The University of Melbourne
School of Computing and Information Systems
SWEN20003 Object Oriented Software Development

Project 2, Semester 2, 2020

Released: 18/09/2020
Project 2A Due: 02/10/2020 at 20:59 AEST
Project 2B Due: 23/10/2020 at 20:59 AEST

Overview

In this project, you will create a graphical simulation of a world and its inhabitants, continuing from your work in Project 1. We will provide a full working solution to Project 1 one week after the release (allowing for late submissions). You **may** use all or part of it—however, any files containing code taken from the solution must clearly indicate this at the top of the file.

This is an **individual project**. You may discuss it with other students, but all of the implementation must be your own work. You may use any platform and tools you wish to develop the project, but we officially support IntelliJ IDEA for Java development.

There are two tasks in this project, with different submission dates.

The first task, **Project 2A**, asks you to produce a class design demonstrating how you intend to implement the classes for your simulation. This should be in the form of a UML diagram showing the classes you plan to implement, their attributes, their public methods, and the relationships (e.g. inheritance and associations) between them. You **do not** need to show constructors, getters/setters, dependency, composition, or aggregation relationships. You **must** submit your diagram as a PDF file. You will be marked on your delegation of tasks between classes, your use of object-oriented principles such as inheritance and encapsulation, use of Java conventions, and other design aspects. Remember, associations should be used **instead of** attributes. There will be a Canvas submission link closer to the due date.

The second task, **Project 2B**, asks you to complete your implementation of the simulation described in the rest of this specification. You **do not** need to strictly adhere to the design you submitted for Project 2A—it is intended as a high-level design based only on reading the specification, and to practice designing using object-oriented principles before you begin programming. You will likely find ways to further improve the design as you implement it. Submission for Project 2B will be via Gitlab. **You must make at least 5 commits throughout your project.** This will be enforced.

Shadow Life

Simulation overview

Shadow Life is a graphical simulation of a world inhabited by creatures called *gatherers*. Their purpose in life is to gather fruit from the trees, and deposit them at *stockpiles*. Once they have gathered all the fruit from their trees, they rest in front of *fences*.

Making their life difficult is the *thief* who aims to steal fruit from the stockpiles and place it in their *hoards*. The thief and gatherers follow rigid rules, and once they all reach their final goals (the fence), the simulation *halts*. They are quite industrious workers—with enough time, they could calculate anything that any computer can!

The behaviour of the simulation is entirely determined by the *world file* loaded when the Shadow Life program starts: each gatherer, thief, and other element begins at a specified location and follows a set of rules to determine their behaviour. Once all gatherers and thieves have reached a fence, the simulation halts, and the amount of fruit at each stockpile and hoard is tallied up. The simulation proceeds in *ticks*, with the tick rate (time between ticks) determined by a command-line parameter. If more than a maximum number of ticks (also determined by a command-line parameter) pass before halting, the simulation *times out*. Otherwise, the number of elapsed ticks, together with the amounts of fruit at each location, is printed to form the *result* of the world file.

Figure 1 shows a screenshot of a simulation in progress.

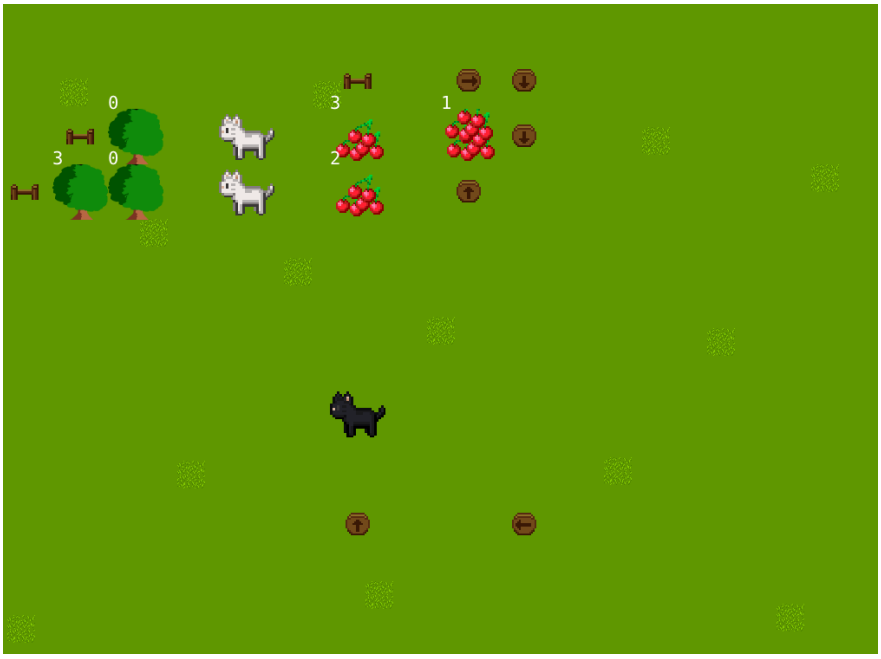

Figure 1: A simulation in progres.

Command-line arguments

Your simulation must take three command-line arguments: the *tick rate* (in milliseconds), the *maximum number of ticks*, and the *world file*. Here are some example argument lists for the program:

```
500 100 res/worlds/harvest.csv
100 1000 res/worlds/sum.csv
```

- If more or fewer than 3 arguments are provided, the simulation should print (to the terminal) the line

  ```
  usage: ShadowLife <tick rate> <max ticks> <world file>
  ```

  and exit with return code `-1` (for example, using `System.exit()`). It's OK if the Bagel version line `Bagel v1.9.3 (August 13th, 2020)` appears as well.

- If the first and/or second arguments are not valid non-negative integers (zero is allowed), the simulation should print the usage line and exit as above.

Update: there is a bug in MacOS preventing command-line arguments from working. Instead, you may read the command-line arguments from a file called `args.txt`. Code that can do this is below.

```
private static String[] argsFromFile() {
    try {
        return Files.readString(Path.of("args.txt"), Charset.defaultCharset())
            .split(" ");
    } catch (IOException e) {
        e.printStackTrace();
    }
    return null;
}
```

Ticks

The simulation proceeds in discrete **ticks**. Each tick, an actor can perform an action such as moving and/or changing direction. The tick rate is determined by a command-line argument as above; you should ensure at least this many milliseconds pass between ticks.

If more than the maximum number of ticks pass (including the first one) during a simulation, the simulation should print the line

```
Timed out
```

and exit with return code `-1`.

Simulation Elements

Below is an outline of the different simulation elements you will need to implement.

The background

As for Project 1, you will need to show the *background* below all other elements.

Actors

An **actor** is an object with an associated image that is located at a particular tile, and may perform an action every tick.

- **Trees:** a tree is stationary and takes no action upon a tick. Its image is located at `res/images/tree.png`. It begins with 3 fruit. The current number of fruit at each tree should be drawn at the top-left of its image. (We have provided a font, `res/VeraMono.ttf`, for you to use if you desire.)
- **Golden Trees:** a golden tree is stationary and takes no action upon a tick. Its image is located at `res/images/gold-tree.png`. It has an infinite reserve of fruit, and no number should be drawn.
- **Stockpiles:** a stockpile is stationary and takes no action upon a tick. Its image is located at `res/images/cherries.png`. It begins with 0 fruit. The current number of fruit at each stockpile should be drawn at the top-left of its image.
- **Hoards:** a hoard is stationary and takes no action upon a tick. Its image is located at `res/images/hoard.png`. It begins with 0 fruit. The current number of fruit at each hoard should be drawn at the top-left of its image.
- **Pads:** a pad is stationary and takes no action upon a tick. Its image is located at `res/images/pad.png`.
- **Fences:** a fence is stationary and takes no action upon a tick. Its image is located at `res/images/fence.png`.
- **Signs:** a sign is stationary and takes no action upon a tick. It serves to redirect gatherers and thieves. There are four types of signs, with images located at
  - `res/images/left.png`
  - `res/images/right.png`
  - `res/images/up.png`
  - `res/images/down.png`
- **Mitosis Pools:** a mitosis pool is stationary and takes no action upon a tick. Its image is located at `res/images/pool.png`.
- **Gatherers:** a gatherer contains state, and is initialised according to Algorithm 1. Each tick, it should follow the procedure of Algorithm 2. **Update:** the order of the move and other actions has changed. Its image is located at `res/images/gatherer.png`.
- **Thieves:** a thief contains state, and is initialised according to Algorithm 3. Each tick, it should follow the procedure of Algorithm 4. **Update:** the order of the move and other actions has changed. Its image is located at `res/images/thief.png`.

Algorithm 1 Gatherer initialisation

Set `direction = LEFT`.
Set `carrying = false`.
Set `active = true`.

Algorithm 2 Gatherer tick

if `active == true` **then**
    Move 64 pixels (1 tile) in its `direction`.
if the gatherer is standing on a fence **then**
    Set `active = false`.
    Move to the position the gatherer was at in the previous tick.
if the gatherer is standing on a mitosis pool **then**
    Create one gatherer at its position with its `direction` rotated 90 degrees **counter-clockwise**.
    Create one gatherer at its position with its `direction` rotated 90 degrees **clockwise**.
    Move each of the created gatherers 64 pixels (1 tile) in their `direction`s. (They should do nothing else this tick.)
    Destroy this gatherer. (It should do nothing else this tick.)
if the gatherer is standing on a sign **then**
    Set `direction` to match the sign.
if the gatherer is standing on a tree and `carrying == false` **then**
    if the tree has at least 1 fruit **then**
        Decrease the tree's fruit by 1.
        Set `carrying = true`.
        Rotate `direction` by 180 degrees.
if the gatherer is standing on a hoard or a stockpile **then**
    if `carrying == true` **then**
        Set `carrying = false`.
        Increase the hoard/stockpile's fruit by 1.
    Rotate `direction` by 180 degrees.

Algorithm 3 Thief initialisation

Set `direction = UP`.
Set `carrying = false`.
Set `consuming = false`.
Set `active = true`.

Algorithm 4 Thief tick

1: **if** `active == true` **then**
2: Move 64 pixels (1 tile) in its `direction`.
3: **if** the thief is standing on a fence **then**
4: Set `active = false`.
5: Move to the position the thief was at in the previous tick.
6: **if** the thief is standing on a mitosis pool **then**
7: Create one thief at its position with its `direction` rotated 90 degrees **counter-clockwise**.
8: Create one thief at its position with its `direction` rotated 90 degrees **clockwise**.
9: Move each of the created thieves 64 pixels (1 tile) in their `directions`. (They should do nothing else this tick.)
10: Destroy this thief. (It should do nothing else this tick.)
11: **if** the thief is standing on a sign **then**
12: Set `direction` to match the sign.
13: **if** the thief is standing on a pad **then**
14: Set `consuming = true`.
15: **if** the thief is standing on a gatherer (after the gatherer's tick has been performed) **then**
16: Rotate `direction` by 270 degrees **clockwise**.
17: **if** the thief is standing on a tree and `carrying == false` **then**
18: **if** the tree has at least 1 fruit **then**
19: Decrease the tree's fruit by 1.
20: Set `carrying = true`.
21: **if** the thief is standing on a hoard **then**
22: **if** `consuming == true` **then**
23: Set `consuming = false`.
24: **if** `carrying == false` **then**
25: **if** the hoard has at least 1 fruit **then**
26: Set `carrying = true`.
27: Decrease the hoard's fruit by 1.
28: **else**
29: Rotate `direction` by 90 degrees **clockwise**.
30: **else if** `carrying == true` **then**
31: Set `carrying = false`.
32: Increase the hoard's fruit by 1.
33: Rotate `direction` by 90 degrees **clockwise**.
34: **if** the thief is standing on a stockpile **then**
35: **if** `carrying == false` **then**
36: **if** the stockpile has at least 1 fruit **then**
37: Set `carrying = true`.
38: Set `consuming = false`.
39: Decrease the stockpile's fruit by 1.
40: Rotate `direction` by 90 degrees **clockwise**.
41: **else**
42: Rotate `direction` by 90 degrees **clockwise**.

World files

The world file follows the same structure as in Project 1. Below is an example:

```
Tree,384,384
Stockpile,768,384
Gatherer,512,384
Fence,256,384
```

- If the provided file does not exist or you fail to open it, the simulation should print the line

  `error: file "<file name>" not found`

  (with the provided file in the quotes) and exit with return code `-1`.

- If the file exists, but one of its line is invalid (e.g. does not contain two commas, the type in the first part is not recognised, or the second two parts are not valid integers), the program should print the line

  `error: in file "<file name>" at line <line number>`

  with the appropriate file name and line number, and exit with return code `-1`. (You only need to print the message for the first invalid line seen.)

The simulation **halts** once all gatherers and thieves have set `active` to `false`. When the simulation halts, you should print the total number of ticks that have elapsed. You should then print the amount of fruit at each stockpile and hoard, in the order they appear in the world file. The program should then exit with return code `0`. For example, for the below world file:

```
Tree,384,384
Stockpile,768,384
Gatherer,512,384
Fence,256,384
```

the correct output is:

```
Bagel v1.9.3 (August 13th, 2020)
40 ticks
0
3
```

(IntelliJ will also display more information such as `/usr/lib/jvm/jdk-14.0.1/bin/java` and `Process finished with exit code 0` . This information is not part of your program's output—you can ignore it.)

We have provided you with several sample world files. Your simulation will be tested on each of these, as well as several secret tests. Sample output for the provided world files is in the appendix of this specification.

Your Code

You are required to submit the following:

- a class named `ShadowLife` that contains a `main` method to run the simulation described above
- at least one other class (the purpose of which is up to you)

You may choose to create as many additional classes as you see fit, keeping in mind the principles of object oriented design discussed so far in the subject. You will be assessed based on your code running correctly, as well as the effective use of Java concepts. As always in software engineering, appropriate comments and variables/method/class names are important.

Implementation Checklist

To help you get started, here is a checklist of the required functionality, with a suggested order of implementation (assuming you begin with your solution to Project 1):

- Implement command-line arguments
- Load all of the actor types (they can be stationary for now)
- Implement gatherer functionality for trees, stockpiles, and fences
- Implement the sign and hoard functionality for gatherers
- Implement basic thief functionality for signs, stockpiles, fences, and hoards (without worrying about `consuming`)
- Implement the pad functionality
- Implement the rest of the thief functionality
- Implement the mitosis pool functionality

Supplied Package

You will be given a package `res.zip`, which contains all of the graphics and other files you need to build the simulation. Here is a brief summary of its contents:

- `res/` – The resources for the simulation.
  - `images/`: – The image files for the simulation.
  - `worlds/`: – The worlds for the simulation.
    - `harvest.csv` – a simple test with no thieves
    - `sum.csv` – a slightly more difficult test with a thief
    - `product.csv` – a challenging test with a thief

Customisation

Optional: we want to encourage creativity with this project. We have tried to outline every aspect of the simulation design here, but if you wish, you may customise any part of the simulation, including the graphics, types of actors, variety of worlds, etc. You can also add entirely new features. However, to be eligible for full marks, you **must** implement all of the features in the above implementation checklist.

For those of you with far too much time on your hands, we will hold a competition for the best simulation extension or modification, judged by the lecturer and tutors. The winning three will be demonstrated at the final lecture, and there will be a prize for our favourite. Past modifications have included drastically increasing the scope of the simulation, implementing jokes and creative , adding polish to the game, and even introducing networked interaction.

If you would like to enter the competition, please email the head tutor, Eleanor McMurtry, at `eleanor.mcmurtry@unimelb.edu.au` with your username and a short description of the modifications you came up with. I can't wait to see what you've done!

For your customisation, you **may** use additional libraries (other than Bagel and the Java standard library). However, the version you submit via your repository **must not use additional libraries.** If you would like to use extra libraries for customisation, please commit these to a different repository and let Eleanor know.

Submission and marking

Technical requirements

- The program must be written in the Java programming language.
- Comments and class names must be in English **only**.
- The program must not depend upon any libraries other than the Java standard library and the Bagel library (as well as Bagel's dependencies).
- The program must compile fully without errors.
- For full marks, **every** public class, method, and attribute must have a descriptive Javadoc comment (covered later in the semester).

Submission will take place through GitLab. You are to submit to your `<username>-project-2` repository. An example repository has been set up here[1] showing an ideal repository structure. At the **bare minimum** you are expected to follow the following structure. You **can** create more files and directories in your repository.

[1] `https://gitlab.eng.unimelb.edu.au/emcmurtry/emcmurtry-project-2`

```
username-project-2
├─ res
│  └─ resources used for project
└─ src
   ├─ ShadowLife.java
   └─ other Java files, including at least one other class
```

On 23/10/2020 at 21:00 AEST, your latest commit will automatically be harvested from GitLab.

Commits

You are free to push to your repository post-deadline, but only the latest commit on or before 23/10/2020 at 20:59 AEST will be marked. You **must** make at least 5 commits throughout the development of the project, and they must have meaningful messages (commit messages must match the code in the commit). If commits are anomalous (e.g. commit message does not match the code, commits with a large amount of code within two commits which are not far apart in time) you risk penalization.

Examples of **good, meaningful** commit messages:

- display background and tree graphics
- implemented gatherer movement logic
- refactored code for cleaner design

Examples of **bad, unhelpful** commit messages:

- fesjakhbdjl
- i'm hungry
- fixed thingzZZZ

Good Coding Style

Good coding style is a contentious issue; however, we will be marking your code based on the following criteria:

- You should *not* go back and comment your code after the fact. You should be commenting as you go. (Yes, we can tell.)
- You should be taking care to ensure proper use of visibility modifiers. Unless you have a very good reason for it, all instance variables should be `private`.
- Any constant should be defined as a `static final` variable with a name **that describes its purpose**. Don't use magic numbers!
  - A string value that is **used once** and is self-descriptive (e.g. a file path) does not need to be defined as a constant.

- Think about whether your code is written to be easily extensible via appropriate use of classes.
- Make sure each class makes sense as a cohesive whole. A class should have a single well-defined purpose, and should contain all the data it needs to fulfil this purpose.

Extensions and late submissions

If you need an extension for the project, please email Eleanor at `eleanor.mcmurtry@unimelb.edu.au` explaining your situation with some supporting documentation (medical certificate, academic adjustment plan, wedding invitation). If an extension has been granted, you may submit via Gitlab as usual; please do however email Eleanor once you have submitted your project.

The project is due at **20:59 AEST sharp**. Any submissions received past this time (from 21:00 AEST onwards) will be considered late unless an extension has been granted. There will be no exceptions. There is a penalty of 1 mark for a late project, plus an additional 1 mark per 24 hours. If you submit late, you **must** email Eleanor so that we can ensure your late submission is marked correctly.

Marks

Project 2 is worth **22** marks out of the total 100 for the subject. You are **not required** to use any particular features of Java. For example, you may decide not to use any interfaces or generic classes. You will be marked based on the **effective and appropriate** use of the various object-oriented principles and tools you have learnt throughout the subject.

- Project 2A is worth **6 marks**.
  - Notation (includes Java conventions):
    - Correct UML notation for methods: **1 mark**
    - Correct UML notation for attributes: **1 mark**
    - Correct UML notation for associations: **1 mark**
  - Design:
    - Good breakdown into classes: **1 mark**
    - Good delegation and use of associations: **1 mark**
    - Appropriate use of inheritance and/or interfaces: **1 mark**
- Project 2B is worth **16 marks**.
  - Features implemented correctly: **8 marks** (based on automated test cases, including the ones we have provided and some hidden cases)
  - Use of object-oriented principles: **6 marks**

    This includes but is not limited to:
    - Delegation: breaking the code down into appropriate classes

    - Use of methods: avoiding repeated code and overly complex methods
    - Cohesion: classes are complete units that contain all their data
    - Coupling: interactions between classes are not overly complex
    - Encapsulation: classes abstract their implementation details away from their users
- General code style: visibility modifiers, magic numbers, commenting etc.: **1 mark**
- Use of Javadoc documentation: **1 mark**

A Sample outputs

For `harvest.csv`:

```
Bagel v1.9.3 (August 13th, 2020)
40 ticks
3
```

For `sum.csv`:

```
Bagel v1.9.3 (August 13th, 2020)
202 ticks
0
0
9
```

For `product.csv`:

```
Bagel v1.9.3 (August 13th, 2020)
758 ticks
0
3
6
0
18
```