Middle East Technical University - Department of Computer Engineering

CENG 443

Introduction to Object Oriented
Language and Systems

Fall '2022-2023

Homework 1 - Maze Game

Due Date: 11 November 2022, Saturday, 23:55
Late Submission Policy will be explained below

Objectives

The objective of this assignment is to learn the basics of object-oriented design principles, Unified Modeling Language (UML) and design patterns. You will build a small graphical Pac-Man style **Maze** game. Finally, you will write javadoc for your implementation and create a UML class diagram. Specifications of the game is given Section 1.
Keywords: Java, Object-oriented programming, Strategy Pattern, Singleton Pattern, Observer Pattern, Decorator Pattern.

1 Specifications

The game consists of **Actors**, specifically Enemies, Bullets, Power-ups, Walls, and finally the player. **Player** will be the user-controlled actor and the player will try to get all the power-ups in order to win the game. **Enemies** will patrol around the maze and if the player and an enemy collide, they will kill the player. **Walls** defines the impassible locations on the game map. **Bullets** are created by the player and damage enemies if they collide. **Power-ups** are spread around the maps and can be collected by the player. The player "wins" the game when all of the power-ups are collected. The player will "lose" the game if he is killed by an enemy.

You are required to design the system and provide a UML diagram of the game in order to show your design. A basic diagram of classes are shown in Figure 1. Please note that this is **not** the UML diagram that we are asking for.

While designing your game, you should utilize Design patterns. Design patterns are typical solutions to common problems in software design and help us to follow the design principles. In this homework, you are expected to use four design patterns; Strategy Pattern, Singleton Pattern, Observer Pattern, and Decorator Pattern. Specifics of the provided class skeletons and fully or partially implemented classes can be seen in Section 1.1.

1.1 Class Definitions

The classes that are provided to you can be seen in Figure 1. Some of the hierarchy that is displayed in the Figure are not mandatory and can be changed by you. It is the class hierarchy of the reference implementation.

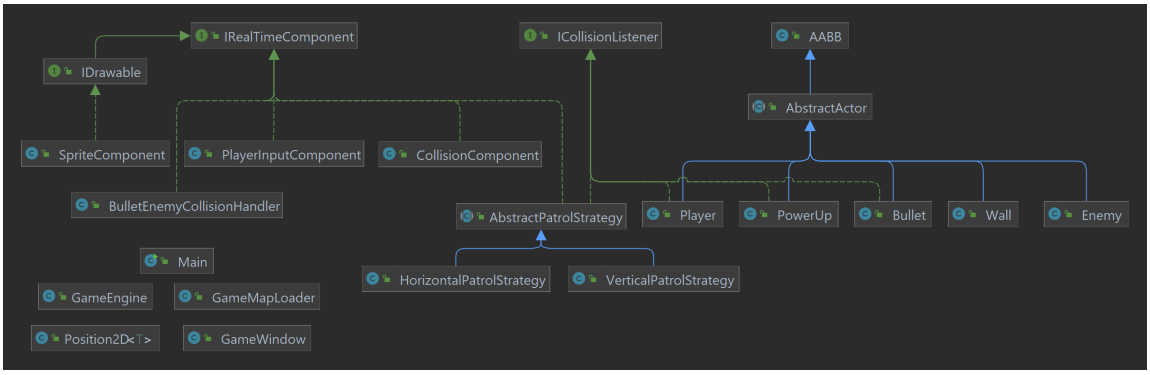

Figure 1: IDE generated Java Class Diagram

- **`Main:`** Main entry point of the game. The main function expects a single argument which should be the map file path. *This class is fully implemented for you and it should not be changed.*. The main function initializes a **`GameEngine`** class and provides it to the singleton **`GameWindow`** class.
- **`GameDisplay:`** Singleton display window that the game runs on. Can be acquired globally due to its singleton nature. *This class is fully implemented for you and it should not be changed.*

  > **Hint:** Use the singleton nature of this class to implement **`PlayerInputComponent`**.

- **`GameEngine:`** GameEngine is the heart of the game. Its update function is called continuously by the display. *This class is partially implemented for you. You can add anything you like except for the update function.*

  > **Notice:** You should not change the **update** function of this class. It already does everything you need it to do. In order to complete the assignment you can add statements. However, such implementations will be penalized. Game loop code can be seen on Listing 1

- **`GameMapLoader:`** Loads the map file according to the screen size. Creates Axis-aligned Bounding Boxes (AABB) for the actors. *This class is fully implemented for you and it should not be changed.*
- **`Position2D<T>:`** generic position class just holds two variables of type "T". It is used to improve readability. *This class is fully implemented for you and it should not be changed.*
- **`AABB:`** Base class of the actors, defines a surface area on the screen. It also has collision methods that can be used to handle collision. *This class is fully implemented for you and it should not be changed.*
- **`AbstractActor:`** Abstract class of the actors. This class should be implemented using **Decorator Pattern**. Decorators in this case should be **`IRealTimeComponent`**. *This class is bare bones. And it should be implemented with respect to your design.*

  > **Info:** One of the most popular game engine; Unity, uses this design pattern extensively.

- **`IRealTimeComponent:`** Fundamental interface of a real-time system. Most of the things in a game needed to be updated on every iteration of the update function. This interface provides that. *This interface is already defined for you.*
- **`IDrawable:`** Further extends the interface **`IRealTimeComponent`** interface, and adds draw functionality. *This interface is already defined for you.*

- **`SpriteComponent:`** Only drawable in the system. Handles drawing sprites (2D images) inside a surface area defined by an AABB. *This class is already implemented for you and it should not be changed.*
- **`PlayerInputComponent:`** Handles player inputs. Should move the attached actor according to key inputs, creates bullets if fire key is released. *This class is partially implemented. Rest should be implemented with respect to your design.*
- **`CollissionComponent:`** Handles collision between the attached actor and list of AABBs. Moves the attached actor away if a collision happens between any of the AABBs. This class should be implemented using **Observer Pattern**. Observers in this case should be **`ICollisionListener`**. Thus, it should notify the observers when a collision happened. *This class is bare bones and it should be implemented according to your design.*

  > **Info:** Many GUI libraries (well Java swing for example), use this pattern to notify GUI input operations (i.e. "button clicked", "key pressed" etc.).

- **`ICollisionListener:`** Interface of the collision listener, **`CollissionComponent`** will call the implementers of this interface if attached. *This interface is already defined for you but you can change the signature or the name of the function if you want.*
- **`AbstractPatrolStrategy:`** Strategy of certain enemies. This class should be implemented using **Strategy Pattern**. Strategies should be attached to certain enemies. *This class is bare bones. And it should be implemented with respect to your design.*
- **`(Horizontal,Vertical)PatrolStrategy:`** Concrete implementations of patrol strategy. Patrolling actors should "turn back" if they hit a wall. *These classes are bare bones. And it should be implemented with respect to your design.*
- **`BulletEnemyCollisionHandler:`** Special component that is attached to the game itself. Checks collisions between bullets and enemies; damages the enemies and destroys the bullets if a collision happens. *This class is bare bones. And it should be implemented with respect to your design.*
- **`Player, PowerUp, Bullet, Wall, Enemy:`** Concrete classes for the game actors. These should hold information about the concrete classes. (See Section 2). *This class is bare bones. And it should be implemented with respect to your design.*
- **You can also add additional classes as well. Don't forget to show it in your design.**

2 Game Details

2.1 `Enemy`

Enemies have health and get damaged by bullets. Moves horizontally, vertically or does not move at all. (depending on the strategy)

- **Health:** 100 units
- **MovementSpeed:** 120 pixels per second
- **SpritePath:** "./data/img/enemy.png"

2.2 `Player`

Players do not have health they immediately die if they collide with an enemy. Moves using player input.

- **MovementSpeed:** 110 pixels per second
- **SpritePath:** "./data/img/player.png"

```
public synchronized void update(float deltaT, Graphics2D currentDrawBuffer)
{
    // ================================== //
    // YOU SHOULD NOT CHANGE THIS FUNCTION  //
    // ================================== //
    // Do update first
    walls.forEach(actor -> actor.update(deltaT, currentDrawBuffer));
    enemies.forEach(actor -> actor.update(deltaT, currentDrawBuffer));
    powerUps.forEach(actor -> actor.update(deltaT, currentDrawBuffer));
    bulletsInCirculation.forEach(actor-> actor.update(deltaT, currentDrawBuffer));
    player.update(deltaT, currentDrawBuffer);
    miscComponents.forEach(c -> c.update(deltaT));
    // Now stuff would die etc. check the states and delete
    enemies.removeIf(actor -> actor.isDead());
    powerUps.removeIf(actor -> actor.isDead());
    bulletsInCirculation.removeIf(actor -> actor.isDead());
    // If player dies game is over reset the system
    if(player.isDead())
    {
        ResetGame();
    }
    // Player won the game!, still reset
    if(powerUps.isEmpty())
    {
        ResetGame();
    }
    // And the game goes on forever...
}
```

Listing 1: Game Loop code

2.3 Wall

Walls don't do much. They never die, only there to collide with others.

- **SpritePath:** "./data/img/wall.png"

2.4 Bullet

Bullets are created by the player when the fire key is pressed. They move toward the last moved direction of the player. Bullets die if their time is ended. over time. If

- **SpritePath:** "./data/img/bullet.png"
- **Bullet Life:** 0.7 seconds
- **Bullet Speed:** 300 pixels per second

2.5 Power-Up

Power-ups are just there to get collected. They die when it collides with the player.

- **SpritePath:** "./data/img/scroll.png"

3 UML Class Diagram

To properly document your code, you are required to draw a class diagram of your application exported as a PDF document. Try to keep your diagram as simple as possible by only including important fields, methods, and relations between the classes. Please, show the relations between classes such as Composition, Aggregation, Dependency, etc. Using StarUML might be useful for this task.

4 FAQ

1. **Q: What should the game look like? Are there any references?**

   A: A sample screenshot of the game can be seen in Figure 2. There is also a video on the ODTUClass.

2. **Q: Can you explain the map file and the loader?**

   A: `GameMapLoader` class reads the map file and generates AABBs for stationary enemies, horizontally patrolling enemies, walls, power-ups and the player. The map file is very basic it is a 32 by 18 grid of characters each character represents the types of actors that should be put in. Space character represents empty space.

3. **Q: Any platform requirements?**

   A: That is the beauty of Java. You can develop on the Linux platform or Windows platform whichever you like. Although, I will check your code on the Windows platform.

4. **Q: Do you have any IDE suggestions?**

   A: IntelliJ Idea is a well-known IDE for Java development. It is also **free** for students. I would suggest you use that.

5. **Q: Performance of the game is really bad? What's up?**

   A: Probably the `JFrame` is not hardware accelerated. Before loading the game, `JFrame` shows the status of the graphics configuration of the frame. If you have Intel integrated graphics, please check this post. Basically, set an environment variable named J2D_D3D_NO_HWCHECK as true. If you still have an issue we can discuss it on the forums.

Regulations

1. **Programming Language:** You will use Java (version 8 or above).
2. **Late Submission Policy:** A penalty of $5 \times Lateday^2$ will be applied for submissions that are late at most 3 days. If unusual circumstances truly beyond your control prevent you from submitting an assignment, you should discuss this with the course staff as soon as possible. If you contact us well in advance of the deadline, we may be able to show more flexibility in some cases.
3. **Cheating:** We have a **zero tolerance** policy for cheating. In case of cheating, all parts involved (source(s) and receiver(s)) get zero. People involved in cheating will be punished according to university regulations. Remember that students of this course are bounded to the code of honor and its violation is subject to severe punishment. Please be aware that there are very advanced tools that detect if two codes are similar.
4. **Newsgroup:** You must follow the Forum (`odtuclass.metu.edu.tr`) for discussions and possible updates on a daily basis.
5. **Submission:** Submission will be made via ODTUClass. Create a zip file named "hw1.zip" that contains all your source code files inside "src" directory. Additionally, "hw1.zip" should contain an UML file (pdf file format), README, javadoc in "docs" directory. Please provide a README file which describes how to compile and run your code.

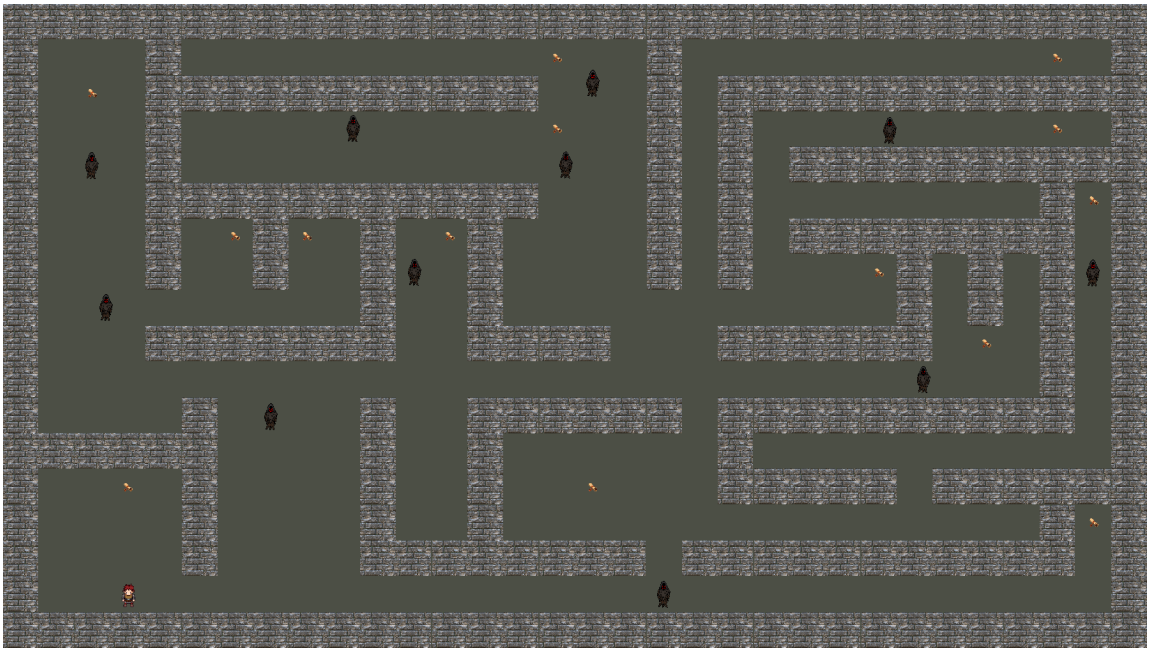

Figure 2: IDE generated Java Class Diagram

Evaluation

Your codes will be evaluated **subjectively**; meaning that a perfectly working game with a bad design may get a bad score. Your UML design and your provided source files will be examined to check for similarity.

You may get partial grades, so even if you are not able to implement all requirements, please submit the completed code. However, be sure that your submission can be compiled.

Some points to consider:

- The subjective evaluation will value the "well-designed games with buggy functionality" more than the "working but badly designed games".
- You should design the classes with the given design pattern or you will lose points.
- If you use **"*instanceof*"** or you will lose points.
- If you change the code at Listing 1, you will lose points. However, if you want to add extra functionality (i.e. GUI for how many power-ups are left, time, etc.) you may do so. Please write it on the README file, if it is not written **I will deduct points even if I notice that the modification obviously adds new functionality.**
- Do not worry about the visual details like colors, sizes, etc., and do not spend so much time on it. It would be best if you focus on object-oriented principles and design patterns.