Middle East Technical University Department of Computer Engineering

CENG 443

Introduction to Object Oriented Programming Languages and System

Spring 2018-2019

Homework 1 - Zombie/Soldier Simulation

Due date: 06 04 2019, Saturday, 23:59

1 Introduction

The objective of this assignment is to learn the basics of object oriented design, java programming language and reflection. Your task is to implement a zombie soldier simulation on a continuous map with fixed dimensions. You will write javadoc for your implementation and explain your design choices in a simple document. There are three types of zombies and soldiers. Zombies and soldiers have certain states and will take action based on their types and states. Soldier use bullets to to shoot down zombies, and bullets are also part of the simulation. All simulation objects will use the same function to act in the simulation and simulation controller will use this function to simulate every object. The design of your solution should be based on object oriented design principles and it is explained to you in the homework text. In this homework, you will not implement a main function and we will use your classes to run the simulation.

2 Simulation Classes and Their Details

Your implementation should consists of following classes:

- **SimulationObject:** Soldiers, Zombies, Bullets, and a parent class that encapsulates the common features of these objects are SimulationObject classes. Each simulation object will have a name, position, direction, speed attributes and step method for simulation controller to simulate these objects.
  - **Zombie:** There are three zombie types you need to implement in the simulation. Zombie represents the parent class that all zombie types derived from. Each type of zombie derived from this base class behaves according to its type and state. It has zombie type, collision range, detection range as constant double attributes and zombie state as a changeable enumeration attribute. SlowZombie, RegularZombie and FastZombie are the zombie types. Your design is expected to consider the fact that some new zombie types can be added in the future whereas you may assume that zombie states are fixed.
  - **Soldier:** There are three soldier types you need to implement in the simulation. Soldier represents the parent class all the soldier types derived from. Each type of soldier derived from this base class behaves according to its type and state. It has soldier type, collision range, shooting range as constant double attributes and soldier state as a changeable enumeration attribute. Commando, RegularSoldier and Sniper are the soldier types. Your design is expected to consider the fact that some new soldier types can be added in the future whereas you may assume that soldier states are fixed.

- **Bullet:** Represent the bullets shot by soldiers in the simulation. It has no additional attributes other that what it should inherit from the SimulationObject.

- **ZombieState:** Enumeration to represent the two states a zombie can be in. Assumed to be non-extendable in the future. Possible values are:
  1. WANDERING
  2. FOLLOWING

- **SoldierState:** Enumeration to represent the three states a soldier can be in. Assumed to be non-extendable in the future. Possible values are:
  1. SEARCHING
  2. AIMING
  3. SHOOTING

- **ZombieType:** Enumeration to represent the three types of a zombie. Assumed to be extendable in the future. Possible values are:
  1. SLOW
  2. REGULAR
  3. FAST

- **SoldierType:** Enumeration to represent the three types of a soldier. Possible values are:
  1. COMMANDO
  2. REGULAR
  3. SNIPER

- **Position:** Contains double x and y parameters to represent coordinates and direction of your simulation objects. It has already been provided to you.
  Important Note: Direction of your objects should always be a normalized value. Do not forget to normalize your the values when it is used for direction.

- **SimulationController:** Most important class in the entire simulation. Controls all simulation objects and map dimensions. All simulation objects interact with this class. Internal representation of this class is mostly left to you. You need to implement four methods in this class that I will be using to black-box grade your homework. The methods are:
  - **addSimulationObject:** Adds the simulation object given in the parameter to the simulation. You can store the simulation object in any way you want. You can store different simulation objects in different lists or arrays by determining their class using Reflection or store them in the same list.
  - **removeSimulationObject:** Removes the simulation object given in the parameter from the simulation.
  - **stepAll:** Simulates the objects in the given simulation. There is no specific order of simulation. You can simulate them in any order you want. Please note that any simulation object added or removed from the simulation during iteration of objects will be problematic. Therefore simulation objects should be added after all step functions are called. Similarly removed objects should be removed after all step functions are called. Moreover, if an object is removed prior to calling their step function, they should not be simulated.
    For example: If a bullet hits a zombie before zombie's step function is called, zombie should not be simulated and removed after all other object simulations are finished.

- **isFinished:** Checks whether there are only zombies or soldiers left in the simulation. If it is, should return `true`. If there is no zombie or soldier left, should also return `true`.

Behaviour and default values of the bullet, zombie and soldiers classes are explained in the next section.

3 Class Behaviour and Default Values

In this section, default values and behaviour of bullet, all zombie and soldier types are explained. During operation objects report their changes and actions to standard output in a specific format. This will be covered in the next section.

3.1 Bullet

3.1.1 Bullet Values

Name of the bullet object should be in this format:

```
Bullet<bullet_no>
```

`bullet_no` is the current avaliable bullet number starting from 0 and incremented for every bullet instance. Bullet position and direction should be the same position and direction of the soldier firing it. Speed depends on the soldier type. Values are given below:

- **RegularSoldier:** 40.0
- **Commando:** 40.0
- **Sniper:** 100.0

3.1.2 Bullet Behaviour

Bullets should enter simulation in the next stepAll function call. This means that bullets should be added to the simulation after all objects are simulated in the current stepAll. Bullets should only be simulated for one step. After that one step, they should be removed from the simulation.
Bullets move fastest in the simulation. Therefore, if you calculate the next position of the bullet directly and check for collision with a zombie, your conclusion may be wrong. To solve this problem, you should divide the single bullet step into multiple small steps depending on its speed. Overall a bullet instance should follow this behaviour:

- Divide the speed into 1.0 to calculate the N small steps
- For every small steps between $[0, N)$:
  - Calculate the euclidean distance to the closest zombie using bullet's and zombie's position. Formula:

$$distance = \sqrt{(Object1_X - Object2_X)^2 + (Object1_Y - Object2_Y)^2}$$

  - If the distance is smaller than or equal to the collision distance of the zombie, remove the zombie and the bullet from the simulation and exit loop
  - Calculate the next position of the bullet with formulate:

$$position = position + direction * 1.0$$

  - If the bullet moved out of bound, exit loop

3.2 Soldier

3.2.1 Soldier Values

Name and position of all soldiers are given to the soldier in the constructor. Direction of every soldier should be randomly selected and normalized at the beginning of the first step call. All soldiers start in SEARCHING state value. In short:

- **Name:** Given in constructor
- **Position:** Given in constructor
- **Direction:** Randomly selected during first step call
- **State:** SEARCHING

Values specific to the soldier types are given below:

- **RegularSoldier:**
  - **Type:** REGULAR
  - **CollisionRange:** 2.0
  - **ShootingRange:** 20.0
  - **Speed:** 5.0
- **Commando:**
  - **Type:** COMMANDO
  - **CollisionRange:** 2.0
  - **ShootingRange:** 10.0
  - **Speed:** 10.0
- **Sniper:**
  - **Type:** SNIPER
  - **CollisionRange:** 5.0
  - **ShootingRange:** 40.0
  - **Speed:** 2.0

3.2.2 Soldier Behaviour

Soldier behaviour depends on its state and its type. For every soldier type and state behaviour is explained.
RegularSoldier:

- **State:** SEARCHING
  - Calculate the next position of the soldier with formula:

$$new\_position = position + direction * speed$$

  - If the position is out of bounds, change direction to random value
  - If the position is not out of bounds, change soldier position to the `new_position`.
  - Calculate the euclidean distance (3.1.2) to the closest zombie.
  - If the distance is shorter than or equal to the shooting range of the soldier, change state to AIMING.
- **State:** AIMING
  - Calculate the euclidean distance (3.1.2) to the closest zombie.
  - If the distance is shorter than or equal to the shooting range of the soldier, change soldier direction to zombie and change state to SHOOTING. Do not forget to normalize the direction. Formula for direction vector:

$$direction(X, Y) = (target\_position_X - object\_position_X, target\_position_Y - object\_position_Y)$$

Next two parts are for normalization

$$length = \sqrt{(direction_X^2) + (direction_Y^2)}$$
$$direction(X, Y) = (direction_X / length, direction_Y / length)$$

  - If not, change state to SEARCHING
- **State:** SHOOTING
  - Create a bullet. As mentioned before, bullet's position and direction should be same as soldier's. Speed depends on the soldier which for RegularSoldier is 40.0. Add the bullet to the simulation after all step function are executed.
  - Calculate the euclidean distance (3.1.2) to the closest zombie.
  - If the distance is shorter than or equal to the shooting range of the soldier, change state to AIMING.
  - If not, randomly change soldier direction and change state to SEARCHING.

Commando:

- **State:** SEARCHING
  - Calculate the euclidean distance (3.1.2) to the closest zombie.
  - If the distance is shorter than or equal to the shooting range of the soldier; change soldier direction to zombie, change state to SHOOTING and return.
  - Calculate the next position of the soldier (3.2.2).
  - If the position is out of bounds, change direction to random value.
  - If the position is not out of bounds, change soldier position to the `new_position`.
  - Calculate the euclidean distance (3.1.2) to the closest zombie.
  - If the distance is shorter than or equal to the shooting range of the soldier; change soldier direction to zombie, change state to SHOOTING.
- **State:** SHOOTING
  - Create a bullet. As mentioned before, bullet's position and direction should be same as soldier's. Speed depends on the soldier which for Commando is 40.0. Add the bullet to the simulation after all step functions are executed.
  - Calculate the euclidean distance (3.1.2) to the closest zombie.
  - If the distance is shorter than or equal to the shooting range of the soldier, change soldier direction to zombie.
  - If not, randomly change soldier direction and change state to SEARCHING.

Sniper:

- **State:** SEARCHING
  - Calculate the next position of the soldier (3.2.2).
  - If the position is out of bounds, change direction to random value
  - If the position is not out of bounds, change soldier position to the `new_position`.
  - Change state to AIMING.
- **State:** AIMING
  - Calculate the euclidean distance (3.1.2) to the closest zombie.
  - If the distance is shorter than or equal to the shooting range of the soldier, change soldier direction to zombie and change state to SHOOTING. Do not forget to normalize the direction(3.2.2).
  - If not, change state to SEARCHING
- **State:** SHOOTING
  - Create a bullet. As mentioned before, bullet's position and direction should be same as soldier's. Speed depends on the soldier which for Sniper is 100.0. Add bullet to the simulation after all step function are executed.
  - Calculate the euclidean distance (3.1.2) to the closest zombie.
  - If the distance is shorter than or equal to the shooting range of the soldier, change state to AIMING.
  - If not, randomly change soldier direction and change state to SEARCHING.

3.3 Zombie

3.3.1 Zombie Values

Name and position of all zombies are given to the zombie in the constructor. Direction of every zombie should be randomly selected and normalized at the beginning of the first step call. All zombies start in WANDERING state value. In short:

- **Name:** Given in constructor
- **Position:** Given in constructor
- **Direction:** Randomly selected during first step call
- **State:** WANDERING

Values specific to the zombie types are given below:

- **RegularZombie:**
  - **Type:** REGULAR
  - **CollisionRange:** 2.0
  - **DetectionRange:** 20.0
  - **Speed:** 5.0
- **SlowZombie:**
  - **Type:** SLOW
  - **CollisionRange:** 1.0
  - **DetectionRange:** 40.0
  - **Speed:** 2.0
- **FastZombie:**
  - **Type:** FAST
  - **CollisionRange:** 2.0
  - **DetectionRange:** 20.0
  - **Speed:** 20.0

3.3.2 Zombie Behaviour

All zombie types have one common behaviour they perform at the beginning of their step call. It is given below:

- Calculate the euclidean distance (3.1.2) to the closest soldier.
- If the distance is shorter than or equal to the sum of collision distance of zombie and soldier ($distance <= collision\_distance_{zombie} + collision\_distance_{soldier}$), remove the soldier from the simulation and return.

Remaining zombie behaviour depends on its state and its type. For every zombie type and state behaviour is explained.
RegularZombie:

- **State:** WANDERING
  - Calculate the next position of the zombie (3.2.2).
  - If the position is out of bounds, change direction to random value.
  - If the position is not out of bounds, change zombie position to the `new_position`.
  - Calculate the euclidean distance (3.1.2) to the closest soldier.
  - If the distance is shorter than or equal to the detection range of the zombie, change state to FOLLOWING.
- **State:** FOLLOWING
  - Calculate the next position of the zombie (3.2.2).
  - If the position is out of bounds, change direction to random value.
  - If the position is not out of bounds, change zombie position to the `new_position`.
  - Count the number of step zombie has been in FOLLOWING state.
  - If the count is 4, change state to WANDERING.

SlowZombie:

- **State:** WANDERING
  - Calculate the euclidean distance (3.1.2) to the closest soldier.
  - If the distance is shorter than or equal to the detection range of the zombie, change state to FOLLOWING and return.
  - If not calculate the next position of the zombie (3.2.2).
  - If the position is out of bounds, change direction to random value.
  - If the position is not out of bounds, change zombie position to the `new_position`.
- **State:** FOLLOWING
  - Calculate the euclidean distance (3.1.2) to the closest soldier.
  - If the distance is shorter than or equal to the detection range of the zombie, change direction (3.2.2) to soldier.
  - Calculate the next position of the zombie (3.2.2).
  - If the position is out of bounds, change direction to random value.
  - If the position is not out of bounds, change zombie position to the `new_position`.
  - Use the calculated distance to the closest soldier in the first step. If the distance is shorter than or equal to the detection range of the zombie, change state to WANDERING.

FastZombie:

- **State:** WANDERING
  - Calculate the euclidean distance (3.1.2) to the closest soldier.
  - If the distance is shorter than or equal to the detection range of the zombie, change direction (3.2.2) to soldier, change state to FOLLOWING and return.
  - If not calculate the next position of the zombie (3.2.2).
  - If the position is out of bounds, change direction to random value.
  - If the position is not out of bounds, change zombie position to the `new_position`.
- **State:** FOLLOWING
  - Calculate the next position of the zombie (3.2.2).
  - If the position is out of bounds, change direction to random value.
  - If the position is not out of bounds, change zombie position to the `new_position`.
  - Change state to WANDERING.

4 Input&Output

Bullets, zombies and soldiers print certain information to `stdout` during simulation. This section explains the outputs your classes need to print to `stdout` during operation. These classes need to print position information in certain situations, therefore it is explained first.

4.1 Position

Position information printed to the screen in this format:

`(<pos_x>, <pos_y>)`

with only two decimal points after zero. **For example:**

`(54.32, 98.12)`

Position information is never printed to the screen alone. It is printed when soldier, zombie or bullet objects need to print information that contains position class.

4.2 Bullet

Bullet objects need to print the following three type of information to the output.

1. When bullet collides with a zombie:

   `<bullet_name> hit <zombie_name>.<newline>`

   For example:

   `Bullet0 hit Zombie1.`

2. When bullet moves out of simulation bounds:

   `<bullet_name> moved out of bounds.<newline>`

   For example:

   `Bullet0 moved out of bounds.`

3. When bullet completes its step without going out of bounds or hitting a zombie:

   `<bullet_name> dropped to the ground at <bullet_position>.<newline>`

   For example:

   `Bullet0 dropped to the ground at (12.37, 34.43).`

4.3 Zombie

Zombie objects need to print the following four type of information to the output.

1. When zombie changes states:

   `<zombie_name> changed state to <state_name>.<newline>`

   For example:

   `Zombie1 changed state to WANDERING.`

2. When zombie changes position:

   `<zombie_name> moved to <position>.<newline>`

   For example:

   `Zombie1 moved to (12.37, 34.43).`

3. When zombie changes direction:

   `<zombie_name> changed direction to <direction>.<newline>`

   For example:

   `Zombie1 changed direction to (0.33, -0.94).`

4. When zombie kills a soldier:

   `<zombie_name> killed <soldier_name>.<newline>`

   For example:

   `Zombie1 killed Soldier1.`

4.4 Soldier

Soldier objects need to print the following four type of information to the output.

1. When soldier changes states:

   `<soldier_name> changed state to <state_name>.<newline>`

   For example:

   ```
   Soldier1 changed state to AIMING.
   ```

2. When soldier changes position:

   `<soldier_name> moved to <position>.<newline>`

   For example:

   ```
   Soldier1 moved to (12.37, 34.43).
   ```

3. When soldier changes direction:

   `<soldier_name> changed direction to <direction>.<newline>`

   For example:

   ```
   Soldier1 changed direction to (0.33, -0.94).
   ```

4. When soldier fires a bullet:

   `<soldier_name> fired <bullet_name> to direction <direction>.<newline>`

   For example:

   ```
   Soldier1 fired Bullet0 to direction (0.51, 0.86).
   ```

Example simulation output files will be provided to you with your homework.

5 Javadoc and Design Document

You are required to write javadoc for all your classes and their methods, you have implemented. You are not required to write for the ones we supplied completely, however partial class definition we have provided is not included in this exemption as they will be mostly implemented by you as well.
You are also required to write a brief (no longer than 1500 words) document (1) highlighting your design choices and (2) commenting on your design considering the OO design principles covered in the class

6 Specifications

- The codes must be in Java. Templates are provided for you to implement the necessary functions. You are not allowed to change the four methods explained in the SimulationController class and you are also not allowed to change the parameters of specific zombie/soldier constructors. You should be able to grasp the design of the system from homework text and design your solution in this manner. You are also allowed to add your own classes, add fields and methods to the given and your own classes.
- You must code your classes based on the object oriented principles covered in class.
- Your codes will be evaluated with both black and white box testing. For black box texting, make sure you print your outputs in the correct format. White box testing will be done to determine whether you have correctly applied object oriented principles to your solution.
- Put all your source codes under package of your metu username `eXXXXXX` under a Source folder.
- Put your design document under a Doc folder.
- Using any piece of code that is not your own is strictly forbidden and constitutes as cheating. This includes friends, previous homeworks, or the Internet. The violators will be punished according to the department regulations.
- Follow the course page on ODTUClass for any updates and clarifications. Please ask your questions on ODTUClass instead of e-mailing if the question does not contain code or solution.

7 Submission

Submission will be done via ODTUClass. You will submit a single tar file called "hw1.tar.gz" that contain all your source code in Source folder and your design document Doc folder. Your javadoc should be inside your source codes. You do not need to generator javadoc html files. All your source codes should be able to compile with following commands.

```
> tar -xf hw1.tar.gz
> cd Source\student_username
> javac *.java
```

Similarly, I should be able to generate your javadoc files with following commands.

```
> tar -xf hw1.tar.gz
> cd Source\student_username
> javadoc *.java
```

After that I will add my main containing class files to test your solutions before examining them.