**DESIGN RATIONALE**

ASSIGNMENT 1

Cedric Liang, Nathan Vaughan

**Zombie Attacks**

We inspect the existing code to note that the randomisation for the hit probability is attached to AttackAction.execute. The generation for the hit probability is the same for each weapon, at 50%. We need to handle the possibility for each weapon to have its own hit probability.

Since hit probability is not natively supported by the Weapon class, we can only crudely hardcode it into the AttackAction class. We create new WeaponItem classes ZombieArm, ZombieLeg, ZombieMace, ZombieClub, and new IntrinsicWeapon classes ZombiePunch, HumanPunch and ZombieBite. We then check in the AttackAction class which of these the Weapon is an instance of, retrieving and hardcoding the weapon attack probabilities in there.

We considered creating a subclass of Weapon that would add variables corresponding to the hit probability, then pointing *all* the weapons in the game class to be a subclass of this new subclass of weapon, but this runs into two issues: first, the Actor.getWeapon method only refers to a Weapon and this is in the Engine package, and two, this implementation is ‘cheating’ in a way, and doesn’t seem like good code practice.

If the weapon is of type ZombieBite, we call the Actor.heal method on the attacker to heal the zombie.

When checking what Action a Zombie must execute on their turn, their top priority is to pick up any WeaponItem at their location. If it can, then we simply return the PickUpItemAction in playTurn() before it can check the other behaviours. We only allow Zombies to pick up at most one WeaponItem. This is to ensure that the Zombie does not continuously pick up any WeaponItems available to them. This implementation is based upon the idea that it is not realistic for Zombies to have an inventory where they can store all of their Weapons and Items because they are not meant to be smart.

Similarly, we can do the same for saying brains, except this should be a simple print calling the println method in the Display class, and in playTurn it returns a DoNothingAction to consume a turn.

**Beating up the Zombies**

We implement new private attributes in the Zombie class that correspond to the number of legs and arms it has.

We implement a new method in ActorInterfaces called takeDamage that handles all actors in the game’s damage taking logic. We make it also have a reference to the location of the actor, by feeding in 'map', then overwrite the takeDamage(int damage, GameMap map) method in zombie to handle game logic about the subtraction of limb count and the creating of an object at the location.

The reason for this is because the basic Actor.hurt method does not contain a reference to the location of any of the actors, and implementing a method in ActorInterfaces allows for each Actor to handle being damaged with different logic, and reduces bloating of the AttackAction class.

Adding onto the override of the takeDamage method for Zombie, we include two more methods limbFallOff(GameMap map) and dropWeapon(GameMap map) to simplify the takeDamage method. The limbFallOff method handles the logic behind knocking a limb off a Zombie, choosing a random limb to knock off. Extending this limbFallOff method, the dropWeapon method handles the logic behind whether or not a Zombie loses its WeaponItem when an arm is knocked off. If both arms are knocked off, then it is guaranteed that the Zombie will drop its WeaponItem, but 50% chance if only one arm is knocked off.

We can code the impact these private attributes have on the zombie’s attack probabilities by coding the randomisation as being dependent on the zombie’s arm and leg private attributes in the getIntrinsicWeapon override in the Zombie class.

Since takeDamage has a reference to the location on the map, we can use the extended methods limbFallOff and dropWeapon to create/present objects such as ZombieLeg, ZombieArm or the WeaponItem the Zombie was previously holding at the location of the Zombie.

World already stores information about what each Actor did in the previous turn, and this can be accessed in the playTurn method in the Zombie class. We can then simply perform a type check on the previous move depending on the number of legs the zombie has (if lastAction instanceof MoveActorAction). This way, we can slow or immobilise the Zombie by altering what behaviours are available to it on a given turn depending on its last action.

**Crafting Weapons**

We need to create a new class called CraftAction. This action takes in an instance of the parent item, the child item, and the Action simply removes the parent Item from the map or the Actor’s inventory and the child item is added to the map or inventory.

To allow the Player to craft certain Items, we need to add CraftAction as an allowableAction to the parent Item, where the argument passed into CraftAction are the parent item itself and a newly created instance of the child Item. As such, the Item associations are stored in each parent item. Each parent knows its child, but the child does not know its parent.

When an Actor stands near a craftable Item, the associated CraftAction will become an option available to the Actor, and the Actor can execute that Action. Since only Players have direct access to Actions (NPCs must have their Actions governed by Behaviours or their own PlayTurn logic), only Players can perform CraftAction through the allowableAction mechanism.

We then create these new weapon classes.

**Rising from the Dead**

It doesn’t make sense to support the possibility to move corpses (almost all industry AAA shooters and RPGs don’t support this capability), so we will create a new class called StationaryItem. We create Corpse as a subclass of StationaryItem. Since we need to differentiate between corpses that were killed by zombies and corpses that were killed by human (to support any future human vs human combat), we create a new class called InfectedCorpse, which is a subclass of Corpse. We give this corpse a private attribute called age, and a private attribute called reanimateWhen. The age starts at 0 and increments one per turn, representing the age. reanimateWhen initialises at a random value between 5 and 10. On the turn when age increments above reanimateWhen, the InfectedCorpse is removed from the map and an instance of Zombie is created at that location.

When a human dies, we perform the check of whether a human or a zombie killed them simply by performing a check in AttackAction of the attacking actor’s undeadness Capability. This is simple to implement because this is already where the code for the *creation* of the corpse item is being handled.

**Farmers and Food**

We create an extension to Human, called Farmer. We add Crop as an extension of Ground.

We create a new set of Action, called SowAction, HarvestAction and FertiliseAction, each of which handle the changing and manipulation of the map around it accordingly, by changing the Ground at that Location. In order to facilitate the behaviour of the Farmer, we create a new class FarmingBehaviour.

FarmingBehaviour will handle the prioritisation of SowAction, HarvestAction and FertiliseAction for the farmer. The farmer will have FarmingBehaviour as a behaviour from which it can choose an action on any given turn. The player will have access to HarvestAction only through the allowableActions mechanism.

We can handle the *permissions for access to these actions* by creating a new Capability class, called CropCapability. This enum will have two modes, *HARVESTABLE and FERTILISABLE.* These correspond to the state a Crop is in and removes the need to create separate classes for ripe crops and unripe crops. A Crop is created with the *FERTILISABLE* capability. We allow a crop to only be fertilised once, and the *FERTILISABLE* capability is removed. When the crop matures, the displayChar is changed and it takes on the *HARVESTABLE* capability.

The presence of the CropCapability associated with Ground will dictate what actions are available to various Actors that can interact with Crop.

We create a new class called Food. This is an extension of PortableItem since food can be carried in the inventory. We create a new Action called EatFoodAction that restores HP to the Actor. We can code this action into a Human’s playTurn method in order to give them the condition to eat food (for example, only a Human that has lost HP can eat food).

In addition to the food mechanism required by the specifications, I added HuntGroundBehaviour and HuntItemBehaviour to facilitate the ability for AI to hunt down more than just Actors. If a Farmer does not perform a farming Action in a certain turn, instead of Wandering, they now move towards the nearest existing Crop within a five square radius so that crops tend to be grown next to existing crops. Humans will now hunt for Food and add it to their inventory if possible, instead of wandering. They will consume this food when they lose HP.

**Assignment 3**

**Mambo Marie**

We implement a new actor called MamboMarie that is a subclass of the Zombie Actor. We set MamboMarie with the UNDEAD capability as she is an enemy to Humans. Additionally, we give MamboMarie a private attribute called turnCount, which will keep count of how many turns she has played through. This will allow us to be selective in when she can play her abilities. If MamboMarie has no ability she can perform, then she wanders randomly using the WanderAction selected by the WanderBehaviour class.

**Chanting**

To implement the chanting ability, we create a new class called ChantAction which is a subclass of Action. In the constructor, we iterate through 5 Zombie names with an included number attached to the end of the name, and then create the new Zombies and store them into a list to be used in the execute method. In the execute method, we retrieve the boundaries of the current GameMap by calling the getXRange() and getYRange() methods in GameMap. We then iterate through the 5 Zombies created, and find a random Location to spawn each of the new Zombies at. We must also consider conditions where the Zombie should not be able to spawn at Locations that currently contain an Actor or if the Location cannot be entered by an Actor.

We want to allow MamboMarie to chant every 10 turns, and this is done within the playTurn() method of the MamboMarie Actor class where we keep track of the number of turns the Actor has gone through. After every 10 turns, we return the ChantAction.

This design was chosen because it is a simple way of adding new Zombies to the GameMap when the ChantAction is executed. This implementation demonstrates maintainability as newly implemented actors or current actors can use this ChantAction, meaning that it is not limited to just MamboMarie. Only one problem that arises from this implementation is that if the entire GameMap is filled with Actors or Locations that are not accessible, then the execute method will be running an infinite loop attempting to find Locations available to spawn the new Zombies at. However, this problem is highly unlikely to occur since the Player will be killing Zombies and MamboMarie is limited on when she can activate her ChantAction ability.

**Vanishing**

Initially, we considered making a VanishAction that makes MamboMarie invisible and teleport to the edge of the GameMap. However, this proved to have apparent issues where MamboMarie is still on the GameMap and if an Actor happens to reach her Location, they would be able to interact with her and be prevented from moving onto her current Location. Additionally, there are no current features that allow us to effectively make an Actor disappear from the GameMap while staying in the game.

To implement the vanishing ability, we create a new class called VanishAction which is a subclass of Action. To counter the problem above, we add a new StationaryItem called MamboMarieTracker. The MamboMarieTracker keeps track of the health of MamboMarie and the Location at which MamboMarie will spawn at when it is time to reappear on the GameMap. In the execute method of VanishAction, we retrieve the boundaries of the current GameMap by calling the getXRange() and getYRange() methods in GameMap. With some extra randomisation logic, we can create a MamboMarieTracker at a random Location at the edge of the GameMap and remove MamboMarie from the game. We must also consider conditions where the Location at which we add the MamboMarieTracker must not be occupied by an Actor or if an Actor cannot enter that Location.

We also want to allow MamboMarie to vanish after 30 turns of being on the GameMap. This is done within the playTurn() method of the MamboMarie Actor class where we keep track of the number of turns the Actor has gone through. When 30 turns have been played, we return the VanishAction.

We decided to go with this design because the VanishAction can simply add a new StationaryItem called MamboMarieTracker which keeps a new reference of MamboMarie in the Game with the previous MamboMarie’s stats. To simulate MamboMarie vanishing from the GameMap, we remove her from the GameMap and create a MamboMarieTracker at the edge of the Map with the same displayChar as the ground it is currently on. This makes the MamboMarieTracker invisible to the Player while also making it non-interactable as it is a StationaryItem.

A problem that arises from this design decision is that this MamboMarieTracker is a tracker only for MamboMarie. It cannot be used for any other Actor as it is associated with MamboMarie. This can be fixed by creating a universal tracker that allows any Actor to use in order to keep the Actor in the Game but not in the GameMap. However, for the purpose of this assignment, we are not required to do so.

**Appearing**

If MamboMarie is not currently on the map, then she has a 5% chance of appearing. To implement this logic, we utilise the override of the tick() method in MamboMarieTracker. We add some logic to the tick() method to determine whether MamboMarie can appear back onto the map. This is checked on every turn as the MamboMarieTracker is an item that remains on the map while being invisible. If Mambo Marie can appear back on the map, we check if an Actor is currently at the Location of the MamboMarieTracker and if not, then we simply create a new MamboMarie Actor at that Location and remove the MamboMarieTracker from the map.

**Mambo Marie Dies**

When MamboMarie is killed, she simply dies like a normal Zombie and will not appear on the map again.

**Ending The Game**

Initially, we considered creating methods within the Player class that checks if the Player has won or lost the game and do these checks within the PlayTurn() method. This proved to be problematic as the only way to prevent the Game from continuing was to remove the Player from the GameMap. Additionally, this implementation would mean that we cannot remove the “Game Over” message that would occur after our “Player Wins” or “Player Loses” message. Removing the Player deemed to be poor programming practise and so we decided to look into what we can do with the World class to end the Game.

In order to implement new endings to the game, we see that the World class holds a method called the stillRunning() which is protected. This is the only method we can use to end the game, so we decided to create a new class called WorldExtended which extends the World class and allows use to override this stillRunning() method. We implement two private methods checkIfPlayerWins() and checkIfPlayerLoses() to return whether the player has won or lost the game. In the stillRunning() method, we call the private methods to identify if the player has won or lost.

The decision to implement a new class called WorldExtended that extends the World class allows simple overriding of methods that are in World which demonstrates good use of the protected methods in World.

**Player Wins**

The checkIfPlayerWins() private method checks the entire GameMap to see if there are any hostile enemies that still exist.

If the player has won the game, it means that all hostile enemies are no longer on the compound GameMap. The stillRunning() method will return the False.

**Player Loses**

The checkIfPlayerLoses() private method checks the entire GameMap to see if there are no longer any Humans besides the Player alive.

If the player has lost the game, it means that either the Player was killed, or all Humans besides the Player has been killed in the compound GameMap. The stillRunning() method will return the False.

**Game Continues**

If both of these methods have not returned True, then the stillRunning() method will return True to continue to game.

**End Game Message**

To display the end game messages associated to winning and losing, we can override the endGameMessage() method in World and add some logic to return the end game message associated to whether the player has won or lost the game.

If the player wins, it will return “Player Wins”.

If the player loses, it will return “Player Loses”.

Otherwise, it will return “Game Over”.

**Quit Game**

We also implement a way for the player to quit the game whenever they like. We add a new class called QuitAction which is a subclass of Action and when the execute() method is used on QuitAction, it removes the player from the game and returns a message “Player quits the game”. By removing the player from the game, the original World class automatically identifies that the player is no longer on the GameMap and ends the game with the “Game Over” message.

**Sniper Rifle**

The SniperRifle is a weapon that rewards patience. When it is fired at a target, and the Actor using the SniperRifle has spent time aiming, then the amount of damage will be increased alongside the probability of hitting the target.

We create a subclass of the WeaponItem abstract class called RangedWeaponItem to allow the ability to use the RangedWeaponItems as melee Weapons too. Within the RangedWeaponItem class, we include private attributes that hold values for the rangedDamage and rangedVerb. These work similar to how they did for normal WeaponItems, but instead for RangedWeaponItems.

To implement a SniperRifle, we create a subclass of the RangedWeaponItem and construct a SniperRifle with the base damage for melee attacks and base rangedDamage for ranged attacks.

To implement a SniperRifle with variable damage and hit chance, we add a new concept called concentration. We give all Actors an attribute called concentration which can be used in various ways. In this case, we are using concentration to allow for better precision in aiming and also an increase in damage for the SniperRifle.

Initially, we had the idea of adding a charge counter to the SniperRifle which will count how many times it has been charged up or used for aiming. However, this brought upon problems where we could not account for when the Actor loses concentration on a target when the Actor is attacked or uses a different Action.

**Range Damage**

Since we have a RangedWeaponItem, it requires a new method to determine the output of the ranged attack. We use rangeDamage() as a method that determines what damage to output when an Actor shoots the SniperRifle. This uses the concept of the Actor’s concentration level which changes what the damage output will be.

We want to provide a SniperRifle that does base rangedDamage when the Actor has not aimed at the target in a previous turn. If the Actor has aimed once and then shoots the same target, then the rangedDamage is doubled. If the Actor has aimed twice and then shoots the same target, then the rangedDamage output is 1000 hit points. We know that a SniperRifle is a lethal Weapon and that an output of 1000 hit points is enough for instant killing the Actors currently found in the Game. When the Actor holding the SniperRifle is hurt or if they decide to execute an Action that is not aiming, then the Actor’s concentration will be reset to 0.

Additionally, we implement a chargeWeapon() method that increases the Actor’s concentration level by 1 and keeps track of the current target that the Actor is aiming at. So, whenever this chargeWeapon() method is called, it means that the Actor has chosen a target to aim at, thus increasing the concentration of the Actor by 1.

**Aim and Shoot**

When an Actor has a SniperRifle in their inventory, they have the ability to aim at a target or shoot a target. If there are no hostile Actors currently visible on the map, there will be no option for the Actor holding the SniperRifle to aim or shoot.

**Aim Sniper Action**

We implement a new Action called AimSniperAction, which involves creating the submenu for the Action. Instead of the execute method returning straight to the Player, it executes another Action called SelectTargetActorAction, which contains the main execution of Aiming and Shooting. When the Player selects aim and a target, the SelectTargetActorAction performs the aiming.

**Shoot Sniper Action**

Similarly, we implement a new Action called ShootSniperAction, which involves creating the submenu for the Action. Instead of the execute method returning straight to the Player, it executes another Action called SelectTargetActorAction. When the Player selects Shoot and a target, the SelectTargetActorAction performs the shooting.

**Selecting Different Targets**

In SelectTargetActorAction, if the Actor chooses a different target, then the concentration of the Actor is reset. This is done by keeping track of the target Actor who is stored as a private attribute called target in the SniperRifle class. This ensures that we can check who the previous target was and compare with the current target allowing us to increase concentration based on same target selections.

**Weapon Hit Probability**

Now, to determine the Weapon hit probability, we hardcoded the values into the SelectTargetActorAction class similar to how it worked in the AttackAction class. We used checks on the Actor’s concentration level and returned the respective Weapon hit chance.

Despite the SelectTargetActorAction containing repeated code from the AttackAction class, it was inevitable as we needed a way to differentiate the ranged attacks from the melee attacks. This decision was made to prevent excessive amounts of code in one class and also, the ranged Weapons were not compatible with the current code found in the AttackAction, meaning that a further refactorization of the entire AttackAction class and inherited classes would be required, where some classes are found within the Engine package and cannot be modified.

**Zombies Using Sniper Rifle**

When Zombies pick up a SniperRifle, they can only use the SniperRifle as a melee Weapon which has its own base damage, relatively lower than the ranged damage. This feature was already implemented as Weapons that are WeaponItems can be used as melee Weapons. So, by extending the WeaponItems, the RangedWeaponItems will have the ability to be used as melee Weapons too.

**Humans Using Sniper Rifle**

Additionally, the override of the tick() method in SniperRifle determines what allowableActions the Actor has when they have the SniperRifle. Since the Humans do not have any behaviours that allow them to shoot or use the SniperRifle, the allowableActions to aim and shoot are only limited to the Player.