Design Rationale For OOD Assignment 1

Documentation of Each Task And How It Is Implemented Into The System

Zombie Attacks

Zombies should be able to bite

The Actor Class has a getWeapon() method which iterates through the Actor's inventory and returns the first weaponItem in the inventory if it exists, if no weaponItem exists, it then returns an intrinsic weapon from getIntrinsicWeapon() method. The getIntrinsicWeapon() method will be overwritten in the Zombie class to have a 50% probability to return a bite instead of a punch.

There were no changes to this part of the design rationale as it was all able to be implemented into code.

A successful bite attack restores 5 health points to the Zombie

Actors have a heal() function that increases the actor's hitPoints by a given integer. The Zombie class will need to know when the heal() method can be used.

The AttackAction() has a Execute() method which has an Actor who is doing the attack as a parameter. The Execute() method will check if the Actor is a Zombie, and if the Zombie did a successful bite attack. If so, Execute will call the heal() method of the Zombie.

There were no changes to this part of the design rationale as it was all able to be implemented into code.

If there is a weapon at the Zombie's location when its turn starts, the Zombie should pick it up.

Since Zombie extends ZombieActor which extends Actor, Zombie has an inventory attribute and the method addItemToInventory(). However, Zombie has no ability to pick up a WeaponItem and place it in its inventory. Hence a new PickUpBehaviour() class that implements behaviour must be made so that the when playTurn() in Zombie is called, the Zombie can call the addItemToInventory() method when a WeaponItem and the Zombie is at the same location.

We only made a small change to this part from Assignment 1, which is having the PickUpBehaviour class take in a class of the item in the constructor so that getAction()

method is able to check whether the class within each of the list of items at the location are the same.

The Zombie will use the weapon instead of its intrinsic punch attack.
The Actor class already has a getWeapon() method that iterates the inventory list and returns the first instance of a Weapon if it exists, else it returns the Actor's intrinsic weapon. Since getWeapon() is already called in AttackBehaviour, the Zombie will automatically use the weapon if it has been picked up.

There were no changes to this part of the design rationale when implementing it as code.

Every turn, each Zombie should have a 10% chance of saying "Braaaaains"
A new Behaviour called ZombieBehaviour() can be implemented, and added to the Zombie's behaviours attribute. ZombieBehaviour() will be the first in the behaviours list and will have a 90% chance of returning a null. The 10% when ZombieBehaviour() does not return null, will return a String that will make the Zombie sound like it is doing something Zombie-like

There we no changes to this part of the design rationale when implementing it as code.

Beating Up The Zombies

On creation, a Zombie has two arms and two legs
We define a new Enum type with the following enumerations of: BOTHARM, ONEARM, NOARM, BOTHLEG, ONELEG, NOLEG to denote the number of limbs a Zombie has. Each Zombie will be initialised with BOTHARM and BOTHLEG Capabilities. When the Zombie loses an arm or a leg, the respective Capability will update: removing capability BOTHARM and adding capability ONEARM. Adding and removing capabilities will be done by using the Capabilities class addCapability() and removeCapability() methods

There were no changes to this part of the design rationale when implementing it as code.

Any attack on a Zombie that causes damage has a chance to knock at least one of its limbs off.
On a successful attack, the AttackAction will call the hurt() method of a target to reduce its hitPoints. The hurt() method of the Zombie class will be overwritten to include the ability for a Zombie to lose its limbs. Everytime the hurt() method is called, if the Zombie does not have the LimbCapability of NOARMS or NOLEGS, then there will be a 25% chance for a Zombie to lose either of its limbs. When a limb is lost, it would call either LoseArm() method or LoseLeg() method depending on whether the Zombie does not have Capability of NOARM or NOLEGS. These 2 methods would be add and remove capabilites of the Zombie depending on whether it has both arms, both legs, one arm, one leg , no arm and no leg.

We added 2 new methods called LoseArm() and LostLeg() so that we are able to change LimbCapability of a Zombie object more easily without cluttering the hurt(int points) method with alot of hasCapability and removeCapability method calls.

Effects of Zombie losing an arm

When a Zombie loses its arm its probability of punching instead of biting is halved. For this functionality, we use getIntrinsicWeapon() method to add this additional logic for punching, biting and checking with ArmCapability is as such: if it has BOTHARM capability then punch 50% and bite 50%; if it had ONEARM capability then punch 25% and bite 75%; if NOARM capability then punch 0% and bite 100%.

When the Zombie is holding a weapon it has a 50% chance of dropping it when it loses an arm, and 100% chance of dropping it when it loses both arms. This logic will be implemented in the LoseArm() method when the Zombie loses an arm.

We used the LoseArm() method to drop the weapon the Zombie is holding instead of the hurt() method because it would be too cluttered, and also we added the LoseArm() method to drop the Zombie's arm and used the hurt() method to check whether a Zombie has LimbCapabilities of NOARM, NOLEG.

Effects of Zombie losing a leg

When a Zombie loses a leg, it can only move every second turn and when it loses both legs, the Zombie would not be able to move at all. This logic is implemented in the Zombie's playTurn() method. It will have a condition that will pass on any behaviour related to moving when it has NOLEG capability in LegCapability.

The Zombie has an additional counter attribute that will start incrementing, called stagger, when Zombie contains a ONELEG capability. The playTurn method checks if ONELEG capability is true and whether the behaviour is an instance of HuntBehaviour or WanderBehaviour. After that, the method would then check if (age - stagger) %2 = 0, if it is true, then it would return a new StaggerAction(). Else, it would return the action which is not part of Hunt or Wander behaviour.

The StaggerAction() class is very similar to a DoNothingAction() class, except that it is used for Zombies.

We added a new class called StaggerAction() class and although we could use DoNothingAction() of StaggerAction(), however in context, the name of StaggerAction fits in more well because a Zombie would be "stunned" if it had lost a leg, rather than just doing nothing.

Lost limbs drop to the ground and can be wielded as simple clubs.

The LoseArm() and LoseLeg() methods adds the broken limbs onto the ground by getting the adjacent locations relative to the Zombie and randomly choosing a location to add the limb to the ground so anybody can pick it up.

We use LoseArm() and LoseLeg() method instead of hurt() method to drop limbs onto the ground because it makes more sense, in a design perspective, to implement it in LoseArm() and LoseLeg() since we have already implemented these methods. We also dropped the limbs on the adjacent locations instead of directly at the location of the Zombie because we do not want to have the Zombie immediately picking up their own limbs since Zombies are not that intellectual.

Crafting Weapons

For our design, we would create a new class called CraftAction class that would extend from Action class and override its Execute() method. Also, we add a new capability enum called CraftableCapability which has Capability CRAFTABLE.

For Item objects in the game that can be used as resources for crafting, they will have a CRAFTABLE capability. The Player class will have a new method called getAdditionalActions(), where it will iterate through the player's inventory and add new CraftAction() Actions for every item that has CRAFTABLE capability. The getAdditionalActions() method will be called in the Player class playTurn() method so that the new actions can be added to the Player's existing possible actions. The CraftAction() takes in an item object with CRAFTABLE capability, and in its Execute() method, will delete the item from the player's inventory and add a new item into the player's inventory. Execute() has a helper method called getCraftItem() that will return a new Item depending on the item passed though the CraftAction(Item item) item parameter. In this case, if the item passed through CraftAction() is a ZombieArm object, getCraftItem() will return a new ZombieClub object. Likewise, it will return a ZombieMace object for a ZombieLeg object.

We create and use the getAdditionalActions() method in Player class rather than using the execute method in CraftAction class because the execute method in CraftAction is supposed to execute the action of crafting a Zombie leg or arm into a Zombie mace and club, rather than return a new CraftAction. We also added a new method called getCraftItem() in CraftAction to create a new Zombie club or mace is so that we can reduce clutter in the execute method.

Rising From The Dead

Since the portability of corpses of humans and zombies is true in Item class, this means that the Actor is able to pick up and drop them. For our design, we created a Corpse Class that extends Item. We also modify corpse Item in the Execute method of AttackAction class from PortableItem object class to of Corpse object class.Also, we assume for our design that the corpse will still experience the passage of time, whether the player is holding the corpse of a human actor or the corpse is on the ground .

So from that, We utilised the 2 tick() methods, tick(Location currLocation) and tick(Location currLocation, Actor actor), that are supposed to be implemented in Corpse class to check whether the item is between 5 to 10 turns.

For the tick(Location currLocation) checks the passage of time for a corpse on the ground, if it has been 5 to 10 turns and the 50% chance of rising from the dead is true, then we would remove the item from that location and create a new zombie object and place it back at that same location. For the tick(Location currLocation, Actor actor) method, it is almost the exact same thing as tick(Location currLocation), however this method would loop through the actor's inventory and if the Item in the inventory is an instance of Corpse, it would remove the Item from the actor's inventory and call placeZombie(Location location).

For placeZombie(Location location) method, it would check whether the current location contains an actor or not. If it does not, then it would add a zombie object to that location. Else, we would get the adjacent locations relative to the current location and use a loop to check whether these locations contain an actor or not, if they don't have an actor, then we place the Zombie object there and break out of the loop.

We used the 2 tick methods rather than using getAgeOfCorpse method because tick keeps track of the corpse's age more efficiently. We also created placeZombie() method in order to avoid the error of placing the zombie at a place that is not allowed by the engine.

Farmers and Food

when standing next to a patch of dirt, a Farmer has a 33% probability of Sow a crop on it

For our design, we would create a new class in the game package called Farmer that inherits methods from Humans (extend Human), a Crop class that inherits from Ground class, a SowBehaviour class that implements Behaviour and a SowAction class that inherits from Action.

The crop class would have a constructor with its displayChar(), '-', at this point.

The SowBehaviour class would have a getAction() method which would use the method getExits() from GameMap (map.locationOf(actor).getExits()) in order to get all the adjacent cells around the farmer. Then for each exit, we would use method getDestination() in Exit class in order to get the location for each of them and use method canActorEnter() from Location class to check if the terrain is passable by overriding the canActorEnter() method in Ground. We also check if the location of the ground is not an instance of Crop. If both are True, then the farmer takes the first exit from destinations to have a 33% chance of Sow a crop on the dirt. If a farmer does Sow the crop (33% is true), then the method would return a new SowAction() class. Else, it would return null.

The Farmer class would have the same methods as Human class except that Farmers have a variable which is an array of Behaviours which contains WanderBehaviour , SowBehaviour, EatBehaviour, FertiliseBehaviour and HarvestBehaviour.

For the SowAction class, we would have an execute method that would create a new Crop object and replace the ground object at the location that a farmer would sow, with the Crop object and return a string that states that the farmer that sowed the crop on that location. The SowAction class would also take in parameter of Location cause the class would need to know at which location that the farmer would like to sow the crop at. Other than that, this class also has a menuDescription(Actor actor) method that returns a string description of the action in the menu.

We changed the method for execute method of SowAction class because changing the displayChar of the ground is not enough. We have to place a new Crop object on that location so that we can fertilise and harvest it.

left alone, a crop will ripen in 20 turns

We created a new CropCapability enum class with a state, RIPE and the Crop class has a tick() method that counts the number of turns until it turns ripe, which is 20 turns, and then it would add a capability of RIPE to the crop and change the displayChar of the Crop from '-' to '='.

We change the tick() method to add capability of RIPE to a Crop object when it reaches 20 turns for its age because we need to check whether it's ripe or not to be able to harvest it.

when standing on an unripe crop, a Farmer can fertilize it, decreasing the time left to ripen by 10 turns

For our design, We created 2 new classes called FertiliseBehaviour and FertiliseAction in the game package. We also created 2 new methods called getAgeOfCrop() and setDisplayChar() in Crop class.

In FertiliseBehaviour class, we have an override method getAction() which uses method locationOf() from GameMap (map.locationOf(actor).) in order to get the location that the farmer is standing on. For the current location of where the farmer is standing, we would use method getGround() from Location class and then use method hasCapability() from Ground class to check whether the ground that the location currently has an unripe crop. We also check whether the ground is of instance Crop.If both are true, We would cast the Crop at that location to a new Crop variable, then the getAction() method returns a new FertilizeAction(crop). Else, it would return null.

For the FertiliseAction class, the constructor takes in a Crop object because we need to know the age of the Crop object in order to fertilise it. We also created an Execute override method that calls the getAgeOfCrop() from the Crop class in order to get the age of the crop. With that, we added 10 to the age of the crop since fertilising the crop would reduce the time to ripen by 10 turns. Then, we check whether the age of the crop is >= 20. If this is True, then we add the CropCapability of RIPE to the Crop object and set its display character from '-' to '='. The method also returned a String stating that the crop is fertilised. The class also has a menuDescription(Actor actor) method that returns string description of the action in the menu.

We added small changes such as for FertiliseBehaviour checking whether the ground is an instance of Crop and having the FertiliseAction take in only parameter of Crop because the class needs to know the age of the Crop to be able to harvest it. We also added to FertiliseAction where it would add the capability of RIPE to the crop when it passes age of 20 to indicate that it is ready to harvest.

when standing on or next to a ripe crop, a Farmer (or the player) can harvest it for food. If a Farmer harvests the food, it is dropped to the ground. If the player harvests the food, it is placed in the player's inventory

We created 3 classes called HarvestAction class that inherits from Action class, HarvestBehaviour class that implements Behaviour and Food class that inherits from PortableItem class. We also made a new enum called EatCapability that has Capability of EATABLE.

For Food class, we extended from Item class, would have a constructor which would have a parameter of health points it recovers. The constructor would also add a EatCapability of EATABLE to each food object.

For HarvestBehaviour class, we made an override getAction() that would use the method getExits() from GameMap (map.locationOf(actor).getExits()) in order to get all the adjacent cells around the farmer and also get the current location of the actor. From there, we first check if the current location has Crop which has CropCapability of RIPE, if it does, then we return a new HarvestAction(Location location). If not, then for each exit, we would use method getDestination() in Exit class in order to get the location for each of them. After that, we checked whether the ripe crop is at adjacent locations. If it is true, then the method would return a new HarvestAction() method. Else, it would return null.

For HarvestAction class, we have a constructor that takes in Location HarvestAction needs to know where to harvest the crop. We also made an override Execute method that would removeCapability of a ripe crop and created a new Dirt object and use setGround() method to the Dirt object. After that, we would create a Food object and then check if the actor taken in as a parameter of the method is an instance of a Player or not. If it is, we add the item to the Player's inventory and return that the actor has harvested the crop. Else, we place the food that was harvested onto the same location and return the string. The class also has a menuDescription(Actor actor) method that returns a string containing the action in the menu.

We changed the execute method of HarvestAction class from just removing capability of ripe crop and using addItem() method to creating a new Dirt and Food object, the Dirt object is used so that when the Food object is picked up from the ground, the ground would not be a ripe crop. The Food object is needed to be placed on the ground so that a human actor is able to pick it up. Other than that, we changed the getAction method of HarvestBehaviour class to check the current location where the actor is standing on to see if there is a ripe crop and then check the adjacent locations of the actor to check for the ripe crop. This is because the previous design was too clunky and hard to design, although the concept for both is similar.

Food can be eaten by the player, or by damaged humans, to recover some health points

We made a EatAction class that inherits from Action class and a EatBehaviour class that implements a Behaviour interface. We also added a new method in Food class called getHealthPointsRecovered() to get the health points that the Food object recovers.

For EatBehaviour class, we have a getAction method that gets the list of Item's at the current location of the actor. We also check if the current health of the actor is bigger or equals the max health of the actor, if it is then we return null. If not, we loop through the list of Item's at that location, if any of those Item's is an instance of Food, then we cast that Food item to a new variable called foodObj and return a new EatAction(foodObj). Else, we return null.

For EatAction class, we have a constructor that takes in parameter Food because EatAction needs to know the health of the Food object. We also have an execute method that check if

the actor is an instance of Player, if it is, then we remove the food from the player's inventory and use it to heal the player. If not, then we assume that it's a Human object which would remove the food from the ground and use it to heal themselves. If the actor's health that has been regained equals to his max health then we return a string that actor has regained to full HP. Else, we return a string that actor has regained to a certain amount out of his full HP.

The reason we change the name of IsHurtBehaviour and UseAction classes to EatBehaviour and EatAction classes is because it would align more with how these 2 classes work. We also dont use the asFood() because the method in EatBehaviour would do the same thing as that method.