German University in Cairo
Media Engineering and Technology
Prof. Dr. Slim Abdennadher

Computer Programming Lab, *Spring 2021*
Empire Building: **Milestone 2**

Deadline: 11.06.2021 @ 23:59

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.

- Please note that you are not allowed to change the hierarchy provided in milestone 1 nor the visibility/access modifiers of variables except as explicitly stated in this milestone description. You should also conform to the method signatures provided in this milestone. However, you are free to add more helper methods. You should implement helper methods to include repetitive pieces of code.
- 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 the MET website. 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.
- Throughout the whole milestone, you should always think about which exceptions should be thrown and when and where they should be thrown. You can refer to milestone 1 description document for the description of each exception and when it can occur.
- **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 boundaries value of all variables. For example, you need to make sure that certain variables don't fall bellow zero**

Game Logic

1 Buildings

1.1 `Building` Class

You should add the following methods to the previously built class:

1. `public void upgrade() throws BuildingInCoolDownException, MaxLevelException` : perform the action of upgrading the building. Carefully think where this method should be implemented. According to each building type, you should upgrade the `level` and `upgradeCost` and `recruitmentCost` based on the below table. Make sure to update the `coolDown` value after performing the action

Building Type	Building Level	upgradeCost
Farm	2	700
Market	2	1000

Building Type	Building Level	upgradeCost	recruitmentCost
Archer Range	2	700	450
Archer Range	3	-	500
Barracks	2	1500	550
Barracks	3	-	600
Stable	2	2000	650
Stable	3	-	700

1.2 `EconomicBuilding` Class

You should add the following methods to the previously built class:

1. `public abstract int harvest()`: Abstract method to be implemented in the respected subclass. should return an amount of gold that this building produces for each building level. The information of each building is given below.

Building Type	Building Level	Gold
Farm	1	500
Farm	2	700
Farm	3	1000
Market	1	1000
Market	2	1500
Market	3	2000

1.3 `MilitaryBuilding` Class

You should add the following methods to the previously built class:

1. `public abstract Unit recruit() throws BuildingInCoolDownException, MaxRecruitedException`: abstract method to be implemented in the respected subclass. Should return a unit based on the building type and level, refer to M1 for the information of each unit. **Make sure to update the `currentRecruit` value after performing the action.**

2 Units

2.1 `Unit` Class

2.1.1 Attributes

You should add the following attribute to the previously built class. All the class attributes are READ and WRITE unless otherwise specified.

1. `Army parentArmy`: An attribute representing the parent army of a unit.
2. `int currentSoldierCount`: The current soldier count of the unit should start with the `maxSoldierCount`, make sure to update the previously implemented variable in the constructor.

2.1.2 Methods

You should add the following methods to the previously built class:

1. `public void attack(Unit target) throws FriendlyFireException`: This method performs the attacking of a unit on the target unit. If a unit attacks another unit it will decrement from the `currentSoldierCount` of the target unit by a certain factor based on the attacking unit soldier count, this factor will be given later under each unit type. For example, if Unit A attacks unit B and the factor is 0.3, then unit B soldier counts will decrease by 0.3* Unit A's soldier count. Think about what will happen if the target unit's soldiers count reaches zero. **Make sure to call the appropriate method for handling the case when the target's soldier count reaches zero**

2.2 `Archer` Class

You can find below what will happen if an Archer attacks another unit with respect to the unit level.

Archer Level	Target type	Factor
1	Archer	0.3
2	Archer	0.4
3	Archer	0.5
1	Infantry	0.2
2	Infantry	0.3
3	Infantry	0.4
1	Cavalry	0.1
2	Cavalry	0.1
3	Cavalry	0.2

2.3 `Infantry` Class

You can find below what will happen if an Infantry attacks another unit with respect to the unit level.

Infantry Level	Target type	Factor
1	Archer	0.3
2	Archer	0.4
3	Archer	0.5
1	Infantry	0.1
2	Infantry	0.2
3	Infantry	0.3
1	Cavalry	0.1
2	Cavalry	0.2
3	Cavalry	0.25

2.4 `Cavalry` Class

You can find below what will happen if a Cavalry attacks another unit with respect to the unit level.

Cavalry Level	Target type	Factor
1	Archer	0.5
2	Archer	0.6
3	Archer	0.7
1	Infantry	0.3
2	Infantry	0.4
3	Infantry	0.5
1	Cavalry	0.2
2	Cavalry	0.2
3	Cavalry	0.3

3 Army

You should add the following methods to the previously built class:

1. `public void relocateUnit(Unit unit) throws MaxCapacityException`: This method adds the given unit to the ArrayList `units`. Additionally, it should remove the unit from the previous army and add it to the corresponding army.
2. `public void handleAttackedUnit(Unit u)`: This method should remove the given unit from the ArrayList of units if all soldiers in the given unit have died in the battle.
3. `public double foodNeeded()`: A method that calculates the food needed from each unit inside the army based on the unit's status. For each unit, you should multiply the relative keepUp value by the unit `currentSoldierCount` to get the amount needed by this unit.

4 City

4.1 `City` Class

4.1.1 Attributes

You should update the previously implemented variables.

1. `Army defendingArmy`: make sure to initialize the defending army of the city in the constructor.

5 Player

5.1 `Player` Class

You should add the following methods to the previously built class:

1. `public void recruitUnit(String type,String cityName) throws BuildingInCoolDownException, MaxRecruitedException, NotEnoughGoldException`: This method should perform the action of recruiting units. First, it should get the given city from the ArrayList `controlledCities`, then should check on the unit type and get the corresponding building from ArrayList `militaryBuildings`. Finally, it should add the recruited units in the defending armies of the given city and updates the player's `treasury` by removing the unit's `recruitmentCost` value from it. Don't forget to check that the player has enough gold for recruiting the unit before recruiting units. Think about the `parentArmy` of the recruited unit.
2. `public void build(String type,String cityName) throws NotEnoughGoldException`: This method should perform the action of adding a new building to the given city. First, it should get the given city from the ArrayList `controlledCities`, then should create a new building based on the building type. Carefully think about whether you should add this building to Economical Buildings or to Military Buildings. Finally, you should update the player's `treasury` by removing the building's `cost` value from it. Don't forget to check that the player has enough gold for adding a new building. Make sure to update the `coolDown` value after performing the action. **As per the game rules, player can only have one building from each type.**
3. `public void upgradeBuilding(Building b) throws NotEnoughGoldException, BuildingInCoolDownException, MaxLevelException` : This method should upgrade the given building by calling the appropriate method and updating the player's `treasury` by removing the building's `upgradeCost` value from it. Don't forget to check that the player has enough gold for upgrading the building.
4. `public void initiateArmy(City city,Unit unit)`: This method handles initiating a new army to attack, the method should perform the following:
   - add the given units to the new army units
   - remove the given unit from the given city's defending army
   - update the `parentArmy` of the given unit.
   - adds the new army to the controlled armies.
5. `public void laySiege(Army army,City city) throws TargetNotReachedException, FriendlyCityException`: This method should handle the sieging process by updating the `currentStatus` of the given army and setting the given city under siege by updating the corresponding values.

6 Game

6.1 `Game` Class

You should add the following methods to the previously built class:

1. `loadArmy(String cityName, String path)`: You should update the implemented method to initialize the parent army of the created units.
2. `public void targetCity(Army army, String targetName)`: This method assigns a target to the given army by updating the `distancetoTarget` to be the distance between the current city and the target city. Think about how you will get the correct distance between the two cities. If the army is on road to another city then you can't send it to another city unless it reaches the target city first.

3. `public void endTurn()`: This method performs the actions needed at the end of each turn, you can find below the logic of ending turn:
    - should increment the turn count
    - should return the cooldown of all buildings in the player cities to its initial status.
    - should reset the `currentRecruit` value of all military buildings in the player cities.
    - should increment the resources of all economical buildings in the player cities by the corresponding values.
    - should calculate the food needed by each army in the player's controlled armies.
    - if the army has a target, then each turn the distance to target should be decremented by 1. If the army reaches its target, you should update the corresponding values.
    - if the player food reaches 0, all army's units should lose 10 percent of its `currentSoldierCount`
    - if any city is under siege, you should increment `turnsUnderSiege` by 1 and decrements the `currentSoldierCount` of the city's defending army units by 10 percent
4. `public void occupy(Army a,String cityName)` : This method should add the given city to the player `controlledCities`. Additionally it should assign the defending army of this city to the given army. Make sure to update the `underSiege` and `turnsUnderSiege` value.
5. `public void autoResolve(Army attacker, Army defender) throws FriendlyFireException`: This method should perform the autoResolve action. A player can choose to auto-resolve the attacking process. The auto resolve method should work as follows:
    - The battle should keep running until one of the attacker or defender army's unit size reaches 0.
    - it should work on a turn-based.
    - each turn you should get a random unit from the attacker and defender armies
    - these units will keep fighting each other each turn by calling attack method. For example, if the attacker units attack this turn, then the defender unit will attack the next turn and so on till the battle ends.

    At the end of this method, you should check if the attacker armies won the battle and if so, occupy method should be called with the appropriate parameters.
6. `public boolean isGameOver()` : This method should check if the game is over or not. The game ends if the player conquers all the cities or the player exceeded the maximum amount of turns.