CENG 443
Introduction to Object-Oriented Programming Languages and System
Spring 2018-2019
Homework 1 - Zombie/Soldier Simulation Design Document

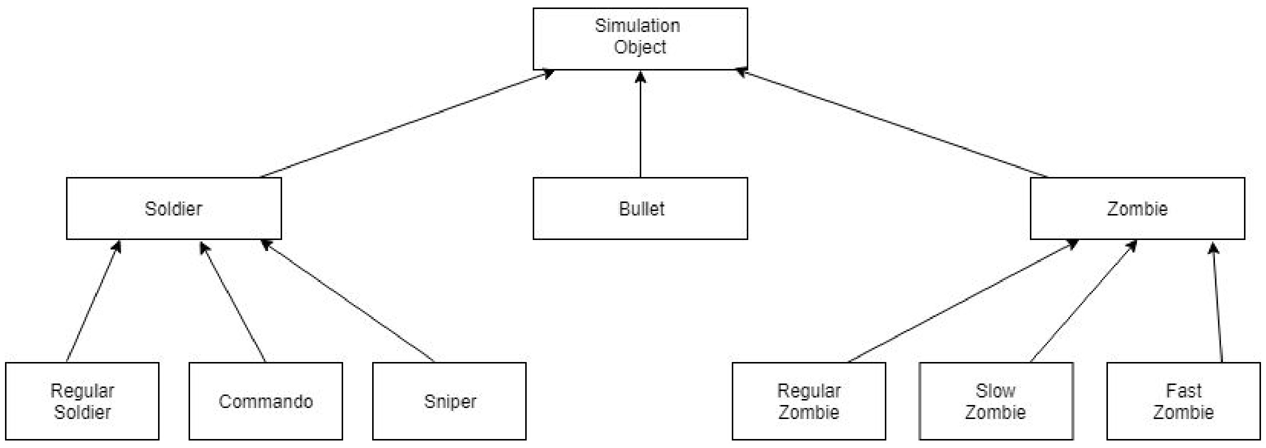
1) Introduction

The purpose of this document is to highlight the design choices and to comment on the design of the project considering the OO principles.

2) Design Choices

2.1) Simulation Object Hierarchy

Simulation objects structure is based on inheritance. SimulationObject.java is the root class for the whole simulation objects. The structure of the simulation objects is as follows



This way of designing omits the code duplication. For example, all objects behave the same when they turn to some position. Therefore, only one implementation of this action is enough. It omits the type checking and lets new types of objects. When a new type of object is added, existing classes will not be affected. Since this structure constructs a “is a” relation, parameter passing is easy and generic. For example, the addSimulationObject() method takes a simulation object of all kind.

2.2) Constants

The constant values of the program are collected in Constants.java file with meaningful names. This increases readability since it removes magic numbers and brings names that are self-describing. It also makes it easy to control and change a value since the values are not spread to the whole project.

2.3) Static Final Variables

I declared variables that are constant and the same for all instances as static final. This prevents unnecessary object creations and saves from memory. For example, the maximum number that a regular zombie is in following state is the same, which is 4, for all regular zombies. So, making it static will prevent the creation of N int variable for N regular zombie instance. Instead, only 1 int is allocated in memory no matter how many regular zombies are created.

2.4) Bullet Factory Functions

Different soldiers create bullets at different speeds. I wrote factory functions that create bullets with different speeds. When a commando creates a bullet, it does not think about the speed and calls the factoryCommandoBullet method without giving the speed. This prevents possible mistakes in giving the wrong value of speed in different soldiers. It makes it easy to read and write the code. For example, in Sniper.java, factorySniperBullet() method is much more meaningful than a constructor like Bullet(some_speed_value).

3) Implementation Choice

This section explains why I removed or changed some of the provided functions and files that are not forbidden to do so. I did not change the four methods explained in the SimulationController class and the parameters of specific zombie/soldier constructors.

3.1) Access Modifiers

I change some of the public methods as package-private since all files are in the same package.

3.2) Soldier & Zombie Types

I did not perform any type checking. So, I removed the soldier and zombie type enum since I did not use them.

3.3) Generating Random Direction

All direction has to be normalized. There is no use of generalizeRandomDirection with false as a parameter. Therefore, I removed the parameter and if statement. The generalizeRandomDirection function return normalized direction all the time as it supposed to.

3.4) String Concatenation

Since strings are mutable, concatenating strings will create a new string object. So, using a string builder is more efficient. Therefore, I first used string builder but IDE gave a warning saying that *"Inspection info: Reports any usages of StringBuffer, StringBuilder or StringJoiner which can be replaced with a single java.lang.String concatenation. Using a String concatenation makes the code shorter and simpler. This inspection only reports when the resulting concatenation is <u>at least as efficient or more efficient</u> than the original code."* So, I concatenate the string directly while printing.

4) OO Design Principles

This section explains the Object Oriented Principles that are applied in this project.

4.1) Open-Closed Principles

This project is easily extendible for new features and different types of objects without requiring any modification to the existing code. For example, when a new type of soldier is added, no need to change the functions like findClosestSoldier because there is no type checking in it. Implementing only the new functionality or the class will be enough and the rest of the program will work fine.

4.2) Single Responsibility Principle

A class in this project has one and only one reason to change, meaning that a class has one job. Changing a requirement will affect only one class, not the others. For example, if the stepping behavior of a slow zombie in wandering state changes, only the one function will be affected by this change.

4.3) Liskov Substitution Principle

In this project, every subclass/derived class can be substitutable for their base/parent class. For example, any type of zombie (slow zombie, fast zombie, regular zombie) can be substitutable for zombie class or simulation class.

4.4) Minimize The Accessibility of Classes and Members

Every class, method, and variable has the narrowest access modifier that it can. This increases the abstraction level of the program. High abstraction level means classes do not know about each other and it reduces complexity. In this project, many variables and methods are private and only their class takes cares of them. Minimizing access variables also provides information hiding which has many benefits like performing side effects or putting constraints on values. For example, setter method for position variable prints the action after setting the position.