Design Rationale – Assignment 2

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.