location passed to adjacentLocations";
164         // The list of locations to be returned.
165         List<Location> locations = new LinkedList<>();
166         if(location != null) {
167             int row = location.getRow();
168             int col = location.getCol();
169             for(int roffset = -1; roffset <= 1; roffset++) {
170                 int nextRow = row + roffset;
171                 if(nextRow >= 0 && nextRow < depth) {
172                     for(int coffset = -1; coffset <= 1; coffset++) {
173                         int nextCol = col + coffset;
174                         // Exclude invalid locations and the original location.
175                         if(nextCol >= 0 && nextCol < width && (roffset != 0 ||
    coffset != 0)) {
176                             locations.add(new Location(nextRow, nextCol));
177                         }
178                     }
179                 }
180             }
181 
182             // Shuffle the list. Several other methods rely on the list
183             // being in a random order.
184             Collections.shuffle(locations, rand);
185         }
186         return locations;
187     }
188 
189     /**
190      * Return the depth of the field.
191      * @return The depth of the field.
192      */
193     public int getDepth()
194     {
195         return depth;
196     }
197 
198     /**
199      * Return the width of the field.
200      * @return The width of the field.
201      */
202     public int getWidth()
203     {
204         return width;
205     }
206 
207     /**
208      * Return a shuffled list of locations adjacent to the given one.
209      * The list will not include the location itself.
210      * All locations will lie within the grid.
211      * @param location The location from which to generate adjacencies.
212      * @return A list of locations adjacent to that given.
213      */
214     public List<Location> closeLocations(Location location)
```

```
215     {
216         assert location != null : "Null location passed to adjacentLocations";
217         // The list of locations to be returned.
218         List<Location> locations = new LinkedList<>();
219         if(location != null) {
220             int row = location.getRow();
221             int col = location.getCol();
222             for(int roffset = -20; roffset <= 20; roffset++) {
223                 int nextRow = row + roffset;
224                 if(nextRow >= 0 && nextRow < depth) {
225                     for(int coffset = -20; coffset <= 20; coffset++) {
226                         int nextCol = col + coffset;
227                         // Exclude invalid locations and the original location.
228                         if(nextCol >= 0 && nextCol < width && (roffset != 0 ||
    coffset != 0)) {
229                             locations.add(new Location(nextRow, nextCol));
230                         }
231                     }
232                 }
233             }
234 
235             // Shuffle the list. Several other methods rely on the list
236             // being in a random order.
237             Collections.shuffle(locations, rand);
238         }
239         return locations;
240     }
241 }
242 
```

```java
import java.util.List;
import java.util.Iterator;
import java.util.Random;

/**
 * A plant is a type of actor which can't move or breed. It randomly spawns throughout the field
 *
 * @author Christian Impollonia and Samiira Mohamed
 * @version 2020.02.17
 */
public class Plant extends Actor
{
    //The food value of a plant
    private static final int FOOD_VALUE = 15;
    // the probabilty of a plant being created when the simulation is first started
    private static final double CREATION_PROBABILITY = 0.08;
    // the probability that a plant will grow during the day
    private static final double DAY_REPRODUCTION_PROBABILITY = 0.06;
    // the probability that a plant will grow during the night
    private static final double NIGHT_REPRODUCTION_PROBABILITY = 0.004;
    // the max number of years a plant can live
    private static final int MAX_AGE = 80;


    // A shared random number generator
    private static final Random rand = Randomizer.getRandom();

    //The age of the plant at the moment
    private int age;



    /**
     * Constructor for objects of class Plant. If the plants are created with the simulator, they are assigned a random age. Else (they are newborn), they are created at age 0.
     */
    public Plant (boolean randomAge, Field field, Location location)
    {
        super(field, location);
         age = 0;
        if(randomAge) {
            age = rand.nextInt(MAX_AGE);
        }
        else {
            age = 0;
        }
    }

    /**
     * Make this plant act - that is: age
     */
```

```
51      public void act(List<Actor> newPlants)
52      {
53          incrementAge();
54      }
55
56      /**
57       * Increase the age. This could result in the plant's death.
58       */
59      private void incrementAge()
60      {
61          age++;
62          if(age > MAX_AGE) {
63              setDead();
64          }
65      }
66
67      /**
68       * Returns the food value of the plant
69       */
70      public static final int getFoodValue()
71      {
72          return FOOD_VALUE;
73      }
74
75      /**
76       * Returns the probability this plant is created
77       */
78      public static final double getCreationProbability()
79      {
80          return CREATION_PROBABILITY;
81      }
82
83      /**
84       * Returns the probability this plant grows during the day
85       */
86      public static final double getDayReproductionProbability()
87      {
88          return DAY_REPRODUCTION_PROBABILITY;
89      }
90
91      /**
92       * Returns the probability this plant grows during the night
93       */
94      public static final double getNightReproductionProbability()
95      {
96          return NIGHT_REPRODUCTION_PROBABILITY;
97      }
98
99
100
101 }
102
```

```java

/**
 * Rain is a weather event that increases the growth of plants
 *
 * @author Christian Impollonia and Samiira Mohamed
 * @version 2020.02.17
 */
public class Rain extends Weather
{
    // the probability that rain will occur
    private static final double EVENT_PROBABILITY = 0.33;
    // the string displayed on screeen during rain
    private String eventString = "Raining";

     /**
     * Constructor for objects of class Rain. Calls to the Weather superclass
     */
    public Rain(Field field)
    {
        super(field);
    }

    /**
     * returns the probabilty of rain occuring
     */
    public double getEventProbability()
    {
        return EVENT_PROBABILITY;
    }

    /**
     * increases the growth of plants
     */
    public void happen()
    {
        Simulator.boost();
    }

    /**
     * stops the effect of rain
     */
    public void stopHappening()
    {
        Simulator.modifierReset();
    }

    /**
     * returns the displayed String for rain
     */
    public String getEventString()
    {
        return eventString;
    }

```

```
55
56
57 }
58
```

```java
import java.awt.Color;

/**
 * Provide a counter for a participant in the simulation.
 * This includes an identifying string and a count of how
 * many participants of this type currently exist within
 * the simulation.
 *
 * @author David J. Barnes and Michael Kölling
 * @version 2016.02.29
 */
public class Counter
{
    // A name for this type of simulation participant
    private String name;
    // How many of this type exist in the simulation.
    private int count;

    /**
     * Provide a name for one of the simulation types.
     * @param name  A name, e.g. "Fox".
     */
    public Counter(String name)
    {
        this.name = name;
        count = 0;
    }

    /**
     * @return The short description of this type.
     */
    public String getName()
    {
        return name;
    }

    /**
     * @return The current count for this type.
     */
    public int getCount()
    {
        return count;
    }

    /**
     * Increment the current count by one.
     */
    public void increment()
    {
        count++;
    }

    /**
     * Reset the current count to zero.
```

```
55          */
56         public void reset()
57         {
58             count = 0;
59         }
60 }
61
```

```java

/**
 * An earthquake is a natural disaster that kills most animals. It has a limited effect on plants.
 *
 * @author Christian Impollonia and Samiira Mohamed
 * @version 2020.02.17
 */
public class Earthquake extends NaturalDisaster
{
    //the probabilty that an earthquake will occur
    private static final double EVENT_PROBABILITY = 0.0005;
    // an earthquake occuring is initialized as false. There is a chance it can become true
    private boolean isOccuring = false;
    //the probability of an earthquake killing an animal
    private static final double KILLING_PROBABILITY = 0.9;
    /**
     * Constructor for objects of class Earthquake. Calls to the NaturalDisaster superclass
     */
    public Earthquake(Field field)
    {
       super(field);
    }

    /**
     * returns the probabilty of an earthquake happening
     */
    public double getEventProbability()
    {
        return EVENT_PROBABILITY;
    }

    /**
     * the earthquake occuring becomes true
     */
    public void happen()
    {
        isOccuring = true;
    }

    /**
     * returns whether the eathquake is occuring or not
     */
    public boolean getIsOccuring()
    {
        return isOccuring;
    }

    /**
     * returns the probability of an earthquake killing an animal
     */
    public double getKillingProbability()
```

```
52      {
53          return KILLING_PROBABILITY;
54      }
55
56      /**
57       * stops the effects of the earthquake.
58       */
59      public void stopHappening()
60      {
61          isOccuring = false;
62      }
63
64  }
65
```

```java
import java.util.List;
import java.util.Iterator;
import java.util.Random;

/**
 * An actor is an element of the simulation which is graphically shown and acts.
 *
 * @author Christian Impollonia and Samiira Mohamed
 * @version 2020.02.17
 */
public abstract class Actor extends Element
{
    // Whether the animal is alive or not.
    private boolean alive;

    // The animal's position in the field.
    private Location location;

    /**
     * Constructor for objects of class Actor. When actors are created they are set as alive.
     */
    public Actor(Field field, Location location)
    {
        super(field);
        alive = true;
        setLocation(location);
    }

    /**
     * Make this actor act - that is: make it do
     * whatever it wants/needs to do.
     * @param newActors A list to receive newly born actors.
     */
    abstract public void act(List<Actor> newActors);


     /**
     * Check whether the Actor is alive or not.
     * @return true if the actor is still alive.
     */
    protected boolean isAlive()
    {
        return alive;
    }

    /**
     * Indicate that the actor is no longer alive.
     * It is removed from the field.
     */
    protected void setDead()
    {
        alive = false;
        if(location != null) {
```

```
            getField().clear(location);
            location = null;
            setFieldNull();
        }
    }

    /**
     * Return the actor's location.
     * @return The actor's location.
     */
    protected Location getLocation()
    {
        return location;
    }

    /**
     * Place the actor at the new location in the given field.
     * @param newLocation The actor's new location.
     */
    protected void setLocation(Location newLocation)
    {
        if(location != null) {
            getField().clear(location);
        }
        location = newLocation;
        getField().place(this, newLocation);
    }


}

```

```java
import java.util.List;
import java.util.Iterator;
import java.util.Random;

/**
 * A simple model of a lion.
 * lions age, move, eat zebras, and die.
 *
 * @author David J. Barnes, Michael Kölling, Christian Impollonia and Samiira Mohamed
 * @version 2020.02.17
 */
public class Lion extends Predator
{
    // Characteristics shared by all lions (class variables).

    // The age at which a lion can start to breed.
    private static final int BREEDING_AGE = 15;
    // The age to which a lion can live.
    private static final int MAX_AGE = 151;
    // The likelihood of a lion breeding.
    private static final double BREEDING_PROBABILITY = 0.015;
    // The maximum number of births.
    private static final int MAX_LITTER_SIZE = 2;
    // The probability of a lion being created
    private static final double CREATION_PROBABILITY = 0.02;
    // A shared random number generator to control breeding.
    private static final Random rand = Randomizer.getRandom();

    // Individual characteristics (instance fields).
    // The lion's age.
    private int age;
    // The lion's food level, which is increased by eating zebras.
    private int count;

    /**
     * Create a lion. A lion can be created as a new born (age zero
     * and not hungry) or with a random age and food level.
     *
     * @param randomAge If true, the lion will have random age and hunger level.
     * @param field The field currently occupied.
     * @param location The location within the field.
     */
    public Lion(boolean randomAge, Field field, Location location)
    {
        super(field, location);
        if(randomAge) {
            age = rand.nextInt(MAX_AGE);
            setRandomFoodLevel();
        }
        else {
            age = 0;
            setMaxFoodLevel();
        }
```

```
54          count = getDeathCount();
55      }
56
57      /**
58       * This is what the lion does most of the time: it hunts for
59       * preys. In the process, it might breed, die of hunger,
60       * or die of old age. If diseased, it will last very little.
61       * @param field The field currently occupied.
62       * @param newlions A list to return newly born lions.
63       */
64      public void act(List<Actor> newLions)
65      {
66          incrementAge();
67          incrementHunger();
68          if(isAlive()) {
69              giveBirth(newLions);
70              // Move towards a source of food if found.
71              Location newLocation = findFood();
72              if(newLocation == null) {
73                  // No food found - try to move to a free location.
74                  newLocation = getField().freeAdjacentLocation(getLocation());
75              }
76              // See if it was possible to move.
77              if(newLocation != null) {
78                  setLocation(newLocation);
79                  if(getIsDiseased()) count--;
80              }
81              else {
82                  // Overcrowding.
83                  setDead();
84              }
85          }
86          if(count==0) setDead();
87
88      }
89
90      /**
91       * Increase the age. This could result in the lion's death.
92       */
93      private void incrementAge()
94      {
95          age++;
96          if(age > MAX_AGE) {
97              setDead();
98          }
99      }
100
101
102
103     /**
104      * Check whether or not this lion is to give birth at this step.
105      * New births will be made into free adjacent locations.
106      * @param newlions A list to return newly born lions.
107      */
```

```
108     private void giveBirth(List<Actor> newLions)
109     {
110         // New lions are born into adjacent locations.
111         // Get a list of adjacent free locations.
112         Field field = getField();
113         List<Location> free = field.getFreeAdjacentLocations(getLocation());
114         int births = breed();
115         for(int b = 0; b < births && free.size() > 0; b++) {
116             Location loc = free.remove(0);
117             Lion young = new Lion(false, field, loc);
118             newLions.add(young);
119         }
120     }
121
122     /**
123      * Generate a number representing the number of births,
124      * if it can breed. Only females will breed. If diseased, it will give its
    disease to the animal it's breeding with.
125      * @return The number of births (may be zero).
126      */
127     private int breed()
128     {
129         int births = 0;
130         if(canBreed() && getGender() == false && rand.nextDouble() <=
    BREEDING_PROBABILITY) {
131             getPartner().giveDisease();
132             births = rand.nextInt(MAX_LITTER_SIZE) + 1;
133         }
134         return births;
135     }
136
137     /**
138      * A lion can breed if it has reached the breeding age and if a suitable
    partner is close.
139      * @return true if the lion can breed, false otherwise.
140      */
141     private boolean canBreed()
142     {
143         if((age >= BREEDING_AGE) && isPartner())
144             return true;
145             else
146                 return false;
147     }
148
149
150     /**
151      * Returns the probability this animal is created
152      */
153     public static final double getCreationProbability()
154     {
155         return CREATION_PROBABILITY;
156     }
157 }
158
```

```
1
2  /**
3   * Fog is a weather event that makes Predators see less.
4   *
5   * @author Christian Impollonia and Samiira Mohamed
6   * @version 2020.02.17
7   */
8  public class Fog extends Weather
9  {
10     //the probabilty that fog will occur
11     private static final double EVENT_PROBABILITY = 0.25;
12     //the string displayed on screen during fog
13     private String eventString = "Foggy";
14     //the probabilty that predators will find prey. it is initialised at 1
15     // and halved when foggy
16     private static double fogDepower = 1;
17
18
19     /**
20      * Constructor for objects of class Fog
21      */
22     public Fog(Field field)
23     {
24         super(field);
25     }
26
27     /**
28      * retuns the probabilty that fog will occur
29      */
30     public double getEventProbability()
31     {
32         return EVENT_PROBABILITY;
33     }
34
35     /**
36      * returns the string displayed when weather is foggy
37      */
38     public String getEventString()
39     {
40         return eventString;
41     }
42
43     /**
44      * reduces the probablity of a predator finding and killing a prey
45      */
46     public void happen()
47     {
48         fogDepower = 0.5;
49     }
50
51     /**
52      * stops the effects of fog
53      */
54     public void stopHappening()
```

```
55      {
56          fogDepower = 1;
57      }
58
59      /**
60       * returns the value of fog depower
61       */
62      public static double getFogDepower()
63      {
64          return fogDepower;
65      }
66
67  }
68
```

```java
import java.awt.Color;
import java.util.HashMap;

/**
 * This class collects and provides some statistical data on the state
 * of a field. It is flexible: it will create and maintain a counter
 * for any class of object that is found within the field.
 *
 * @author David J. Barnes and Michael Kölling
 * @version 2016.02.29
 */
public class FieldStats
{
    // Counters for each type of entity (fox, rabbit, etc.) in the simulation.
    private HashMap<Class, Counter> counters;
    // Whether the counters are currently up to date.
    private boolean countsValid;

    /**
     * Construct a FieldStats object.
     */
    public FieldStats()
    {
        // Set up a collection for counters for each type of animal that
        // we might find
        counters = new HashMap<>();
        countsValid = true;
    }

    /**
     * Get details of what is in the field.
     * @return A string describing what is in the field.
     */
    public String getPopulationDetails(Field field)
    {
        StringBuffer buffer = new StringBuffer();
        if(!countsValid) {
            generateCounts(field);
        }
        for(Class key : counters.keySet()) {
            Counter info = counters.get(key);
            buffer.append(info.getName());
            buffer.append(": ");
            buffer.append(info.getCount());
            buffer.append(' ');
        }
        return buffer.toString();
    }

    /**
     * Invalidate the current set of statistics; reset all
     * counts to zero.
     */
    public void reset()
```

```
55      {
56          countsValid = false;
57          for(Class key : counters.keySet()) {
58              Counter count = counters.get(key);
59              count.reset();
60          }
61      }
62
63      /**
64       * Increment the count for one class of animal.
65       * @param animalClass The class of animal to increment.
66       */
67      public void incrementCount(Class animalClass)
68      {
69          Counter count = counters.get(animalClass);
70          if(count == null) {
71              // We do not have a counter for this species yet.
72              // Create one.
73              count = new Counter(animalClass.getName());
74              counters.put(animalClass, count);
75          }
76          count.increment();
77      }
78
79      /**
80       * Indicate that an animal count has been completed.
81       */
82      public void countFinished()
83      {
84          countsValid = true;
85      }
86
87      /**
88       * Determine whether the simulation is still viable.
89       * I.e., should it continue to run.
90       * @return true If there is more than one species alive.
91       */
92      public boolean isViable(Field field)
93      {
94          // How many counts are non-zero.
95          int nonZero = 0;
96          if(!countsValid) {
97              generateCounts(field);
98          }
99          for(Class key : counters.keySet()) {
100             Counter info = counters.get(key);
101             if(info.getCount() > 0) {
102                 nonZero++;
103             }
104         }
105         return nonZero > 1;
106     }
107
108     /**
```

```
109     * Generate counts of the number of foxes and rabbits.
110     * These are not kept up to date as foxes and rabbits
111     * are placed in the field, but only when a request
112     * is made for the information.
113     * @param field The field to generate the stats for.
114     */
115    private void generateCounts(Field field)
116    {
117        reset();
118        for(int row = 0; row < field.getDepth(); row++) {
119            for(int col = 0; col < field.getWidth(); col++) {
120                Object animal = field.getObjectAt(row, col);
121                if(animal != null) {
122                    incrementCount(animal.getClass());
123                }
124            }
125        }
126        countsValid = true;
127    }
128 }
129
```

```java
import java.util.Iterator;
import java.util.List;
import java.util.Random;

/**
 * A simple model of a giraffe.
 * Giraffes age, move, breed, and die.
 *
 * @author David J. Barnes, Michael Kölling, Christian Impollonia and Samiira Mohamed
 * @version 2020.02.17
 */
public class Giraffe extends Prey
{
    // Characteristics shared by all giraffes (class variables).

    // The age at which a giraffe can start to breed.
    private static final int BREEDING_AGE = 7;
    // The age to which a giraffe can live.
    private static final int MAX_AGE = 44;
    // The likelihood of a giraffe breeding.
    private static final double BREEDING_PROBABILITY = 0.37;
    // The maximum number of births.
    private static final int MAX_LITTER_SIZE = 5;
    //The food value of a giraffe
    private static final int FOOD_VALUE = 10;
    //The probability that a giraffe is created
    private static final double CREATION_PROBABILITY = 0.09;

    // A shared random number generator to control breeding.
    private static final Random rand = Randomizer.getRandom();

    // Individual characteristics (instance fields).

    // The giraffe's age.
    private int age;
    private int count;

    /**
     * Create a new giraffe. A giraffe may be created with age
     * zero (a new born) or with a random age.
     *
     * @param randomAge If true, the giraffe will have a random age.
     * @param field The field currently occupied.
     * @param location The location within the field.
     */
    public Giraffe(boolean randomAge, Field field, Location location)
    {
        super(field, location);
        age = 0;
        if(randomAge) {
            age = rand.nextInt(MAX_AGE);
            setRandomFoodLevel();
        }
```

```
54          else {
55
56              setMaxFoodLevel();
57          }
58          count = getDeathCount();
59      }
60
61      /**
62       * This is what the giraffe does most of the time - it runs
63       * around and eats plants. Sometimes it will breed or die of old age.
64       * If diseased, it will last very little.
65       * @param newGiraffes A list to return newly born giraffes.
66       */
67      public void act(List<Actor> newGiraffes)
68      {
69          incrementAge();
70          incrementHunger();
71          if(isAlive()) {
72              giveBirth(newGiraffes);
73              // Try to move into a free location.
74              Location newLocation = findFood();
75              if(newLocation == null)   {
76                      newLocation = getField().freeAdjacentLocation(getLocation());
77                  }
78              if(newLocation != null) {
79                  setLocation(newLocation);
80                  if(getIsDiseased()) count--;
81              }
82              else {
83                  // Overcrowding.
84                  setDead();
85              }
86          }
87          if(count==0) setDead();
88
89      }
90
91      /**
92       * Increase the age.
93       * This could result in the giraffe's death.
94       */
95      private void incrementAge()
96      {
97          age++;
98          if(age > MAX_AGE) {
99              setDead();
100         }
101     }
102
103     /**
104      * Check whether or not this giraffe is to give birth at this step.
105      * New births will be made into free adjacent locations.
106      * @param newGiraffes A list to return newly born giraffes.
107      */
```

```
108     private void giveBirth(List<Actor> newGiraffes)
109     {
110         // New giraffes are born into adjacent locations.
111         // Get a list of adjacent free locations.
112         Field field = getField();
113         List<Location> free = field.getFreeAdjacentLocations(getLocation());
114         int births = breed();
115         for(int b = 0; b < births && free.size() > 0; b++) {
116             Location loc = free.remove(0);
117             Giraffe young = new Giraffe(false, field, loc);
118             newGiraffes.add(young);
119         }
120     }
121
122     /**
123      * Generate a number representing the number of births,
124      * if it can breed. Only females will breed. If diseased, it will give its
disease to the animal it's breeding with.
125      * @return The number of births (may be zero).
126      */
127     private int breed()
128     {
129         int births = 0;
130         if(canBreed() && getGender() == false && rand.nextDouble() <=
BREEDING_PROBABILITY) {
131             getPartner().giveDisease();
132             births = rand.nextInt(MAX_LITTER_SIZE) + 1;
133         }
134         return births;
135     }
136
137     /**
138      * A giraffe can breed if it has reached the breeding age and if a suitable
partner is close.
139      * @return true if the zebra can breed, false otherwise.
140      */
141     private boolean canBreed()
142     {
143         if((age >= BREEDING_AGE) && isPartner())
144             return true;
145             else
146                 return false;
147     }
148
149
150     /**
151      * Returns the food value of the giraffe
152      */
153     public static final int getFoodValue()
154     {
155         return FOOD_VALUE;
156     }
157
158     /**
```

```
159         * Returns the probability this animal is created
160         
161        public static final double getCreationProbability()
162        {
163            return CREATION_PROBABILITY;
164        }
165    }
166    
```

```java
import java.util.List;
import java.util.Iterator;
import java.util.Random;

/**
 * A prey, an animal that is hunted by predators.
 *
 * @author Christian Impollonia and Samiira Mohamed
 * @version 2020.02.17
 */
public abstract class Prey extends Animal
{

    /**
     * Prey objects constructor, calls to the Animal superclass
     */
    public Prey(Field field, Location location)
    {
        super(field, location);
    }

    /**
     * This method has the prey look for food around it. If the item it finds is a plant, then it eats it and the food level of the Prey increases based on the food value of the plant.
     */
     protected Location findFood()
     {
         Field field = getField();
         List<Location> adjacent = field.adjacentLocations(getLocation());
         Iterator<Location> it = adjacent.iterator();
          while(it.hasNext()) {
             Location where = it.next();
             Object item = field.getObjectAt(where);
             if(item instanceof Plant) {
                 Plant plant = (Plant) item;
                 if(plant.isAlive()) {
                     plant.setDead();
                     foodLevel += plant.getFoodValue();
                     return where;
                 }

             }
     }
     return null;
    }

    /**
     * sets the initial food level of a prey when created. It's a number between 0 and twice the food value of a plant
     */
    protected void setRandomFoodLevel()
     {
         foodLevel = Randomizer.getRandom().nextInt(2*Plant.getFoodValue());
```

```
52      }
53
54      /**
55       * sets the food level of a new born prey. It is equal to twice the food
value of a plant
56       */
57      protected void setMaxFoodLevel()
58      {
59          foodLevel = 2*Plant.getFoodValue();
60      }
61
62
63  }
64
```

```java

/**
 * Weather is an event that can randomly occur during the simulation.
 * Weather changes frequently and can have small effects on animal and plant behaviour
 *
 * @author Christian Impollonia and Samiira Mohamed
 * @version 2020.02.17
 */
public abstract class Weather extends Event
{

    /**
     * Constructor for objects of class Weather. Calls to the Event superclass
     */
    public Weather(Field field)
    {
        super(field);
    }

    public abstract String getEventString();


}

```

```java
import java.util.List;
import java.util.Iterator;
import java.util.Random;

/**
 * A class representing shared characteristics of animals.
 *
 * @author David J. Barnes, Michael Kölling, Christian Impollonia and Samiira Mohamed
 * @version 2020.02.17
 */
public abstract class Animal extends Actor
{
    // The food level of the animal
    protected int foodLevel;
    // whether the animal is male or female
    private final boolean isMale;
    // whether the animal is diseased or not
    private boolean isDiseased;
    //the probablilty that an animal created is diseased
    private static final double DISEASE_PROBABILITY = 0.01;
    // the temporary variable that is used to store the animals partner when breeding
    private Animal partner;
    //the number of steps before an animal that is diseased dies
    private static final int DEATH_COUNT = 10;



    private static final Random rand = Randomizer.getRandom();

    /**
     * Create a new animal at location in field.
     *
     * When the animal is created, it has a 50/50 chance to be male or female.
     * It also has a very low chance to be diseased.
     * @param field The field currently occupied.
     * @param location The location within the field.
     */
    public Animal(Field field, Location location)
    {
        super(field, location);
        int genderValue = rand.nextInt(2);
        if(genderValue==1) isMale = true;
            else isMale = false;
        if(rand.nextDouble()<DISEASE_PROBABILITY)
            isDiseased = true;
        else isDiseased = false;
    }

    /**
     * Make this animal act - that is: make it do
     * whatever it wants/needs to do.
     * @param newAnimals A list to receive newly born animals.
```

```
53      */
54     abstract public void act(List<Actor> newAnimals);
55
56
57     /**
58      * Make this animal more hungry. This could result in the animal's death.
59      */
60     protected void incrementHunger()
61     {
62         foodLevel--;
63         if(foodLevel <= 0) {
64             setDead();
65         }
66     }
67
68
69     /**
70      * returns the animals gender
71      */
72     protected boolean getGender()
73     {
74         return isMale;
75     }
76
77     /**
78      * returns whether the animal can breed with another animal. It checks
79      * whether they are of opposite gender and the same species.
80      */
81     protected boolean isPartner()
82     {
83         Field field = getField();
84         List<Location> adjacent = field.closeLocations(getLocation());
85         Iterator<Location> it = adjacent.iterator();
86         while(it.hasNext()) {
87             Location where = it.next();
88             Object item = field.getObjectAt(where);
89             if(item instanceof Animal)
90             {
91                 Animal animal = (Animal) item;
92                 if((animal.getClass() == this.getClass()) && (animal.getGender()
!= getGender()))
93                 {
94                 partner = animal;
95                 return true;
96             }
97             }
98
99         }
100        return false;
101    }
102
103    /**
104     * returns the animal that a given animal has bred with
105     */
```

```
106     protected Animal getPartner()
107     {
108         return partner;
109     }
110
111     /**
112      * makes the animal diseased
113      */
114      public void giveDisease()
115      {
116          isDiseased = true;
117      }
118
119     /**
120      * returns the number of steps a diseased animal has left to live
121      */
122     protected int getDeathCount()
123      {
124          return DEATH_COUNT;
125      }
126
127     /**
128      * returns whether an animal is diseased or not
129      */
130     protected boolean getIsDiseased()
131     {
132         return isDiseased;
133     }
134
135     protected abstract Location findFood();
136
137 }
138
139
140
```

```java
import java.util.Iterator;
import java.util.List;
import java.util.Random;

/**
 * A simple model of an antelope.
 * Antelopes age, move, breed, and die.
 *
 * @author David J. Barnes, Michael Kölling, Christian Impollonia and Samiira Mohamed
 * @version 2020.02.17
 */
public class Antelope extends Prey
{
    // Characteristics shared by all antelopes (class variables).

    // The age at which a antelope can start to breed.
    private static final int BREEDING_AGE = 5;
    // The age to which an antelope can live.
    private static final int MAX_AGE = 32;
    // The likelihood of an antelope breeding.
    private static final double BREEDING_PROBABILITY = 0.335;
    // The maximum number of births.
    private static final int MAX_LITTER_SIZE = 4;
    //The food value of an antelope
    private static final int FOOD_VALUE = 8;
    //The probability that an antelope is created
    private static final double CREATION_PROBABILITY = 0.09;

    // A shared random number generator to control breeding.
    private static final Random rand = Randomizer.getRandom();

    // Individual characteristics (instance fields).

    // The antelope's age.
    private int age;
    private int count;

    /**
     * Create a new antelope. A antelope may be created with age
     * zero (a new born) or with a random age.
     *
     * @param randomAge If true, the antelope will have a random age.
     * @param field The field currently occupied.
     * @param location The location within the field.
     */
    public Antelope(boolean randomAge, Field field, Location location)
    {
        super(field, location);
        age = 0;
        if(randomAge) {
            age = rand.nextInt(MAX_AGE);
            setRandomFoodLevel();
        }
```

```
        else {
            age = 0;
            setMaxFoodLevel();
        }
        count = getDeathCount();
    }

    /**
     * This is what the antelope does most of the time - it runs
     * around and eats plants. Sometimes it will breed or die of old age.
     * If diseased, it will last very little.
     * @param newAntelopes A list to return newly born antelopes.
     */
    public void act(List<Actor> newAntelopes)
    {
        incrementAge();
        incrementHunger();
        if(isAlive()) {
            giveBirth(newAntelopes);
            // Try to move into a free location.
            Location newLocation = findFood();
            if(newLocation == null)    {
                    newLocation = getField().freeAdjacentLocation(getLocation());
                }
            if(newLocation != null) {
                setLocation(newLocation);
                if(getIsDiseased()) count--;
            }
            else {
                // Overcrowding.
                setDead();
            }
        }
        if(count==0) setDead();

    }

    /**
     * Increase the age.
     * This could result in the antelope's death.
     */
    private void incrementAge()
    {
        age++;
        if(age > MAX_AGE) {
            setDead();
        }
    }

    /**
     * Check whether or not this antelope is to give birth at this step.
     * New births will be made into free adjacent locations.
     * @param newAntelopes A list to return newly born antelopes.
     */
```

```
108     private void giveBirth(List<Actor> newAntelopes)
109     {
110         // New antelopes are born into adjacent locations.
111         // Get a list of adjacent free locations.
112         Field field = getField();
113         List<Location> free = field.getFreeAdjacentLocations(getLocation());
114         int births = breed();
115         for(int b = 0; b < births && free.size() > 0; b++) {
116             Location loc = free.remove(0);
117             Antelope young = new Antelope(false, field, loc);
118             newAntelopes.add(young);
119         }
120     }
121
122     /**
123      * Generate a number representing the number of births,
124      * if it can breed. Only females can breed.
125      * @return The number of births (may be zero).
126      */
127     private int breed()
128     {
129         int births = 0;
130         if(canBreed() && getGender() == false && rand.nextDouble() <=
BREEDING_PROBABILITY) {
131             getPartner().giveDisease();
132             births = rand.nextInt(MAX_LITTER_SIZE) + 1;
133         }
134         return births;
135     }
136
137     /**
138       * Aa antelope can breed if it has reached the breeding age and if it has a
suitable partner near it
139       * @return true if the zebra can breed, false otherwise.
140       */
141     private boolean canBreed()
142     {
143         if((age >= BREEDING_AGE) && isPartner())
144             return true;
145             else
146                 return false;
147     }
148
149
150     /**
151      * Returns the food value of the antelope
152      */
153     public static final int getFoodValue()
154     {
155         return FOOD_VALUE;
156     }
157
158     /**
159      * Returns the probability this animal is created
```

```
160         */
161         public static final double getCreationProbability()
162         {
163             return CREATION_PROBABILITY;
164         }
165 }
166
167
```

```java

/**
 * An event is an element of the simulation that randomly occurs
 *
 * @author Christian Impollonia and Samiira Mohamed
 * @version 2020.02.17
 */
public abstract class Event extends Element
{

    /**
     * Constructor for objects of class Events. Calls to the Element superclass
     */
    public Event(Field field)
    {
        super(field);
    }

    public abstract void happen();

    public abstract void stopHappening();

    public abstract double getEventProbability();


}

```

```java
import java.util.List;
import java.util.Iterator;
import java.util.Random;

/**
 * An animal that hunts preys
 *
 * @author Christian Impollonia and Samiira Mohamed
 * @version 2020.02.17
 */
public abstract class Predator extends Animal
{


    /**
     * Predator objects constructor. It calls to it's Animal superclass
     */
    public Predator(Field field, Location location)
    {
        super(field, location);
    }

    /**
     * Look for preys adjacent to the current location.
     * Only the first prey is eaten.
     * @return Where food was found, or null if it wasn't.
     */
    protected Location findFood()
    {
        Random rand = Randomizer.getRandom();
        Field field = getField();
        List<Location> adjacent = field.adjacentLocations(getLocation());
        Iterator<Location> it = adjacent.iterator();
        while(it.hasNext()) {
            Location where = it.next();
            Object animal = field.getObjectAt(where);
            if(animal instanceof Prey)  {

            if(animal instanceof Zebra) {
                Zebra zebra = (Zebra) animal;
                if(zebra.isAlive() && rand.nextDouble()<Fog.getFogDepower()){
                    zebra.setDead();
                    foodLevel += zebra.getFoodValue();
                    return where;
                }

            }
            else if(animal instanceof Giraffe) {
                Giraffe giraffe = (Giraffe) animal;
                if(giraffe.isAlive() && rand.nextDouble()<Fog.getFogDepower()) {
                    giraffe.setDead();
                    foodLevel += giraffe.getFoodValue();
                    return where;
                }
```

```
55                  }
56                  else if(animal instanceof Antelope) {
57                      Antelope antelope = (Antelope) animal;
58                      if(antelope.isAlive() && rand.nextDouble()<Fog.getFogDepower())
{
59                          antelope.setDead();
60                          foodLevel += antelope.getFoodValue();
61                          return where;
62                      }
63                  }
64              }
65
66              }
67              return null;
68      }
69
70      /**
71       * sets the foodlevel of a predator when created. It's a number between 0 and
the sum of the food value of all preys
72       */
73      protected void setRandomFoodLevel()
74      {
75          foodLevel =
Randomizer.getRandom().nextInt(Simulator.getTotalFoodValue());
76      }
77
78      /**
79       * sets the food level of a new born predator. It is equal to the sum of the
food value of all preys
80       */
81      protected void setMaxFoodLevel()
82      {
83          foodLevel = Simulator.getTotalFoodValue();
84      }
85
86
87
88
89  }
90
```

```java
import java.util.Random;
import java.util.List;
import java.util.ArrayList;
import java.util.Iterator;
import java.awt.Color;

/**
 * A predator-prey simulator, based on a rectangular field
 * containing animals in the Savannah
 *
 * @author David J. Barnes, Michael Kölling, Christian Impollonia and Samiira Mohamed
 * @version 2020.02.17
 */
public class Simulator
{
    // Constants representing configuration information for the simulation.
    // The default width for the grid.
    private static final int DEFAULT_WIDTH = 120;
    // The default depth of the grid.
    private static final int DEFAULT_DEPTH = 80;
    // The food value of all animals. A number between 0 and this is given to predators when created
    private static final int TOTAL_FOOD_VALUE = Zebra.getFoodValue() + Giraffe.getFoodValue() + Antelope.getFoodValue() ;

    // List of actors in the field.
    private List<Actor> actors;
    //List of the different types of weather
    private List<Weather> weatherEvents;
    // The current weather
    private Weather currentWeather;
    // Variable that stores the earthquake object
    private Earthquake earthquake;
    // The current state of the Field
    private Field field;
    // The current step of the simulation.
    private int step;
    // A graphical view of the simulation.
    private SimulatorView view;
    // A boolean that specifies if it is day or night
    private boolean isDay = false;
    // An integer that keeps track of the time
    private int time;
    //A modifier used in the creation of plants
    private static int modifier = 1;
    //The defaul weather string
    private String weather = "Clear";


    /**
     * Construct a simulation field with default size.
     */
    public Simulator()
```

```
52      {
53          this(DEFAULT_DEPTH, DEFAULT_WIDTH);
54      }
55
56      /**
57       * Create a simulation field with the given size.
58       * @param depth Depth of the field. Must be greater than zero.
59       * @param width Width of the field. Must be greater than zero.
60       */
61      public Simulator(int depth, int width)
62      {
63          if(width <= 0 || depth <= 0) {
64              System.out.println("The dimensions must be greater than zero.");
65              System.out.println("Using default values.");
66              depth = DEFAULT_DEPTH;
67              width = DEFAULT_WIDTH;
68          }
69
70          actors = new ArrayList<>();
71          weatherEvents = new ArrayList<>();
72          field = new Field(depth, width);
73
74          Rain rain = new Rain(field);
75          Fog fog = new Fog(field);
76          Snow snow = new Snow(field);
77
78          earthquake = new Earthquake(field);
79
80          // Create a view of the state of each location in the field.
81          view = new SimulatorView(depth, width);
82          view.setColor(Zebra.class, Color.ORANGE);
83          view.setColor(Lion.class, Color.BLUE);
84          view.setColor(Tiger.class, Color.MAGENTA);
85          view.setColor(Giraffe.class, Color.LIGHT_GRAY);
86          view.setColor(Antelope.class, Color.RED);
87          view.setColor(Plant.class, Color.GREEN);
88
89          //create an arraylist of the different types of events that can occur
90          weatherEvents.add(rain);
91          weatherEvents.add(fog);
92          weatherEvents.add(snow);
93
94          // Setup a valid starting point.
95          reset();
96      }
97
98      /**
99       * Run the simulation from its current state for a reasonably long period,
100      * (4000 steps).
101      */
102     public void runLongSimulation()
103     {
104         simulate(4000);
105     }
```

```
106
107     /**
108      * Run the simulation from its current state for the given number of steps.
109      * Stop before the given number of steps if it ceases to be viable.
110      * @param numSteps The number of steps to run for.
111      */
112     public void simulate(int numSteps)
113     {
114         for(int step = 1; step <= numSteps && view.isViable(field); step++) {
115             simulateOneStep();
116
117         }
118     }
119
120     /**
121      * Run the simulation from its current state for a single step.
122      * Iterate over the whole field updating the state of each
123      * actor
124      * Weather events or natural disasters can happen
125      */
126     public void simulateOneStep()
127     {
128         Random rand = Randomizer.getRandom();
129         step++;
130         time++;
131         if((step%24) == 0)
132             time = 0;
133         if(time==6)
134             isDay = true;
135         else if(time==18)
136             isDay = false;
137
138         if(rand.nextDouble()<earthquake.getEventProbability())
139         earthquake.happen();
140
141
142             //resets the current weather with a small chance of a weather event
    happening. It does it every 24 steps
143             if((step%24) == 0)
144             {   if(currentWeather != null) currentWeather.stopHappening();
145                 currentWeather =
    weatherEvents.get(rand.nextInt(weatherEvents.size()));
146             if (rand.nextDouble() < currentWeather.getEventProbability()) {
147                 currentWeather.happen();
148                 weather = currentWeather.getEventString();
149             }
150             else weather = "Clear";
151             }
152
153         // Provide space for newborn actors.
154         List<Actor> newActors = new ArrayList<>();
155         // Let all actors act.
156         for(int i=0; i<modifier; i++)
157         {
```

```
158             for(int row = 0; row < field.getDepth(); row++) {
159                 for(int col = 0; col < field.getWidth(); col++) {
160                     if((field.getObjectAt(row, col)) == null)
161                     {
162                         double plantProbability;
163                         if(isDay) plantProbability =
    Plant.getDayReproductionProbability();
164                         else    plantProbability =
    Plant.getNightReproductionProbability();
165                         if(rand.nextDouble() <= plantProbability) {
166                         Location location = new Location(row, col);
167                         Plant plant = new Plant(true, field, location);
168                         newActors.add(plant);
169                     }
170                 }
171             }
172         }
173         }
174          for(Iterator<Actor> it = actors.iterator(); it.hasNext(); ) {
175             Actor actor = it.next();
176             if(isDay)
177                actor.act(newActors);
178             if(earthquake.getIsOccuring())
179             {
180                 if(rand.nextDouble()<earthquake.getKillingProbability())
181                     actor.setDead();
182                 }
183             if(! actor.isAlive()) {
184                 it.remove();
185             }
186         }
187
188         if(!isDay)
189             delay(100);
190          earthquake.stopHappening();
191         // Add the newly born lions and zebras to the main lists.
192         actors.addAll(newActors);
193         view.showStatus(step, field, time, weather);
194     }
195
196     /**
197      * Reset the simulation to a starting position.
198      */
199     public void reset()
200     {
201         step = 0;
202         actors.clear();
203         populate();
204
205         // Show the starting state in the view.
206         view.showStatus(step, field, time, weather);
207     }
208
209     /**
```

```
210      * Randomly populate the field with animals and plants
211      */
212     private void populate()
213     {
214         Random rand = Randomizer.getRandom();
215         field.clear();
216         for(int row = 0; row < field.getDepth(); row++) {
217             for(int col = 0; col < field.getWidth(); col++) {
218                 if(rand.nextDouble() <= Lion.getCreationProbability()) {
219                     Location location = new Location(row, col);
220                     Lion lion = new Lion(true, field, location);
221                     actors.add(lion);
222                 }
223                 else if(rand.nextDouble() <= Zebra.getCreationProbability()) {
224                     Location location = new Location(row, col);
225                     Zebra zebra = new Zebra(true, field, location);
226                     actors.add(zebra);
227                 }
228                 else if(rand.nextDouble() <= Tiger.getCreationProbability()) {
229                     Location location = new Location(row, col);
230                     Tiger tiger = new Tiger(true, field, location);
231                     actors.add(tiger);
232                 }
233                 else if(rand.nextDouble() <= Giraffe.getCreationProbability()) {
234                     Location location = new Location(row, col);
235                     Giraffe giraffe = new Giraffe(true, field, location);
236                     actors.add(giraffe);
237                 }
238                 else if(rand.nextDouble() <= Antelope.getCreationProbability()) {
239                     Location location = new Location(row, col);
240                     Antelope antelope = new Antelope(true, field, location);
241                     actors.add(antelope);
242                 }
243                 else if(rand.nextDouble() <= Plant.getCreationProbability()) {
244                     Location location = new Location(row, col);
245                     Plant plant = new Plant(true, field, location);
246                     actors.add(plant);
247                     // else leave the location empty.
248                 }
249             }
250         }
251 }
252
253
254     /**
255      * Pause for a given time.
256      * @param millisec  The time to pause for, in milliseconds
257      */
258     private void delay(int millisec)
259     {
260         try {
261             Thread.sleep(millisec);
262         }
263         catch (InterruptedException ie) {
```

```
264             // wake up
265         }
266     }
267
268     /**
269      * returns the total food value of all preys
270      */
271     public static final int getTotalFoodValue()
272     {
273         return TOTAL_FOOD_VALUE;
274     }
275
276     /**
277      * alters the value of plants that grow, making it grow at double the rate
278      */
279     public static void boost()
280     {
281         modifier = 2;
282     }
283
284     /**
285      * stops the growth of plants
286      */
287     public static void depower()
288     {
289         modifier = 0;
290     }
291
292     /**
293      * resets the growth of plants to normal
294      */
295     public static void modifierReset()
296     {
297         modifier = 1;
298     }
299
300 }
301
```

```java
/**
 * Represent a location in a rectangular grid.
 *
 * @author David J. Barnes and Michael Kölling
 * @version 2016.02.29
 */
public class Location
{
    // Row and column positions.
    private int row;
    private int col;

    /**
     * Represent a row and column.
     * @param row The row.
     * @param col The column.
     */
    public Location(int row, int col)
    {
        this.row = row;
        this.col = col;
    }

    /**
     * Implement content equality.
     */
    public boolean equals(Object obj)
    {
        if(obj instanceof Location) {
            Location other = (Location) obj;
            return row == other.getRow() && col == other.getCol();
        }
        else {
            return false;
        }
    }

    /**
     * Return a string of the form row,column
     * @return A string representation of the location.
     */
    public String toString()
    {
        return row + "," + col;
    }

    /**
     * Use the top 16 bits for the row value and the bottom for
     * the column. Except for very big grids, this should give a
     * unique hash code for each (row, col) pair.
     * @return A hashcode for the location.
     */
    public int hashCode()
    {
```

```
55          return (row << 16) + col;
56      }
57
58      /**
59       * @return The row.
60       */
61      public int getRow()
62      {
63          return row;
64      }
65
66      /**
67       * @return The column.
68       */
69      public int getCol()
70      {
71          return col;
72      }
73
74
```

```
Project: Savannah Predator-Prey Simulation
Authors: Michael Kölling, David J. Barnes, Christian Impollonia and Samiira
Mohamed

A predator-prey simulation involving animals, plants and random events in
an enclosed rectangular field. It is set in the Savannah.

This version uses inheritance.

How to start:
    Create a Simulator object.
    Then call one of:
        + simulateOneStep - for a single step.
        + simulate - and supply a number (say 10) for that many steps.
        + runLongSimulation - for a simulation of 500 steps.

```