**2099 Assignment 3 Design Rationale**



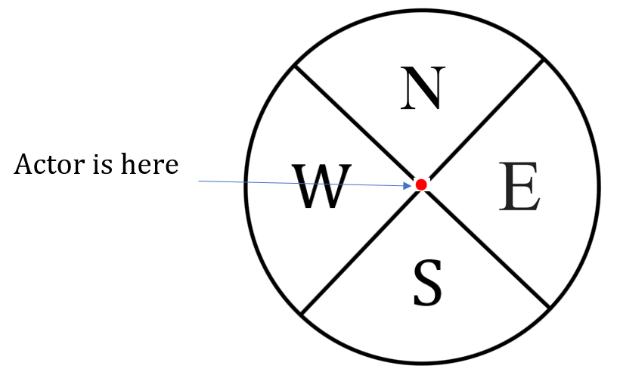
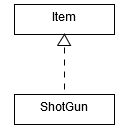
Done by: Ahsan Zafar – 29805805 AND Hin Seng (28088956)

**Contents**

1. ShotGun and Ammunition – Design Choices
2. ShotGun and Ammunition - Code OverView
3. Sniper and Ammunition – Design Choices
4. Sniper and Ammunition – Code Overview.
5. Ending the Game
6. Going to Town
7. Mambo Marie

**ShotGun and Ammunition – Design Choices – Ahsan Zafar**

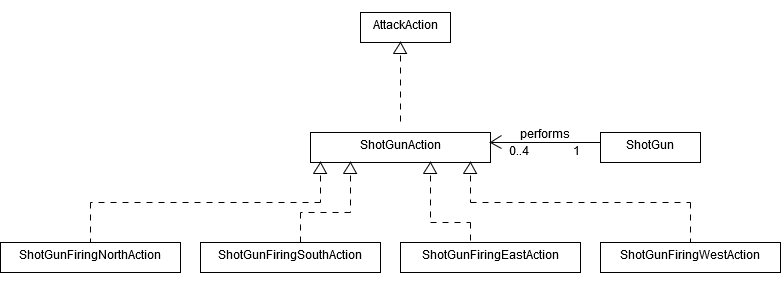
To implement a shotgun, we create a class called “ShotGun” which extends from “Item” class.   
A shotgun has the ability to shoot North, South, East, West. The effect of a shot in each direction is shown in the image below.  
Why ? This is done so that we can re-use the game functionality of a player able to pick an item with no extra code. A ShotGun is an item, so a player can pick it up just like it can pick up other. The Reusability **Principle** allows us to implement easily with no unnecessary code.



To implement a firing action, we create an abstract class “ShotGunAction” which extends “AttackAction”. We then have 4 new classes which inherit the “ShotGunAction” abstract class as shown below. This is shown below.

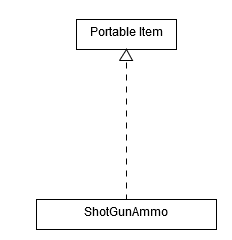
Why ? The use of abstract class allows us to implement the **DRY principle**. By utilizing abstract classes, we can reduce code where functionality is common across those classes.  
“AttackAction” and “ShotGun” action are similar in the sense that both actions are executing an attack on a target, and both need to handle the case when a target is killed (create a corpse etc.).

The “ShotGunFiringNorthAction”, “ShotGunFiringEastAction”, “ShotGunFiringWestAction”, and “ShotGunFiringSouthAction” are all similar once again in the sense that they all perform a firing shot on a target. The only difference is the direction where the bullet is shot. The use of abstract class allows these classes to share common functionality.



To implement ammunition for shotgun, we create a class “ShotGunAmmo” which extends from “Portable item”.  
This ammunition is placed on multiple locations throughout the map.

Why ? By extending from “Portable item”, we can simply re-use the game functionality of a player able to pick an item with no extra code. The Reusability **Principle** allows us to implement easily with no unnecessary code.



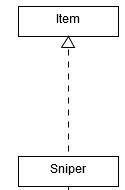
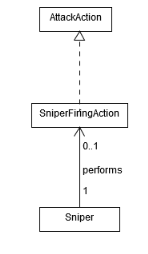
**ShotGun and Ammunition – Code OverView – Ahsan Zafar**

In every turn of the game, a ShotGun object will check if it is on the ground or if it held by a player via the tick( ) method.  
If it is on the ground, the ability of firing a bullet will be removed.  
If it is on a player, it will check if player has ammunition. If ammunition is there, the ShotGun will add the ability of shooting. The options to shoot will be displayed to the console.

To execute a shot, we obtain all the possible location that can be affected by a shotgun firing (there will be 15 such locations). For each location which has an actor on it, we set a hit chance, and if it succeeds, the damage is applied to the actor.

**Sniper and Ammunition – Design Choices – Ahsan Zafar**

To implement a Sniper, we create a class called “Sniper” which extends from “Item” class.   
A Sniper has the ability to shoot any zombie throughout the map.   
Why ? This is done so that we can re-use the game functionality of a player able to pick an item with no extra code. The Reusability **Principle** allows us to implement easily with no unnecessary code.



To implement a firing action, we create a class called “SniperFiringAction” which extends  
 from AttackAction.   
Why ? Similar to sniper, the use of abstract class allows us to implement the **DRY principle**.  
 By utilizing abstract classes, we can reduce code where functionality is common  
 across those classes. “AttackAction” and “SniperFiringAction” action are similar in the sense  
 that both actions are executing an attack on a target, and both need to handle the case when  
 a target is killed.

To implement ammunition for Sniper, we create a class “SniperAmmo” which extends from “Portable item”.  
This ammunition is placed on multiple locations throughout the map.

Why ? Similar to the Shotgun ammunition, by extending from “Portable item”, we can simply re-use the game functionality of a player able to pick an item with no extra code. The Reusability **Principle** allows us to implement easily with no unnecessary code.

**Sniper and Ammunition – Code OverView – Ahsan Zafar**

In every turn of the game, a Sniper object will check if it is on the ground or if it held by a player.  
If it is on the ground, it will remove the capibility of firing a bullet.  
If it is on a player, it will check if player has ammunition. If ammunition is there, the Sniper will add the capibility of shooting. The options to shoot will be displayed to the console.

If it is the first round of using a sniper, it will ask the user to set a target to shoot. Then it will prompt user to either spend a round aiming or shoot. As mentioned in the assignment specification, the longer the aiming, the more accurate the shot. This implementation is performed in the execute( ) method of “SniperFiringAction”.

Additionally, in every turn of the player, we check if any action is performed that will break the concentation of player and lose sight of target. In the playturn( ) method of player, we check what actions were performed in the last round, and alter the concetration accordingly. If a player is attacked and loses health points, we also check this in the execute( ) method in the “AttackAction” class, where we break the concentration.

There is a lot going on in the execute( ) method of “SniperFiringAction”, which is why we have created multiple functions to make the code readable to programmers, allowing us to implement the KIS **Principle** (KIS). Further more, where possible, fields have been kept private, and

**Ending The Game – Ahsan Zafar**

**A “quit game” option in the menu**

To add a QUIT GAME option to the game, we simply add a new class called “quit action” which extends “action”.  
From the way the system has been designed, any option that is displayed on the console is associated with an action.

Now, in each turn, a quit action is added to the allowable actions of the player. This way, the user is always prompted to quit the game.

In the execute( ) method of “QuickAction”, we simply just remove the actor from the map. The rest of the functionality is handled by the engine.

Why ? This allows us to make us of the Reusability **Principle** of the system. We just add a new action to the system, and the engine package will take care of stopping the game. This prevents redundant code and maintains understandibility for a user. Additionally, because this is a very simple design, we implement the KISS **principle** (Keep It Simple).

**A “player loses” ending for when the player is killed, or all the other humans in the compound are killed.  
A “player wins” ending for when the zombies and Mambo Marie have been wiped out and the compound is safe.**

To implement these features, we create a new class “World2” which extends the current “Word” class.

We override the stillrunning( ) method. Previously, it was only checking whether the actor is present on the map. Now we check the following conditions:  
1. Check whether the player is present, or humans are present. If either of them are not present, then stillrunning( ) will return False so that the game is stopped.  
2. Check whether all the zombies (including Mamo Marie) have been killed. If this is the case, then stillrunning( ) will return False so that the game is stopped.  
3. If the above to conditions are not met, then it means that the game still needs to be run, so it will return True.

We utilise the capability of zombies (UNALIVE); to check if any zombies are left on the map, we just need to visit every location of the map and see if there is an actor. If the actor’s capability is UNALIVE, this means that this actor is either a zombie or MamboMarie so the game should continue.

We also override the endGameMessage( ) method in the “World2” class. Now, the stillrunning( ) method communicates with the endGameMessage( ) method and informs the state of the game (i.e. whether player has won the game or player has lost the game). Then, the appropriate message is displayed.

Why ? The use of inheritance here allows us to implement the **DRY principle**. By utilizing inheritance, we can reduce code where functionality common across those classes.  
“World2” essentially operates just like “World”, except for a few additional functionalities. We also design “World2” class to have the **Open/Closed Principle,** similar to “World” class. For maintainability purposes, this code can further be extended to apply new extensions, but complying to the principle, it is closed for modification.

Now, by using the if statements in the stillrunning( ) function, we prevent prevent “premature optimization”. Premature optimisation is when you spend a lot of time on something that you may not actually need. The above approach of having multiple if statements might be a bad idea in another situation as it could potentially lead to hundreds of conditional statements (if there are that many new features to implement). But given the current assignment specification, by using only 2 conditional statements which result in a very low computational cost, we are able to meet the assignment requirement, keep the code simple (therefore abide by the KISS **Principle** (Keep it simple), and prevent premature optimization.

**Going to town - HinSeng**

**6.1 - Place a vehicle somewhere on your existing map**  
*6.1.1 – Place a vehicle somewhere on your existing map. The vehicle should provide the player with the option to move to a town map.*We will first create a vehicle class. Then we will add MoveActorAction to the vehicle it provides an option for the player to move to the town map. The town map will be a list of string just like the existing map.

Why this way has been chosen:

Vehicle is a new item so we decide to create a Vehicle class inherited from Item class and set that it is not portable. The **Reusability Principle** allows us to implement easily with no unnecessary code. We then add MoveActorAction for the vehicle so when player steps on it, it provides an option for player to move to the town map.

**6.2 - Somewhere on the town map, place a sniper rifle and a shotgun**  
*6.2.1 – Somewhere on the town map, place a sniper rifle and a shotgun.*We will first create sniper and shotgun class, then we place these items on the town map.

Why this way has been chosen:

It is the same as the plank in the existing map, so we just simply follow the way.

**6.3 - When you leave a map, any creatures in the old map should continue to move and act**  
*6.3.1 – When you leave a map, any creatures in the old map should continue to move and act.*

When we create the new town map, we will add this town map to the same world as the existing map did. The two map will then run synchronously so even the player leave a map, any creatures in the old map will continue to move and act.

Why this way has been chosen:

World class is the main class to let the creatures move and act, so we just need to ensure all the maps are added into the same world so these maps can run synchronously.

**Mambo Marie - HinSeng**

**8.1 - Mambo Marie is a Voodoo priestess and the source of the local zombie epidemic. If she is not currently on the map, she has a 5% chance per turn of appearing**

*8.1.1 – Mambo Marie is a Voodoo priestess and the source of the local zombie epidemic. If she is not currently on the map, she has a 5% chance per turn of appearing.*We will first create a MamboMarie class as she is a new zombie. Then we will create a tracker class to keep track whether she is existing on the map, and set the probability to 5% of appearing.

Why this way has been chosen:

Mambo Marie is a new zombie so she is inherited from zombie actor. The **Reusability Principle** allows us to implement easily with no unnecessary code. We will create a new class called Tracker to keep track the Mambo Marie. This class is inherited from gameMap class and we will override the tick method to check whether Mambo Marie exists on the map. Tracker also uses the **Reusability Principle** allows us to implement easily with no unnecessary code.

**8.2 - She starts at the edge of the map and wanders randomly**

*8.2.1 – She starts at the edge of the map and wanders randomly.*We just need to add WanderBehaviour for Mambo Marie to let her wander randomly. In Tracker class, we need to get the X range and Y Range of the map and the minimum and maximum value of them. Then, add these locations to a list and spawns the Mambo Marie at one of them.

Why this way has been chosen:

We already have WanderBehaviour so we can just apply this behaviour for Mambo Marie. For getting the edge of the map, this is a easy way to get the edges as the gameMap already have the functions getXRange() and getYRange().

**8.3 - Every 10 turns, she will stop and spend a turn chanting. This will cause five new zombies to appear in random locations on the map. If she is not killed, she will vanish after 30 turns**

*8.3.1 – Every 10 turns, she will stop and spend a turn chanting. This will cause five new zombies to appear in random locations on the map. If she is not killed, she will vanish after 30 turns.*

Within the MamboMarie class, we will have a variable to count the turns of Mambo Marie and perform different actions according to the turns. We will create a class called ChantingBehaviour which generates ChantingAction and let Mambo Marie to spawn five new zombies in random locations on the map.

Why this way has been chosen:

It is because there is a method called playTurn in Actor class and we can override this method to let it count the turns of Mambo Marie. Chanting is a new behaviour so we need to create it and let Mambo Marie to use it.

**8.4 - Mambo Marie will keep coming back until she is killed**  
*8.4.1 – Mambo Marie will keep coming back until she is killed.*

We just need to use the isConscious method of the Actor class to check whether Mambo Marie is alive. If yes, she can come back after she vanishes. If no, she cannot come back.

Why this way has been chosen:

This is the easiest way to determine whether Mambo Marie is alive and perform different actions according to this. **DRY Principle** is used as the method isConscious is already in Actor class so we can just simply use it.