Homework 1 - Design Document

ALPUL, Yasin Fatih (e2098739)

April 6, 2019

1 Overall Design

The following are the classes used in the design:

- Bullet
- Commando
- FastZombie
- Position
- RegularSoldier
- RegularZombie
- SimulationController
- SimulationObject
- SimulationRunner (driver)
- SlowZombie
- Sniper
- Soldier
- SoldierState
- SoldierType

- Zombie
- ZombieState
- ZombieType

The `SimulationObject` is the abstract base class for `Soldier`, `Zombie` and `Bullet` classes. Moreover, `Soldier` is the abstract base class for `RegularSoldier`, `Commando` and `Sniper` classes whereas `Zombie` is the abstract base class for `RegularZombie`, `SlowZombie` and `FastZombie` classes.

In `SimulationController` class, adding new objects is done by checking whether the object is of which derived type of `SimulationObject`. Then, it adds the object to the relevant `List`. Removing is similar.

For simulating a single step, the order is chosen to be as the following:

1. Simulate all soldiers
2. Simulate all zombies
3. Simulate all bullets

Later, necessary operations are taken care of such as clearing the battlefield from dead objects.

For storing the objects in the simulation, bullets are chosen to be stored in `ArrayList` container types, whereas soldier and zombie types are stored in `LinkedList` containers. The reason for that is, since there will be a possibility that a soldier/zombie can be removed after some step in simulation, an efficient remove operation will be needed. One cannot anticipate which soldier/zombie will be removed next. Therefore, in which exact position in the container will be removed cannot be known. So in this scenario, a `LinkedList` would be more efficient than an `ArrayList`. Bullets are stored in `ArrayList` since the list is completely replaced after each step in the simulation.

2 Design Principles

2.1 Minimizing the Accessibility

Data fields of classes have been made `private` where possible. The reason for that is to not expose the internal details of the objects. For instance, `shootingRange` and `collisionRange` fields have been made private and public methods have been provided using those fields. However, note that no direct setters/getters have been provided. Instead, to improve abstaction, high level methods have been provided. For instance, zombie types have a method with the signature `boolean canDetect(Soldier)`, which internally uses its `detectionRange`. Additionally, in the program code, expressions that are very similar to natural language can be written such as `if (this.canDetect(closestSoldier))`. This design improves abstraction.

2.2 Open/Closed Principle

This principle states that, as covered in class, *an entity should be open for extension, yet closed for modification.* This principle can be honored by using abstraction, polymorphism and inheritance. As it is also covered in the next section, it is achieved by using polymorphic types instead of `enum` types for soldier/zombie types.

2.3 The Single Choice Principle

This principle, as covered in class, states that *whenever a software system must support a set of alternatives, ideally only one class in the system knows the entire set of alternatives.* This principle is honored in the sense that, only `SimulationController` class knows about the entire set of soldier/zombie classes. Even then, it doesn't know about the classes which are derived from soldier/zombie abstract base classes. When a `SimulationObject` is added, it only checks for whether the object is a `Soldier`, `Zombie` or a `Bullet`.

2.4 The Liskov Substitution Principle

This principle states that, as covered in class, *functions that use references to base classes must be able to use objects of derived classes without knowing it.* This principle is honored in the sense that, in the `Zombie` class, there are a few methods that take a `Soldier` reference and call methods of it. However, `Soldier` is an abstract base class so there is no actual `Soldier` type whatsoever. In reality, it is either a `RegularSoldier`, a `Commando` or a `Sniper`. But it doesn't know that, it is able to use the reference of the base class without knowing it.

3 Zombie and Soldier Types

In the implementation, *enum* types of zombies and soldiers are not used. Instead, virtual method invocation facilities of JVM have been favored. There are several reasons for that.

1. **Extendability:** In the case of a need for addition of a new soldier/zombie type, a number of changes would be needed to be made to source files. Assuming a check for the type of a soldier/zombie, additional checks for new types would be added. Also, the *enum* types would be recompiled. In this way however, new types would be just plugged and it will work.

2. **Abstraction:** Soldier/zombie type is an internal detail. Users will only need to know whether it is a soldier/zombie or not. User will need a generic soldier object and JVM will call the right method according to its type using `invokevirtual` bytecode.

4 Javadoc Style

While writing javadoc comments, the following conventions have been followed.
Official Oracle Documentation:
`https://www.oracle.com/technetwork/java/javase/documentation/index-137868.html`