KATOUZIAN Pouria s192865
HOUYON Manuel s203802

Project 2

INFO0027 : Tower Defense game

Université de Liège
Faculté des Sciences Appliquées
Année académique 2023-2024

1 Introduction

This project consists of developing a tower defense game coded in Java programming and employing different types of design patterns. To launch the project, simply run the commands make all then make run.

2 Choices of patterns

Based on several research and referencing the course INFO0027 - Programming Techniques, the patterns chosen for each class in this project are as follows :

In the Base, EnemiesManager, Game, and Map classes, the singleton pattern is employed. The state pattern is used to implement the WaveState and StandbyState classes. also, some classes in this project, such as Tower and Enemy, are developed using the factory design pattern.

2.1 Singleton Pattern

The Singleton Pattern is implement in the Game, Map and StateManager classes.

2.1.1 Game

```
/*
 * This class is used to represent the game.
 */
public class Game implements TowerDefenseEventsHandlerInterface {

    private Base base;
    private int level;
    private int money;
    private Vector<Tower> towers;
    private EnemyManager enemy_manager;
    private StateManager state;
    private Map map;
    private static Game game = null;

    private Game() {
        this.money = 50;
        this.level = 0;
        this.base = Base.get_base();
        this.towers = new Vector<Tower>();
        this.enemy_manager = EnemyManager.get_enemiesManager();
        this.state = StateManager.get_StateManager();
        this.map = Map.get_map();

    }

    public static Game get_game() {
        return game == null ? game = new Game() : game;
    }
    //Other methods
}
```

The Game used singleton instance of the Map class. This allows the Game class to interact with the map without needing to create multiple instances of Map.The Game class also holds a reference to the singleton instance of the StateManager class in order to manage the current state of the game and transition between different states.

2.1.2 Map

```java
/*
 * This class is used to represent the map in the game.
 */
public class Map {

    private final short width = 12;
    private final short height = 9;
    private int[][] grid = new int[width][height];
    private Vector<Point2D> path = new Vector<Point2D>();
    private static Map map = null;

    /*
     * Constructor for the Map class
     */
    private Map() {
        set_map();
        set_path();
    }

    public static Map get_map() {
        return map == null ? (map = new Map()) : map;
    }

    //Methods to set & get Map and path.
    //Method to get x & y cordinate of each cell
    //method to set occupied a cell
}
```

2.1.3 State Manager

```java
//This class is used to represent the state manager in the game.
public class StateManager {
    private StateInterface state;
    private static StateManager instance = null;

    //Constructor for the StateManager class
    private StateManager() {
        this.state = new StandbyState();
    }

    // getter and setter methods

    public void run(Game currentGame, TowerDefenseView
       towerDefenseView) {
        state.run(currentGame, towerDefenseView);
    }
    public void launch_wave(Game currentGame) {
        state.launch_wave(currentGame);
    }

}
```

The StateManager class manages the state of the game by holding a reference to the current state (an instance of StateInterface). It can transition between different states, such as StandbyState and WaveState, and performs state-specific operations on the Game instance.

By employing the Singleton pattern in these classes and ensures that there is only one instance of each class throughout the application, providing a centralized and consistent point of access for managing game states and map operations. This promotes efficient resource usage and simplifies the interactions between these core components.

2.2 Observer Pattern

```java
//This class is used to represent the enemies in the game.
public class Enemy extends Entity {

    private final int maxIndex = 35;
    private float health;
    private float speed;
    private int rewards;
    private int currentPathIndex;
    private long lastMoveTime;
    private List<EnemyObserver> observers;

    // Constructor for the Enemy class
    public Enemy(int type, int level, Point2D startPoint) {

        super((int) startPoint.getX(), (int) startPoint.getY(), 10 *
           type, type + 2, level + type + 1);
        this.observers = new ArrayList<EnemyObserver>();
        switch (type) {
            //consstruct class based on the type of the Enemy
        }
        this.rewards = 5 * type;
        this.set_image(EntityFlyweight.getEnnemiesImg(type));
    }

    public void registerObserver(EnemyObserver observer) {
        observers.add(observer);
    }

    public void unregisterObserver(EnemyObserver observer) {
        observers.remove(observer);
    }

    private void notifyObservers() {
        for (EnemyObserver observer : observers) {
            observer.anEnemyInRange(this);
        }
    }
    //getter and setter methods
}
```

2.2.1 EnemyObserver

```java
public interface EnemyObserver {
    void anEnemyInRange(Enemy enemy);
}
```

The Observer Pattern is used in the Enemy and Tower classes. This pattern is used to allow Tower objects to respond to changes in the state of Enemy objects. The classes involved are Enemy (Subject), EnemyObserver (Observer Interface) and Tower (Concrete Observer).

The Enemy class has a list of observers that are implemented the EnemyObserver interface which provides methods to add and remove observers and a method to notify all registered observers about a state change. The Tower class implements the EnemyObserver interface, allowing it to be notified when an Enemy object changes state. The Tower reacts to these notifications by attacking the enemy. The interaction between these classes are as follows, the Enemy object notifies all registered EnemyObserver instances (Tower) when it changes state. When notified, the Tower objects perform an action, such as attacking the enemy.

2.3 Facotry Pattern

The Factory pattern is used to create instances of the Enmey class and the classes as EnemyFactory, Enemy, EnemyManager and Game are involved in this dessign pattern. The Factory pattern defines an interface for creating an object and letting subclasses alter the type of objects that will be created.

2.3.1 EnemyFactory

```
//This class is used to create enemies.
public class EnemyFactory {

    public static Enemy createEnemy(int type, int level, Point2D
        startPoint) {
        return new Enemy(type, level, startPoint);
    }
}
```

2.3.2 EnemyManager

```
public class EnemyManager {

    private List<Enemy> enemies;
    private Vector<Point2D> path;
    private List<Tower> towers;
    private static EnemyManager enemyManager = null;

    private EnemyManager() {
        this.enemies = new Vector<Enemy>();
        this.path = Map.get_map().get_path();
        this.towers = new ArrayList<>();
    }

    public static EnemyManager get_enemiesManager() {
        //getter method
    }
    public void generate_ennemies(int level) {
        //generating ennemies
    }
    //Other methods
}
```

Others methods are provided previously, providing redendent code example make report longer.

The Game class interacts with the EnemyManager class to generate and manage enemies for each wave. The Game class does not directly create enemies and it relies on the EnemyManage which calls the EnemyFactory to use its createEnemy() method to create different type of Enemy.

2.4 State Pattern

The State Pattern is used in StateManager, StandbyState, WaveState, StateInterface and Game classes to manage the different states of the game. The state pattern allows an object to change its behavior when its internal state changes. The Game and StateManager are provided previously. The Game class interacts with the StateManager to manage the current state of the game by using the methods of the StateManager to run current state and launch new waves. The StateManager class holds a reference to the current state, represented by an object that are implemented in StateInterface class. The StateInterface interacts with Game and TowerDefenseView. The StateManager uses StandbyState and WaveState classes.

2.4.1 StandbyState

```
1  //This class is used to represent the StandbyState in the game.
2  public class StandbyState implements StateInterface {
3
4    public void update_base(Base base, TowerDefenseView towerDefenseView)
5        throws WrongBasePositionException, EmptySpriteException {
6      //Implementation of the update_base() method
7    }
8
9    private void update_towers(Vector<Tower> towers, TowerDefenseView
         towerDefenseView)
10       throws WrongTowerPositionException, EmptySpriteException {
11     //Implementation of the update_tower() method
12   }
13
14   private void update_money(int money, TowerDefenseView
         towerDefenseView) {
15     towerDefenseView.updateMoney(money);
16   }
17
18   public void run(Game currentGame, TowerDefenseView towerDefenseView) {
19     //implementation of the run method
20   }
21
22   public void launch_wave(Game currentGame) {
23     currentGame.generate_ennemies();
24     currentGame.get_state().set_state(new WaveState());
25   }
26 }
```

2.4.2 StateInterface

```
1  //This class is used to represent the StateInterface in the game.
2  interface StateInterface {
3
4      void run(Game currentGame, TowerDefenseView towerDefenseView);
5
6      void launch_wave(Game currentGame);
7  }
```

2.4.3 WaveState

```
1  //This class is used to represent the wave state in the game.
2  public class WaveState implements StateInterface {
3
4      //Constructor for the WaveState class
5      public void update_base(Base base, TowerDefenseView
           towerDefenseView)
6              throws WrongBasePositionException, EmptySpriteException {
7          base.heal();
8          towerDefenseView.updateBase((int) base.get_position().getX(),
               (int) base.get_position().getY(),
9                  base.get_health(), base.get_image());
10     }
11
12     private void update_towers(Vector<Tower> towers, TowerDefenseView
           towerDefenseView)
13             throws WrongTowerPositionException, EmptySpriteException {
14             //Implementation of the update_tower() method
15     }
16     public void update_ennemies(List<Enemy> ennemies, TowerDefenseView
           towerDefenseView)
17             throws WrongAttackerPositionException, EmptySpriteException
                   {
18             //Implementation of the update_ennemies() method
19     }
20     public void attack_ennemies(Game currentGame, Vector<Tower> towers,
           List<Enemy> ennemies,
21             TowerDefenseView towerDefenseView) {
22             //Implementation of the attack_ennemies() method
23
24     }
25     public void wave_end(StateManager currentState, Game currentGame,
           TowerDefenseView towerDefenseView)
26             throws UnknownTowerException {
27             //Implementation of the wave_end() method
28     }
29     public void run(Game currentGame, TowerDefenseView
           towerDefenseView) {
30
31             //Implementation of the run() method
32     }
33     public void launch_wave(Game currentGame) {
34         return;
35     }
36 }
```

2.5 Flyweight Pattern

The Flyweight pattern is used to sahre and reuse common data in order to minimize memory usage or computational expenses, as image in this project, among multiple instances of other classes, such as Enemy and Tower. The classes involved in this pattern are EntityFlyweight (Flyweight Factory), Enemy and Tower. This pattern is useful when dealing with large numbers of objects that share common properties. The code of Tower and Enemy class are provided previously.

2.5.1 EntityFlyweight

```
//This class is used to represent the entity flyweight in the game.
public class EntityFlyweight {

    private static final Map<Integer, ImageIcon> EnnemiesMap = new
        HashMap<Integer, ImageIcon>();
    private static final Map<Integer, ImageIcon> TowersMap = new
        HashMap<Integer, ImageIcon>();

    public static ImageIcon getEnnemiesImg(int type) {
        ImageIcon img = EnnemiesMap.get(type);
        if (img == null) {
            img = new ImageIcon("../resources/attackers/attacker" +
                type + ".png");
            EnnemiesMap.put(type, img);
        }
        return img;
    }

    public static ImageIcon getTowersImg(int type) {
        ImageIcon img = TowersMap.get(type);
        if (img == null) {
            img = new ImageIcon("../resources/towers/tower" + type +
                ".png");
            TowersMap.put(type, img);
        }
        return img;
    }
}
```

2.6 Command Pattern

2.6.1 Command

```
public interface Command {
    void execute();
}
```

The command pattern turns a request into a stand-alone object that contains all information about the request. The command is used to encapsulate a request as an object, thereby allowing for parameterization of clients with queues, requests and operations. The classes operate which each other, are Command, AttackCommand, Tower, Enemy and Game. All Code except the Command class are provided previously.

3 Advantages of using these patterns

The design patterns used in this project promote code reuse, enhance flexibility and simplifying the maintenance. Employing these design patterns collectively brings significant advantages in this project. To manage resource efficiently Singleton pattern is used and to opetimize memory usage, Flyweight pattern. The state pattern enhance modularity and ease of feature extension and command pattern facilitate a robust and scalable architecture. The reason of using each pattern are listed below.

3.1 singleton

Using the singleton pattern in this project (Game, Map, StateManager) ensures a single, consistent source of truth for game state, map layout, and management. Employing this pattern reduces inconsistencies and simplifies debugging. Singletons also control object creation, saving memory in resource-constrained environments.

3.2 Factory

The factory pattern (EnemyFactory) allows to create different enemy types (FastEnemy, StrongEnemy, etc.). This simplifies object creation and allows for easy addition of new enemies without changing existing code.

3.3 Observer

The observer pattern lets objects (Enemy in this project) notify others (as, Towers) about changes without tight coupling. This makes the code more modular and easier to manage.For example, enemies move, towers within range get notified to attack.

3.4 State

The State pattern lets an object change its behavior based on its internal state and provides smooth transitions between states.This pattern keeps the code clean and organized.

3.5 Flyweight

The flyweight pattern helps reduce memory usage by sharing data such as image between similar objects as Enemy and Tower. This pattern is useful for games with many objects and it reduce memory usage of the game.

3.6 Command

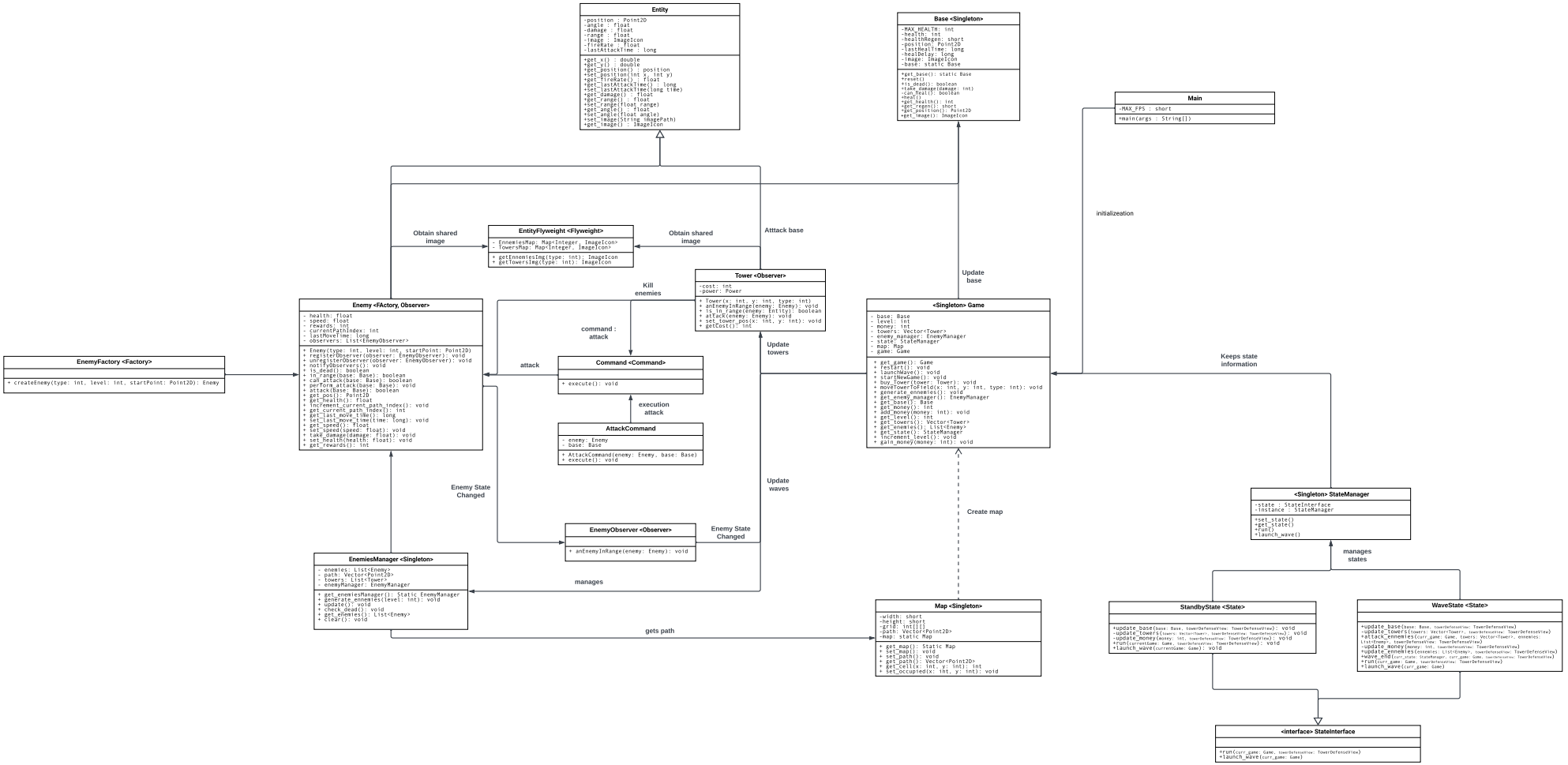
The command pattern encapsulate action such as attacking in case of Towers into objects. This makes the code more flexible for adding new commands in future and allows for features like undo/redo or command queuing which is demonstrated by the Command interface and Attackcommand class.

4 Feedback

This project proved to be quite interesting.Selecting the appropriate design pattern posed a challenge initially, prompting the need for some research and reviewing the course material. However, the implementation phase was relatively complicated. This project took 3 weeks to analyze the problem and finding the right path to develop and reach the final stage of the development.

5 UML

This report contains a UML diagram on the last page, which details the program's functionality.


Entity
-position : Point2D
-angle : float
-damage : float
-range : float
-image : ImageIcon
-fireRate : float
-lastAttackTime : long
+get_x() : double
+get_y() : double
+get_position() : position
+set_position(int x, int y)
+get_fireRate() : float
+get_lastAttackTime() : long
+set_lastAttackTime(long time)
+get_damage() : float
+set_range(float range)
+set_damage(float damage)
+get_angle() : float
+set_angle(float angle)
+set_image(String imagePath)
+get_image() : ImageIcon
Base <Singleton>
-MAX_HEALTH: int
-health: int
-healthRegen: short
-lastHealTime: long
-lastHealTime: long
-healDelay: long
-image: ImageIcon
-base: staitc Base
+get_base(): static Base
+reset()
+is_dead(): boolean
+take_damage(damage: int)
+can_heal(): boolean
+heal()
+get_health(): int
+get_regen(): short
+get_position(): Point2D
+get_image(): ImageIcon
Main
-MAX_FPS : short
+main(args : String[])
initializeation
Obtain shared image
EntityFlyweight <Flyweight>
- EnnemiesMap: Map<Integer, ImageIcon>
- TowersMap: Map<Integer, ImageIcon>
+ getEnnemiesImg(type: int): ImageIcon
+ getTowersImg(type: int): ImageIcon
Obtain shared image
Atttack base
Tower <Observer>
-cost: int
-power: Power
+ Tower(x: int, y: int, type: int)
+ anEnemyInRange(enemy: Enemy): void
+ is_in_range(enemy: Entity): boolean
+ attack(enemy: Enemy): void
+ set_tower_pos(x: int, y: int): void
+ getCost(): int
Update base
Kill enemies
Enemy <FActory, Observer>
- health: float
- speed: float
- reward: int
- currentPathIndex: int
- lastMoveTime: long
- observers: List<EnemyObserver>
+ Enemy(type: int, level: int, startPoint: Point2D)
+ registerObserver(observer: EnemyObserver): void
+ unregisterObserver(observer: EnemyObserver): void
+ notifyObservers(): void
+ is_dead(): boolean
+ is_in_range(base: Base): boolean
+ call_attack(base: Base): boolean
+ perform_attack(base: Base): void
+ attack(base: Base): boolean
+ get_pos(): Point2D
+ get_health(): float
+ increment_current_path_index(): void
+ get_current_path_index(): int
+ get_last_move_time(): long
+ set_last_move_time(time: long): void
+ get_speed(): float
+ set_speed(speed: float): void
+ take_damage(damage: float): void
+ set_health(health: float): void
+ get_reward(): int
<Singleton> Game
- base: Base
- level: int
- money: int
- towers: Vector<Tower>
- enemy_manager: EnemyManager
- state: StateManager
- map: Map
- game: Game
+ get_game(): Game
+ restart(): void
+ launchWave(): void
+ startNewGame(): void
+ buy_tower(tower: Tower): void
+ moveTowerPosition(x: int, y: int, type: int): void
+ get_enemy_manager(): EnemyManager
+ get_base(): Base
+ get_money(): int
+ add_money(money: int): void
+ get_level(): int
+ get_towers(): Vector<Tower>
+ get_enemies(): List<Enemy>
+ get_state(): StateManager
+ increment_level(): void
+ gain_money(money: int): void
Keeps state information
EnemyFactory <Factory>
+ createEnemy(type: Int, level: Int, startPoint: Point2D): Enemy
command : attack
attack
Command <Command>
+ execute(): void
Update towers
execution attack
AttackCommand
- enemy: Enemy
- base: Base
+ AttackCommand(enemy: Enemy, base: Base)
+ execute(): void
Enemy State Changed
Update waves
Create map
EnemyObserver <Observer>
+ anEnemyInRange(enemy: Enemy): void
Enemy State Changed
EnemiesManager <Singleton>
- enemies: List<Enemy>
- path: List<Point2D>
- level: int
- enemyManager: EnemyManager
+ get_enemiesManager(): Static EnemyManager
+ generate_enemies(level: Int): void
+ update(): void
+ check_dead(): void
+ get_enemies(): List<Enemy>
+ clear(): void
manages
Map <Singleton>
-width: short
-height: short
-grid: Int[][]
-path: Vector<Point2D>
-map: static Map
+ get_map(): Static Map
+ get_map(): void
+ set_path(): void
+ get_path(): Vector<Point2D>
+ get_cell(x: int, y: int): int
+ set_occupied(x: int, y: int): void
gets path
<Singleton> StateManager
-state : StateInterface
-instance : StateManager
+set_state()
+get_state()
+fun()
+launch_wave()
manages states
StandbyState <State>
+update_base(base: Base, towerDefenseView: TowerDefenseView): void
-update_towers(towers: Vector<Tower>, towerDefenseView: TowerDefenseView): void
-update_money(money: int, towerDefenseView: TowerDefenseView): void
+run(currentGame: Game, towerDefenseView: TowerDefenseView): void
+launch_wave(currentGame: Game): void
WaveState <State>
+update_base(base: Base, towerDefenseView: TowerDefenseView)
-update_towers(towers: Vector<Tower>, towerDefenseView: TowerDefenseView)
+attack_ennemies(curr_game: Game, towers: Vector<Tower>, ennemies: List<Enemy>, towerDefenseView: TowerDefenseView)
-update_money(money: int, towerDefenseView: TowerDefenseView)
+wave_end(curr_state: StateManager, curr_game: Game, towerDefenseView: TowerDefenseView)
+run(curr_game: Game, towerDefenseView: TowerDefenseView)
+launch_wave(curr_game: Game)
<interface> StateInterface
+run(curr_game: Game, towerDefenseView: TowerDefenseView)
+launch_wave(curr_game: Game)