Design Rationale – Assignment 1

Group: 404

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

1) Zombie Attack

Main Classes involved:

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

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; this will be done by generating a random number between 1 and 0 and if the number is below a 0.5, it chooses the bite and if it is above then it chooses the punch. If a successful bite attack is implemented, the zombie restores 5 health, this can be done within the AttackAction class where it checks if the chosen attack weapon is bite and if it lands and that the actor is zombie in which case it increases the zombie health by 5 if the health is less than 95; otherwise would just restore the zombie to full health.

There is also the option 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)
- 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 class; 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. 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. Next it will create either a ZombieClub or ZombieMace object and use the addItemToInventory method to add the new stronger weapon to the Player inventory. We will also override the getAllowableAction method in the ZombieLimb class due to the DRY principle because both ZombieArm and ZombieLeg will inherit the method so we do not have to code it twice. The getAllowableAction of the ZombieLimb class will be called in the processActorTurn method of the World class and we plan to make it initialize a CraftWeaponAction and return that Action so it will be added to the actions array. If the player chooses the CraftWeaponAction, it will pass the Player as a parameter into the execute method so that the inherited Actor methods like addItemToInventory and removeItemFromInventory can be used.

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 AttackAction class to handle the chances of a zombie losing its arms and legs when it is damaged and if it does actually lose any limbs, the AttackAction class will call our new methods from the Zombie class to remove the limbs so there is a new dependency between AttackAction and Zombie.

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 extra 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 only return either an AttackAction from AttackBehaviour or DoNothingAction if no attack is available. 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.

When a zombie loses any arms, the method to remove arms from Zombie class will be called in the AttackAction class to remove the specified number of arms from the arms ArrayList. After removing the arm from the ArrayList, that method 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. If the zombie has no arms remaining, it will definitely drop any weapon it is holding. This method will return a boolean of true if the

weapon is dropped and false if the weapon is not dropped, so when the loseLimb method in the AttackAction class calls this method, it can use the return value to decide if it should create a new DropItemAction and call its execute method to drop the weapon. The method to remove arms in the Zombie class will also call another new method whose purpose will be to modify the percentage chance for a punch to happen instead of a bite depending on the number of arms left, assuming that the zombie has no weapon and must use its intrinsic weapon.

To make the game harder for the player, we decided to make the lost limbs of a zombie drop onto the ground that it is standing on so it can immediately pick up the limb as a weapon the next turn. This is implemented in the new loseLimb method for the AttackAction class by getting the location of the zombie and then calling the addItem function to add the dropped limbs as weapons on the location of the zombie.

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. 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:

- Farmer (new)
- SowCrop(new)
- Fertilize (new)
- Crop (new)
- Harvest (new)

For farmers and food features, it is required to let a farmer have 33% probability of sowing a crop when standing next to a patch of dirt. To do this, we plan to create a Farmer class and a SowCrop class. Firstly, we will implement a sowCrop() method inside the Application class. The method will check whether the farmer is standing next to a patch of dirt. If the farmer is next to a patch of dirt, it will give the farmer a 33% probability to sow the crop. If the farmer is given the ability to sow the crop, SowCrop will call Crop class to replace the patch of dirt.

Secondly, if the patch of dirt is left alone, it will ripen in 20 turns. To implement this feature, we will create a method called “ripen” to increment the ripeness of the crop each turn. However, if a farmer is standing next to an unripe crop, the farmer will decrease the time left for the crop to ripen by 10 turns. A simple method called decreaseTime() will be created to reduce the remaining time.

Next, Harvest class will be added to support the ability for a farmer or the player to harvest a ripe crop for food. To complete this support, the Application class will have a harvest() method to check whether the farmer or player is standing next to a ripe crop. If

they are, Harvest class will be called and the crop will drop to the ground or added into the player's inventory depending on who is standing next to the crop.

Finally, the food can either be eaten by the player or damaged by humans to recover some health points. To implement this, a simple method called eatFood() and damageFood() will be added in Player class and Human class respectively. The content of the methods would be similar, which is to wear down the crop. Last but not least, health points of the human or player will be added if the crop is destroyed or eaten.