# Design Rationale – Assignment 3

# Group: 404


1) Nur Farhanah binti Baharudin (29969470)
2) Morad Abou Shadi (29799260)
3) Lai Khairenn (29959381)


## Main Design Plan:

For all the actions possible in the game, we decided to implement an Action or Behaviour, so we ended up creating many new Actions and Behaviours. This helps to increase encapsulation in examples like the zombie picking up weapons whenever it finds a weapon, and swapping it for its current weapon if it already has a weapon. The logic behind how it will swap the weapon can be encapsulated inside the PickUpWeaponBehaviour, and we just need to assign the behaviour to the zombie to allow the zombie to pick up weapons. Using Actions and Behaviours for all the actions in the game also adheres with the DRY principle because it allows multiple actors to have the same behaviours and do the same actions. For example, both the Player and Farmer can harvest ripe crops, so by implementing a HarvestBehaviour, we can add this functionality to both Player and Farmer without writing duplicate code to do the same thing by using the HarvestBehaviour. Furthermore, implementing actions and behaviours also greatly increases the maintainability of the code. For example, new actors might be added into the game in the future and we can allow them to carry out certain actions by just giving them the existing behaviour instead of implementing it all over again. Or if for example we want to allow characters to automatically search for to pick up instead of waiting to stand on top of food, then we can just change it once in the PickUpFoodBehaviour instead of finding all the actors that can pick up food and editing all that code.

# 1) Zombie Attack

Main Classes involved:

- Zombie (modifying)
- AttackAction (modifying)
- Display (inheriting methods)
- ZombieActor/Actor (inheriting methods)
- ActorLocations/Location (inheriting methods)

- PickupItemAction
- PickUpWeaponBehaviour (new)

Firstly, we decided to encapsulate the miss chance of each actor in the getWeapon method of their own class so that each actor can have their own customisable miss rate, which increases maintainability because new actors that are added can have their own miss chance, instead of being forced to follow a 50% chance in AttackAction. Since the getWeapon method is called whenever an actor attacks, we will make it calculate if the attack misses, and it will return null if the attack missed. To implement a zombie bite, we have decided to create a new form of intrinsic weapon called “bite” within the zombie class. The probability regarding which of the two intrinsic weapons to be used will be determined in the getIntrinsicWeapon method in Zombie using the nextBoolean() method. If a successful bite attack is implemented, the zombie restores 5 health. When a bite attack is returned, the getWeapon method will have a further 50% chance to miss, on top of the base 50% chance, and if it successfully returns a bite attack, then the zombie will be healed by 5hp.

There is also the requirement that if there is a weapon at the zombies location, it picks it up and uses it instead of the intrinsic weapons. This functionality will be implemented by creating a new class called PickUpWeaponBehaviour that implements the Behaviour interface. In the PickUpWeaponBehaviour class we will get the location of the zombie and get all the items at the zombies location. If the zombie has at least one arm (arm is explained below), there is a weapon on the ground and the zombie does not have a weapon in its inventory, then the getAction method of the behaviour will return a PickUpItemAction so the zombie can pick it up, or else if will return null. Hence, PickUpWeaponBehaviour will have a dependency on the PickUpItemAction class. Then in the AttackAction class, when the execute function runs it always checks the actor inventory for a weapon and uses that over intrinsic weapons to attack humans.

To display a zombie like message, we have decided to put it in the zombie class under playTurn method; we generate a random number from 0 to 1 and if it is less than 0.1 (representing 10% probability) then it prints out (“*Zombie name* says “BRAAAAAAINS””) and plays turn normally otherwise would just execute without displaying anything.

# 2) Crafting Weapons

Main classes involved:

- ZombieMace (new)
- ZombieClub (new)
- CraftbleWeapon

- WeaponItem
- Player
- ZombieArm and ZombieLeg (new)

With Zombies dropping limbs everywhere when they get hit, the player has the option to use these limbs as weapons with more damage than the basic intrinsic weapon attack. For the functionality of crafting weapons, we have decided to create two classes for zombie mace and zombie club which inherit features and methods from the WeapoItem; they would contain all the relevant information regarding the effects and properties of the weapon so we do not have to code it all over again, which is in accordance with the DRY principle. Then, we will make the ZombieArm and ZombieLeg class implement a new interface, which is the CraftableWeapon interface, which has a method called getCraftedWeapon that we must override in the ZombieArm and ZombieLeg class to return their respective upgraded versions. Next we will create a new action called CraftWeaponAction that inherits the Action abstract class. We will override the menuDescription method to display a suitable message and we will also override the execute method so that when it is called, it will call the removeItemFromInventory method on the Player to remove either the ZombieArm or ZombieLeg weapon from the Player inventory, and then use the getCraftedWeapon method to get an instance of their own upgraded weapon and add that to the Player's inventory. We did not implement the crafting inside the weapon's getAllowableActions method because this method will be called even

when the item is on the ground, so it would not make sense to be able to craft a weapon before even picking it up.

# 3) Beating up the zombies

Main classes involved:

- Zombie
- ZombieLimb (new)
- ZombieArm and ZombieLeg (new)
- AttackAction

This requirement is mainly about the mechanism behind allowing zombies to drop their limbs after being attacked, so we have decided to implement two new classes, ZombieArm and ZombieLeg to represent a zombie's arm and leg respectively. We chose to use implement arms and legs separately instead of using a general limb class because losing arms and losing legs will have very different effects on the zombie, so separating them will be  These two classes will extend another new class which is ZombieLimb because we decided that both the arm and leg are limbs will have the same damage and verb, so the ZombieLimb

class will allow us to practice the DRY principle by using the same damage and verb from the ZombieLimb constructor instead of repeating them in each subclass constructor. The ZombieLimb further extends the WeaponItem class, because it already contains all the functionality needed for weapons in the game so we do not have to code it from scratch all over again, which is in accordance with the DRY principle also.

The Zombie class has an association with the ZombieArm and ZombieLeg classes because the Zombie has 2 ZombieArm objects and 2 ZombieLeg objects when it is created. The Zombie class will be responsible for keeping track of its own arm and legs, so two new attributes which are ArrayLists will be added, one to hold 2 ZombieArm instances and the other one will hold 2 ZombieLeg instances. We decided to use ArrayLists to hold them for easy removal of any limbs and also easy checking for remaining limb count. We will also add two new methods for the Zombie class, one for removing arms and one for removing legs. We plan to add a method called loseLimb in the Zombie class to handle the chances of a zombie losing its arms or legs when it is damaged and another method called removeLimbs to handle the chances of the zombie losing more than one limb. We decided to move our Zombie losing limb mechanism into the Zombie class instead of implementing it in the AttackAction class to follow good oop principles and encapsulate the losing of limbs within the Zombie class, since only zombies are able to lose limbs so other parts of the code do not need to know about it. However, we still had to downcast the actor into a Zombie and use a special zombieHurt method inside AttackAction so that we could handle dropping limbs and weapons onto the ground because the original hurt method does not return any information about the loss of limbs and weapons, and the Zombie class does not know about map too, so we had to drop the items onto the map inside AttackAction.

A zombie will have difficulty moving after losing either 1 or 2 legs so we have implemented this by modifying the playTurn action of the Zombie class instead of the Hunt and Wander Behaviours to avoid unnecessary dependencies and also to reduce repeating code. So every turn, the playTurn method is called and it will check if the zombie has no legs or has one leg and the last action is a movement action. If any of the statements is true, then the playTurn method will not be allowed to return a MoveActorAction. This fulfills the requirement of a zombie being allowed to move every two turns only if it has one leg, or not move at all if it has no legs.

To handle the consequences of losing arms, we will use the removeArm method, which is called whenever the Zombie loses an arm. After the method removes any arms, it will check how many arms are remaining. If the zombie has one arm remaining, it means the zombie only dropped one arm in that turn, so we will use rand.NextBoolean() to give the zombie a 50% chance of dropping its weapon and set the punch chance attribute of the Zombie to half of its original value. If the zombie has no arms remaining, it will definitely drop any weapon it is holding and the punch chance will also be set to 0%.

All limbs and weapons that are dropped are placed on the map in the AttackAction class, where it will get a random location that is adjacent to the zombie and drop each item so that other actors have a chance to pick them up, instead of just dropping it on the zombie location for the zombie to pick up. The special zombie hurt method will return an array list of

items that were dropped during the attack, and we will implement a method in AttackAction called DropItemOnMap that will handle this.

# 4) Rising from the dead

Main classes involved:

- PortableItem
- Corpse (new)

We implement this requirement by first making a new class called Corpse that extends from the PortableItem class so that we do not have to re-code the functionality of a portable item, which is in accordance with the DRY principle and also to encapsulate all the functionalities of a human corpse. We plan to modify the execute method of the AttackAction class to instantiate a new Corpse object whenever a human dies. The corpse class will have an integer attribute called turnCount to count the number of turns that have passed after the human was knocked unconscious and another integer attribute called turnsNeeded to set the number of turns needed for the corpse to turn into a zombie. The Corpse constructor will initialize turnCount to 0 and generate a random number between 5 to 10 (inclusive) to initialize turnsNeeded. Next we will override the tick method inherited from the Item class to check if turnCount is equal to turnsNeeded. When turnCount reaches turnsNeeded, then the Corpse item is either removed from the Player inventory or removed from the ground. Next, a new Zombie instance is created and placed at either a random adjacent location if the player was previously holding the corpse or placed directly at the corpse location using the addItem method from the Location class. Lastly, if nothing happens, the tick method of the Corpse will increase the turnCount by 1.

# 5) Farmers and food

Main classes involved:

- Crop (new)
- EatFoodAction (new)
- EatFoodBehaviour (new)
- Farmer (new)
- FertilizeAction (new)

- FertilizeBehaviour (new)
- Food (new)
- HarvestAction (new)
- HarvestBehaviour (new)
- Human (modify)
- PickUpItemAction
- PickUpFoodBehaviour (new)
- Player (modify)
- SowCropAction (new)
- SowCropBehaviour (new)

For the farmers and food section, a lot of the behaviours are unique. Hence, we will create several classes to support the behaviours which are EatFoodBehaviour, FertilizeBehaviour, HarvestBehaviour, PickUpFoodBehaviour and SowCropBehaviour.

To add on that, we plan to create new action classes to support the behaviour classes. For example, HarvestBehaviour will call a method from HarvestAction to execute the action. This design will make the program run efficiently and smoothly.

Next, this section needs one new actor, which is a farmer. To support this, we will create a Farmer class which represents a farmer. Next, a farmer has an ability to sow a crop. Hence, along with the SowCropAction and SowCropBehaviour, we will also create a Crop class which is a ground that represents a crop because a Crop should not be able to be picked up. To implement this feature, SowCropBehaviour will be instantiated in Farmer class. Next a method in SowCropBehaviour will check if there's at least one dirt next to the farmer and give the farmer 33% probability to sow the crop. If it is approved, this class will call SowCropAction to execute the action. When the action is executed, a new Crop will replace the dirt on the ground. Next, a crop will ripen in 20 turns if it's left alone. A method will be created in the Crop class to increment its turn count. When the number of turns reach 20, the crop will be known as a ripe crop and. To verify whether a crop is ripe, a method in Crop class will check whether the total turns is equal to 20.

A farmer or player is allowed to harvest a ripe crop. Therefore, we plan to create

HarvestAction class to support HarvestBehaviour and this behaviour will be added in both Farmer and Player classes. Although harvesting is allowed by both farmers and players, the methods are slightly different so there'll be two overloaded getAction methods for each of them. For the Player getAction, they are allowed to choose which crop to harvest. Hence, it'll return HarvestAction for all crops next to the player. When a player harvest a crop, a new

Food object will be created and added to the player's inventory. For getAction for a normal actor such as the Farmer, since they cannot make any choices, it will randomly choose a ripe crop and return the harvestAction for that crop. The harvestAction will also check who is harvesting the crop. If a Farmer is harvesting a crop, then we will place the food on a random location adjacent to the Farmer for other actors to pick it up. If other actors, such as Player and Human, harvest the crop, then the food will be added to their inventory for them to eat.

Last but not least, food can be eaten by a player or human. Hence,

PickupFoodBehaviour and EatFoodBheaviour alongside with EatFoodAction will be implemented to support the feature inside Player class and Human class. When EatFoodBehaviour's getAction is called from Player class, the player's inventory will be checked whether it contains food. If it contains food, the first Food object obtained from the player's inventory will be eaten by the player by calling EatFoodAction. When a food is eaten, the Food object will be removed from the player's inventory and the player will be healed by a certain amount of health points. Meanwhile, for the humans to eat food and recover some health points, PickUpFoodBehaviour will be called and it will check whether there is a Food object on the ground. When food is found, the human will first pick up the food using PickUpItemAction and add it to the human's inventory. Then, the getAction of EatFoodBehaviour will return an action instead of null, and EatFoodAction will be executed which is performed just like how players eat their food. We also could not implement eating food inside the getAllowableActions of the Food item because that method will be called even when the Food is on the ground, so it does not make sense to be able to eat food before even picking it up.

# 6) Going to town

Main class involved:

- Car (new)
- DriveCarBehaviour (new)
- DriveCarAction (new)
- Player

For going to town features, we plan to add a new class called Car which extends Item class to use the existing item functionality, mainly the getAllowableActions method. This class is essential as cars are the main object used to travel to another destination. We also

encapsulated all the information needed for the Car, which is the destination location and the console display message.

Next, to implement this feature, we override the getAllowableActions method of the Car class to return a DriveCarAction. This method will be called whenever the player steps on the location of the Car and the player will have the option to use the Car to move to another map.

When the player chooses to drive to another game map, DriveCarAction will be executed. The DriveCarAction class will also need to know information about the destination location, so the Car class will pass this information as a parameter when creating the action. When the execute method is called, the program will take the player to the desired different game map and continue their journey there. Although the player now plays on a different map, the game will keep playing on the old map.

# 7) New weapons: shotgun and sniper rifle

Main classes involved:

- Gun (new)
- Sniper (new)
- Shotgun (new)
- Ammo (new)
- SniperAmmo and ShotgunAmmo (new)
- ShootingSubmenu (new)
- SingleTargetShootingAction (new)
- DirectionalShootingAction (new)

To implement this feature, we created a base Gun abstract class that holds the common functionality for guns, to prevent repeated code. Next, we created a shotgun and sniper that inherits from the gun class. These two guns need separate classes because they have very different shooting mechanisms and therefore, have different attributes and methods. We also created a base Ammo abstract class to represent the ammo that guns will consume for each shot. Ammo for specific guns will inherit from this Ammo abstract class to keep them consistent. To create ammo for specific guns, the ammo will have a String attribute that

represents the gun that it is compatible with. To implement shooting guns, we created a ShootingSubmenu class that inherits from Actions, so it can be added to the player actions list and be displayed as a menu option. The submenu that is displayed will vary according to which type of gun is chosen, which allows for expanding in the future because more guns can be easily added to the ShootingSubmenu class. If the shotgun is chosen, then it will generate a DirectionalShootingAction for all 8 directions for the player to choose. If the sniper is chosen, then it will prompt the player if he wants to aim or fire the gun, then ask the player for a target and finally, generate a SingleTargetShootingAction for the chosen target. To reduce repeated code, our submenu system uses the existing engine Menu class to get user inputs instead of trying to code our own way of getting the player choice. We also separated the shooting mechanism into directional shooting and single target shooting to increase encapsulation, because these two shooting actions are very different and have nothing in common with each other, so we shouldn't just make a big shooting action class and dump all the code there because these two shooting actions shouldn't know anything about each other.

For reloading the guns, we decided to override the tick method that gets called when the item is in an actors inventory. The method will check if the actor currently has a compatible gun in its inventory, then reload that gun and remove the ammo from the inventory. We did not use an Action to reload the guns because we wanted the guns to be reloaded automatically whenever an actor picks up ammo. Hence, using the tick method from the base ammo class reduces repeated code because different kinds of ammo that inherit from our base class can also reload using this method as long as the compatible gun is set correctly. Using an Action is also unnecessary in this case as we can encapsulate all the reloading functionality within the ammo itself so other class do not have to know about how the gun is reloaded and it also does not create any new extra dependencies between the Player, the reload Action and the gun and ammo classes. The tick method already has access to the actor and its inventory, so we are not creating any new dependencies here.

# 8) Mambo Marie

Main classes involved:

- Mambo Marie (new)
- ChantingBehaviour (new)
- ChantingAction(new)
- MyWorld (new)
- GameMap
- ZombieActor

- Zombie

Mambo Marie has quite a unique functionality, We will create a Mambo Marie class with all its relevant methods and variables and encapsulate all her functionality within the class. Mambo Marie will extend the ZombieActor class so that it inherits all the basic abilities and methods, it extends ZombieActor instead of Human and her capability is set to undead rather than alive to prevent the Zombies from attacking her. Mambo Marie has only two behaviours and those are Wanderbehaviour which allow her to move around the map in the same way as the other characters in the game as well as a newly defined ChantingBehaviour which is responsible for the chanting section. Alongside the ChantingBehaviour class, will have an accompanying ChantingAction class specifically to create the 5 zombies and place them randomly around the map, we will implement it this way to encapsulate her functionality and allow for other action classes to be implemented in the future if other similar characters are to be added.

We have initialized Mambo Marie and control all its external operations from the Myworld class, it is done this way as the Myworld class is constant throughout the game and once the player dies, the game ends (world stops running); which is exactly what we want. Within the MyWorld class we have initialized an instance of Mambo Marie, which means that once the game starts Mambo Marie will be instantiated but not placed on the map. While MamboMarie is not on the map we generate a random number and if it is found to be less than 0.05 (representing 5%) it will then proceed to generate all the valid random positions on the edge of the given map that the player is currently on and then randomly spawn her on one of these locations.  It has been implemented in a way where it generates all the edge positions (top, bottom, right and left) of the given map, hence it would work on any new maps added to the game as well. Within the MamboMarie class we have added a turn counter that will allow us to keep track of how many turns the priestess has played, it only increments by one if she is placed on the map. By doing this it allows us to remove her from the map (vanish) once she reaches 30 turns without being killed and also allows us to use this counter to call the ChantingBehaviour every 10 turns.

Within the MamboMarie class itself, we only try to get actions from the chanting and wander behaviours when she is on the map and we encapsulated the conditions for when she can spawn zombies and vanish within the chanting behaviour itself, which is consistent with all our other actor actions. In the chanting behaviour, we check if the turn counter has reached a multiple of 10, if so, it calls the ChantingBehaviour which inturn calls the ChantingAction. The ChantingAction is responsible for generating 5 new zombies and adding them to the map. In the execute method, we generate random X and Y coordinates, we then check if an actor can enter this location to check its a valid location, if so then it generates the new zombie instance and adds it to the given location; otherwise enters the while loop of generating random locations until a valid one is found. This process is repeated 5 times using a for loop. Every turn it checks if her counter has reached a multiple of 30 where she's alive, if so, she gets removed from the map representing her vanishing. Considering she is no longer on the map, she now has a 5% chance to return every round just like she did initially. Also as we make use of the same instance, her health/damage before she vanishes will be the same when she returns and will not restore.

We will also add a check before adding her to a map to ensure she doesn't exist on another map. We use a boolean variable onmap which indicates whether the priestess exists on a map in the game; onmap is initialized to false and gets updated as she is added to or removed from a map accordingly, we use this variable to avoid duplication and having her on multiple maps at the same time.

# 9) Ending the game

Main class involved:

- ExitAction (new)
- MyWorld (new)
- Player

In order to be able to end the game, we created an action called ExitAction and add this to the actions list of the player before displaying the menu so that the player can choose to exit the game. When this action is executed, the player is removed from the map which will trigger the existing end game condition. To end the game when all zombies and Mambo Marie is killed, we decided to create a new class called MyWorld that extends from the engine World class. Then, we override the stillRunning method to add the condition that when Mambo Marie and all zombies are eliminated, the method returns false, which ends the game. This also improves code maintainability because in the future, more game ending conditions can be added.

# 10) Bonus Feature: Chopping trees

Main class involved:

- Axe (new)
- Tree
- CutTreeAction and CutTreeBehaviour (new)

FIrst, we made an Axe weapon item that can be used as a normal melee weapon and can also be used to chop trees to produce three planks when the tree is 20 turns old. To allow the player to cut trees using the axe, we decided to implement it in a similar way to all our other actor actions using Behaviour which returns an Action if conditions are met. We could not implement this directly in the getAllowableActions method of the Axe because then we would have to check if the conditions are met to chop trees inside the Axe class which is not possible because the Axe class should not know anything about which Location the actor is on or how old the tree is. We also decided not to implement it in the allowableActions method of the Tree class because firstly, this method is called even when the actor is adjacent to the tree, but we wanted the actor to only be able to chop trees when he is standing on a tree. Secondly, we wanted to increase the maintainability of the code. If any

actors created in the future also have the ability to cut trees using an axe, then it would be easier to just add the behaviour for that actor and get the CutTreeAction whenever possible. If we override the allowableActions of the Tree class, then we would have to catch this action inside the actor's playTurn method by looping through the Actions that are available, and repeat it again for all other actors that can cut trees. Therefore, we decided that using a behaviour and Action is the best way to implement this to keep consistent with previously implemented actions and also to make maintenance easy.