***\*\*Blue texts are the new implementations for Assignment 2\*\****

***\*\*Green texts are the new implementations for Assignment 3\*\****

**Beating up the Zombies**

**Limb**

It is an Interface class which Arm and Leg class implements this interface. Each *Actor* contains an *ArrayList<Limb>* with a length of 4 (2 of them are from *Leg* class and the other 2 are from *Arm* class).

**Advantages:** When actor limbs are added to an ArrayList the type of the ArrayList can be limbs instead of Item. Therefore, the Limb class applies the rule of DRY (don’t repeat yourself).

**AttackAction**

When an Actor object(as a target gets attacked) type is Zombie Object with a possibility of 25% one of the zombies limbs will be dropped. Also, when a target dies using an if statement (instanceof syntax) to get the type of the Actor who died. If the actor is a human it assigns the corpse to PortableItem which means after 5 turns, that corpse will rise from the ground as a Zombie object. And if the Zombie object dies, the corpse variable will be assigned as a PortableItemZombie object which means that the zombie is dead till the end of the game. Furthermore, in this class another if statement is used when the zombie attack is successful and if the zombie object uses a bite attack, the zombie object will be healed for 5 hit points.

**Advantages**: This class is modified because it is the only class which shows the changes where an actor attacks another actor. And it can be modified to create changes in the GameMap where an actor dies.(rise of dead humans from the ground and change that type to zombies.)

**Zombie:**

1. In the getIntrinsicWeapon() method, bite attack has been added with a probability of 50%. For better implementation, since if a zombie loses its arm the chance of punching will be ⅓ and chance of biting will be ⅔, therefore the probability changes and because of that, the value of probability for punching will be assigned to a variable which makes it easier for us to change that value.
2. When a player attacks a zombie by a chance of 25% a limb of a zombie drops, with respect to that limb the zombie behaviour changes which they operate in each turn. For instance, if a leg drops, the speed halves. Or if two legs drop the zombie cannot move. For implementing these scenarios, if/elif statements can be used to state what change in behaviour can happen to the zombie (inside of playTurn() method in zombie class).
3. Override hurt method is created for the zombies. Which drops one of the zombie limbs from items array list when non-zombie actor attacks a zombie actor. The hit point of the zombie will be reduced as well
4. If the zombie has one arm, the chance of punching will be 25% and if the zombie has no arms, the chance of punching will be 0%.

**HuntBehaviour:**

In this behaviour zombies will follow one of the non-zombie if they are in some certain distance to the zombie. This behaviour happens when the zombie has two legs or (it has one leg and the zombie is playing its turn in even turns) (because it is the condition which a zombie can move)

**WanderBehaviour:**

This behaviour is for human and zombies which they can wander around the location which they currently standing on. This behaviour happens when the zombie has two legs or (it has one leg and the zombie is playing its turn in even turns) (because it is the condition which a zombie can move)

**ZombieActor:**

In the ZombieActor class, variable of “turn” is declared for all of the actors which should be incremented in the playTurn method in actor’s class. Also, since currently all of the actors have 2 arms and 2 legs, the base ArrayList of Limbs is created in the ZombieActor which is known as ‘items’ and from this design all of the actors can have 2 arms and 2 legs.

Also, the ZombieActor class contains the AllowableAction() method which provides actions for the player such as eatFood, harvestFood, craftItem etc. which helps the system to implement DRY since the player use the super.AllowableAction to access all of the actions which the player needs to perform.

**PickUpBehaviour:**

This class is using the PickUpItemAction object to let zombies pick up an Item if that item on the ground can be counted as a weapon (which is in zombie’s current location). The zombie limbs can also be picked up by zombies (if the player at least has one arm).

**Crafting weapons**

**Leg**

Leg Class extends WeaponItem functionality (therefore it’s a kind of WeaponItem) it also implements Limbs method. it represents the Leg of a zombie object which they can be dropped if zombie objects get attacked (with the possibility of 25%) also, Leg class has craft method which states which item is crafted instead of leg(which that item is Mace).

**Arm**

Arm Class extends WeaponItem functionality (therefore it’s a kind of WeaponItem) it also implements Limbs method. it represents the Arm of a zombie object which they can be dropped if zombie objects get attacked (with the possibility of 25%) also, Arm class has craft method which states which item is crafted instead of arm(which that Item is Club).

the player can use the Leg as a Club() which itself is a weapon Item. it has more damage than the simple Leg and Arm which is used as Weapons.

**Advantages:** Arms and Legs got separate classes since they used as different weapons when a player craft them (Leg is crafted to the Mace and Arm is crafted to the Club)

**Club**

This class has extended WeaponItem which has the same type of the Arm which the player might have in its inventory.

It is a class in which a player can craft an Arm weapon to the Club object (which is a weapon item too) which has greater damage rather than Arm.

**Mace**

This class has extended WeaponItem which has the same type of the Leg object which the player might have in its inventory.

It is a class which a player can craft an Arm weapon to the Club object (which is a weapon item too) which has greater damage rather than Leg

**CraftAction:**

CraftAction is a subclass of Action class which shows that if a player has Leg in its inventory, the Leg can be crafted as a Mace weapon and if the player has Arm in its inventory, that Arm can be crafted and used as a Club.

The Craft action execute method, checks for every item in the Actor’s inventory if the Item is craftable, calls the “craft” override method for that Item.

**Advantage1:**

The Override craft function has been used instead of instance of statement, which makes the design better and makes functions and classes more cohesive.

**RangedWeapon:**

RangedWeapon is an abstract class that extends *WeaponItem* class. Has abstract methods for

**ReloadAction:**

ReloadAction class extends *Action* class.

**Shotgun:**

Shotgun class extends *RangedWeapon* abstract class.

**ShotgunAimMenuAction:**

This *Action* class brings up a sub-menu for *Player*  to choose which direction to perform the *ShotgunShootAction* in.

**ShotgunAmmo:**

ShotgunAmmo class extends *Ammo* class. Can only be used in a Shotgun.

**ShotgunShootAction:**

This *Action* class gets all the possible exits (direction) of the Actor to determine which directions to display in the sub-menu.

**SniperRifle:**

SniperRifle class extends *RangedWeapon* abstract class.

**SniperRifleAimAction:**

This *Action* class brings up a sub-menu for *Player* to choose which actor to aim at.

**SniperRifleAimMenuAction:**

This *Action* class brings up a sub-menu for *Player* to choose which actor to perform the *SniperRifleAimAction* on.

**SniperRifleAmmo:**

*SniperRifleAmmo* class extends *Ammo* class. Can only be used in a SniperRifle.

**SniperRifleShootAction:**

This *Action* class gets all the actors on the *gameMap* and brings up a sub-menu for *Player* to choose which actor to snipe.

**SniperRifleShootMenuAction:**

This *Action* class does damage (depending on the aim) to the selected target.

**Ammo:**

Extends Item class from the engine package.

**Advantage:** Allows further implementation for different kinds of ammo for different types of ranged weapons.

**\*\*All shoot/aim and submenu actions\*\***

**Advantage:** For other ranged weapons (further implementations), classes for shooting can be created like how it was created for sniper and shotgun.

**Rising from the dead**

**PortableItem**

If a Human object dies after 5 to 10 turns the human corpse will be changed to a Zombie object with Zombie behaviour. Since all of the actors who is killed by the zombie rise from the dead, their ZombieCapability will be ZombieCapability.Alive. and this idea has been used to make the difference between non-zombie actors consequence after death and zombie object actors consequence

**Advantages:** This method is useful when it is shown that the dead human can rise from the ground as a zombie object.

**DropAction**

In drop action the actor will drop the weapon which it is holding in an adjacent location to the actor. And this design decision is made for actors to not to pick up the dropped item in the next turn. So they can perform other actions such as attack action as well.

**DropBehaviour**

This behaviour is for zombies and it states if the zombie has one arms, it has the chance of 50% to drop the weapon which it is holding (in each turn) but if the zombie doesn’t have any arms, the zombie should drop its weapon in every turn.

**ZombieExpressionAction**

it is an ability for the zombie to say zombie expression such as: “Brainnnnnnns”

in each turn

**ZombieExpressionBehaviour**

this behaviour will control the ZombieExpressionAction to be occurred in 10% of the turns.

**Farmers and food**

**Farmer**

Extends the *Human* class. *Farmer* objects have a *HarvestBehaviour* object that allows the *Farmer* to perform the *HarvestAction*.

**Advantages:** *Farmer* class can be extended further into different types of farmer sowing different types of *UnripeCrop* which heals for different amounts.

**HarvestBehaviour**

Implements the *Behaviour* interface. Only *Farmer* objects (*Farmer* objects are type *Human too*) have a *HarvestBehaviour* object.

**HarvestAction**

Extends the *Action* class. *HarvestAction* allows a *Farmer* object to harvest a *RipeCrop* and instantiates a *Food* object and adds it into its inventory (if the object that is performing the action is a *Player* object) or drops it on the ground (if the object that is performing the action is a *Farmer* object).

**Advantages:** When the *HarvestAction* object is added into *ripen crop*’s *allowableAction* field, the option for the *Player* to harvest a *RipeCrop* will be added into the menu.

**HarvestAction**

It is no longer added into the *Food*’s *allowableAction* field. It is now created in *addHarvestAction(Actions actions, Actor actor, GameMap map)* method in *ActorInterface* and called in the *Player* class to give the *Player* the menu option to harvest ripe crop around them.

**FertilizeBehaviour**

Implements the *Behaviour* interface. Only *Farmer* objects have a *FertilizeBehaviour* object. It allows a *Farmer* to perform *FertilizeAction* that fertilizes an Unripe Cropthat the *Farmer* is standing on.

**FertilizeAction**

Extends the *Action* class. *FertilizeAction* changes the age of the *UnripeCrop* object to decrease the time for it to ripe, given that there is an *UnripeCrop* object that the *Farmer* object is standing on.

**Advantages:** When the FertilizeAction object is added into *Player*’s *allowableAction* field, the option for the *Player* to fertilize an *UnripeCrop* will be added into the menu.

**SowBehaviour**

Implements the *Behaviour* interface. Only *Farmer* objects have a *SowBehaviour* object. It allows a *Farmer* to perform *SowAction* that sow an *UnripeCrop* on a patch of dirt, given that there is a patch of dirt next to the *Farmer*.

**SowAction**

Extends the *Action* class. *SowAction* has a 33% probability of successfully sowing a UnripeCrop on a patch of dirt given that the *Farmer* object performing the action is next to a patch of dirt.

**Advantages:** When the *SowAction* object is added into *Player*’s *allowableAction* field, the option for the *Player* to sow on a patch of dirt next to them will be added into the menu.

**EatFoodBehaviour**

Implements the *Behaviour* interface. Only *Human* objects have an *EatFoodBehaviour*. Only *Human* objects have an *EatFoodBehaviour* object. It allows *Human* objects to perform *EatFoodAction* that consumes food and heals the *Human* object.

**EatFoodAction**

Extends the *Action* class. *EatFoodAction* removes a *Food* object in a *Human* object’s inventory and heals the *Human* for a certain amount of health points.

**Advantages:** When *EatFoodAction* object is added into *Food allowableAction* field, the option for the *Player* to eat food in their inventory will be added into the menu, given that they have a *Food* object in their inventory.

**EatFoodAction**

It is no longer added into the *Food*’s *allowableAction* field. It is now created in *addEatFoodAction(Actions actions, Actor actor)* method in *ActorInterface* and called in the *Player* class to give the *Player* the menu option to eat food.

**UnripeCrop**

Extends the *Item* class. *UnripeCrop* interacts with *FertilizeAction* because *FertilizeAction* will decrease the *age* of the *UnripeCrop* object. *UnripeCrop* cannot be harvested and can only be fertilized by *Farmer*s.

**Advantages:** This class is initiated in the *SowAction* class.

**RipeCrop**

Extends the *Item* class. A *RipeCrop* object is instantiated within an *UnripeCrop* object (when *UnripeCrop* object age > 20). It replaces a *UnripeCrop* object on the map with *RipeCrop*.

**Advantages:** This class is separated from the *UnripeCrop* class because when the player and a farmer want to harvest food from a *RipeCrop*, in the code “instanceof” syntax can be used for *Farmer* and the *Player* to see if the *item* in actor’s location is a *RipeCrop* or not.

**Crop (new class implemented to replace UnripeCrop and RipeCrop classes)**

Extends the *Item* class. *Crop* class will have a boolean variable that determines whether the *Crop* is ripe or not, the boolean variable will be changed according to the *Crop*’s age variable.

**Food**

Extends the *Item* class. *Food* objects are instantiated and either added into *Player* inventory (if the object is not *Farmer* type) or dropped to the ground (if the object is not *Player* type and is *Farmer* type) when a *RipeCrop* is harvested by a *Farmer*.

***INTERFACES:***

**ActorInterface:**

In the ActorInterface, getNumArm() and getNumLeg() default method has been created for actors to use default value for leg and arm but the logic of these functions will be changed in the ZombieActor by calculating the number of arms and legs which is in the items arraylist.

The same thing is done for getNumTurn() which the logic will be changed in the ZombieActor class. Also, the crafter method is used as a default value of false and this value will change to true if an actor implements the craft interface.

The harvester() and perThatEatFood() method works the same way as crafter() method.

**ItemInterface:**

Craftable() method gives an actor the value of false as a default value. Though, this value can be overrided for each item which becomes craftable.

The craft() function returns the value of 0 as an output for default value and if craftable item can override this method and after writing its logic in craft method returns the value of one which means it is craftable.

Car:

The car class represents the vehicle in the game. The object of this class has the method which add action to its allowable action (addAction()). Since the allowable actions of two cars are different (one moves from compound to town map and another moves player from town to compound map) two cars with their allowable action which is added in their addAction() method, will be added to the Application class. The reason that the cars along with their allowable actions were added to the application was that two cars get different functionalities and since cars get to be on the maps forever, there is no harm in creating their allowable actions in the Application class.

TransferAction():

Is the Action which Car class uses it (as an allowable action) and it allows the player(actor) to move from one map location to another. This class uses newMap, x, y as constructor parameters which newMap is the map which the player is going to be transferred to and x and y is the location of the player in the new map.

WorldSub:

Since world class is in the engine package and cannot be modified a sub class of world (WorldSub class) is created and the protected attributes and methods can be reused and override in the WorldSub class.

stillRunning():

This method is overridden from World class to check other than Player is all of the zombies are there and all human are existed and if all zombies are wiped out, is the mambo is vanished or dead. This implementation is done by getting the type of the actors (UNDEAD/ALIVE) and it is done by looping the actors through actorLocation. For getting the status of the mambo we can use the static getIsAlive() to find out whether the mambo is alive or not. The Boolean value of playerWins and playerLoses is updated when the zombies and mambo is wiped out and all humans are dead respectively. For finding whether the mambo marie is alive or not the static variable from MambaMarie class which is called isAlive() is used

The advantage of this implementation is that if in the future we got other Actors in our game the UNDEAD and ALIVE ZombieCapability can group Actors and we don’t need to get the number of the existed actors in the game for every Actors.

endGameMessage():

if the value of playerLoses is true it returns “player loses”. if the value of playerWins is true it returns “player wins” if the map does not contain player it returns “player loses”.

mambaCreation():

since the mamba is vanished when the game starts, some other class should create it and since the world has the control in all components of the game world is a good option to create mambo. The mamboCreation() is called in the stillRunning(). The mambo will be created based on the probability of 5 percent per chance (when the mambo is created till its vanished it won’t re-create in future turns of World class) and the mambo will be located in one of edges in the map. For finding out if the mambo is vanished, the isMamboVanished variable will be used. When the mambo is created, isMamboVanished will be false otherwise it always will be true.

The advantage of doing this method in World class is that it is rational to user that since the world has connected and organised all of the component of the game. And because of that it’s non sense that some other classes such as player or zombie would create the MambaMarie.

MambaMarie:

When mambo appear in the map they have the WanderBehaviour in their behaviour and the value of mambo increased by one and isBorn variable change to true(means the mambo has been created initially). When the mambo dies the mamboNum static variable decreases by one. (which was declared initially in world sub)

in the turns which is divisible by 10 the ChantingBehaviour will be added to the behaviour list (in the first index). If it’s not divisible by 10, the ChantingBehaviour will be removed from the behaviour list and if the turn is divisible by 30 the mambo will be removed from the map. For determining the whether the zombie is dead or not, the static variable “isAlive” will be false when the mambo is not conscious (overriding the isConcious() method in mambo class) in any other situations the isAlive will always be true.

When the mambo vanishes in the 30th turn, the isMamboVanished static variable (defined initially in WorldSub ) becomes true.

ChantingBehaviour:

It is the behaviour which uses ChantingAction. In ChantingAction 5 zombies will be created on the map in random location if that random location does not contain an actor.

DieAction:

This actions comes to player’s menu every single turn as “quit game” and when the user clicks on that option, the map which the player is on it, remove the player.