Design Rationale

Jiten Verma and Ayesha Ali

Going to town

Feature Number	Feature description
1	Place a vehicle somewhere on your existing map. The vehicle should provide the player with the option to move to a town map. The design of the town can be customised
2	Somewhere on the town map, place a sniper rifle and a shotgun.
3	When you leave a map, any creatures in the old map should continue to move and act.

Feature 1

A new class called Vehicle which extends item would be created, which would require the map to be transported to. The constructor class would add an allowable action for MoveActorAction to allow us to transport the actor onto the town map.
In application, the item of Vehicle will be created.
Modify the Application class to create a new map and add the townMap into the World.

Feature 2

On the map created in Application, place sniper rifle and shotgun in a randomly selected location by the programmer.

Feature 3

The run method in the World class ticks all maps stored already every turn.

Rationale

The choice of creating a new Vehicle class was to ensure that the vehicle item could be created again if necessary without having to copy and paste code. This would save us time from having to refactor it in the future. This also allows us to easily recreate the vehicle in the town map to bring the player back to the compound. Furthermore, this helps us to maintain a single responsibility rule, as we have one designated class responsible for moving actors between maps.

New weapons: shotgun and sniper rifle

Feature Number	Feature Description
1	Create a shotgun weapon
2	Shotgun spread: The shotgun has a short range, but sends a 90◦ cone of pellets out that can hit more than one target – that is, it does area effect damage. Rather than being fired at a target, the shotgun is fired in a direction. Its range is three squares. So, if the shotgun is fired north, it can hit anything in the three squares north of the shooter, northeast of the shooter, northwest of the shooter, or anything in between.
3	It has a 75% chance of hitting any Actor within its area of effect and decides the amount of damage for a shotgun.
4	Create sniper rifle weapon
5	When the player fires your sniper rifle, they should be presented with a submenu allowing them to choose a target. Once they have selected a target, they have the option of shooting straight away or spending a round aiming. Spending time aiming improves the result as follows: • No aim: 75% chance to hit, standard damage • One round aiming: 90% chance to hit, double damage • Two rounds aiming: 100% chance to hit, instakill
6	If the shooter takes any action other than aiming or firing during this process, their concentration is broken and they lose their target. They will also lose their target if they take any damage.
7	Create ammunition boxes

Feature 1

Create a new class named Shotgun which extends weaponItem. Player would be able to pick up and use this weapon.

Feature 2

Create a new class named FireShotgunAction which would be added to a players list of actions if they possess a shotgun in their inventory. This method would present a submenu for the player to select the direction in which they'd like to shoot.
Another class would be created named ShootAction which would take the direction to shoot in and perform the 90° cone of pellets.

Creating a new class for this action allows us abide by the Single Responsibility Principle in order to ensure that a single class is responsible for everything regarding shooting with a shotgun. The current AttackAction randomly selects a weapon to use when attacking, hence will not be suitable for the use of shotgun as a weapon.

Feature 3

If an actor is within the range of the shotgun, then there will be a randomly generated number between 0-99. If the number is below 75 then the target actor will be damaged.

Feature 4

Create a new class named SniperRifle which extends weaponItem. Player would be able to pick up and use this weapon.

Feature 5

Create new class named UseSniperRifleAction which enables player to use the sniper rifle. In the execute method a method called selectAim is called which helps displays a submenu to allow player to select an enemy to attack with the use of a new class called SelectTargetAction. An attribute named turnsSpentAiming is initialised at 0 in the player class and passed through this class. If this integer is less than 2, an action is added to aim as well as shoot with the sniper, else only shoot action is added. A submenu of these actions is displayed for the player to choose from.

Create a new class named AimSniperRifleAction which enables player to spend a turn aiming. This class increments timeSpentAiming.

Create a new class named ShootSniperAction which enables player to shoot the sniper. The AttackAction class is modified to have a new constructor which allows it to take in rangedWeapons. A new sniper rifle is passed through and its damage depends on the number of turns spent aiming. timeSpentAiming is reset to 0.

Creating new classes for these actions allows us to abide by the Single Responsibility Principle in order to ensure that a single class is responsible for everything regarding shooting with a sniper rifle. The current AttackAction randomly selects a weapon to use when attacking, hence will not be suitable for the use of a sniper rifle as a weapon.

Feature 6

If the LastAction was not FireSniperRifleAction then it sets the value of concentration to 0. An override hurt method in player is created which sets value of concentration to 0 as well. If the value of concentration is 0 then a new instance of FireSniperRifleAction is added to the player menu, else the lastAction used is simply added again into the list of possible actions for the player.

Feature 7

A new class named Ammunition which extends PortableItem. If this item is picked up, then the player will gain 15 bullets.
An attribute called Ammunition would be created in the Player class to keep track.

Rationale

These design methods help us maintain the single responsibility principle as each class has one role within our system. This helps us to ensure that we are not creating a god class and that all the functionality for a given class is contained within the one class itself. This helps to ensure that classes do not become too large and unreadable or difficult to maintain. Furthermore, the design chosen ensures that the open/closed principles are being met, as we are able to create more methods within the classes without having to modify the existing code. The existing code should work well independently, and would not be dependent on future methods added to the classes.
In addition, the design will aim to uphold the interface segregation principle to ensure that each abstraction remains clean, small and extensible. We would be ensuring that the ranged weapons extend WeaponItem to showcase they behave as a weapon, the actions associated with shooting and picking up ammunition will all extend Action to easily access and execute them. This will help ensure that our system remains extensible and that common classes are grouped together, and segregated from the rest.

Mambo Marie

Feature Number	Feature Description
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.

2	She starts at the edge of the map and wanders randomly
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.
4	Mambo Marie will keep coming back until she is killed.

Feature 1 and 4

Make a new class called MamboMarie which extends ZombieActor. This class will have all the functionality related to MamboMarie. In the Endgame class which is a subclass of World, we can generate a random number from 1 to 100 and if that number is from 1 to 5 and no other instance of Mario Mambie exists on the the player is in map and it is conscious, an object of Mambo Marie can be placed on any of the edge locations on the map the player is in.This is done in Endgame class as it has some methods like run() and stillRunning() which are being called at each turn which is important for keeping track of the chances of creating Mambo Marie. Also after Mambo Marie is being vanished after 30 turns, a new object is created to make the game more challenging for the player as it has to be killed in 30 turns only.

Feature 2

Make a list of possible edge locations and randomly choose one of them to place MamboMarie on it. This method is being made in Endgame as that method is responsible for placing Mambo Marie on the map at which the player is in.To make it wander add an object of WanderBehaviour in MamboMarie which causes it to walk randomly when called in MamboMarie's PlayTurn.

Feature 3

Make a new attribute turns which is incremented by 1 by overriding the tick location in the Mambo Marie class.If the number of turns reaches a multiple of 10, we can choose 5 locations by generating a random x and y value and if an actor can be placed at the x and y value, place Zombie objects in that .At 30 turns, if she is conscious then we can remove her object from the map.

Rationale

The D.R.Y principle is being maintained as Mambo Marie which is a subclass of ZombieActor shares similar functionalities and attributes to ZombieActor like PlayTurn and we are adding are own methods as well to implement features specific to MamboMarie.SRP is also being followed as single classes are being created wherever possible like MamboMarie and Endgame to reduce dependencies on other classes.It also follows Open/Closed Principle as the base class Item is not being modified but it is being extended by adding subclasses like MamboMarie and Endgame.It also follows the Interface Segregation Principle as each functionality uses the needed interface only like all Behaviours are implemented by the Behaviour interface and all Actions are based on the ActionInterface.

Ending the game

Feature Number	Feature Description
1	A “quit game” option in the menu
2	A “player loses” ending for when the player is killed, or all the other humans in the compound are killed
3	A “player wins” ending for when the zombies and Mambo Marie have been wiped out and the compound is safe

Feature 1

Create a new class called ExitAction with a hotkey ‘E’.It is added to list Player’s actions and whenever it clicks on ‘E’, the game is finished by removing the player from the map.The World class in engine automatically stops the game when player is not in the map.

Feature 2

Make a subclass of World called Endgame.The run() method in World determines when the game should end and prints the endGameMessage when the game ends.We override run and check if any Humans exist on the compound map or not. This is done by using the maximum values of x and y values of the compound map with the getXRange and getYRange in GameMap class.If there is not a single Human in the compound map, the game ends and the player has lost.

Feature 3

Make a subclass of World called Endgame.The run() method in World determines when the game should end and prints the endGameMessage when the game ends.We override run and check if any Zombies or Mambo Marie exist on the compound map or not. This is done by using the maximum values of x and y values of compound map with the getXRange and getYRange in GameMap class.If there is not a single Zombies or Mambo Marie on the compound map the game ends and the player has lost.To compensate for a new Mambo Marie being initialised after vanishing, I have assumed that the game can end if the Mambo Marie does not exist on the map rather than being dead.

Rationale

The D.R.Y principle is being maintained as EndAction which is a subclass of Action shares similar functionalities and attributes to Action and we are modifying the features specific to Action.SRP is also being followed as single classes are being created wherever possible like ExitAction and Endgame to reduce dependencies on other classes.It also follows Open/Closed Principle as the base class Item is not being modified but it is being extended by adding subclasses. It also implements LSP as Endgame can be substituted in any place that needs World and it has been done Application.It also follows the Interface Segregation Principle as each functionality uses the needed interface only like all Actors are implemented by the Actorinterface and all Actions are based on the ActionInterface.