Syddansk Universitet
IMADA

18-12-2020
DS801

# INTRODUKTION TIL OBJEKTORIENTERET PROGRAMMERING:

Projekt 3/3 (Gruppe 25)

**Koordinator:** Stefan Randa

Sarah Pedersen

Simon Lorentzen

# Indhold

**1. Indledning** ..... **2**

**2. Kravsspecifikation** ..... **2**

**3. Implementering** ..... **3**

3.1. Colony ..... 3

3.2. Graph ..... 4

**4. Test** ..... **6**

4.1. Test af de udleverede klasser ..... 6

4.2. Test af de udviklede klasser ..... 8

**5. Konklusion** ..... **10**

**Referencer** ..... **10**

**Bilag I (Colony.java)** ..... **11**

**Bilag II (Graph.java)** ..... **12**

# 1. Indledning

Denne rapport dokumenterer den tredje og sidste fase i eksamensprojektet til kurset *Introduktion til objektorienteret programmering* (DS801). Kravene til projektet er specificeret i det kommende afsnit. Derefter beskrives nogle af de designbeslutninger, som projektgruppen har taget, og derpå uddybes implementeringen af relevante metoder. Til sidst foreligger et afsnit om testscenarier, en konklusion og relevante bilag.[1]

# 2. Kravsspecifikation

Projektgruppen skal i denne fase af projektet udarbejde to klasser (*Colony* og *Graph*), hvortil følgende krav er specificeret. For klassen *Colony*:

- En standardkonstruktør uden argumenter, som opretter en ny koloni med et tomt sukkerlager.
- En metode, **void topUp(int sugar)**, som øger koloniens sukkerlager med en given mængde.
- En metode, **void consume()**, som reducerer koloniens sukkerlager med én enhed.
- En metode, **boolean hasStock()**, som indikerer, om sukkerlageret i kolonien er ikke-tomt.

For klassen *Graph*:

- En konstruktør, som tager følgende argumenter: en bredde, en dybde, en array af kolonier samt sandsynligheden for, at en knude indeholder sukker til at begynde med og den gennemsnitlige mængde sukker i en sådan knude. Denne konstruktør opretter således en ny gittergraf med de specificerede dimensioner.
- En konstruktør, som tager følgende argumenter: et filnavn, en array of kolonier samt sandsynligheden for, at en knude indeholder sukker til at begynde med og den gennemsnitlige mængde sukker i en sådan knude. Denne konstruktør oprettet således en ny graf baseret på knude- og kantværdierne fra den angivne fil.
- En metode, **int pheromoneLevel(Node source, Node target)**, som returnerer niveauet af feromoner i kanten, der forbinder kilde- og målknuden i grafen.

[1] OBS: Formuleringerne i dette afsnit er delvist kopieret fra projektgruppens første og/eller anden rapport.

- En metode, **void raisePheromones(Node source, Node target, int amount)**, som niveauet af feromoner i kanten, , der forbinder kilde- og målknuden i grafen med en given mængde.
- En metode, **Node[] adjacentTo(Node node)**, som returnerer en array med alle de knuder, der støder op til en given knude i grafen.
- En metode, **void tick()**, som reducerer niveauet af feromoner i grafen med én enhed, og muligvis aflægger sukker i en tilfældig knude.

Samt en rapport, som inkluderer

- En beskrivelse af og eksempler på den implementerede kode.
- Relevante designbeslutninger.
- Scenarier, metoder og resultater fra test af koden (med dedikeret kode).
- Et bilag, som inkluderer kildekoden.

Disse tre filer skal afleveres i én samlet ZIP-fil uden undermapper.[2]

# 3. Implementering

Dette afsnit indeholder en forklaring af klassernes interne betydning samt en uddybende beskrivelse af den implementerede kode og/eller andre relevante overvejelser.

## 3.1. Colony

Denne klasse repræsenterer de knudepunkter, hvor de aktive kolonier befinder sig. Bemærk, at den enkelte koloni ikke opbevarer transportabelt sukker, men sukker til konsumption.

*Colony* er en subklasse, der nedarver fra *Node*. Nedarvningen initialiseres, idet subklassens konstruktør kalder superklassens konstruktør, men eftersom projektgruppen har valgt ikke at overskrive nogle attributter eller metoder, er det redundant at lave et eksplicit super-kald – compileren tilføjer nemlig dette kald implicit. Derudover implementeres tre simple metoder, som tilsammen styrer koloniens sukkerlager: **public void topUp(int sugar)** tilføjer sukker til koloniens lager fra indgående myrer, **public void consume()** fjerner én enhed af sukker fra koloniens lager, og **public boolean hasStock()** returnerer koloniens ikke-tomme sukkerlager.

[2] Formuleringerne i dette afsnit er delvist kopieret fra projektgruppens første og/eller anden rapport.

## 3.2. Graph

Denne klasse repræsenterer grafen, hvorpå myrerne bevæger sig i løbet af simulationen. Den udfylder en todimensionel array med instanser af *Node*, *Edge* og *Colony*, der fungerer som en tekstbaseret version af den tabel, *Visualizer* tegner op under eksekveringen. *Graph* kontrollerer således mængden af feromoner og sukker i kant- og knudeinstanserne samt forbruget af disse ressourcer over tid. Foruden de to konstruktører og fire metoder fra kravspecifikationen, har gruppen implementeret diverse hjælpermetoder (konsulter evt. bilaget for at se selve koden).

**public Graph(int width, int depth, Colony[] colonies, double sugarProbability, int sugarAverage):**
Denne konstruktør skaber en todimensionel array af knude- koloniinstanser med dimensioner dikteret af brugeren gennem kommandolinjegrænseflade, sandsynligheden for at sukker opstår tilfældigt i Node instanser over tid samt den gennemsnitlige mængde af sukker, der opstår tilfældigt.

**public Graph(String filename, Colony[] homes, double sugarProbability, int sugarAverage):**
Denne konstruktør skaber også en todimensionel array af knude- koloniinstanser, men ud fra specifikationer læst fra en tekstfil, og gør dette med forbehold for eventuelle formateringsfejl i den givne tekstfil, der læses fra.

**public int pheromoneLevel(Node source, Node target):**
Denne metoder er en getter, som henter feromonniveauet i en bestemt kantinstans mellem to knudeinstanser.

**public void raisePheromones(Node source, Node target, int amount):**
Denne metode øger mængden af feromoner i en bestemt kantinstans mellem to knudeinstanser.

**public Node[] adjacentTo(Node node):**
Denne metode tjekker hvilke knudeinstanser, der ligger ved siden af den, som kaldte denne metode i tabellen (baseret på knudearrayet).

**public void tick():**

Denne metode formindsker feromonniveauet i instanser af *Edge*, og beslutter tilfældigt, om sukker skal opstå i en vilkårlig knudeinstans.

**private Node[][] fillingInNodesTo2dArray(int width, int depth):**

Denne hjælpemetode tilføjer instanser af *Node* til knudearrayet, indtil alle celler er udfyldte.

**private void insertingColoniesToMap(Node[][] nodeGrid, Colony[] colonies, int width, int depth):**

Denne hjælpemetode konverterer et vilkårlig antal af knudeinstanser i celler af knudearrayet til instanser af *Colony*.

**private Edge[] createEdgesBetweenNodes(int width, int depth, Node[][] nodeGrid):**

Denne metode skaber en instans af *Edge* med forbindelse til to knudeinstanser i knudearrayet.

**private void spawnSugar():**

Denne hjælpemetode tilføjer sukker til en vilkårlig instans af *Node*.

**private boolean isPositionColony(Node[][] map, int widthPostion, int depthPosition):**

Denne hjælpemetode tjekker, om objektet i en specifik celle af knudearrayet er en instans af *Colony* eller ej.

**private void addColoniesToNodeArray(String[] colonyInformation, Node[] nodeLocation, Colony[] homes):**

Denne hjælpemetode tilføjer instanser af *Colony* knudearrayet.

**private void addNewNodesToEmptyIndexes(Node[] nodeLocation):**

Denne hjælpemetode udfylder tomme celler i knudearrayet.

**private void createEdges(Scanner filescanner, Node[] nodeLocation):**

Denne hjælpemetode skaber en instans af *Edge* forbundet til to knudeinstanser.

**private void checkDuplicateEdge(Edge edge, List<Edge> edges):**

Denne hjælpemetode tjekker om flere kantinstanser eksisterer mellem de samme knudeinstanser.

**private void addSugarToNodes(Node[] nodeLocation):**

Denne hjælpemetode øger mængden af sukker i en bestemt knudeinstans.

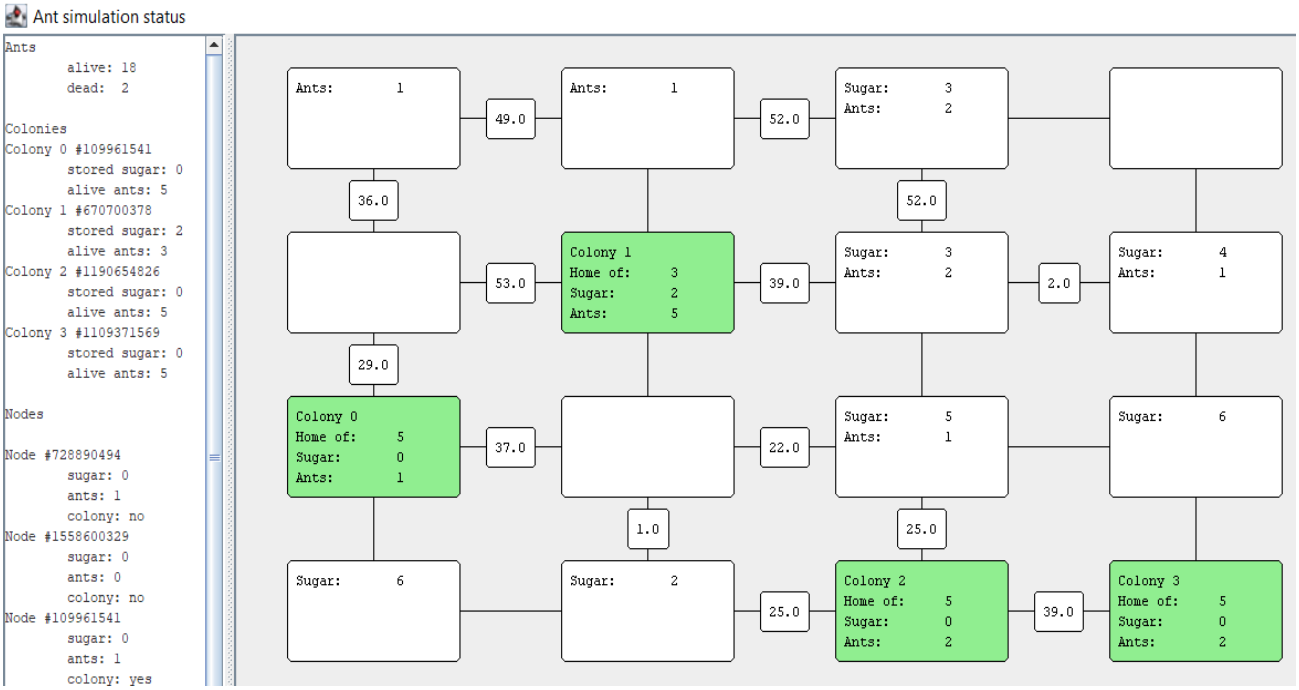
# 4. Test

I denne fase af projektet er testen blevet udført i to dele: en test med de udleverede klasser og en test med udviklede klasser. Eftersom test med printudsagn og test med dedikeret kode i form af modultest allerede er blevet adresseret i de to tidligere faser af projektet, vil den endelig test, som foreligger her. fokusere på programmet som helhed. Hvis en klasse fejler, vil det kunne ses i hele programmet. Tesen vil således lægge grundlaget for en vurdering af det udviklede programs funktionalitet ift. det udleverede program såvel som projektbeskrivelsen.

## 4.1. Test af de udleverede klasser

I dette afsnit testes *Graph* og *Colony* sammen med de udleverede klasser. Programmet opfører sig som forventet. Ved et lavt feromonniveau bevæger myrerne sig længere væk fra kolonien og vice versa. Stærke myrer kan bære mere sukker hjem, og kolonilageret stiger derfor drastisk, hvis der er meget på ruten. Nedenfor ses en eksekveret simulation af følgende argumenter:

| Efterspørgsler | Argumenter |
|---|---|
| What is the probability that a node will have sugar? | 0,5 |
| What is the average amount of sugar in a node with sugar? | 5 |
| How many units of sugar can an ant carry? | 3 |
| How many units of pheromones are dropped by ants as they move? | 2 |
| For how long should the simulation run? | 20 |
| Please insert the number of ant colonies: | 4 |
| How many ants live at colony 1 | 5 |
| How many ants live at colony 2 | 5 |
| How many ants live at colony 3 | 5 |
| How many ants live at colony 4 | 5 |
| What kind of graph do you want (1:grid / 2:read from a file)? | 1 |
| Please insert the width of the grid (at least 3): | 4 |
| Please insert the height of the grid (at least 3): | 4 |
| What simulation mode do you want (1:resumed / 2:visual)? | 2 |




Under testen blev følgende funktionalitet desuden identificeret i de udleveret klasser:

| **Beskrivelse** | **Resultat** |
|---|---|
| Ved brugerinput | Programmet fejler ved indtastning af engelsk decimalnotering, giver servicebesked ved negative værdier og fejler ved alle andre ulogiske indtastninger såsom a og !. |
| Fillæsning | Læsning af *graph1.txt* fejler som forventet, da den indeholder duplikater. Efter rettelse af filen (ved at fjerne duplikater) kører programmet som forventet. |
| Visualisering | Se eksemplet ovenfor. |

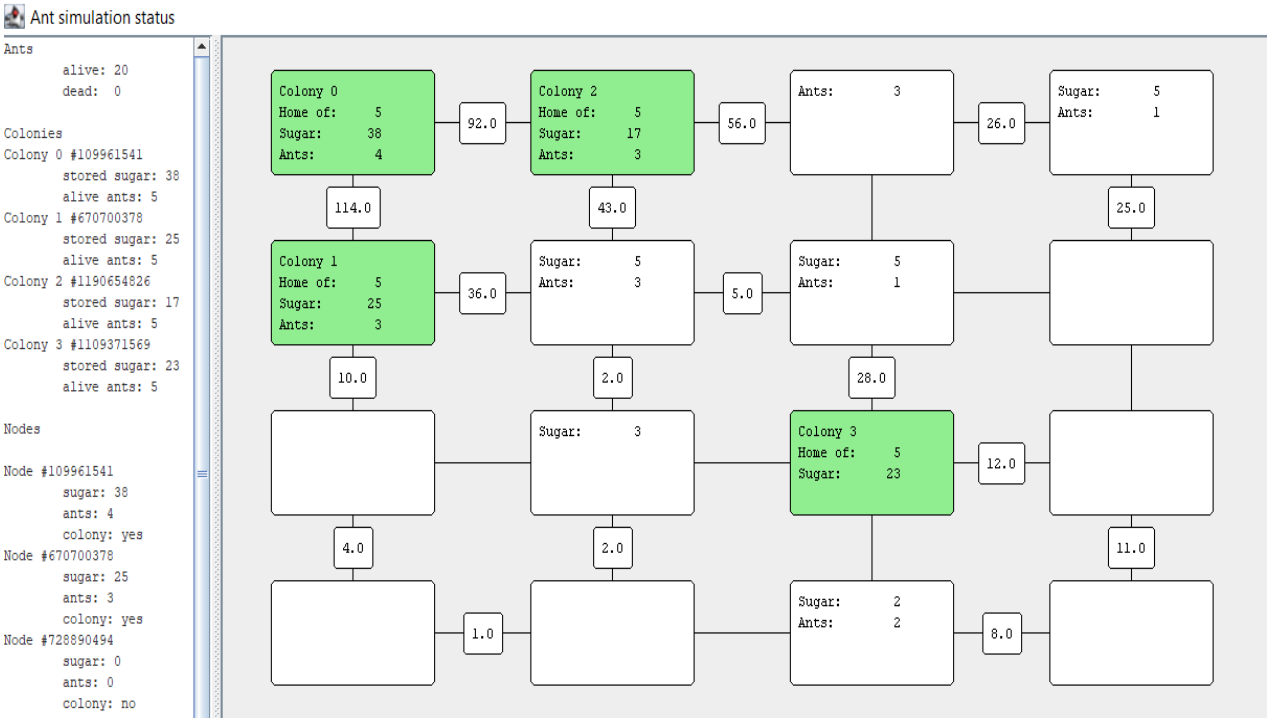
For at teste filindlæsningen med en fil, som lever op til kravet om ingen duplikater, blev filen *graph1.txt* fra første fase af projektet modificeret ved at slette linje 7, som indeholdte 2 1, hvilket allerede optrådte tidligere i filen. Efter sletningen kunne filen indlæses uden problemer.

## 4.2. Test af de udviklede klasser

I dette afsnit testes *Graph* og *Colony* sammen med de udviklede klasser. Under testen af de samtlige udviklede klasser blev der fundet en fejl i klassen *Simulator*. Modsat dokumentationen gav klasse den totale mængde sukker med til kolonien, når en myre lagde sukker, i stedet for kun at medgive den mængde sukker, som myren leverede til kolonien. Dette blev løst ved at fjerne sukkerkaldet i **dropSugar()** i *Simulator*. På billede nedenfor ses rettelsen:

```
76      public void dropSugar(Ant ant) {
77          if (ant != null) {
78              Colony home = ant.home();
79              if (ant.isAtHome() && ant.carrying()) {
80                  ant.dropSugar();
81                  // Corrected after submission
82                  home.topUp(sugarCapacity);
83                  // home.topUp(home.sugar() + sugarCapacity);
84              }
85          }
86      }
```

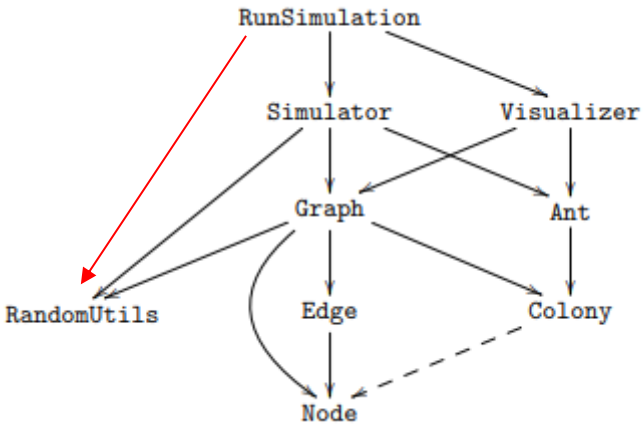
Her ses en eksekveret simulation for de udviklede klasser med de samme argumenter, som blev brugt i den forrige test:



Simulationen opfører sig som forventet jf. beskrivelsen i rapporten fra projektets anden fase

> *Simulationen er påvirket af argumenter fra brugeren. Dette ses bl.a. ved myrernes levetid: hvis der er mange myrer, lidt sukker og myrerne kan bære meget lidt sukker, så dør myrerne hurtigt. Hvis der derimod er få stærke myrer, som kan bære meget sukker, så lever myrerne længere. Hvis feromonniveauet er højt, forbliver myrerne tættere på kolonien, men hvis det er lavt, bevæger de sig længere væk fra kolonien. Størrelsen på grafen og antallet af kolonier påvirker, hvor hurtigt sukkeret indsamles, og dermed også hvor ofte, der bliver lagt nyt sukker på knuderne*. (s. 7)

Jf. projektbeskrivelsen har projektgruppen udviklet klasserne svarende til det fremviste klassediagram fra projektintroduktionen. *Graph* er klient af *RandomUtils*, *Edge* og *Colony*. Samtidig servicerer *Graph*, *Simulator* og *Visualizer*. *Colony* nedarver fra *Node*, og servicerer både klasserne *Graph* og *Ant*:




Som det ses på billedet, har projektgruppen yderligere valgt at lade *RunSimulation* benytte sig af biblioteket fra *RandomUtils* for at skabe ekstra brugervenlig funktionalitet i form af myreinput i intervaller. Dette var ikke inkluderet i projektbeskrivelsen, men det blev udviklet for at imødekomme simulationer med høje antal myreinput. En anden ekstra funktionalitet, som er udviklet for brugervenlighedens skyld, er den automatiske indlæsning af en standardfil, hvis brugeren ingen fil har. Dette har dog ikke krævet adgang til andre klasser. Testen viser at det udviklede program kan håndtere brugerargumenter, og eksekverer simulationer med forventet output – ligesom det udleverede program.

# 5. Konklusion

Kravspecifikationen til koden såvel som rapporten er opfyldt. Sammen med de udleverede klassefiler, eksekveres de udviklede klasser for den tredje og sidste fase af dette projekt problemfrit i kommandolinjegrænsefladen, hvor alle relevante argumenter og informationer bliver indsamlet og fremvist. Design og implementering er forklaret, og programmets funktionalitet er testet gennem flere scenarier. Derfor kan det konkluderes, at gruppen efterlever kravspecifikationerne.[3]

# Referencer

Foruden den udleverede klassedokumentation, har gruppen konsulteret lærebogen og Oracles officielle kodekonventioner efter behov:


- Oracle (1999). *Code Conventions for the Java™ Programming Language*. Lokaliseret den 01/12/2020: https://www.oracle.com/java/technologies/javase/codeconventions-contents.html.
- Savitch, W. (2019). *An Introduction to Problem Solving & Programming*. Harlow, UK: Pearson Education Limited.


[3] OBS: Formuleringerne i dette afsnit er delvist kopieret fra projektgruppens første og/eller anden rapport.

# Bilag I (Colony.java)

```java
/**
 * This class represents nodes of ant colonies, extends the class Node and
 * manages the colonies' sugar stock.
 */

public class Colony extends Node {

    /**
     * Default constructor creates a new colony with an empty sugar stock.
     */
    public Colony() {
    }

    /**
     * Increases the colony's sugar stock by a given amount.
     */
    public void topUp(int sugar) {
        setSugar(sugar() + sugar);
    }

    /**
     * Decreases the colony's sugar stock by one unit.
     */
    public void consume() {
        decreaseSugar();
    }

    /**
     * Returns a non-empty sugar stock in the colony.
     */
    public boolean hasStock() {
        return sugar() > 0;
    }
}
```

# Bilag II (Graph.java)

```
import java.io.File;
import java.util.Scanner;
import java.util.List;
import java.util.ArrayList;

/**
 * The Graph class creates and fills out a grid of Node, Edge and Colony
 * instances. Additionally, Graph handles the pheromone levels of Edge
 * instances, the amount of sugar in Node and Colony instances, as well as the
 * periodic reduction of pheromones in Edges.
 */
public class Graph {
    private double sugarProbability;
    private int sugarAverage;
    private Edge[] edges;

    /**
     * The first constructor creates a 2D array of Node and Colony instances with
     * user-specified variables for the probability and the amount of sugar being
     * added to Nodes randomly over time.
     */
    public Graph(int width, int depth, Colony[] colonies, double sugarProbability, int sugarAverage) {
        this.sugarProbability = sugarProbability;
        this.sugarAverage = sugarAverage;

        Node[][] nodeGrid = fillingInNodesTo2dArray(width, depth);
        insertingColoniesToMap(nodeGrid, colonies, width, depth);
        this.edges = createEdgesBetweenNodes(width, depth, nodeGrid);
    }

    /**
     * The second constructor creates a 2D array of Node and Colony instances from a
     * user-specified text file, while handling possible exception that might be
     * thrown by a malformed text file from the user.
     */
    public Graph(String filename, Colony[] homes, double sugarProbability, int sugarAverage) {
        this.sugarProbability = sugarProbability;
        this.sugarAverage = sugarAverage;

        try {
            Scanner filescanner = new Scanner(new File(filename));

            int amountOfNodes = filescanner.nextInt();
            filescanner.nextLine();
            Node[] nodeLocation = new Node[amountOfNodes];

            String[] colonyInformation = filescanner.nextLine().trim().split(" ");

            addColoniesToNodeArray(colonyInformation, nodeLocation, homes);
            addNewNodesToEmptyIndexes(nodeLocation);
            createEdges(filescanner, nodeLocation);
            addSugarToNodes(nodeLocation);
            filescanner.close();
        } catch (Throwable t) {
            // Use in case of debugging
            // t.printStackTrace();
            System.err.println("File is not well-formed, error:  " + t.getMessage());
            System.exit(-1);
        }
    }

    /**
     * Gets the amount of pheromones in a given Node instance.
     */
    public int pheromoneLevel(Node source, Node target) {
        for (Edge edge : edges) {
            if (edge.source() == source && edge.target() == target) {
                return edge.pheromones();
            }
            if (edge.target() == source && edge.source() == target) {
                return edge.pheromones();
```

```
        }
    }
    return 0;
}

/**
 * Increases the amount of pheromones in a given Node instance.
 */
public void raisePheromones(Node source, Node target, int amount) {
    for (Edge edge : edges) {
        if (edge.source() == source && edge.target() == target) {
            edge.raisePheromones(amount);
        }
        if (edge.target() == source && edge.source() == target) {
            edge.raisePheromones(amount);
        }
    }
}

/**
 * Checks for Node instances next to the one calling this method.
 */
public Node[] adjacentTo(Node node) {
    ArrayList<Node> nodes = new ArrayList<>();
    for (Edge edge : edges) {
        if (edge.source() == node) {
            nodes.add(edge.target());
        } else if (edge.target() == node) {
            nodes.add(edge.source());
        }
    }
    return nodes.toArray(new Node[0]);
}

/**
 * Decreases the amount pheromones in edges and randomly decides whether to add
 * sugar.
 */
public void tick() {
    for (Edge edge : edges) {
        edge.decreasePheromones();
    }
    if (RandomUtils.coinFlip(sugarProbability)) {
        spawnSugar();
    }
}

/**
 * Creates a 2D array to represent the position of Node instances relative to
 * each other.
 */
private Node[][] fillingInNodesTo2dArray(int width, int depth) {
    Node[][] nodeGrid = new Node[width][depth];
    for (int i = 0; i < width; i++) {
        for (int j = 0; j < depth; j++) {
            if (RandomUtils.coinFlip(sugarProbability)) {
                nodeGrid[i][j] = new Node(RandomUtils.randomPoisson(sugarAverage));
            } else {
                nodeGrid[i][j] = new Node();
            }
        }
    }
    return nodeGrid;
}

/**
 * Takes 2D array and converts a given amount of them from Nodes to Colony
 * instances.
 */
private void insertingColoniesToMap(Node[][] nodeGrid, Colony[] colonies, int width, int depth) {
    for (int i = 0; i < colonies.length;) {
        final int widthPostion = RandomUtils.randomInt(width);
        final int depthPosition = RandomUtils.randomInt(depth);
        if (!(isPositionColony(nodeGrid, widthPostion, depthPosition))) {
```

```
                nodeGrid[widthPostion][depthPosition] = colonies[i];
                i = i + 1;
            }
        }
    }

    /**
     * Creates connecting Edge instance between Node instances with source and
     * targets.
     */
    private Edge[] createEdgesBetweenNodes(int width, int depth, Node[][] nodeGrid) {
        List<Edge> edges = new ArrayList<>();
        for (int i = 0; i < width; i++) {
            for (int j = 0; j < depth; j++) {
                if (i + 1 < width) {
                    edges.add(new Edge(nodeGrid[i][j], nodeGrid[i + 1][j]));
                }
                if (j + 1 < depth) {
                    edges.add(new Edge(nodeGrid[i][j], nodeGrid[i][j + 1]));
                }
            }
        }
        return edges.toArray(new Edge[edges.size()]);
    }

    /**
     * Randomly decides whether to increase the amount of sugar in a Node by a
     * random amount.
     */
    private void spawnSugar() {
        int randomEdgeIndex = RandomUtils.randomInt(edges.length);
        Edge randomEdge = edges[randomEdgeIndex];
        if (RandomUtils.coinFlip(0.50)) {
            randomEdge.source().setSugar(RandomUtils.randomPoisson(sugarAverage));
        } else {
            randomEdge.target().setSugar(RandomUtils.randomPoisson(sugarAverage));
        }
    }

    /**
     * Checks if the instance calling this method is currently in an instance of
     * Colony.
     */
    private boolean isPositionColony(Node[][] map, int widthPostion, int depthPosition) {
        return map[widthPostion][depthPosition] instanceof Colony;
    }

    /**
     * Adds Colony instances to Node array.
     */
    private void addColoniesToNodeArray(String[] colonyInformation, Node[] nodeLocation, Colony[]
homes) {
        int colonyCounter = 0;
        for (String colony : colonyInformation) {
            int colonyIndex = Integer.parseInt(colony) - 1;
            nodeLocation[colonyIndex] = homes[colonyCounter];
            colonyCounter = colonyCounter + 1;
        }
    }

    /**
     * Fills out empty slot in Nodes array.
     */
    private void addNewNodesToEmptyIndexes(Node[] nodeLocation) {
        int i = 0;
        while (i < nodeLocation.length) {
            if (nodeLocation[i] == null) {
                Node node = new Node();
                nodeLocation[i] = node;
            }
            i = i + 1;
        }
    }
```

```
    /**
     * Creates a connecting Edge instance between two Node instances.
     */
    private void createEdges(Scanner filescanner, Node[] nodeLocation) {
        List<Edge> edges = new ArrayList<>();
        while (filescanner.hasNextLine()) {
            int nodeA = filescanner.nextInt() - 1;
            int nodeB = filescanner.nextInt() - 1;
            filescanner.nextLine();
            Edge edge = new Edge(nodeLocation[nodeA], nodeLocation[nodeB]);
            checkDuplicateEdge(edge, edges);
            edges.add(edge);
        }
        this.edges = edges.toArray(new Edge[edges.size()]);
    }

    /**
     * Checks whether multiple Edge instances target the same Node in the same
     * direction.
     */
    private void checkDuplicateEdge(Edge edge, List<Edge> edges) {
        for (Edge other : edges) {
            boolean isSameWay = other.source() == edge.source() && other.target() == edge.target();
            boolean isOtherWay = other.source() == edge.target() && other.target() == edge.source();
            if (isSameWay || isOtherWay) {
                throw new RuntimeException("Duplicate edges");
            }
        }
    }

    /**
     * Increases the amount of sugar in a given Node instance.
     */
    private void addSugarToNodes(Node[] nodeLocation) {
        for (Node node : nodeLocation) {
            if (RandomUtils.coinFlip(sugarProbability)) {
                node.setSugar(RandomUtils.randomPoisson(sugarAverage));
            }
        }
    }
}
```