FIT2099 Assignment 3(Updates)

Design Rationale

(Braaaains)

Group Name	:	Highroll
Group Members	:	Tan Weihan
		Wong Jun Kang
Tutor	:	Dr Jasbir Kaur Dhaliwal
Tutorial Datetime	:	Tuesday, 4:00pm-6:00pm
Assignment	:	Assignment 3

Design Rationale

Package

All the classes are all grouped under the game package. This helps ensure that elements that depend on each other are kept inside an encapsulation boundary. Sub packages like weapons, actions, actors and behaviours will also be created to provide further encapsulation to the application.

Zombie

Zombie arm and zombie limb will be implemented in Zombie class as variables, along with it's getters. This is because we want to fulfil the Single-responsibility principle and let zombie class deal with functionalities of zombie class. We split the zombie's limbs into arms and legs as they affect the zombie in different ways when the limbs are lost. Also, the variables of Zombie arm and leg are both declared as private instead of public. By doing so, we have declared things on the tightest possible scope. This ensures that other methods have no access to the variable besides from the Zombie class itself. All processes associated with modifying or getting the variable must be performed through the accessor or the mutator.

When a zombie is attacked, there will be a 25% of the zombie to lose at least 1 limb. We implement this in Zombie class by using a random number generator that determines the number of limbs lost when the zombie is attacked, while dropArm and dropLeg handles the losing of arms and legs separately. We feel that this implementation is best as all the calculations and the loss of limbs are done in Zombie class, which fulfils the single responsibility principle, along with the principle that classes should be responsible for their own properties. DropArm and dropLeg methods are implemented separately because while both do make the zombie lose limbs, dropArm needs to calculate another event caused by losing arms, which is dropping weapons held by zombies.

When a zombie loses a number (N) of arms, the zombie will go through a method dropArm() where the arm number reduces by N and N amount of ZombieLimb objects will be created. A random number generator in dropArm() will also be used to determine whether the weapon used by the zombie will be dropped. All weapons will be dropped if the zombie loses all arms and the zombie will only attempt to bite the target. We do this by limiting the number of weapons zombies can pick to 1. This ensures that when a weapon is dropped, zombies can't just use another weapon in inventory to attack. This also ensures that zombie does not use weapons to attack when they have no arms, along with making zombie feel more zombielike.

When a zombie loses a number (N) of legs, the zombie will go through a method dropLeg() where the leg number reduces by N and N amount of ZombieLimb objects will be created.

Movement Speed
In the PlayTurn() method, there will be an if condition that checks the last Action of the zombie if it only has 1 leg. It will only allow certain behaviours depending on the last action of the zombie. Any movement behaviours will not be allowed if the zombie contains no legs. Other behaviours such as attackBehaviour will not be restricted by the number of legs. This is done to fulfil the encapsulate what varies principle, as other classes that call this method do not need to know details like the number of legs the zombie object has.

Zombie's Bite
GetIntrinsicWeapon() function will be modified to include "bite" as a potential attack. The chance to bite will vary according to the number of arms the zombie has. The chances will be calculated using a random number generator within the method. This is done to fulfil the encapsulate what varies principle, as other classes that call this method do not need to know details like the chance of zombie using bite attack or the number of limbs the zombie has.

Zombie's speech action and speech behaviour will be implemented as separate classes that extend Action class and behaviour class separately. This is to maintain consistency within the program. Speech behaviour will have a 10% chance to be called.

Zombies will have access to PickUpBehaviour(), which contains PickUpItemAction(). This allows zombies to pick up and use weapons on the ground. This is done to fulfil the single responsibility principle, as we call PickUpBehaviour(), which handles item pickup for all non-playable characters(NPC). This also fulfils the don't repeat yourself (DRY) principle, as we do not write similar code multiple times in different locations to complete a similar functionality.

AttackAction
The healing effects and hit chance of Zombie's bite attack are implemented in AttackAction. We implement it as such because hit chance itself is already calculated in AttackAction. If we implement it anywhere but AttackAction, we would not be following the single responsibility principle, and we would also not fulfil the DRY principle.

CraftableItem
CraftableItem is a class that extends PortableItem class. CraftableItem will take in newItem, a PortableItem object, as a parameter. This newItem is the item that the item will be crafted to when CraftAction() is called. We took into account the open closed principle when deciding on this implementation. CraftableItem can perform any functionalities that can be performed by its superclass PortableItem without breaking the application. This is because CraftableItem behaves similarly like PortableItem but with additional functionalities (ability to craft into another item). By ensuring this, we have successfully fulfilled the Liskov Principle which requires the objects of a superclass to be replaceable with objects of its subclasses without causing any errors to the application.

CraftAction
CraftAction class will extend Action class. CraftAction takes in 2 parameters, which is oldItem and newItem. newItem will replace oldItem in the inventory. CraftAction is created as

a class rather than a method in another class as we want to fulfil the single responsibility principle, while also maintaining consistency among subclasses of Action class.

Zombie Limb

ZombieLimb class will extend CraftableItem class and implements Weapon class. Zombie limb will also take in a parameter newItem, which is the item to be crafted when CraftItem Action is called. Zombie limbs will be responsible for creating instances of zombie arms and zombie legs. We decided on this implementation as we took the DRY principle into consideration. This implementation allows us to not write and maintain the same code in 2 similar classes.

Zombie Mace and Zombie Clubs

Both zombie mace class and zombie club class will be separate classes, both extending WeaponItem. We implement it as such because we want to maintain consistency among WeaponItem subclasses, along with ensuring we adhere to the single responsibility principle and Open-closed principle. This ensures the classes only take care of one weapon type in the system, along with allowing any new features to be implemented in the least amount of time to implement.

PickUpItemBehaviour

PickUpItemBehaviour implements Behaviour as its interface. PickUpItemBehaviour allows ZombieActor to pick up items such as weapons from the ground. This behaviour will be added as part of Zombie's behaviour, hence, Zombie objects will be able to pick up the weapon item from the ground. By implementing the Behaviour interface, we are able to achieve total abstraction. The Behaviour interface provides contracts that objects can use to work together without needing to know anything about each other.

HumanCorpse

HumanCorpse class extends PortableItem. Human Corpse will rise from the dead after a number of turns, which creates a new Zombie object and removes the HumanCorpse object (itself) from the game map.We decided to use this implementation as we wanted to fulfil the single responsibility principle, along the open closed design principle. A class level constant will be created to store the number of turns for a Zombie object to be created. This helps avoid excessive use of literals. Also, any changes to the number of turns can be modified easily and safely at the constant itself.

Dirt

Dirt object extends Ground and therefore has all the functionality of the ground object. This implementation of inheritance ensures that no recreation of duplicate code is needed. This fulfills the principle of Don't Repeat Yourself.

Crop

Crop class extends Ground class. Crops are given 2 additional capabilities, and allow only Farmer objects or Player to interact with it. We implement Crop class as Ground class to maintain consistency within the application, along with Crop class only needing 1 tick function, rather than 2 that is implemented in Item class. This way, we can be more efficient (albeit just very slightly), as we can just run tick() function without checking if it is in an Actor object's inventory (it should not be in an actor's inventory anyway).

Enum classes

We have created 4 classes to store enums for items, which are ItemCapability, ItemStatus, ZombieCapability and FarmingCapability. ItemCapability and ItemStatus applies only to Item class and its subclasses, while ZombieCapability and FarmerCapability applies to ZombieActor along with its subclasses and Farmer classes respectively.These enum classes are separated as such because we do not want to allow classes to have access to capabilities that they should not have.

Farmer

Farmers are humans with additional functionalities. Instead of instantiating and recreating all the instances and methods in the Human class, the concept of inheritance is utilised to minimise duplicate codes. This helps fulfil the principle of Don't Repeat Yourself (DRY). By submitting to this principle, we can reduce and avoid potential update anomalies. Hence, any changes made to the Human class will also apply to the Farmer class directly. Farmers class extends Human class and given access to Sowbehaviour, FertilizeBehaviour and HarvestBehaviour. Farmer class is created to separate these additional functionality from Human class, because Human class will never use these classes when running the application.

SowBehaviour and SowAction

SowBehaviour is a class that extends Behaviour, and allows access to SowAction, along with calculating the chance of SowAction being called. SowAction is a class that extends Action. SowAction turns Sowable Dirt objects into Crop objects.

FertiliseBehaviour and FertiliseAction

FertiliseBehaviour is a class that extends Behaviour, and allows access to FertiliseAction. FertiliseAction is a class that extends Action. FertiliseAction increases the age of the Crop object by a set amount, and can only be applied onto unripe Crop objects.

HarvestBehaviour and HarvestAction

HarvestBehaviour is a class that extends Behaviour, and allows access to HarvestAction. HarvestAction is a class that extends Action. HarvestAction turns a ripe Crop object into Dirt object, and creates a Food object on the ripe Crop object's location

<u>Justifications on sow, fertilise and harvest Actions and Behaviour</u>
(Grouped as they have similar/related justifications)
The Action class in the engine package is used as the base class for SowAction, FertiliseAction and HarvestAction. The existence of the Action class allows its subclasses to inherit the common attributes and methods without having the class to recreate them again. This ensures that we don't repeat ourselves by creating DRY code. These classes are implemented as such to allow us to maintain the consistency among Behaviour and Action subclasses, along with taking the single responsibility principle into consideration. Separating them as such also ensures no potential side effects that may arise if they are implemented as 1 collective Action class and Behaviour class will happen.

<u>Food</u>
Food class extends PortableItem, and will heal the user's HP by a certain amount when used in EatAction. This implementation allows us to keep the consistency of the code. Implementing a Food class also allows for more extensibility. This means that additional features (e.g. increased movement speed when a particular type of food is consumed) can be added more easily in the future when required.

<u>EatAction</u>
EatAction extends Action. When invoked, it will heal the user by a set amount of HP, which is listed in the Food object. The Food object will then be removed from the user's inventory. We chose this implementation to ensure consistency between subclasses of Action. This also only allows relevant objects access to it's methods, which reduces redundant code along with ensuring only one class is responsible for this functionality, which is in accordance to the single responsibility principle.

<u>EatBehaviour</u>
EatBehaviour implements Behaviour and allows access to EatAction. This behaviour is only available to Human objects. When invoked, it will check if the user's inventory contains a Food object. If yes, it will return EatAction(). If there is no Food object, it will return null. By implementing the Behaviour interface, we are able to achieve total abstraction. The Behaviour interface provides contracts that objects can use to work together without needing to know anything about each other. This implementation also fulfils the single responsibility principle, as it only deals with checking if the user can invoke EatAction.

<u>MamboTotem</u>
MamboTotem extends Item. This item will be spawned at the beginning of the game, and controls Mambo Marie's spawning. MamboTotem objects are not portable, and only have 1 present in the game at all times. This ensures that at most 1 Mambo Marie exists on the game map.

<u>MamboMarie</u>
MamboMarie extends ZombieActor. MamboMarie has a turn limit of 30, and will spawn 5 zombies every 10 turns. MamboMarie is spawned at coordinates (0,0) on gameMap. We implement it as such because of assignment specifications, along with giving the player a harder time to kill her due to the distance between MamboMarie and player.

<u>Ending the game</u>
We complete this by creating a class called EndableWorld which subclasses World. We use 2 boolean variables to determine if the player won or lost the game. These 2 variables also allow us to know if the player decides to quit the game or completed the game, as the prompt for the player quitting the game is different compared to the player completing the game. We decided to implement the endings as a method in EndableWorld, while the player quitting the game is set as its own Class called QuitAction. We decided this implementation as it allows us to follow the Single Responsibility Principle, where EndableWorld class that's responsible for running the game is also responsible for ending the game when player completes it, while any actions made by the player (including quitting the game) is handled by an Action subclass.

<u>QuitAction</u>
QuitAction is a subclass of Action. QuitAction allows the player to quit the game. We do this by removing the player character from the game. This would trigger the game to end. We decided to implement QuitAction as its own class because we want to fulfil the Single Responsibility Principle, where any Action made by the player is handled by a subclass of Action.

<u>SpawnAction</u>
SpawnAction is a subclass of Action. SpawnAction will spawn 5 zombies in random locations using a random number generator. We implemented SpawnAction as its own class because we want to maintain the consistency of the application, along with fulfilling the single responsibility, where any action made is handled by a subclass of Action.

<u>Vehicle</u>
Vehicle is a subclass of Item. Vehicle allows the player to travel between maps through DriveAction.

<u>DriveAction</u>
DriveAction is a subclass of Action. Driveaction allows the player to travel between maps. We decided on this implementation to fulfil the Single Responsibility Principle, where only subclasses of Action class are allowed to handle actions from actors.

<u>AimAction and ReloadAction</u>
Both AimAction and ReloadAction are a subclass of Action. Both AimAction and ReloadAction are implemented as an Action rather than a method in SniperRifle. We have decided on this implementation where only subclasses of Action class can handle actions from actors. This ensures that the Single Responsibility Principle is fulfilled. Also, by subclassing action class, we have been able to reduce redundant codes. Instead of instantiating and recreating all instances and methods in the Action class, the concept of inheritance is utilised to minimise duplication of codes which help fulfils the principle of Don't Repeat Yourself (DRY). Therefore, reduces the occurrence of potential update anomalies.

So, any changes made to the Action will be applied directly to its children class AimAction and ReloadAction.

- **AimAction**
  AimAction provides a SniperRifle instance with an allowable action to aim. Aiming improves the accuracy and the damage of the SniperRifle.

- **ReloadAction**
  ReloadAction provides all RangedWeapon with an allowable action to reload their ammunition. ReloadActions are only available when both the weapon and the ammo for the weapon type is inside the player's inventory.

<u>RunAwayBehaviour</u>

RunAwayBehaviour implements Behaviour and therefore are required to implement all methods stated in the Behaviour interface. By implementing the Behaviour interface, we are able to achieve total abstraction. The Behaviour interface provides contracts that objects can use to work together without needing to know anything about each other. Additionally, RunAwayBehaviour is implemented as a behaviour rather than a function in Human class.
This is because such implementations allow the RunAwayBehaviour to be reused if needed in the future when similar functionalities were to be implemented on non-human type Actor. Also, we have fulfilled the Single Responsibility Principle as each method only takes care of one single responsibility that is relevant to themselves.

<u>RangedWeapon</u>

A RangedWeapon class is created as the base class of all ranged weapons including SniperRiffle and Shotgun instances. RangedWeapon is an abstract class. All ranged weapons have functionalities for ammunition. RangedWeapon helps ensure similar functionalities like the system of ammunition is not repeated in Shotgun and SniperRifle class. By creating a base class for ranged weapons, we have managed to reduce duplicate codes (fulfil DRY principle) and potential anomalies like update or delete anomalies. By creating an abstract base class we have also fulfilled the open-closed principle. For instance, we are allowed to move the responsibility of actually getting the accuracy away from the getAccuracy() method of the ranged weapon class (by overriding). Hence, both SniperRifle and Shotgun can have their own logic for getAccuracy() without having to make any modification to the base class. This allows flexibility in extending the code. So, SniperRifle can have a very different accuracy system from Shotgun just by extending the code differently. Therefore, making the code open for extension, but closed for modification.

<u>Shotgun and SniperRifle</u>
Both SniperRifle and Shotgun inherit the functionality of RangedWeapon class. RangedWeapon class contains the basic methods that are used by both Shotgun And SniperRifle class. Hence, it greatly reduces the amount of duplicate code and fulfils the DRY principles. By submitting to the DRY principle, we have reduced the chances of occurrence of inconsistency of codes.

Shotgun
A subclass of RangedWeapon that deals damage to all actors(any type) within its range. A Shotgun is given an additional allowable action ShotgunAction. ShotgunAction will be available only if Shotgun with at least 1 ammoCount is inside the player's inventory.

ShotgunAction
ShotgunAction is a subclass of Action that deals damage to all actors within a cone of range 3. We do this by involving AttackAction on all actors within the cone. We decided on this implementation because we want to fulfil the single responsibility principle, where AttackAction is responsible for attack probability and damage calculations of all weapons.

ShotgunMenu
ShotgunMenu extends Action and acts as a submenu for Shotgun. ShotgunMenu will get directions where the player can fire. We implement it as such because we want to declutter the display and provide a better display that doesn't overwhelm the player. This way we can improve on the player's experience when playing the game.

SniperRifle
A subclass of RangedWeapon that can attack any actors with ZombieCapability.UNDEAD on the map. SniperRifle has varying accuracy and damage depending on its aim count. The weapon rewards player patience, the higher the aim count the greater the accuracy and damage. Functionalities relevant only to SniperRifle like getAimCounter, resetCounter, incrementCounter and resetTarger are all implemented on SniperRifle to ensure single responsibility principle is fulfilled.

SniperRifleMenu and SnipeAction
2 Assisting classes are created for SniperRifle which are SniperRifleMenu and SnipeAction. Both action classes will return a menu to show the options available to the actor depending on the state of the SniperRifle. A SniperRifleMenu will return a list of targetable actors (with UNDEAD capability) for the player to choose from. Once an actor is targeted he/ she will be directed to another menu, to select the action to do to the targeted actor. Players can choose to attack directly or take an AimAction to improve the accuracy and damage of the attack, but loses its target when attacked by other actors or selects any other options besides from aiming or shooting. To fulfil the single responsibility principle, these 2 methods are implemented as 2 separate classes instead of 1 single class. This is because SniperRifleMenu and SnipeAction are in charge of very different logic and functionality. SniperRifleMenu will decide on which menu to prompt by checking for the status of the SniperRifle (whether target is null), whereas, SnipeAction is in charge of the actual sniping

process (attack and aim). Also, separating the menu into submenu makes the user interface much cleaner and readable. This will help improve the gameplay experience for the player.

<u>Ammo - ShotgunAmmo and SniperRifleAmmo</u>

Ammo

An Ammo class is created as the base class for all Ammo objects including ShotgunAmmo and SniperRifleAmmo. Ammo class is an abstract class. Ammo is declared as an abstract class to prevent programmers from instantiating an Ammo object directly. Programmers can only create specific ammo objects from the subclasses of Ammo including the ShotgunAmmo or SniperRifleAmmo instances. A base class is created to reduce redundant (DRY) code, as SniperRifleAmmo and ShotgunAmmo share a similar functionality.

ShotgunAmmo and SniperRifleAmmo

ShotgunAmmo and SniperRifleAmmo are implemented as a separate class extended from ammo class. This ensures that the single responsibility principle is fulfilled as ShotgunAmmo only deals with Shotgun and SniperRifleAmmo only deals with SniperRifle.