FIT2099 Assignment 3 Design Rationale

ZombieWorld class

ZombieWorld class inherits from World class. The new class we created here is ZombieWorld class. WorldRun class inheriting from World class means that the methods in World can be used by the ZombieWorld class. In Application code, we called world.run(world is an instance of ZombieWorld class) meaning that we are using the method run to run the world. The role and responsibility of the ZombieWorld class is to implement the features for ending the game and runs the game as intended, where we override the run method to check if the player was killed or all other humans in the compound are killed which means the player loses. It also checks if all zombies and Mambo Marie are killed in the compound so that the player wins. By using inheritance, we apply the principle of "DRY" and by making ZombieWorld class a subclass of World class, it allows us to avoid code duplication as we realized they both share the same methods.

ExitGameAction class

ExitGameAction class inherits from Action class. The new class we created here is the ExitGameAction class. ExitGameAction class inherits from Action class means the methods in Actions can be implemented in ExitGameAction class. In the spec, it is stated that we must add "A "quit game" option in the menu". To achieve this we made it an action to be displayed on the menu. The role and responsibility of the ExitGameAction class is to allow users to choose whether they want to quit the game they are playing anytime, the action shows up as an option in the menu and the player inputs the choice to quit and the map removes the actor from the map causing the game to end. By using inheritance, we apply the principle of "DRY" and by making ExitGameAction class a subclass of Action class, it allows us to avoid code duplication as we realized they both share the same methods.

Ammunition class

Ammunition class inherits from PortableItem class. Ammunition is placed in town map and compound map. Player can pick up the ammunition. Player can only use the shotgun and sniper rifle if they have ammunition.

Shotgun class

Shotgun class inherits from WeaponItem class because shotgun is a weapon. Its default damage value is 10, player or other actor can wield it to attack other actors.

SniperRifle class

SniperRifle class inherits from WeaponItem class. Its default damage value is 10, player or other actor can wield it to attack other actors.

ShotAction class

ShotAction class inherits from Action class. It allows the player to attack all the zombies in a specific direction. It asks the user to input a direction, there is a while loop to ensure the direction is north, northeast, east, southeast, south, southwest, west or northwest. If the player inputs an invalid direction, it will ask the player to input again and again. After getting the input, if Math.random <= 0.75, call the fire method depending on the direction. For example, if the user input north, and Math.random <= 0.75, it will call the firetNorth method, in fireNorth method, use loops to get locations that the player can shoot. In every location, check whether it contains an actor, if yes, call the fire method. In fire method, create a new AttackAction and set the damage to 50. At the end, remove one ammunition from the actor's (player's) inventory.

SnipeAction Class

SnipeAction class inherits from Action class, it allows the player to either snipe a target or aim. In the while loop, ask the user to input the name of the target, use the for loop to go through the whole map to check whether the target is in the map, if not, ask again until the input target is valid. Record the location of the target. In the while loop, ask the user to input their order, input 1 to fire the target, check the value of aimRound(the value of aimRound shows whether the player has aim and how many rounds did the player aim), if the value of aimRound <= 2, input 2 to aim for one round. If user input 1,if Math.random <= probabilityOfDamage, create a new AttackAction (if aimRound == 0, probabilityOfDamage = 0.75, if aimRound == 1, probabilityOfDamage == 0.9, if aimRound == 2, probabilityOfDamage == 1). Set the damage in attackAction to “damage”(the value of damage will change depends on the value of aimRound, if aimRound == 0, damage = 40, if aimRound == 1, damage = 80, if aimRund == 2, damage = 100). Call attackAction.execute, then remove one ammunition from the player's inventory. However, if player input2 chooses to aim for one turn, it creates a new AimAction.

AimAction class

AimAction class inherits from Action class, it increases the value of aimRound. The value of aimRound will make the value of damage and probabilityOfDamage different in SnipeAction class.

MamboMarie class

MamboMarie class inherits from ZombieActor class. It has an attribute turnForChant, its default value is 0. Another attribute is numberOfChant, its default value is 0. Mambo Marie has a wander in its action, it wonders in every turn. MamboMarie class has an attribute, turnForChant, its default value is 0, in every turn, it increases by one. If turnForChant == 10, set it back to 0 and create a new chantAction, set numberOfZombies to 5*numberOfChant (numberOfChant records how many times Mambo Marie has chanted). To set numberOfZombies to 5*numberOfChant is because when creating the new zombie in chantAction, zombies' name can generate uniquely.

ChantAction Class

ChantAction class inherits from Action class, it creates zombies. Use a while loop to create the zombie five times, find a random location, if the location does not contain an actor, create a zombie on the location. The zombie name generated depends on numberOfZombie (as mentioned in MamboMarie class), zombie name is "zombie" + numberOfZombie. For example, if numbeOfZombie is 0, the name of the new zombie is "zombie0". Once a zombie is created, numberOfZombie += 1. The reason for doing that is to ensure that every zombie has a unique name, when the player wants to snipe and the input of the target's name would not crash.

MamboMarie in player class

In player class, there is an attribute, momboMarieHitPoints, its default value is 50. Create a new momboMarie in player class hitPoints is momboMarieHitPoints. In playTurn method, check if the map contains a mamboMarie, if no, if Math.random <= 0.05, add momboMarie into the map. If Math.random <= 0.25, place the mamboMarie at the top of the map (random x, max y), if Math.random <= 0.5, place the mamboMarie at the right most of the map (max x, random y)… If the map contains a mamboMarie, turnForMamboMarieVanish records how many turns the Mambo Marie is in the map, if turnForMamboMarieVanish == 30, remove the mamboMarie from the map.

Extra features

Animal class

The new classes added here are Animal class. Animal class is declared as an abstract class so that it could be used as a base for its subclasses (Cow, Chicken, Pig). The Animal class inherits from Actor class (Actor is the parent class and Animal is the child class). The role and responsibility of the Animal class are to act as a blueprint for us to create animals which will prevent code duplication when creating animals. An example would be the saySomething(Actor actor, String word) method in Animal class where it allows its child classes to implement it to have a probability of 60% to say something like the Cow: “Moo”, Pig: “Oink”, Chicken: “Cluck”. In the AnimalConstructor, added AnimalCapability.Killable which will make all animals which inherits from Animal class to possess that capability. By using inheritance and abstract class concepts, we apply the principle of “Don’t Repeat Yourself” and avoid code duplication. As we realized that the child classes share the same methods as their parent class.

AnimalCapability, ChickenCapability, PigCapability and CowCapability class

The new classes added here are AnimalCapability, ChickenCapability, PigCapability and CowCapability class. All these classes are enum classes which serve great importance to the functionality to the animals created. In AnimalCapability class, KILLABLE is to allow us to identify if an animal can be killed and this capability would be added in the Animal class constructor so that all animals which inherit from animal class would have this capability which allows us to reduce code duplication. In ChickenCapability class, CLUCK is to allow us to identify if we are killing a chicken, which will later be used in AttackAction class to check if we are killing a chicken and it should drop a Drumstick if a chicken was killed. In PigCapability class, OINK is to allow us to identify if we are killing a pig, which will later be used in AttackAction class to check if we are killing a pig and it should drop a pork if a pig was killed. In CowCapability class, MOO is to allow us to identify if we are killing a cow, which will later be used in AttackAction class to check if we are killing a cow and it should drop a steak if a cow was killed.

Cow, Pig, Chicken class

The new classes added here are Cow, Pig and Chicken class. All 3 classes inherit from Animal class. The role and responsibility of the Cow, Pig and Chicken class is to allow us to place them on the game map where they are allowed to wander around the map. For recognition purposes, display characters ‘W’, ‘P’, ‘N’ were given to Cow, Pig and Chicken respectively. In Cow class, cows were given CowCapability.MOO, this would later be used in attack action to help us identify whether we are attacking a cow. In Pig class, pigs were given PigCapability.OINK, this would later be used in attack action to help us identify whether we are attacking a pig. In Chicken class, chickens were given ChickenCapability.CLUCK, this would later be used in attack action to help us identify whether we are attacking a chicken. By inheriting from Animal class, we apply the principle of “Don’t Repeat Yourself” and avoid code duplication. As we realized that the child classes share the same methods as their parent class.

Pork, Steak and Drumstick class

The new classes added here are Pork, Steak and Drumstick class. All 3 classes inherit from the Food class. The role and responsibility of these 3 classes is to allow food to be dropped when an animal is killed and the player can pick the food up and eat it to get health points. For recognition purposes, display characters ‘}’, ‘/’, ‘{‘ were given to Pork, Steak and Drumstick respectively. By inheriting from Food class, we apply the principle of “Don’t Repeat Yourself” and avoid code duplication. As we realized that the child classes share the same methods as their parent class.

HostileMob and HostileMobCapability class

The new classes added here are HostileMob classes. HostileMob class is declared as an abstract class so that it could be used as a base for it's subclasses(Creeper). HostileMob class inherits from Actor class (Actor is the parent class and HostileMob is the child class). The role and responsibility of the HostileMob class is to act as a blueprint for use to create mobs which are dangerous which will prevent code duplication when creating them. The role and responsibility of HostileMobCapability class is to make hostile mobs killable, this would later be used when the player needs to detect whether Creeper is killable or not. It is given in the constructor of HostileMob so that the child classes of HostileMob would inherit that capability thus decreasing code duplication. By using inheritance, we apply the principle of "Don't Repeat Yourself" and avoid code duplication. As we realized that the child classes share the same methods as their parent class.

Creeper and CreeperCapability class

The new classes added here are Creeper and CreeperCapability class. Creeper class inherits from HostileMob class. The role and responsibility of the Creeper class are to allow us to create creepers and place them on the map where they will wander around unless they detect a living thing (zombies included) in their blast radius where they have a 50% chance of exploding. For recognition purposes, the display character '!' was given to Creeper. As the Creeper wanders around the map it would constantly say "Imma Blow!!!!!" to remind the player it is dangerous to go near it. The role and responsibility of CreeperCapability are to allow the creeper to be identified by it is not utilized in our program, however it would be useful in the future when we want to expand the abilities of the Creeper. By using inheritance, we apply the principle of “Don’t Repeat Yourself” and avoid code duplication. As we realized that the child classes share the same methods as their parent class.

ExplodeAction and ExplodeBehaviour class

The new classes added here would be ExplodeAction and ExplodeBehaviour. ExplodeAction class inherits from Action class and ExplodeBehaviour inherits from Behaviour interface so that it becomes a factory to create exploding actions. The role and responsibility of the ExplodeAction class is to allow Creeper to explode as that is the main feature about the Creeper. By using inheritance, we apply the principle of “Don’t Repeat Yourself” and avoid code duplication. As we realized that the child classes share the same methods as their parent class.

DestroyCropAction and DestroyBehaviour class

The new classes added here would be DestroyCropAction and DestroyBehaviour. DestroyCropAction class inherits from Action class and DestroyBehaviour inherits from Behaviour interface so that it becomes a factory to create destroy actions. The role and responsibility of the DestroyCropAction class is to allow actors to destroy crops they are standing on (this case is zombies). By using inheritance, we apply the principle of “Don’t Repeat Yourself” and avoid code duplication. As we realized that the child classes share the same methods as their parent class.

Player increase health points if kill a zombie

In AttackAction class, execute method, if the target is killed, check if the actor is a player, if yes, call heal method to add health points for the player.

Stuff from assignment 2

Crop class and CropCapability class

Crop class inherits from Item class. CropCapability class is an enum class which contains UNRIPE and RIPE. The new classes we have created here are Crop class (child class of Item class) and CropCapability class. The role and responsibility of the Crop class is to allow Farmer to sow crops in dirt. The role and responsibility of the CropCapability class is to give crops a status whether it is UNRIPE or RIPE so that human bots, Player and farmers know when it is suitable to harvest the crop. In Crop class, there is a private integer attribute called age. It is to allow us to track the growth of the Crop so that at a certain age value, it grows and changes the display character (displayChar) of the Crop object. For recognition purposes, we can override the “tick” method in the Item class by having displayChar: ‘o’, ‘c’, ‘C’ to represent the crop at different ages and show on the UI if it was ripe enough to be harvested. By making Crop class inherit from Item class, it allows us to implement methods in the Item class to Crop class. By using inheritance, we apply the principle of “Don’t Repeat Yourself” and avoid code duplication as we realized Crop class and Item class would share the same methods.

Food class and CropFood class

Food class inherits from PortableItem class. CropFood class inherits from Food class. The new classes we have created here are Food class (child class of PortableItem class and parent class of CropFood class) and CropFood class (child class of Food class). The Food class here is declared as abstract because in the future we might need to create more types of food and those foods might share the same methods as Food class. The role and responsibility of the Food class is to allow us to create more types of food from it. The role and responsibility of the CropFood class is to allow us to replace crops harvested to become a food which can be carried around, picked up and dropped. By using inheritance, we apply the principle of “Don’t Repeat Yourself” and by making Food class abstract, we can avoid code duplication as we realized future food types would share the same methods as Food class.

Human class and HumanCapability class

The new class we have created here is HumanCapability class. HumanCapability class is an enum class which contains HARVEST and EAT. The role and responsibility of the HumanCapability class is to allow us to give Humans HARVEST and EAT capability. This allows us to classify what humans can do when they harvest crops (we can differentiate what Human harvest can do and Farmer harvest can do) and eat food.

Farmer class and FarmerCapability class

Farmer class inherits from Human class as it shares the same characteristics and abilities as a Human. FarmerCapability class is an enum class which contains Harvest. The new classes we added here are Farmer class (child class of Human class) and FarmerCapability class. The roles and responsibilities of the Farmer class is to allow us to randomly create Farmers in the GameMap and allow it to roam around the map to sow crops, fertilize crops and harvest the crop. In the Farmer class, there is a private Random attribute called random for the Farmer to have a 33% probability of sowing a crop when standing next to a patch of dirt. For recognition purposes, we can provide the Farmer object a name, displayChar of ‘F’ and hit points of 100 in the super() from Human class. The role and responsibilities of the FarmerCapability class is to allow us to classify what Farmer can do when they harvest crops. By inheriting Farmer class from Human class it allows us to implement Human class behaviours such as WanderBehaviour which allows the Farmer to roam around the game map and methods from Humans parent class. By using inheritance, we apply the principle of

“Don’t Repeat Yourself” and avoid code duplication as we realized Farmer class would share the same methods as their Human class.

EatAction class

The EatAction class inherits from the Action class. The new class we have created here is EatAction class (child class of Action class). The role and responsibilities of the EatAction class is to allow human bots and Player to eat crops they harvested or pick up CropFood on the ground. In the EatAction class there is a private Food attribute called food, to specify which food to eat. By inheriting EatAction class from Action class, it allows us to implement the “execute” method in Action class to let the Actor have an eat functionality. By using inheritance, we apply the principle of “Don’t Repeat Yourself” and avoid code duplication as we realized the EatAction class would share the same methods as the Action class.

HarvestAction class

The HarvestAction class inherits from the Action class. The new class we have created here is HarvestAction class (child class of Action class). The role and responsibilities of the HarvestAction class is to allow human bots, Player and Farmer to harvest RIPE crops. We can use their respective capabilities to specify what they can do when they harvest the crops. In the HarvestAction class there is a private Location attribute called target and a private Item attribute item to specify what item at which location we want to harvest. By inheriting HarvestAction class from Action class, it allows us to implement the “execute” method in Action class to let the Actor have an eat functionality. By using inheritance, we apply the principle of “Don’t Repeat Yourself” and avoid code duplication as we realized the HarvestAction class would share the same methods as the Action class.

FarmingBehaviour, SowAction and FertilizeAction class

The new classes added here would be FarmingBehaviour, SowAction and FertiliseAction class. SowAction and FertiliseAction class inherit from Action class. All these classes are grouped together by the FarmingBehaviour class. The FarmingBehaviour class inherits from the Behaviour interface so that it becomes a factory for creating actions. The FarmingBehaviour class has a dependency with Location class as the actions which are grouped together by FarmingBehaviour needs to know the location of the actor to perform the action. The role and responsibility of FarmingBehaviour class is to allow Farmer class to utilise the Actions in FarmingBehaviour class. This is done by having an association between Farmer class and FarmingBehaviour class. We decided to put SowAction and FertilizeAction class under Farming Behaviour class so that there is less duplicate code. The role and responsibility of SowAction class is to allow Farmer to sow a crop on dirt. It has a private Location attribute called target, so that SowAction knows which location to sow a crop. The role and responsibility of FertilizeAction class is to allow Farmer to fertilize the crop to reduce ripening time required for the crop. It has a private Item attribute called target, so that FertiliseAction knows which item to fertilize. By using inheritance, we apply the principle of “Don’t Repeat Yourself” and avoid code duplication. As we realized that the child classes share the same methods as their parent class.