2099 Assignment 1 Design Rationale

**1 - Zombie Attacks – Ahsan Zafar**

**1.1 - Zombies should be able to bite. Give the Zombie a bite attack as well, with a 50% probability of using this instead of their normal attack. The bite attack should have a lower chance of hitting than the punch attack, but do more damage.** **A successful bite attack restores 5 health points to the Zombie.**

How your proposed system will work:   
*1.1.1 – Giving zombie a bite attack with 50% chance.*We plan on modifying the getInrinsicWeapon( ) method in the Zombie Class.  
Initially, it was only returning a punch Intrinsic weapon.  
Now, it will have a 50% chance of returning a punch Intrinsic weapon, and a 50% chance of returning a bite Intrinsic weapon.

*1.1.2 - The bite attack should have a lower chance of hitting than the punch attack, but do more damage.*  
We plan to modify the execute( ) method in AttackAction class.   
Initially, AttackAction only executed a punch weapon, which had a 50% chance of either hitting or missing.

Now, we use a variable “HitChance” to determine the chance of a hit or miss.  
Within the execute ( ) method, we first check if the weapon to be executed is a bite. If it is a bite, HitChance is 40%, allowing for a 40% probability of hitting, and 60% probability of missing.   
Else, ( if weapon to be executed is punch ), we leave HitChance as it is, which is 50% (allowing for original functionality to be retained).

*1.1.3 - A successful bite attack restores 5 health points to the Zombie.*

In the execute ( ) method, we are already checking what weapon is used.   
We simply perform the following: if the weapon used is bite, we heal the zombie by making using of the heal method in the Action class.

Why this way has been chosen:

[Explaining 1.1.1, 1.1.2, 1.1.3 together].

The decision to modify the getIntrinsicWeapon( ) is made due to the observation that the current functionality of the system uses the above method to return a weapon. By simply modifying the getIntrinsicWeapon to return a punch or a bite (Rather than just a punch), we do not change the original behavior of the code, we just add an additional feature.

Additioanally, this way has been chosen to prevent “premature optimization”. Premature optimisation is when you spend a lot of time on something that you may not actually need.

The above approach would be a bad idea if a Zombie had 100 weapons with different probabilities, because it would potentially mean we would need to use 100 if statements to determine set the hit probability. However, given the assignment specification, a Zombie does not have 100 weapons at this point in time. We only need to change the hit probability if a bite is used.

Using an if statement only once results in a one-line computational cost, which is really not that huge. Using the above approach allows for all Assignment specifications to be reached, prevents us from “premature optimization”, and also allows for greater understandibility.

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

How your proposed system will work:  
*1.2.1 – Zombie should pick up weapon.*

Zombie currently has 3 behaviours, namely attack, wander, and hunt. We create a new class “PickUpBehaviour” which implements the “Behaviours” interface and add this new behavior to Zombie.

The PickUpBehaviour will pick up an item if it is on the location of where the zombie is standing. It picks up an item by using the PickUpItemAction provided in the engine package.   
If there is no item on its location, it will move on to see if it can either attack, hunt, wander or do nothing (as per the normal zombie operation).

Why this way has been chosen:

[Explaining 1.2.1].  
We observed that every action performed by a Zombie is through a behaviour. Initially, we thought of adding the functionality of picking up an item in AttackBehaviour, but this would be incorrect as a Zobie will then be performing 2 turns in one go instead of 1 turn.

To keep consistent with how a Zombie performs, we decided to create a new behaviour to perform a Pick Up item action. This allows us to make use of the given Behaviour interface. Additionally, creating a “PickUpBehaviour” class allows other actors to use this behaviour now, allowing our system to be easily extendable. (Let’s suppose humans are required to pick up items, they can use this behaviour to do so).

**1.3 - Every turn, each Zombie should have a 10% chance of saying “Braaaaains” (or something similarly Zombie-like)**

How your proposed system will work:

*1.3.1 - Every turn, each Zombie should have a 10% chance of saying “Braaaaains”.*  
In the entry to the playTurn ( ) method in Zombie class, we set up a 10% chance of printing brains to console. If the 10% chance is successful, we call the println ( ) method in display class (in engine package) to print to console.

Why this way has been chosen:

[Explaining 1.3.1].  
The playTurn ( ) method is called once per turn, therefore adding this functionality to this method allows for a 10% chance of saying “ **Braaaaains** “ every turn.

Additionally, we choose to call the println ( ) method in display RATHER than printing to console directly because the engine class should only be printing to console.

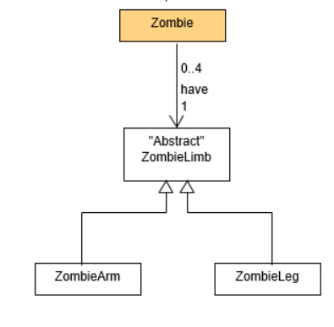
NOTE: The way we have implemented this is, the chance of saying “Brains” is INDEPENDENT on a Zombie executing it’s turn. (Example, a zombie may say that “Brains” and also perform a punch attack).

**2 - Beating up the Zombies - Ahsan Zafar**

**2.1 -   
On creation, a Zombie has two arms and two legs. It cannot lose more than these.  
Any attack on a Zombie that causes** **damage has a chance to knock at least one of its limbs off.   
Lost limbs drop to the ground, either at the Zombie’s location or at an adjacent location.  
Cast-off Zombie limbs can be wielded as simple clubs – you decide on the amount of damage they can do.**

How your proposed system will work:

*2.1.1 – How zombie limbs are being implemented in the game.*

To implement Zombie limbs into the game, we plan on adding a “ZombieLimb” *abstract* class to our system. The ZombieLimb class will extend WeaponItem class. We then create a “ZombieArm” class and a “ZombieLeg” class, which extends “ZombieLimb”.   
  
  
  
  
 \*\* NOTE: Zombie limbs extending from WeaponItem is not shown here \*\*  
  
  
  


A zombie then stores these limbs in a collection.

We plan on instantiating the arms and legs in the constructor of zombie.

*2.1.2 – A successful attack on zombie**has a chance to knock at least one of its limbs off.*

We plan to override the hurt method in the zombie class. In the override, we will add code so that there is a 25% chance to lose a limb. A limb is lost by removing it from the collection that is storing the limbs.

The hurt method is called by engine if and only if damage needs to be done to the actor. We take advantage of this by adding this functionality in the override hurt method in Zombie class.

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

This is requirement is achieved by the fact that the “ZombieLimb” class extends “WeaponItem” class. In simple english, this means that a zombie limb is also a weapon item. We already know that the specified actors can choose to pick up any items on the ground at its location. Given that a zombie limb is now a weapon item, once a zombie’s limb is knocked off, we simply drop it to the ground, and now it can be used as a simple club weapon.

Why this way has been chosen:

[ Explaining 2.1.1 and 2.1.3]  
Conceptually, we can think of zombies having limbs in the following way: A zombie limb is a Weapon item that the Zombie possses (but cannot use it as a weapon; it’s just a limb of its body obviously). Once the limb falls off, that limb is now a weapon item that can be used by the player (or other actors that have the functionality of picking up weapons). This is exactly why we chose for ZombieLimb to extend WeaponItem.

If ZombieLimb class did NOT extend WeaponItem, then once a zombie loses its limb, we would have had to instantiate a new weapon Item (which is redundant, given that we can just use the Zombie limb as a weapon itself).

>> continued…

The ZombieLimb class is chosen to be an abstract because it can be used as a common definition of a base class that multiple derived classes can share. For example, if we need to add a function which all zombie limbs need to have, such as bleeding (this is a very random example), we can just add it to the base abstract class, and now all derived classes can share this function. Additionally, an abstract class allows our system to be extandable; we can add a new zombie limb such as a Zombie head (another random example).

Finally, by instantiating the arms and legs in the constructor of zombie, we adhere to that requirement that on creation, a Zombie must have two arms and two legs.

[ Explaining 2.1.2]We choose to override the hurt method in zombie class because the loosing limbs functionality only applies to Zombies. If we did not override in zombie class, and modifed the existing hurt methord, then all actors (such as humans and players) will be forced to lose limbs as well (since the hurt method applies to all actors).

Additionally, the hurt method is called by engine if and only if damage needs to be done to the actor. This is how we are going to adhere to that requirement that a limb has a chance to fall off if and only if the attack on zombie is successful.

**If a Zombie loses one arm, its probability of punching (rather than biting) is halved and it has a 50% chance of dropping any weapon it is holding. If it loses both arms, it deﬁnitely drops any weapon it was holding.  
If it loses one leg, its movement speed is halved. If it loses both legs, it cannot move at all, although it can still bite and punch.**

How your proposed system will work:

*2.2.1 – If a zombie loses one arm, probability of punching is halved*.   
No new classes are created to implement this feature. We simply just modify the getIntrinsicWeapon( ) method so that it first checks how many arms a zombie has. Initally, a zombie had a 50% chance to make an attempt to punch, and 50% chance to make an attempt to punch. Now, we modify the method in such a way that the chance of a punch attack is reduced to 25% if one arm is lost, and chance of a punch attack is reduced to 0% if it both arms are lost.

*2.2.2 – If a zombie loses one arm, it has a 50% chance of dropping any weapon it is holding. If it loses both arms, it definitely drops any weapon it was holding.*

No new classes are created to implement this feature. In 2.1.2, we override the hurt( )method in Zombie class. We use the same hurt( ) method to check if during an attack on a zombie, if it loses an arm, there should be a 50% chance that the zombie drops the weapon. And if it loses both arms, it should drop all weapons,

*2.2.3 – If it loses one leg, its movement speed is halved. If it loses both legs, it cannot move at all, although it can still bite and punch.*

[Updated]

Depending on the condition of the zombie, we decide which behaviours the zombie is allowed to use.   
The behaviours are stored in an array. By choosing which behaviours to loop over and which behaviours to ignore (given the zombie condition), we can implement this specification.

For example, the playturn( ) method will NOT let a zombie use either a Wander behaviour or Hunt behaviour if it has both legs missing. It will also check if the has 1 leg AND the zombie moved in it’s last action. If this is the case, the actor will not move.

Why this way has been chosen:

[Explaining 2.2.1 and 2.2.2]  
Although the explanation given suits an "implementation rationale" rather than a “design rationale", I thought it was still interesting to mention why we do this. It is done because of maintainability; if the zombie still has both arms, it will still have a 50% chance of punching, adhering to the initial requirment of the system.

[Explaning 2.2.3]  
We do it this way so that we can prevent our system from being very messy and unreadable, and for extendability. Suppose we chose the approach of performing multiple if statements to check what behaviour the zombie should be allowed to perform, then this would be very bad practise as it we were to implement 100 new behaviours in the future, we will potentially have 100 if statements in our playturn( ) method.

By performing this, we adhere to the requirement that a zombie can pick up an item (if it has an arm), perform a punch (if it has an arm) and perform a bite. To summarise, this way has been chosen as it prevents us from changing code that other classes might be relying on, makes the system easily extendable and adheres to all requirements, without modifying the current functionality.

**3 – Crafting weapons - HinSeng**

**3.1 - If the player is holding a Zombie arm, they can craft it into a Zombie club, which does significantly more damage**

How your proposed system will work:   
*3.1.1 – If the player is holding a Zombie arm, they can craft it into a Zombie club, which does significantly more damage.*There will be a class called CraftingAction, it is used to let player craft a weapon to a different weapon. There will be method to check whether there is any zombie arm in player’s inventory list.

Why this way has been chosen:

It is because there is no existing action to craft weapons so we need to create a new class for it and we can use this class to craft weapons (Increasing damage).

**3.2 - If the player is holding a Zombie leg, they can craft it into a Zombie mace, which does even more damage**

How your proposed system will work:   
*3.2.1 – If the player is holding a Zombie leg, they can craft it into a Zombie mace, which does even more damage*There will be a class called CraftingAction, it is used to let player craft a weapon to a different weapon. There will be method to check whether there is any zombie leg in player’s inventory list.

Why this way has been chosen:

It is because there is no existing action to craft weapons so we need to create a new class for it and we can use this class to craft weapons (Increasing damage).

***4 – Rising from The Dead – Ahsan Zafar***

**4 - As everybody knows, if you’re killed by a Zombie, you become a Zombie yourself.** **After a Human is killed, and its corpse should rise from the dead as a Zombie 5-10 turns later.**

How your proposed system will work:

*4.1.1 - After a Human is killed, its corpse should rise from the dead as a Zombie 5-10 turns later.*

We create a new class called “ResurrectableCorpseItem” which extends “PortableItem” class. Inside this new class, we override the tick( ) method; we check if the age of corpse is old enough for a human corpse to risen from the dead as a zombie. If it is, we create a new zombie and place him at a random location on the map (although the spawn location is random, we ensure that it is an acceptable location).

We also plan to modify the AttackAction class. The additional functionality is that if a Zombie is successful in killing a human when performing an action, we instantiate a new ResurrectableCorpseItem instead of a normal corpse item.

Why this way has been chosen:

[ Explaining 4.1.1 ]  
Inheriting the ResurrectableCorpseItem from PortableItem is done because, we only want to override the tick method for a corpse, not any other item. If we chose to override the tick method in portable item, this would be extremely bad as every portable item in the game map will create a new zombie once certain number of turns have passed.   
Additionally, we know that the tick( ) method is automatically called by engine once per turn. By inheriting from Portable item, we can know re-use this functionality, and remove the possibility of implementing duplicate code.

By simply modifying the execute( ) method in attackAction to when to create a resurrectableCorpseItem, we successfully implement this assignment specification without impacting how other parts of the system interact with the execute( ) method.

This approach might be a bad idea if there were a lot of actors in the game, and the way a corpse item of those actors are used is different for each type of actor. For example, if a zombie kills an actor of type x, the actor should be eaten (rather than rise from the dead). Or If a zombie kills an actor of type y, the zombie should store the actor’s limbs in its inventory and should NOT rise from the dead. The reason it is bad is because it could potentially lead to many if statements. However, at this point in time, we only need to implement a human rising from the dead, and this could be achieved by using an if statement which results in one-line computational cost. This not only allows us to adhere to the assignment requirement, but also prevents us from “premature optimisation”, which is considered the “root of all evil” in computer science.

**5 – Farmers and food - HinSeng**

**5.1 - When standing next to a patch of dirt, a Farmer has a 33% probability of sowing a crop on it**

How your proposed system will work:   
*5.1.1 – Standing next to a patch of dirt, a Farmer has a 33% probability of sowing a crop on it.*We will first create a Farmer class. Then we will create a SowingBehaviour class used by the farmer and this behaviour will generate SowingAction which takes dirt’s location as parameter.

Why this way has been chosen:

Farmer is inherited from human and he can perform extra behaviour such as sowing, so we decide to create SowingBehaviour class and it can be used by the farmer.

**5.2 - Left alone, a crop will ripen in 20 turns**

How your proposed system will work:   
*5.2.1 – Left alone, a crop will ripen in 20 turns.*We will create a Crop class. It is similar to the Tree class and it has method that changes its display character once the crop is ripen.

Why this way has been chosen:

Its features are like trees’, so we decide to create a Crop class.

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

How your proposed system will work:   
*5.3.1 – Standing on an unripe crop, a Farmer can fertilize it, decreasing thie time left to ripen by 10 turns.*

We will have a FertilizeBehaviour class for famers and it will generate FertilizeAction that takes farmer’s location as parameter and check whether the farmer is standing on an unripe crop. If yes, decrease the time left to ripen by 10 turns. A method for decreasing the turns should be added into Crop class.

Why this way has been chosen:

Fertilize is a new behaviour so we need to create a new class and let the farmer use it.

**5.4 -** **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**

How your proposed system will work:   
*5.4.1 – 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 will have a HarvestBehaviour class for and famers and this class will generate HarvestAction for both farmers and player. It will check the locations around the actors to check whether there are ripe crop and pass them to HarvestAction. Food class is created and it will drop to the ground or place into player’s inventory depending who harvests the ripe crop.

Why this way has been chosen:

Harvest is a new behaviour so we need to create a new class and let the farmer use it.

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

How your proposed system will work:   
*5.5.1 – Food can be eaten by the player, or by damaged humans, to recover some health points.*

We will have a EatingBehaviour class for human, this class is used to check whether human’s hp is below than his max hp. If yes, then generate EatingAction and heals them. Player can directly use EactingAction to perform eating.

Why this way has been chosen:

Eating is a new behaviour so we need to create a new class and let the human use it.

**6 – Going to town - HinSeng**

**6.1 - Place a vehicle somewhere on your existing map**

How your proposed system will work:   
*6.1.1 – Place a vehicle somewhere on your existing map. The vehicle should provide the player with the option to move to a town map.*We will first create a vehicle class. Then we will add MoveActorAction to the vehicle it provides an option for the player to move to the town map. The town map will be a list of string just like the existing map.

Why this way has been chosen:

Vehicle is a new item so we decide to create a Vehicle class inherited from Item class and set that it is not portable. The **Reusability Principle** allows us to implement easily with no unnecessary code. We then add MoveActorAction for the vehicle so when player steps on it, it provides an option for player to move to the town map.

**6.2 - Somewhere on the town map, place a sniper rifle and a shotgun**

How your proposed system will work:   
*6.2.1 – Somewhere on the town map, place a sniper rifle and a shotgun.*We will first create sniper and shotgun class, then we place these items on the town map.

Why this way has been chosen:

It is the same as the plank in the existing map, so we just simply follow the way.

**6.3 - When you leave a map, any creatures in the old map should continue to move and act**

How your proposed system will work:   
*6.3.1 – When you leave a map, any creatures in the old map should continue to move and act.*

When we create the new town map, we will add this town map to the same world as the existing map did. The two map will then run synchronously so even the player leave a map, any creatures in the old map will continue to move and act.

Why this way has been chosen:

World class is the main class to let the creatures move and act, so we just need to ensure all the maps are added into the same world so these maps can run synchronously.

**8 – Mambo Marie - HinSeng**

**8.1 - Mambo Marie is a Voodoo priestess and the source of the local zombie epidemic. If she is not currently on the map, she has a 5% chance per turn of appearing**

How your proposed system will work:   
*8.1.1 – Mambo Marie is a Voodoo priestess and the source of the local zombie epidemic. If she is not currently on the map, she has a 5% chance per turn of appearing.*We will first create a MamboMarie class as she is a new zombie. Then we will create a tracker class to keep track whether she is existing on the map, and set the probability to 5% of appearing.

Why this way has been chosen:

Mambo Marie is a new zombie so she is inherited from zombie actor. The **Reusability Principle** allows us to implement easily with no unnecessary code. We will create a new class called Tracker to keep track the Mambo Marie. This class is inherited from gameMap class and we will override the tick method to check whether Mambo Marie exists on the map. Tracker also uses the **Reusability Principle** allows us to implement easily with no unnecessary code.

**8.2 - She starts at the edge of the map and wanders randomly**

How your proposed system will work:   
*8.2.1 – She starts at the edge of the map and wanders randomly.*We just need to add WanderBehaviour for Mambo Marie to let her wander randomly. In Tracker class, we need to get the X range and Y Range of the map and the minimum and maximum value of them. Then, add these locations to a list and spawns the Mambo Marie at one of them.

Why this way has been chosen:

We already have WanderBehaviour so we can just apply this behaviour for Mambo Marie. For getting the edge of the map, this is a easy way to get the edges as the gameMap already have the functions getXRange() and getYRange().

**8.3 - Every 10 turns, she will stop and spend a turn chanting. This will cause five new zombies to appear in random locations on the map. If she is not killed, she will vanish after 30 turns**

How your proposed system will work:   
*8.3.1 – Every 10 turns, she will stop and spend a turn chanting. This will cause five new zombies to appear in random locations on the map. If she is not killed, she will vanish after 30 turns.*

Within the MamboMarie class, we will have a variable to count the turns of Mambo Marie and perform different actions according to the turns. We will create a class called ChantingBehaviour which generates ChantingAction and let Mambo Marie to spawn five new zombies in random locations on the map.

Why this way has been chosen:

It is because there is a method called playTurn in Actor class and we can override this method to let it count the turns of Mambo Marie. Chanting is a new behaviour so we need to create it and let Mambo Marie to use it.

**8.4 - Mambo Marie will keep coming back until she is killed**

How your proposed system will work:   
*8.4.1 – Mambo Marie will keep coming back until she is killed.*

We just need to use the isConscious method of the Actor class to check whether Mambo Marie is alive. If yes, she can come back after she vanishes. If no, she cannot come back.

Why this way has been chosen:

This is the easiest way to determine whether Mambo Marie is alive and perform different actions according to this. **DRY Principle** is used as the method isConscious is already in Actor class so we can just simply use it.