‘Braaaains’ design rationale

This brief will contain in depth detail of the implementation behind our design and an explanation behind our decisions. It will focus on 5 categories of implementation: ‘Actors’, ‘Behaviours and Actions’, ‘Items’, ‘Grounds’ and ‘additionals’.

Actors

Actors would always re-use existing implementation to ensure a minimalistic and readable design. If we felt like the functionality of a specific actor would be able to be re used in other actors, we would create an abstract class to give access to their behaviours to other actors being implemented.

Farmers were inherited from Humans, farmers contained all the functionality of a human however they also contained an additional behaviour called FarmBehaviour. Farmers also have an attribute for harvesting, which will drop the fooditem they harvest on the ground.

An abstract class called RespawnableActor was made that would have two additional functions, one for determining if the actor is ready to appear and another function to perform putting the actor on the map, which would also control where the actor appears on the map. This would enable the use of Liskov substitution principle if the code were to be extended to have multiple types of respawnable actors, as they could all use the method from respawnableActor.

Mambo Marie was an actor that extended from RespawnableActor, Mambo Marie would have a tick attribute that would increment by 1 every turn when she is not vanished. Ticks would determine the action to perform on a given turn, every 10 turns she would call a SummonZombieAction and after 30 turns she would call a VanishAction and reset her tick counter to 0. Her behaviour on spawning was solely determined by chance, where we created a random number generator from 0-20 and would only let her back on the map if it landed on 0, giving her a 5% chance to respawn.

Behaviours and Actions

Our design of behaviours and actions would allow behaviours to use multiple types of actions. These actions would be related to behavior but more specific. This would allow behaviours to be reusable and extendible, as we could add more actions to the same behaviour if needed. This also keeps behaviours distinct from eachother despite being able to use the same actions, making their purpose clear whilst reusing actions between them.

We decided to create a BiteAction class which is used by AttackBehaviour. We did this because we felt Bite was a type of attack and it shared many similarities to the already existing class AttackAction. The probability of BiteAction being called vs AttackAction when a zombie called AttackBehaviour would depend on the number of arms a zombie had.

PickupWeaponBehaviour was a new class which inherited from Behaviour. We decided to make picking up items a behaviour as we felt many types of items could be selected to pick up. This would control the

PickUpItemAction by checking if there are any weapons at a given location and if there are, it will create a new PickUpItemAction with the item to be picked up as a parameter in its constructor.

Another class we made which inherited from Behaviour was SayBehaviour. SayBehaviour would allow an actor to use a turn to say something, currently we only made an action called SayBrainsAction which would allow an actor to say 'braiins', but more actions can be added onto it to allow the actor to say different things.

We only needed to count the number of arms/legs a zombie had, so we stored them as attributes which were part of Zombie. As zombies needed to be able to lose limbs, we created a function within zombie called dropLimb, which would be private and have a 25% chance to be called when the zombie has taken damage. In this function, there would be a 50% chance of either dropping a leg or an arm, but if an arm or a leg counter has reached 0, it will never be able to further decrement the counter. When a limb has fallen off, it will decrement the respective attribute counter and drop the zombie arm/leg onto the floor.

When a zombie drops an arm, there is a 50% chance that the weapon will be dropped if they originally had 2 arms, and a 100% chance if they had only 1 arm. This will be handled through another private function in zombie to encapsulate zombie's functionality from the rest of the code.

CraftingAction would be an Action which would allow an item to be turned into another item. If an item is craftable it will have the option to be crafted, if the player selects to craft an item during their turn it will call CraftingAction to destruct the old item object and construct the new item object that it crafts into as a replacement. CraftingAction would not need a Behaviour as it's only performed by the player.

WanderBehaviour is already an existing Behaviour being used by Zombie. We changed it so when zombies lose legs, it will change the rate of which WanderBehaviour can be called per turn. We kept this change in the Zombie class as we believed this was a unique behaviour for Zombies and we didn't want this change to be implemented for other Actors.

We created a private function called reanimateZombie within corpseitem which would check whether a tick counter has reached 8, when the condition has reached true, it will create a zombie object at the same location and deconstruct the class. We decided to not make any additional classes for this as we felt this was a much simpler approach to tackle the problem and this was specialized for corpseitem specifically.

Farmers will be a class that extends from Human, but also our final behaviour we will be adding FarmBehaviour. The FarmBehaviour class contains all actions of sowing, fertilizing and harvesting. We saw these three actions as relatable actions, as only one of these actions could be performed by a farmer per turn depending on the ground farmers were standing on. The FarmBehaviour class would allow the Actor to perform the appropriate Action according to the type of ground they were standing on or are surrounded by.

FarmBehaviour has a 1/3 chance of calling the class SowAction if the Actor calling the behaviour has the same location on the map as a dirt block. Farming is something we saw as needing its own behaviour as its unique compared to other behaviours. This would serve as the base to use a variety of other actions. SowAction will replace the dirt block with an unripe crop. We replaced objects by destructing old ones and constructing new ones to replace them with.

FarmBehaviour will also call the class FertilizeAction if the Actor is in the same location as a new class extending from Ground called UnripeCrop. FertilizeAction will insure the UnripeCrop in the same location as the Actor on the map will ripen in 10 less turns or ripen immediately if it was expected to ripen in less than 10 turns.

Lastly, FarmBehaviour will be able to call the class HarvestAction if the Actor is in the same location or are next to the new class extending from Ground called RipeCrop, they may harvest the crop turning the Ground from RipeCrop to dirt by destructing RipeCrop and constructing a new Dirt. Additionally, if a farmer calls HarvestAction, they will drop the FoodItem at their location. If the player harvests the crop the FoodItem will be added to their inventory.

Players can pick up a specific item using PickupAction. We added behaviours which will control this action in certain ways to allow actors to choose which types of items to pick up. For example, we made a PickupFoodBehaviour class which allows the actor to pick up a FoodItem instead of any item, we also made a PickupWeaponBehaviour to allow the actor to pick up a WeaponItem. The behavior would decide which item the actor should pick up, and it would then call the PickupAction on that item to ensure it picked up. We decided to keep one action and increase the number of behaviours as we felt this was a decision making process by the actor, and would create extendibility issues in the future if we made multiple types of pickupactions, when you can maintain the code with only one action instead.

DriveAction would move the actor performing the action to a specified location in 1 turn. When a DriveAction is constructed, the location of which the actor will travel to has to be given in the parameter to ensure the actor has somewhere to travel to when performing the execution of the action. So far driveaction can only be performed when next to a vehicle however this action can be implemented in various other places.

SummonZombieBehaviour would use SummonZombieAction in a specific way. Upon making a SummonZombieBehaviour, you would be able to specify the number of zombies you'd like to summon when getting the action. This behavior would tell the SummonZombieAction class to return a class containing an execute method to summon the specified number of zombies. We felt the behaviour gives more room for flexibility with how to extend on the code, but we acknowledge this behaviour isn't necessary as you can just call the action directly with the specified amount as of now (instead of using the behaviour as a middle ground).

To allow actors such as Mambo Marie to temporarily leave the game, we created another action called VanishAction. VanishAction would simply perform the removeactor function on the map the actor was on to remove itself. We felt this was a very simple action that did not need a decision-making process (you either want to vanish or you don't).

One of the few functions is added for the new gun weapons is the ShotgunAction that will be used for the shotgun item. This action takes a direction parameter to determine which direction to shoot in. It will shoot the specified direction with a range of 3 blocks wide and also diagonally. This action will damage any actors that is in the range of fire and has a 75% chance of hitting for each actor that is in the fire range.

For the SniperRifle a few more action classes are needed since when the player uses the sniper he or she has to choice to aim for a few turns before shooting or just shoot immediately. One of the functions that

is added for the sniper is SniperAction, this action will create a submenu which allows the player to select the target to shoot. In the submenu a new action is added the ChooseSniperTarget, this function takes an Actor as the parameter and sets the target for player in the Player class. After the player, chooses the target, the player will be prompted with 2 more new functions. One of which, is the ContinueAim function, where this will increase the aimCount in the Player class. The higher the AimCount in the Player class the higher the accuracy will be when the player fires the sniper. After aiming the Player will have the option to continue aiming for an extra turn or to fire now. Which leads to another action the FireNowAction. This action will look at the aimCount of the player, and will adjust the accuracy according to the aimCount. If the player does anything other than ContinueAim, they will lose concentration therefore resetting the aimCount to 0.

The reload action works for both the sniper and shotgun. It will check the amount of bullets in the gun and the ammo magazine. This action will attempt to fill the gun to maximum capacity. If there is not enough bullets in the magazine, then it will put all the bullets from the magazine into the gun and the empty magazine will be deleted from the inventory.

The last action is QuitGameAction where the player can stop the game anytime they want. All the action does it sets the endGameStatus to true in the Player class. The world class will then check this variable every turn and break the loop if the endGameStatus is set to True.

Items

A new abstract class was made called CraftableWeaponItem which extends from WeaponItem. CraftableWeaponItem ensures that any item that's both craftable and a weapon will have a method in it that will return its crafted variant. This was possible by making abstract methods within the CraftableWeaponItem class, ensuring that all children of the class contain specific types of functions.

When zombies drop an arm, a zombie arm will drop as a ZombieArm object that extends from CraftableWeaponItem as you can attack with it, however it will also be a craftable item. The zombie arm will be able to be crafted into a Club which is a child of WeaponItem that does a significant amount of damage. We found it to be appropriate to create a class extending from CraftableWeaponItem as it would increase reusability to make ZombieArm its own object. Otherwise you would need to specify all the parameters of what a ZombieArm is upon constructing each zombie arm in craftableweaponitem.

Zombies would also drop a ZombieLeg which is a child of CraftableWeaponItem and behaves similarly to the zombie arm. The difference is the zombie leg can be crafted into a zombie mace instead which is a WeaponItem object that does even more damage than a zombie club.

Another type of weapon are the guns. There is the shotgun and sniper where the it extends WeaponItem. These two classes are quite similar, they both have an attribute which keeps track of the amount of ammo that is left in the gun. There is also the method in the that will be used to reduce the ammo by 1 each time the gun is shot.

Currently, humans drop a PortableItem on death at their location. We decided to create a new class called CorpseItem which would extend from PortableItem but also contain an attribute with an integer value of 8. A tick method in Item would be called after each turn, incrementing the counter by 1. When the counter reaches this attribute value of 8 it will call the method reanimateZombie, a private method within its own class.

Our last item was FoodItem, this class is a child class of PortableItem however it also has an attribute containing the amount of health that food will heal. We extended this from PortableItem as food can be carried by players.

Grounds

An additional enum was added called FarmerCapabilities. This was to make it easy to keep track of what actions could be performed on specific ground objects. We had 3 different capabilities (DIRT, UNRIPE, RIPE) which indicate all the allowable actions to be performed on these three specific types of ground.

We added an additional object which extended from the Ground class. Crop would contain all methods and attributes of Ground, however it would also contain an attribute of the int 20 signaling when its ready to ripen. Upon each turn, Crop will call tick and when its counter reaches 20, it will replace its capabilities with ripe crop and set a Boolean to true to signal that it has ripened. Crops that are not ripened will allow farmers to fertilize it, increasing its counter by 10 immediately. Crops that are ripened will allow farmers to harvest it, replacing the crop with dirt and constructing a new FoodItem at the location.

We also added a vehicle which extended from ground but would also store a location upon being created using dependency injection. Dependency injection allowed us to minimize the dependencies Vehicle had that the class it inherited from did not have, whilst reusing the code from the parent class. Vehicle has allowable actions which allow the actor to interact with this object. When an actor is near the vehicle, they can perform a DriveAction.

As well as this, we added additional grounds not specified in the assignment doc for aesthetic purposes of town. We added Brick which would not allow the actor to enter it, but also be represented by the '=' character for visual aid. Tile would contain a different characteristic compared to the other ground type objects. Tile would allow the actor to enter and walk in, however you would not be able to perform any farming actions within it.

Additionals

We felt the best way to store vanished actors whilst not changing the existing engine behind the code was to override world, this was using the open closed principle as instead of modifying the original world class, we were extending from it. This would ultimately give us control over the world, which would allow us to add an additional attribute keeping track of the actors that were vanished in an arraylist. The reason we wanted to inherit from the world class was because actors were not bound to a map, actors may be able to vanish across maps. In the case of Mambo Marie, should would always appear on the same map as the player. We added an additional function called addinvisibleactor as we needed to store invisible RespawnableActors separately from the rest of the actors to know which actors to check for whether they should respawn. We used the invisibleactors to check whether they should respawn at the start of every iteration of the main game loop in the run function, and if the

shouldspawn action within the respawnableactor returned true we would place it on the map of the player.

We used the interface segregation principle to add default methods to the interface that abstract classes in the engine used, and because of this we didn't need to change the code within engine. This allowed us to avoid down casting which increased the codes readability as well as its reusability as we would not need to check the classes object type in any situation, we could instead implement the default methods from the interface within our own classes.