German University in Cairo
Media Engineering and Technology
Prof. Dr. Slim Abdennadher
Dr. Nourhan Ehab
Dr. Ahmed Abdelfattah

CSEN 401 - Computer Programming Lab, *Spring 2024*
Attack on Titan: Utopia
Milestone 2
Deadline: 26.4.2024 @ 11:59 PM

This milestone is a further *exercise* on the concepts of **object oriented programming (OOP)**. By the end of this milestone, you should have a working game engine with all its logic, that can be played on the console if needed. The following sections describe the requirements of the milestone. Refer to the **Game Description Document** for more details about the rules.

- In this milestone, you must adhere to the class hierarchy established in Milestone 1. You are not permitted to alter the visibility or access modifiers of any variables, unless such changes are explicitly mentioned in the description of this milestone. Additionally, you should follow the method signatures given in this milestone. However, you have the freedom to introduce additional helper methods as needed.
- All methods mentioned in the document should be `public`. All class attributes should be `private` with the appropriate access modifiers and naming conventions for the getters and setters as needed.
- You should always adhere to the OOP features when possible. For example, always use the `super` method or constructor in subclasses when possible.
- The model answer for M1 is available on CMS. It is recommended to use this version. A full grade in M1 doesn't guarantee a 100 percent correct code, it just indicates that you completed all the requirements successfully :)
- Some methods in this milestone depend on other methods. If these methods are not implemented or not working properly, this will affect the functionality and the grade of any method that depends on them.
- **You need to carefully read the entire document to get an overview of the game flow as well as the milestone deliverables, before starting the implementation.**
- You need to think about the boundary values of all variables. For example, you need to make sure that certain variables don't fall below zero
- Before any action is committed, the validity of the action should be checked. Some actions will not be possible if they violate game rules. **All exceptions that could arise from invalid actions should be thrown in this milestone.** You can refer to milestone 1 description document for the description of each exception and when it can occur.

1 Interfaces

Default interface methods in Java allow interfaces to have method implementations. A default method in an interface provides a default implementation that can be used by all implementing classes. It is declared using the default keyword. Implementing classes can choose to override the default method with their own behaviour, or they can use the default implementation provided by the interface. Default method can (and should, when possible,) utilize other non-default methods of the interface in their own implementation. For example in Interface `I` that has a default method `m1()` and a non default method `m2()`, m1 can call m2() even though m2 doesn't have an implementation yet. In summary, default interface methods allow interfaces in Java to provide default implementations for methods, giving flexibility to both the interface and implementing classes.

1.1 Build the Attackee Interface

Name : `Attackee`

Package : `game.engine.interfaces`

Type : Interface

Description : Interface containing the methods available to all objects that gets attacked within the game.

1.1.1 Methods

This interface should include the following methods:

1. `default boolean isDefeated()`: A method that checks if the attackee's current health has reached or fallen below 0.
2. `default int takeDamage(int damage)`: A method that inflict damage to the attackee's current health and returns the value of gained resources in case the attackee was defeated, otherwise returns 0.

1.2 Build the Attacker Interface

Name : `Attacker`

Package : `game.engine.interfaces`

Type : Interface

Description : Interface containing the methods available to all attackers within the game.

1.2.1 Methods

This interface should include the following methods:

1. `default int attack(Attackee target)`: A method that inflict damage to the target and should return the value of gained resources in case the attackee was defeated.

1.3 Build the Mobil Interface

Name : `Mobil`

Package : `game.engine.interfaces`

Type : Interface

Description : Interface containing the methods available to all objects that has mobility (i.e can move) within the game.

1.3.1 Methods

This interface should include the following methods:

1. `default boolean hasReachedTarget()`: A method that checks if a mobil has arrived at the intended target.
2. `default boolean move()`: A method that moves a mobil by changing it's distance. The method should return true if the target was reached.

2 Classes

2.1 Titans Class

For each subclass, distinct behaviors can be implemented by overriding relevant methods if needed to suit their specific functionalities. Consider the special features of each titan below.

Titan's Types	Special Feature
Pure Titan	-
Abnormal Titan	Performs the attack action on target twice per turn (if not already defeated by the first attack)instead of once.
Armored Titan	Only takes quarter of the intended damage when attacked
Colossal Titan	Speed (distance moved) increases by 1 after every movement action

2.2 Weapons Class

2.2.1 Methods

This class should include the following abstract method that behaves differently for each type of weapon.

1. `abstract int turnAttack(PriorityQueue<Titan> laneTitans)`: A method that performs a certain damage to all titans on a lane and returns the value of resources gained if any of the attackees got defeated. Any defeated titan should be removed.

This certain damage is different for each subclass, so each class should override `turnAttack(laneTitans)` to suit the following features

Weapon's Types	Attacking Feature
Piercing Cannon	Attacks the closest 5 titans if they exist
Sniper Cannon	Attacks the closest titan if it exists
VolleySpread Cannon	Attacks any titan within the specified range indicated by the minimum and maximum
Wall Trap	Attacks the closest titan only if it reached the base/wall

2.3 Lane Class

Name : `Lane`

Package : `game.engine.lanes`

Type : Class.

2.3.1 Methods

1. `void addTitan(Titan titan)`: A method that adds the given titan in the lane.
2. `void addWeapon(Weapon weapon)`: A method that adds the given weapon in the lane.
3. `void moveLaneTitans()`: A method that moves all the titans present in a lane each turn if they have not reached the base/wall yet.
4. `int performLaneTitansAttacks()`: A method that performs the attacks of all the titans present in the lane that has reached the base/wall and returns the number of resources gathered.
5. `int performLaneWeaponsAttacks()`: A method that performs the attack of all weapons present in the lane to the titans present in the same lane and returns the number of resources gathered.
6. `boolean isLaneLost()`: A method that checks if the lane's wall is destroyed.
7. `void updateLaneDangerLevel()`: A method that updates the danger level of the lane based on the number of titans present and their danger level.

2.4 Titan Registry Class

Name : `TitanRegistry`

Package : `game.engine.titans`

Type : Class.

2.4.1 Methods

1. `Titan spawnTitan(int distanceFromBase)`: A method that returns an object of the relevant type of titans based on the registry code attribute. The titan object will be spawned at the input distance.

2.5 Weapon Registry Class

Name : `WeaponRegistry`

Package : `game.engine.weapons`

Type : Class.

2.5.1 Methods

1. `Weapon buildWeapon()`: A method that returns an object of the relevant type of weapon based on the registry code attribute.

2.6 Weapon Factory Class

Name : `WeaponFactory`

Package : `game.engine.weapons.factory`

Type : Class.

Description : A class representing the `WeaponFactory`, which is used to store the available weapons to be bought during each turn.

2.6.1 Methods

1. `FactoryResponse buyWeapon(int resources, int weaponCode) throws InsufficientResourcesException`: A method that takes available resources and weapon code as inputs to purchase the specified weapon from the `weaponShop` indicated by the code if there are available resources. In case of a successful purchase, it returns a `FactoryResponse` containing the weapon and the remaining resources after deducting the weapon's price.
2. `void addWeaponToShop(int code, int price)`: A method that adds a `WeaponRegistry` with the given parameters to the `weaponShop` with its code.
3. `void addWeaponToShop(int code, int price, int damage, String name)`: A method that adds a `WeaponRegistry` with the given parameters to the `weaponShop` with its code.
4. `void addWeaponToShop(int code, int price, int damage, String name, int minRange, int maxRange)`: A method that adds a `WeaponRegistry` with the given parameters to the `weaponShop` with its code.

Game Setup

2.7 Battle Class

Name : `Battle`

Package : `engine`

Type : Class

Description : A class representing the `Game` itself. This class will represent the main engine that manages the flow of the game.

2.7.1 Methods

This class should include the following additional methods:-

1. `void refillApproachingTitans()`: A method that refills approaching titans based on the codes of titans present in the current phase.
2. `void purchaseWeapon(int weaponCode, Lane lane) throws InsufficientResourcesException, InvalidLaneException`: A method that deploys the specific weapon indicated by the input code and adds it to the input lane if it's not lost (Weapons can only be added to active lanes). Resources should be updated after a successful purchase. The rest of the turn actions should be performed afterwards.
3. `void passTurn()`: A method that skips buying weapon action. The rest of the turn actions should be performed afterwards.
4. `private void addTurnTitansToLane()`: A method that adds `Titan`s from `approachingTitans` to the least dangerous active lane based on the the `numberOfTitansPerTurn`. If `approachingTitans` is empty, it should be refilled based on the phase.
5. `private void moveTitans()`: A method that performs the move action for all titans in all of the active lanes present in the game.
6. `private int performWeaponsAttacks()`: A method that performs weapon attacks for all of the active lanes present in the game. It should return the accumulated resources gathered by all the defeated titans.
7. `private int performTitansAttacks()`: A method that performs titan attacks for all of the lanes present in the game. It also returns the attack's result (Wall's resource value in case of its destruction).
8. `private void updateLanesDangerLevels()`: A method that updates the danger level of all of the active lanes present in the game.

9. `private void finalizeTurns()`: A method whose main functionality is to update the number of turns and switch the phases according to the following table.

Number of Turns	Phase
Below 15	EARLY
Below 30	INTENSE
More than or equal to 30	GRUMBLING
More than 30 and divisible by 5	GRUMBLING and doubles the number of titans

10. `private void performTurn()`: A method that performs the main functionalities throughout each turn. The turn cycle encompasses several key actions: titans in all lanes execute movement and attacks, weapons in all lanes carry out their attacks, new titans are introduced, the danger level is adjusted, and the turn is subsequently updated to proceed. Check the game description document for the order of these steps.

11. `boolean isGameOver()`: A method that checks the game over condition of the game. Returns true if game is over (All the lanes are lost) and false otherwise.