**SAAUREN M – ASSIGNMENT 3 DESIGN RATIONALE**

**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.