**DESIGN RATIONALE – ASSIGNMENT 3**

**GOING TO TOWN**

The features discussed in this segment are depicted in Class Diagram – Car and Sequence Diagram – Moving Between Maps.

In order to build the new map, it was necessary to make new Ground subclasses. This is so that the differences in the map appearance can quickly and intuitively convey to the user that they are in a new type of map. Replacing Dirt with Concrete was also necessary to prevent growing crops indoors or in a city street, which would be unrealistic.

I decided to make the new map the same size as the old, for the sake of consistency. A Shotgun and Sniper Rifle are placed on the map indoors, which requires the player to travel through tight quarters dodging Zombies to reach these valuable items. The Shotgun, being generally less powerful, is easier to reach than the Sniper Rifle.

For transport between maps, I decided to make a new Item subclass, a Car. In my original plans I made the Car a ground type and intended to manually add an Exit at the car’s location, but this design would be cumbersome to implement for any new maps added, as well as introducing the possibility for error should there be a mismatch between the Car groundtype location and the Exit location. With the Car as an item, it can be easily added to any new maps as they are made and will always function properly with valid parameters.

The car is non-portable, so cannot be picked up. Its constructor adds a MoveActorAction as an allowable action, and takes the movement destination as a parameter in the form of a Location. Adding the movement as an allowable action is an easy and clean way to add this capability, as the PlayTurn method already checks any items at the Player’s feet for allowable actions and adds them to the menu. Only one line of code was needed to fit this feature in with minimal dependency and no external refactoring, fulfilling the Single Responsibility Principle(SRP) and Liskov Substitution Principle(LRP).

**NEW WEAPONS: SHOTGUN AND SNIPER RIFLE**

The features discussed in this segment are depicted in Class Diagram – Firearm, Sequence Diagram – Firing Shotgun and Sequence Diagram – Firing Sniper Rifle.

To implement the guns, I decided to make an abstract subclass of Item; Firearm. Both Shotgun and Sniper Rifle would then in turn be subclasses of Firearm. This sets a common framework for any other Firearms that may be implemented in the future, fulfilling the Open/Closed Principle(OCP). Firearm contains, alongside its constructor, an abstract shoot method that may be customised for any of its subclasses, and a non-abstract checkAmmo class. Having checkAmmo be non-abstract reduces repetition for all subclasses, and will be equally functional for all subclasses given that the ammo type for the firearm is passed into its constructor, fulfilling LSP.

Each Firearm subclass has the enum GUN added to it during construction. This allows it to be identified during a Player’s PlayTurn action, which will add a ReadyFirearmAction for that firearm to the list of allowable actions. This method was decided on, rather than adding the ReadyFirearmAction as a capability, to ensure that the firearm needs to be picked up to be fired and cannot be fired by Zombies. Zombies can still pick the firearms up off the floor requiring the player to knock their arms off or kill them to get it back, which adds an interesting element of challenge to the game. Using an enum also allows for items in the player’s inventory to be checked without needing to resort to using instanceof. Firearm’s Shoot method was also added to the ItemInterface, so that Firearms may be fired as Items without requiring downcasting, fulfilling LSP.

The ReadyFirearmAction is universal to all firearms, and represents the player shouldering their weapon to prepare to fire. It is necessary because the engine’s Menu class only accepts and returns Action subclasses. The ReadyFirearmAction calls the shoot method of whichever firearm it was instantiated for. This allows the player a choice between multiple Firearms, as many as they have in their inventory. As a subclass of Action it fulfills the LSP, and also contributes to the Firearm abstract class being highly extensible. In combination with this action, any child class of Firearm does not need new or rewritten code up until the shoot method is called, which is the same point at which any unique properties for the firearm would come into effect.

**SHOTGUN**

The Shotgun has one associated action subclass, called FireShotgunAction. When a shotgun is instantiated, one FireShotgunAction for each of the eight possible firing directions is added to an Actions collection, which is then stored in the Shotgun as a variable. This implementation was chosen so that the existing Menu class may be used, as it only accepts Action subclasses as menu options. The FireShotgunAction does not narrow Action’s range of uses, and so fulfills LSP.

When the shotgun shoot method is called, that collection of FireShotgunActions is passed to a new menu, and the user choses the direction of firing through the menu. When the appropriate Fire action is chosen and execute is called on it, the action calls checkAmmo() on the associated firearm. If checkAmmo returns false, the FireShotgunAction’s execute method returns a description String advising the player of this with no further action taken.

While calling checkAmmo() at this stage does introduce a dependency in that the action must message and store the firearm, this implementation does have benefits, in that it keeps consistency with the rest of the program’s UI. No other item prints out messages to the UI or cancels user actions on its own, so having the item itself end the player’s turn would break SRP. Additionally, the firearm would have to create and return a DoNothingAction, which would simply replace one dependency with another.

If checkAmmo() returns true, the affected tiles in the given firing direction are checked for actors, and 75 damage is applied at 75% probability of damage per actor. As shells do not discriminate, the shotgun can damage both Humans and Zombies.

**SNIPER RIFLE**

The Sniper Rifle has two associated action subclasses; AimSniperAction and FireSniperAction, which as with FireShotgunAction are necessary in order to use the engine’s Menu class. When the Sniper Rifle is readied and its shoot method is called, the weapon first checks if a target is already stored.

If there is no target stored, the map is scanned for potential targets. An AimSniperAction with each valid target as a parameter is used to populate a menu for the User to choose their target, and the associated action and target are stored in a HashMap called targetHolder, with the Action as key. When the user makes their menu selection, the action is not executed but instead used to retrieve the target actor from that HashMap, which is then stored in the rifle for future reference. This method of selecting a target is used to accommodate the Menu class only accepting Actions.

With a target selected either earlier in this turn or in a previous turn, the user is presented with a menu giving the option of aiming or shooting. Either of the chosen actions at this point will be executed, and so end the turn. If the user choses to Aim, the Sniper Rifle’s aimUp method will be called, iterating an integer variable named damageLevel up by one.

The user is not given this choice if damageLevel is already at 2, which would indicate two turn of dedicated aiming. If damageLevel reaches this value or the user chooses through the menu to fire early, the action FireSniperAction is executed. FireSniperAction takes the current damageLevel as a variable, and deals 40, 80, or 999 damage depending on its value. In the same manner as the Shotgun, the Sniper Rifle also checks for ammo at this stage.

Executing FireSniperAction calls the clearTarget method in the Sniper Rifle, clearing the stored target and resetting damageLevel to 0, but this is not the only way to reset targets. The player also loses their target on choosing another action or taking damage, and so their inventory is scanned for any Sniper Rifles and targets cleared with that method if either of those events takes place. In order to facilitate this without resorting to instanceof or downcasting, the enum AIMED is applied each sniper rifle as a capability, and clearTarget was added to ItemInterface.

**MAMBO MARIE**

The Mambo Marie feature was implemented into the game. To represent a MamboMarie object, a class named “MamboMarie” was created. This class extends from the “ZombieActor” class. By doing this, we can easily access and use critical methods that are already defined for us in the ZombieActor class, and therefore also the Actor class. An alternative to this design decision may be to have the MamboMarie class inherit Zombie rather than ZombieActor.

This was not done as the Zombie class does not have a constructor (by default or from previous assignments) where we can choose the display character, hitpoints and other attributes. We could overload the constructor in Zombie which will let us do this, however I decided that the Zombie class did not have any methods that would be useful to MamboMarie and ZombieActor already had this sort of constructor defined.

The class MamboMarie, like other classes such as Zombie and Farmer uses an array to hold a set of behaviours. These behaviours are MarieSpawnBehaviour, ChantBehaviour, and WanderBehaviour.

In order to implement MamboMarie appearing 5 percent of the time, the behaviour MarieSpawnBehaviour is created. This class implements the Behaviour interface. By implementing an interface, we can easily attain methods that we will need in order to develop this behaviour such as the getAction() method. This is good practice as this interface requires that all methods that implement it are required to have it’s methods, so we can uphold consistency amongst all Behaviours.

The MarieSpawnBehaviour class implements the getAction method which returns a MarieSpawnAction, or null if it is not. A helper class to determine a random edge location getRandEdgeLocation is utilized which determines and returns a random Location on the top edge of the map. This method is called in the getAction action method. An alternative for this would be to determine the random location inside of getAction, however this would be bad design practice as this means that the code is not modularised and therefore more difficult to interpret. In this manner, using a helper method makes the code more readable.

The MarieSpawnAction inherits the Action class. Doing this is good design practice as it maintains consistency throughout the code as other Actions do this same thing. This means we can simply override the execute and menuDescription methods from the Action class. This also makes sure that we do not write unnecessary code such as adding this Action to the menu manually as the engine handles it for us.

The chanting was implemented by creating a ChantBehaviour, which also implements Behaviour, which returns a ChantAction. Using the Behaviour interface is good design practice as stated above. The getAction method in the ChantBehaviour class uses the Math library to get random spawn locations for 5 Zombies. Each location is added to a predefined ArrayList which will then be passed to the ChantAction in a for loop. Using a loop rather than hard coding the adding of locations to the ArrayList improves readability and is also easier to maintain as changes only need to be made inside the body of the loop. An ArrayList was used as they are mutable unlike arrays, so Locations can easily be added to them.

The ChantAction that is returned by ChantBehaviour is passed this ArrayList of locations. The ChantAction uses these locations and adds Zombie actors to these locations in a for loop. Although ChantAction is currently only triggered by MamboMarie, it may be possible that in the future we may want to give this ability to new actors in the future. In order to cater for this fact, we concatenate “actor” with the execution statement and menu description so that if any other actor was passed in the execute() or menuDescription() methods, no changes to ChantAction will need to be made.

Hiding MamboMarie was done by implementing a setVisibility() method in the MamboMarie class. To avoid downcasting when calling this method, this method was added to the ActorInterface. The alternative would be to simply cast the “actor” parameter in execute to a MamboMarie type, but this is considered bad design practice and hence was avoided.

**ENDING THE GAME**

The game end feature was implemented into the game. In order to implement the quit option in the menu, a QuitAction class had to be created. This class inherited the Action class, meaning we can easily override the execute, menuDescription and hotkey methods and implement them for QuitAction. The hotkey method was used to define “q” as the hotkey.

In order to allow the player to quit the game at any time, we use the Menu object “menu” created in the Player class. In the playTurn() method in the player class, we can simply add a new QuitAction to the “actions” parameter of playTurn(). In doing so, each turn will show the “q” option to quit the game. This is good design practice as we are using a simple implementation rather than trying to manually add the quit option to the menu. We are simply adding a QuitAction() as a possible action for player.

In order to track the winner and loser, AttackAction was modified. Methods named checkIfZombiesAlive and checkIfHumansAlive were added to this class, and checks were added to the execute method to check if Player had lost or won. An enumeration was used to define if the player had won or not. This is useful as we can simply add the capability to the player that they had won or lost, and we can check for that capability when accessing the player Actor.