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

Project 2, Semester 1, 2021

Released: 23/4/2021
Project 2A Due: 5/5/2021 at 20:59 AEST
Project 2B Due: 21/5/2021 at 20:59 AEST

Overview

In this project, you will create a graphical simulation of ShadowTreasure, 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 to Canvas (Note, Project 2A should be submitted to Canvas, not gitlab)**. 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.

Figure 1 shows a screenshot from the simulation after completing Project 2. Project 2 are using the same concept of **tick** as in Project 1.

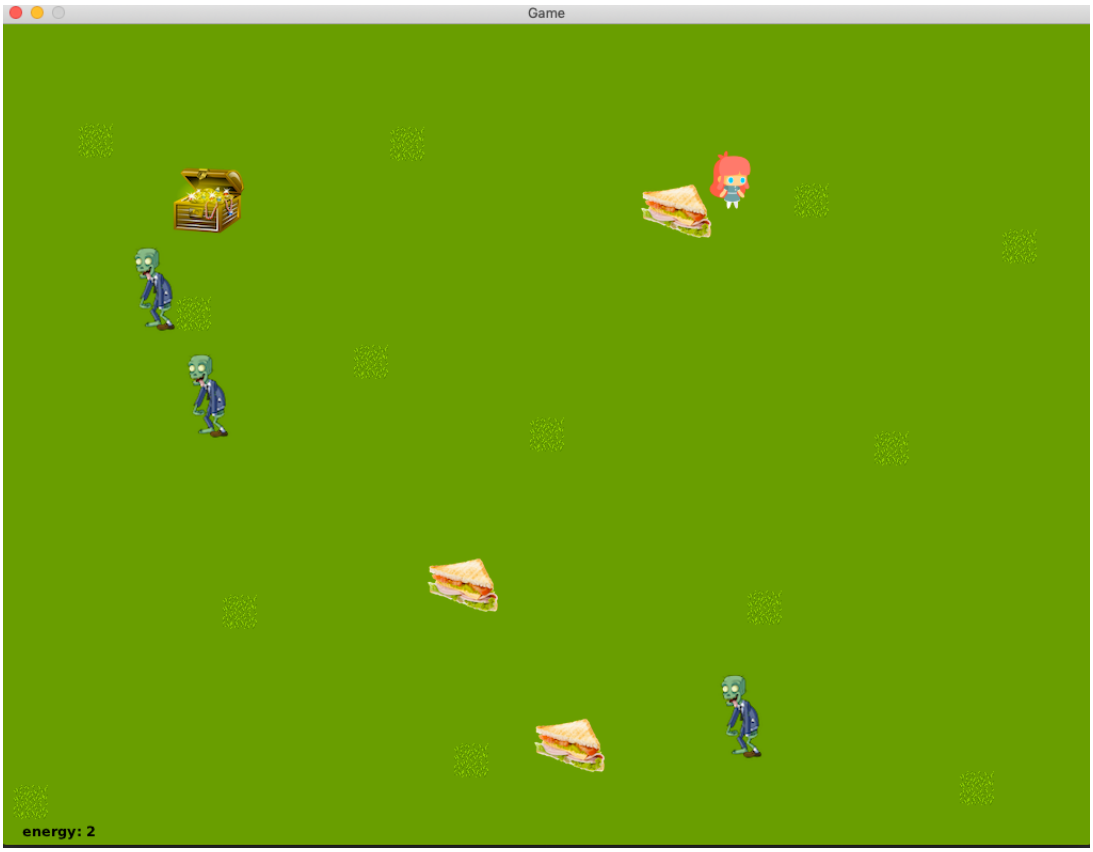

Figure 1: An Example of Project 2 Screenshot (note that the positions of player, sandwich and zombie might be different in your implementation).

Treasure Hunt Game

As said in Project 1, the player enters a tomb to search for a treasure box. The tomb is filled with zombies, that the player needs to fight before reaching the treasure. While the fighting is energy-consuming, the tomb is also filled with the nutritious foods to let the player regain strength.

There is **only one player** in the treasure hunt game. For Project 1, there is one zombie and one sandwich. The character and function descriptions of the player, zombie and sandwich are outlined below.

Background should be rendered/drawn in the same way as Project 1.

Entities

There is only one player, and **a number of** other entities, zombies and sandwich, in the tomb that the Player can interact with.

- **Zombie and sandwich are stationary entity. See Project 1 for where the image files locate.**
- **Treasure:** a stationary entity in the tomb. Its image is located at `res/images/treasure.png`.

(a) Zombie (b) Sandwich (c) Player (d) Treasure (e) Bullet

Figure 2: Images of Characters in ShadowTreasure for Project 2. Note, I enlarge the Bullet in (e), but it is just a small green dot when it's drawn on the screen.

See Figure 2(d). There is **only one treasure** in ShadowTreasure. The final goal of the player is to reach/meet treasure. But, the treasure can be **reached/met only if all zombies in the tomb are killed**. See Algorithm 1 for how to kill zombies.

Player

Image file and attributes energy and step size are the same as Project 1. Note, you still need to print the energy on the screen as in Project 1. In project, the player can shoot the bullet to kill zombies.

- *Bullet*: a moving object in the tomb. Its image is located at `res/images/shot.png`. When it's shooting, the step size of bullet is 25. Each shot consumes energy 3. The same as the player, **bullet movements only happen in each tick**.

Initial Positions

The same as Project 1, the initial positions of the entities are determined by a **environment file**, located at `res/environment.csv`. This is a comma-separated value (CSV) file with rows in the following format for zombie and sandwich:

```
[Type], x-coordinate, y-coordinate
```

where the [Type] is either Zombie, Sandwich or Treasure and the x-coordinate and y-coordinate show the location of the entity. A row in the file for the Player will have the format:

```
Player, x-coordinate, y-coordinate, energy level
```

where x-coordinate and y-coordinate show the initial location of the Player and the last column, energy level, shows the starting energy level of the Player.

You must actually load the environment file—copying and pasting the data, for example, is not allowed. You must load this environment file and create the corresponding entities in your simulation. When we testing your code, we may use a different environment file. You should assume that, in each environment file, there will be **only one player, only one treasure and a number of zombies and sandwiches. Note the number of zombies and sandwiches may vary.**

Algorithm 1: Interaction Logic and Energy Update of Player in **Each Tick**

```
1  if the treasure is met/reached by the player, or the player's energy level is (strictly) less
     than 3 when there are zombies but no sandwiches then
2      print to the console the players energy level;
3      if the game is terminated because the player reaches the treasure then
4          You need also print to console "success!" (insert one comma between energy level
             and the string "success!").
5      endif
6      terminate the game;
7  else
8      // the player interacts with entities
9      if the player meets a sandwich then
10         the player eats the sandwich and increases her energy level by 5. The sandwich
             should disappear, i.e., remove it from the environment.
11     else if the player is within the shooting range of a zombie then
12         the player shoots a bullet toward the zombie and reduces energy by 3.
13     endif
14     // player sets moving direction
15     if all zombies are killed then
16         the treasure is reachable now and the player moves toward the treasure by one step.
17     else if the player's energy level ≥ 3 then
18         the player moves toward the closest zombie by one step.
19     else
20         the player moves toward the closest sandwich by one step.
21     endif
22     // removing dead zombie
23     if a zombie is shot dead, it should disappear, i.e. remove it from the environment.
24 endif
```

Interactions/Function between the Player and other entities and Energy update

The definition of "meet" and one-step move are the same as in Project 1. The Player has strong radar that collects the information (type and position of other entities) in the game. She should interact and set/change the direction of her movement in each tick according to Algorithm 1. See the following notes for Algorithm 1.

- *Shooting Range* is 150. The player cannot shoot the bullet if its distance to any zombie is greater than or equal to the shooting range.
- *death of Zombie*: A zombie is called "shot dead" if it is strictly less than 25 from the bullet.
  - You can assume there is only one zombie shot by the bullet at a time, i.e., you **do not need to consider the situation when there are more than two zombies within the "shot dead" range from the bullet**. The environment file is always set to avoid this situation.

  - According to Algorithm 1, the player (with energy level no less than 3) still needs to move to the zombie after she shoots a bullet to it. However, the step sizes of player and bullet are set in a way such that this zombie will be killed before the player meets it. Therefore, **you do not need to have any part in your code handling the situation when the player meets a zombie**.

    You **do not need to consider when the bullet does not kill, or miss the "shot dead" range of, the zombie, but the player enters the shooting range of another zombie**. We will set each environment file to avoid this situation (if Algorithm 1 is implemented correctly). This also means only one bullet instance/object is needed in your ShadowTreasure game. You do not need to use a bullet array.

- You need to **make sure the treasure is reachable only if all zombies are killed**. If the player meets a sandwich on the way move to the treasure, she still needs to interact with (e.g., eat) this sandwich.
- According to Algorithm 1, the player should interact with sandwich/zombie on the way to zombie/sandwich, which might results in negative energy level. But, you should assume the positions of entities in environment file are always set to prevent this situation. Also, $\#sandwiches * 5 \geq \#zombies * 3$, i.e., the energy level will always be nonnegative. This means you **do not need to have any part in your code handling nonpositive energy**.
- Is should be noted that Algorithm 1 will not be all implemented in one method or one class. In this case, you need to consider **one special case** which is related to dealing with the last zombie in test2. In line 1 of Algorithm 1, you should judge the case "the player's energy level is (strictly) less than 3 when there are zombies but no sandwiches" by discriminating whether the bullet is shooting. This is because, for example, when the player has 0 energy, and at the same time the bullet is moving to the last zombie but has not killed it, she can still reach the treasure with "Success!" after the last zombie is shot dead. In this case, you should make the player move toward the last zombie until it is shot dead. See test2 video. We will not provide pseudocode for this part. You need to decide how to write your code to handle this special case.

Note, the following will cause mark reduction.

- Algorithm 1 should be implemented in **each tick, not each frame**.
- However, you do need to **draw all entities in each frame**.
- You need to make your game **automatically run**, not driven by keyboard inputs.
- You need to make sure the **energy color and position** are correct on the screen. **Do not omit the string "energy"**.
- **Make sure you draw the background**. The background should cover the whole screen of the game.

Output

It is required that you trace the bullet's movement in each tick whenever it's shooting. Make sure you **record the position of bullet only when it is shooting**.

You should write the position of bullet to the file **res/IO/output.csv** in the form of

```
x-coordinate, y-coordinate
```

in one line for each tick.

We will read res/IO/output.csv to check if the bullet is shot correctly.

You should also print to stdout/console as described in Algorithm 1.

You need to print to output.csv and stdout in the exact form described above. **Do not insert other char/string**, e.g., "x-axis:...., y-axis:...". I striped strings for project 1, but will not do it for project 2B.

The outputs above are the main resources I refer to when checking your implementation. Make sure your code produces the expected outputs. **If you don't write to stdout and/or output.csv, you will not get any mark for implementation**, even if it's 100% correct.

Your Code

You are required to submit the following:

- a class named `ShadowTreasure` that contains a `main` method to run the simulation described above
- and all other classes you have designed and implemented to make the game work.

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:

- Draw the background on screen.
- Load the environment file correctly at the beginning of the game.
- Implement Algorithm 1.
- Write to output.csv and stdout.

Supplied Package

Resources: You will be given a package `res.zip` containing all 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.
    - `background.png`: The background image
    - `zombie.png`: The image for zombie
    - `sandwich.png`: The image for sandwich
    - `treasure.png`: The image for treasure
    - `shot.png`: The image for bullet
  - `IO/` – The input/output files for treasure hunt game.
    - `environment.csv`: The environment file.
  - `font/` – The font files for the simulation.
    - `DejaVuSans-Bold.ttf`: The fond file.

Make sure you **do not change anything in sub-directories `images/` and `font/`**: We may load `res/` in your submission when testing your code.

It is **extremely important** that you **do not corrupt the structure of res/** and make sure your code **loads environment from res/**. Otherwise, your code would not pass any test when I mark your code: I will only replace the environment file in res/ with the testing one, run your code and collect outputs.

- Do not load envrironment from test/ folder.
- load all resources from "res/". Do not use "/res/" or insert any thing before "res/...", e.g., do not load from "project-1/res/..."
- If you use windows, replace all \\in file paths with /. I will still use mac for project 2B.

Materials for testing: You will be given the `test.zip`, which contains two examples of the environment files and the expected stdouts of them. They are provided for you to check the correctness of your code.

- `test/`
  - `test_stdout/` – contains tests 1, 2 and 3. You can use them for test your stdout (only). Each folder contains
    - `environment.csv`: the environment file `res/IO/environment.csv`.

    The expected stdouts/console outputs are listed on Canvas.

  - `test_output/` – contains tests 4 and 5. You should use them for test your stdout and output.csv. Each folder contains
    * `environment.csv`: The environment file.
    * `output.csv`: The output file after you load `environment.csv` and run `ShadowTreasure`.

    The expected stdouts/console outputs are listed on Canvas.

To test your code, copy `environment.csv` in any test folder to the folder `res/IO/`, run the game `ShadowTreasure` and then compare `stdout.csv` in the same folder with your stdout.

Also, refer to one of my announcements for further details of the test kit.

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 showing an ideal repository structure. At the **bare minimum** you are expected to follow the following structure. You **can** create more files/directories in your repository.

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

On 21/5/2021 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 21/05/2021 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.

Do not corrupt your repository after your final commit/push before the deadline. Chances are we will reclone from your folder if any problem. In this case, if you push after the deadline, we will consider it late submission.

I have helped recover some students' corrupted submissions for project 1, but will not do this for project 2B.

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

- displayed background and load player and other entities correctly
- implemented Algorithm 1 correctly
- 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. 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 Ni Ding at `ni.ding@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 Ni Ding 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 Ni Ding 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 – **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 – **16 marks**
  - Correct implementation: **8 mark** It is important to get your code run. Otherwise, you get 0 mark for implementation. I have looked into and corrected the errors for all students getting 0/4 for implementation for project 1, but will not do this for project 2B.
    - Correct bullet shooting: **3 mark**
    - Correct implementation of Algorithm 1: **5 mark**
  - Use of object-oriented principles: **6 marks**

    This includes, but 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**