- **Holding a static variable for tracking the number of Bullet instances**

I defined a static and private “bulletNo” variable inside Bullet class to track number of instances of the Bullet class, so that I can determine the name of the specific bullet instance as specified in the homework document.

Since it is “static”, I can reach it from everywhere and since it is “private”, it is reachable from only bullet instances. Thus, it does not violates “information hiding” and “encapsulation” which is one of the most important principle of the object-oriented programming paradigm.

- **Don’t remove the bullet object inside “step” function of Bullet**

I didn’t remove the Bullet instance after it completed its life-cycle in the simulation in the Bullet class methods. Instead, I handed over this job to “SimulationController” class, so that I prevented the concurrent modification problem and also I didn’t mix the data with controller. The controller should modify the objects in the simulation , so that the code would be more maintainable.

- **Number of times that “RegularZombie” has been in FOLLOWING state**

I created “private” methods for doing operations related number of times RegularZombie instance in FOLLOWING state. These methods are :

- resetFollowingCount
- incrFollowingCount
- isOutOfFollowingCountBound

With the help of these methods I am able do operations easily and prevent access except from the instance itself.
Also, these methods make the module more maintainable since no hard-coded sequence of code will be wrote by this way and if the specification of this task changed in the future, only thing is necessary is modifying these methods.

- **Put newly created Bullet instances to the simulation after the step function is finished**

I put newly created bullet instances to another list of bullets (as if it was waiting queue of bullets) in order to leave “Bullet” class out of this operation and handed over this job to the “SimulationController” class. Therefore, the Bullet module did not violate the “Single Responsibility Principle” which is one of the SOLID principles.

In addition, I did not provide a “public” method to add Bullet instances to the main list of the simulation (e.g. addSimulationObject) to prevent adding Bullet instances to main list without noticing by the clients. Instead, I provided “addWaitingBullet” method to add Bullet instances to another list called “waitingBullet” (waiting queue).

- **Add separate simulation objects to separate containers**

I overload the “addSimulationObject” and “removeSimulationObject” methods to separate adding operations of objects to separate different lists. In that way, the client will not have to know which method he/she need to call in order to add separate “SimulationObject” ‘s into the simuation. This supports polymorphism in a way.

- **Reduce code repetition**

I implement some methods for commonly used operations. Some of these methods are :

- SimulationController#getClosestZombie
- SimulationController#getClosestSoldier

These two methods, help the object retrieve closest object easily and reduce the overhead of the functions because it **delegates** the job to controller class.

- SimulationObject#getNextPosition
- SimulationObject#isOutOfBounds
- SimulationObject#directionThrough

These three methods, are used for commonly executed sequence of tasks among state actions of the “SimulationObject” instances. Their access modifier is “protected” to prevent non-related class to access these methods. These methods, support re-usability.

- **Soldier , Zombie and Bullet classes**

I added another abstract class between the different types of soldier and zombie classes and the SimulationObject to support Open/Closed principle and increase the re-usability and polymorphism.
I also did not make “step” function abstract in these class and implemented for the use of subclasses (RegularSoldier, SlowZombie etc.) since their state transition is common. I implemented the general template of state transitions and make another abstract functions for separate state actions and leave subclass to implement them such as “search, aim, shoot” or “wander, follow”. By this way, if any one of the specification of the state actions changes the one will need to modify only these methods. This makes code more maintainable.

Also , I created Bullet class that extends SimulationObject abstract class to implement step function in polymorphic way.

- **Separate utility class for printing (logging) tasks**

I created a “Logger” class for differentiate the logging and the simulation task s of the module. This support “loose coupling”. By that way, in the future if logging task is changed or need to be expanded, the one will not need to modify actual simulation objects, instead it will only modify this “Logger” class easily. This will increase the maintainability of the code.

Also, I made “Logger” class “static” in order to not to instantiate the class over and over again for same job. This will reduce the overhead of the memory during execution of the program.