Middle East Technical University Department of Computer Engineering

CENG 443

Introduction to Object Oriented Programming Languages and System

Fall 2021-2022

Homework 1 - Tower Defense

Due date: 05 12 2021, Sunday, 23:59

1 Introduction

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 **Tower Defense** game. You will write javadoc for your implementation and create a UML class diagram.

In this tower defense game, there are monster waves and towers that try to destroy them. Towers cost money to build and each monster killed will provide certain amount upon death. The player will have three lives and each monster that reach the end zone will take a life from the player. The aim of this game is to kill as many waves as possible before losing all three of lives. The waves and the monsters should get stronger as time goes on. The details of the game is explained in the following Specifications section.

2 Specifications

In this game, we will have four main game entity classes. **Monster** class for simulating the monsters and **TowerRegular**, **TowerIce**, and **TowerPoison** classes for defense towers you will place. You will use these towers to destroy waves of enemies until you lose all three of your lives. The monsters will circle the tower and upon reaching their spawn point again, the player will lose a life.

Additionally, you are required to design the system and to document it by using javadoc. A class diagram in UML is sufficient for the design of the system. You need to comment the code properly to generate javadoc. You will fill in the methods and implementation details of the classes. Do not rename/remove the existing ones. Basic implementation for drawing the panel and images is provided. You can simply use the Display instance in the required classes to draw the entities. You will complete the parts where *TODO* comments are. Stick to the OO standards; do not use public access level everywhere, use getters/setters, polymorphism, etc. It is better to start by designing the application by drawing a UML class diagram. 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 five design patterns; Singleton, Factory Method, State, Strategy and Decorator. Singleton is already implemented in the **Game** and **Display** class for you. The details are explained in the following sections.

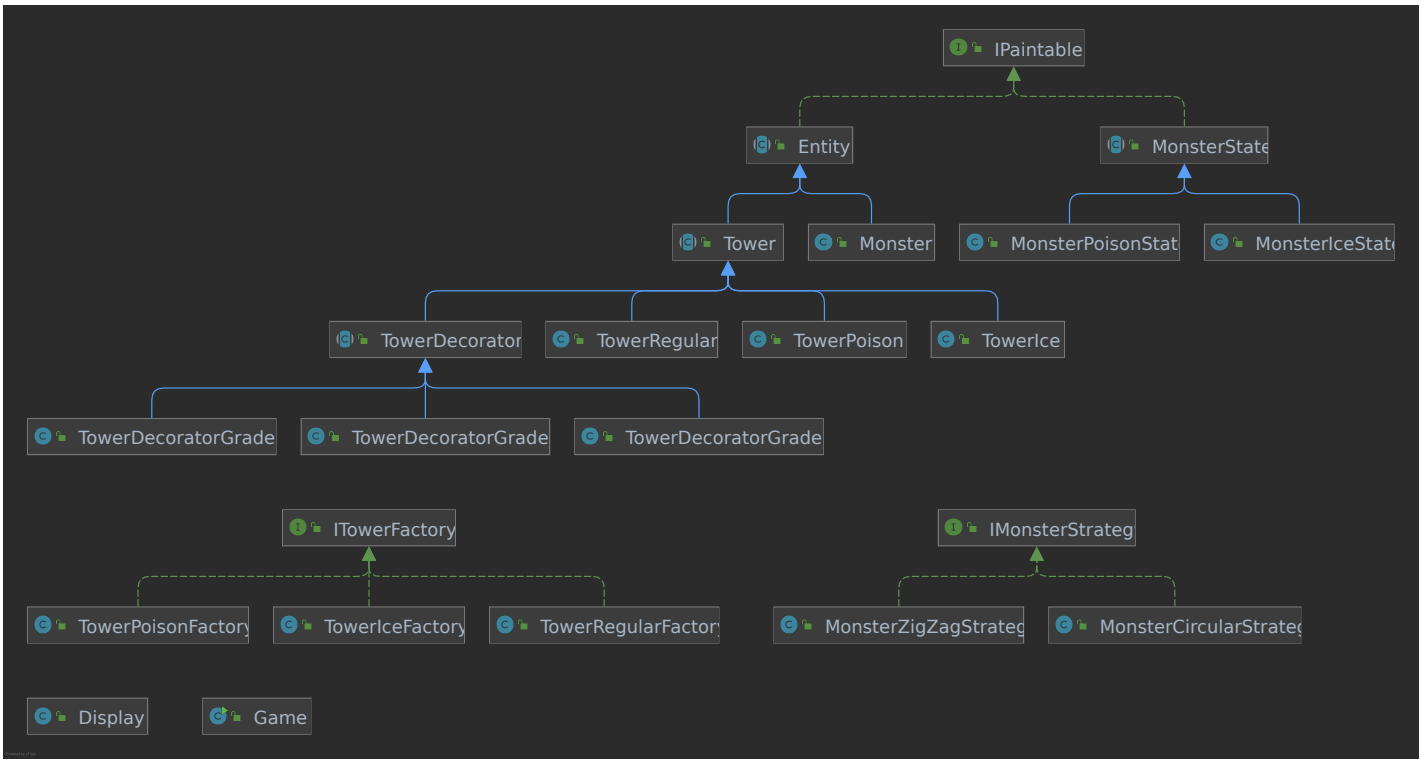

Figure 1: IDE generated Java class diagram.

2.1 Animation Entities

The classes that are provided to you are listed and explained below. In Figure 1, a simple, IDE generated Java class diagram is provided to show the big picture. Be careful, it is not the UML class diagram we expect from you.

- **Game:** Singleton class that contains the main method. It is the responsible class for managing the simulation, and storing the required entities.
- **Display:** Singleton class that represents the display on which animation entities, the remaining lives, current gold, total kills and the current wave number are repeatedly drawn.
- **IPaintable:** Interface for paintable game objects.
- **Entity:** Represents the abstract base class for the animation entities. Towers and Monster are the subclasses of the base class.
- **Monster:** The concrete class for Monster entities in the game. Hold all of their information.
- **Tower:** Abstract class for different Tower classes. Holds basic information.
- **TowerRegular, TowerPoison, TowerIce:** The concrete classes that extend the Tower class.
- **TowerDecorator:** An abstract base class for TowerDecoratorGrade1, TowerDecoratorGrade2, TowerDecoratorGrade3.
- **TowerDecoratorGrade1, TowerDecoratorGrade2, TowerDecoratorGrade3:** The concrete decorator classes that extend the TowerDecorator class that are responsible to draw the symbols indicating kill counts of the towers.

- **ITowerFactory:** An interface for different tower factories.
- **TowerRegularFactory, TowerPoisonFactory, TowerIceFactory:** The concrete factory classes that implements the ITowerFactory interface.
- **IMonsterStrategy:** An interface for the strategy pattern to implement different monster movement strategies.
- **MonsterZigZagStrategy, MonsterCircularStrategy:** The concrete strategy classes that implements the IMonsterStrategy interface.
- **MonsterState:** The abstract class representing state pattern for monster upon receiving special damage.
- **MonsterPoisonState, MonsterIceState:** The concrete state classes extend the MonsterState class.

2.2 Game Details

The game class holds all the information about the game and the main method necessary to run it. It also has timers or such synchronization mechanisms that one would need to run the game. The display class will draw the required window. The side panel will display the information about wave count, total gold, the number of monster kills, and remaining lives.
Starting information is given below:

- **Gold:** 25
- **Lives:** 3
- **Wave:** 1
- **Monster Count:** $Wave \times 1$

The monsters will start in the start zone and circle around in the map. You can designate the start zone at one of the corners of the map. The towers will be placed in the middle and they will attack the monsters when the monster is in range. If a monster comes back to the start zone without dying, the player loses a life. The game ends when the user loses all of its lives. The towers do not attack at every step and different towers can inflict different ailments to the monster.

The stats of the monster class is given below:

- **Monster**
  Health: 100 + (WaveCount) * 20
  Speed: 1.0 pixel (at every animation step) (you need to normalize vectors for direction)
  Reward: 10
  It should start in a random position within the start zone and should have random movement strategy. At each corner, there should be a random chance of being assigned a new strategy. In the given example, Monsters are drawn as squares. It should also show its remaining health and provide an indication whenever it has received damage.

The details of the Tower classes are given below:

- **TowerRegular**
  Range: 150px (from the tower center to the monster center)
  Rate of Fire: 1 in 20 steps
  Damage: 20
  Cost: 20

- **TowerIce**
  Range: 100px (from the tower center to the monster center)
  Rate of Fire: 1 in 20 steps
  Damage: 10
  Cost: 15
  It inflicts MonsterIceState to the monster which should reduce its speed to $speed = 0.2 \times step$ for 5 steps. This state change should be shown when monster is drawn. In the given example, it is drawn as a blue square around the monster shape.

- **TowerPoison**
  Range: 75px (from the tower center to the monster center)
  Rate of Fire: 1 in 10 steps
  Damage: 5
  Cost: 25
  It inflicts MonsterPoisonState to the monster which should reduce its health by 5 for 3 steps. This state change should be shown when monster is drawn. In the given example, it is drawn as a green square around the monster shape.

All towers should attack the closest monster in its range. They should be visualized with different colors and their range should be shown with dotted circular lines. I have used simple circles to represent towers. Tower should be added dynamically during runtime. It can be added with mouse and keyboard operations, buttons or dialog boxes. It is up to you. However, document your solution in your submission.
You will implement following four design patterns:

- **Factory Method / Abstract Factory:** Creations of towers (i.e., regular, ice, and poison) will be accomplished via entity factories. You can visit here to get more information about Factory design pattern. For an excellent object-oriented design demonstration, make sure the object that creates the entities is unaware of the entity types.

- **Strategy:** To provide behavior to monsters, you will need to use the Strategy pattern. You can visit here to get more information about Strategy design pattern. The details of concrete Strategy classes are listed below. Remember that to demonstrate a good design; the entities should not be aware of the strategy that they are assigned to.
  MonsterCircularStrategy: The monster follows a standard path around the towers.
  MonsterZigZagStrategy: The monster follows a zigzag pattern around the towers.

- **State:** State pattern also provide behavior however they do it more indirectly compared to the strategy pattern. States can also be aware of each other. You can visit here to get more information about State design pattern. The details of concrete State classes are listed below.
  MonsterIceState: Reduce the speed of the monster according to the formula given above. It should last for 5 steps and visualized on the screen.
  MonsterPoisonState: The monster take five damage at each step. It should stop after 3 steps and visualized on the screen.

- **Decorator:** Use the decorator pattern at run-time. You can visit here to get more information about Decorator design pattern. Please make sure the decorated entities are not aware of the fact that they are being decorated. Moreover, decorator classes should not know each other. In other words, they should not depend on each other. Each tower will be decorated according to its kill count. There are 3 Grades (Decorators) according to kill. Lines going across the tower visual or dots is sufficient to represent different grades.
  TowerDecoratorGrade1: 10 kills
  TowerDecoratorGrade2: 25 kills
  TowerDecoratorGrade3: 50 kills

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 Sample Output

You can see the empty game panels and some tower visuals in the following figures 2, 3, 4, 5.

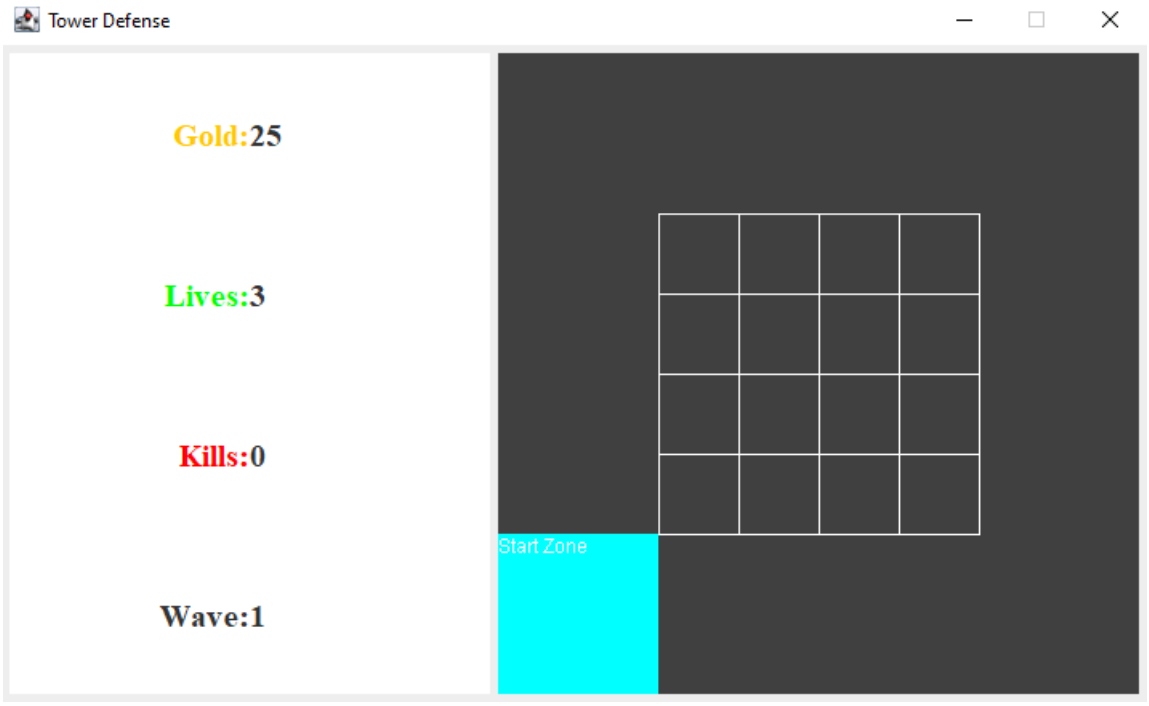

Figure 2: Empty Tower Picture

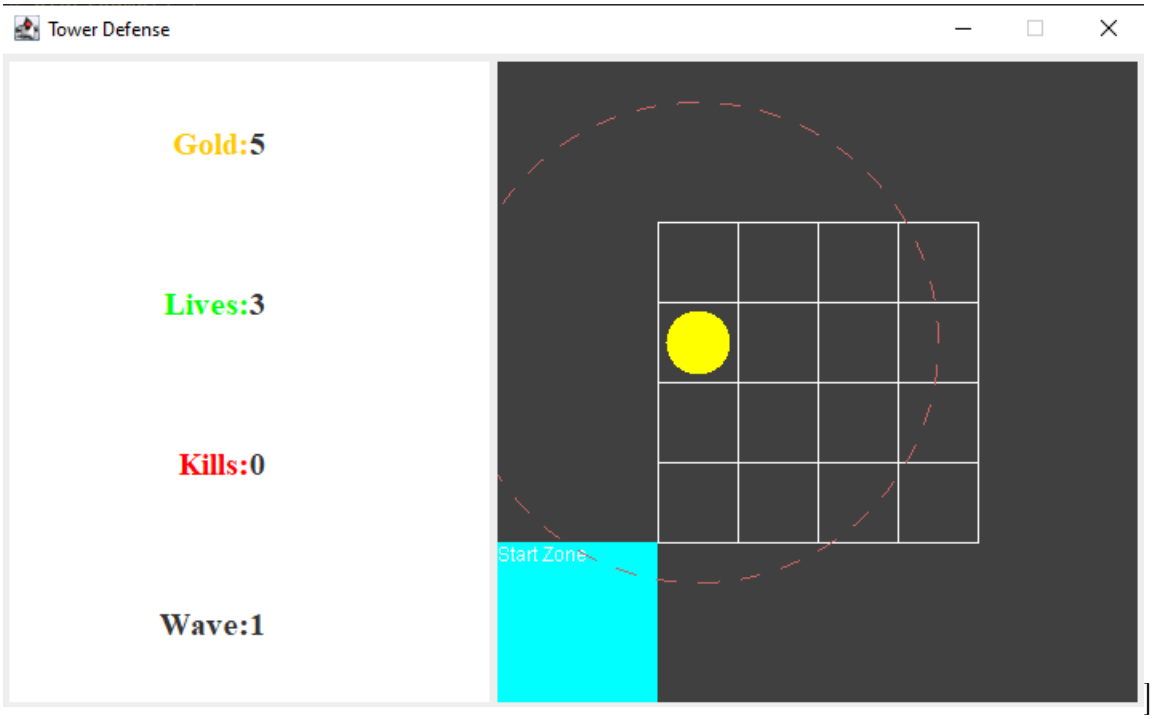

Figure 3: Regular Tower Picture

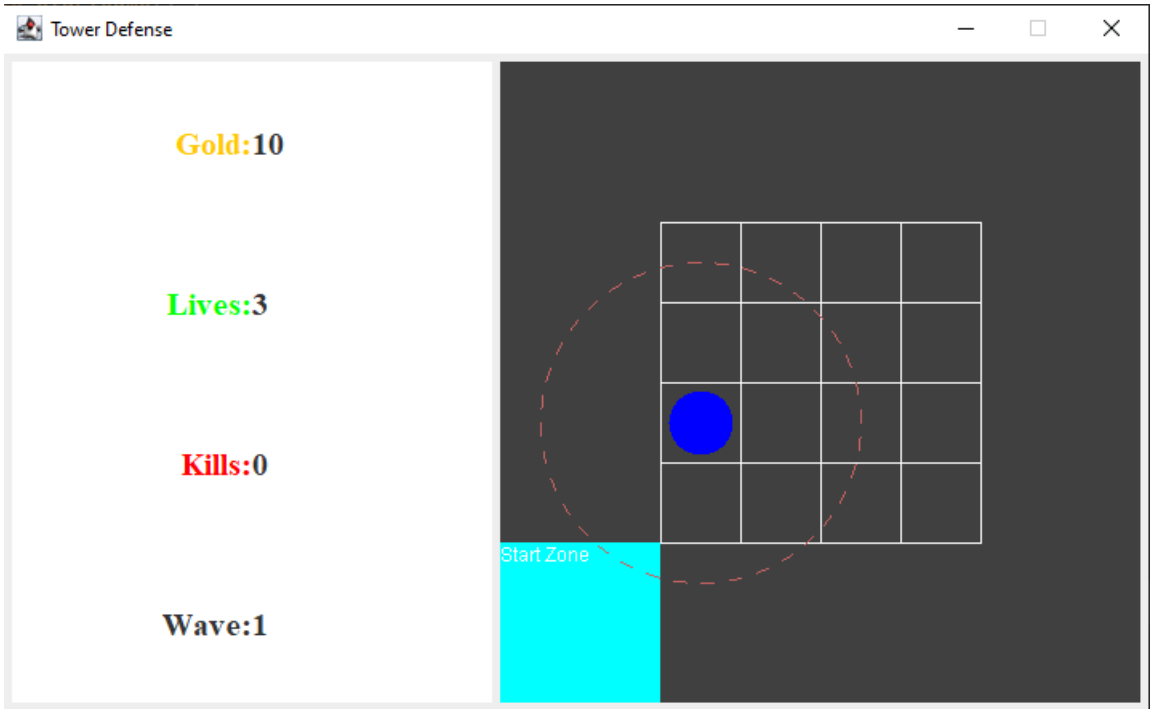

Figure 4: Ice Tower Picture.

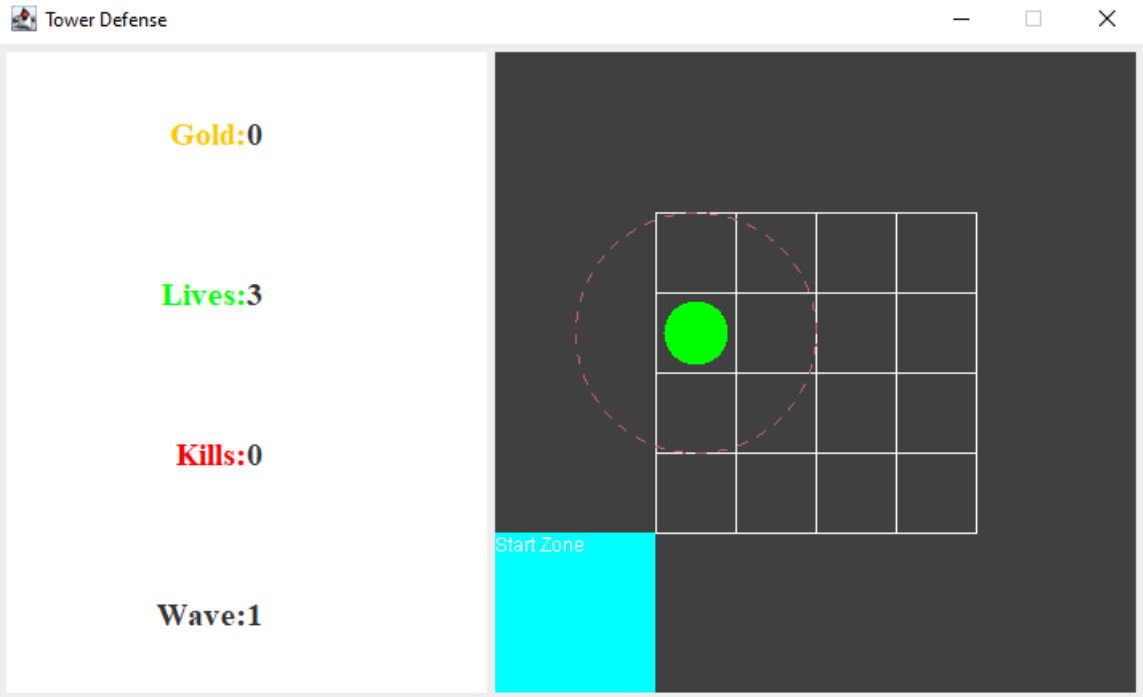

Figure 5: Poison Tower Picture

You can watch the sample video here. It should give a general idea. (I will upload a better one later one with better graphics and larger screen.) To add a tower and runtime, I have used mouse click to select the position and press specific keys (on the keyboard) depending on the tower type. (r for regular, i for ice and p for poison)

5 Specifications

- **Programming Language:** Java (version 8)
- **Late Submission:** A penalty of $5 \times Lateday^2$ will be applied for submissions that are late at most 3 days.
- **Cheating:** We have zero tolerance policy for cheating. Your solution must be original. People involved in cheating will be punished according to the university regulations and will get 0 from the homework. You can discuss algorithmic choices, but sharing code between groups or using third party code is strictly forbidden. In case a match is found, this will also be considered as cheating. Even if you take only a part of the code from somewhere or somebody else, this is also cheating. Please be aware that there are very advanced tools that detect if two codes are similar.
- **Newsgroup:** You must follow the ODTUClass forum for discussions, possible updates, and corrections.
- **Submission:** Submission will be made via ODTUClass. Create a zip file named "hw1.zip" that contains all your source code files inside "Source" directory and UML file, README, javadoc in "Docs" directory. Please provide a README file which describes how to compile and run your code.
- **Evaluation:** Your codes will be evaluated manually; small differences might be tolerated. In addition, your code will be examined to check the correctness of the implementation. 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. Therefore, if you have differences when running the program, that will not be a problem. **Important note:** The visualization details are not so important. It is enough if one can distinguish the monsters, towers, their range, decorators, strategies and states in the simulation visually. 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. You can even change the size of the animation windows. However make sure that you also increase the range of the towers the same coefficient. Please, do not ask for more details about the visualization; the requirements above are sufficient. You are free to demonstrate the simulation as long as it is similar to the provided examples and the video.