Modification

Previously, if a weapon is able to be crafted, it will inherit the CraftableWeaponItem class, as it will have one method called craftWeapon(), which allows an item to craft a new item.

Now, all craftable weapons will instead implement the Craftable interface, which includes CraftableWeaponItem class.

The reason behind this change is that there are some other weapon items that needs to inherit from other class. For instance, the Sniper and Shotgun needs to inherit from the GunWeaponItem class. Unfortunately, Java does not support multiple inheritance. Therefore, although the Sniper and Shotgun can be crafted (into Golden guns), they cannot inherit GunWeaponItem and CraftableWeaponItem at the same time. Therefore, to ensure consistency, all craftable weapon will now implement the Craftable Interface instead of inheriting from CraftableWeaponItem.

Changes to Class: CraftableWeaponItem now implements the Craftable interface.



Previously, if a Player is standing on an item that has allowable action, the player can execute the action even if he does not have the item. For instance, a ZombieLeg can be crafted even the Player has not picked it up. There are items that have allowables action that requires the Player to has them first:

such as

1. The player needs to have a bullet before he can reload
2. The player needs to have a sniper before he can shoot

To overcome this problem, a new interface called MenuActionInterface is added. Essentially what is does is that it requires classes that implement it to have an extra method called getAssociated. It is useful as it will allow the playTurn method in the Player class to filter certain actions that is not executable as the Player does not have them (by calling the getAssociated) aciton

Classes that implement this interface include:

1. CraftWeaponAction (getAssociated = originalPart(the item name to be crafted))
2. FireSniperAction (getAssociated = “sniper”)
3. FireShotgunAction (getAssociated = “shotgun”)
4. ReloadAction (getAssociated = bulletname)

This avoid a lot of “if” statement requires and allows polymorphism.

3. EatingHealAction implements MenuActionInterface

* So that player will not have the option to eat food that they don't have in their inventory

4. Player

* has an attribute of Wallet and NewWorld

Player has a wallet ( Wallet attribute ), able to update the amount of money Player has.

Player has a world ( NewWorld attribute ) as to add EndingGameAction into actions require a parameter of a NewWorld object to end that World, Player has also a setWorld method to set the World created to be the same one as the one involved in the game ( instead constantly creating a new one )

Association is used instead to reduce dependency and is relevant.

* Money == wallet amount

Update so that have when other class use the currency system have lesser dependencies

* Add EndingGameAction into the actions

So that quit game option is shown in the menu

New functionalities (Required)

1. Shotgun and sniper

As shotgun and sniper have similar functionalities such as get bullet count, deduct bullet count and reload, an abstract class called **GunWeaponItem** is created for them to inherit. **GunWeaponItem** can be inherited by extra gun-related weapons (Golden Shotgun and Golden Sniper). This can avoid a lot of repeated code.

1. Bullet + Reloading

**Bullet** is another abstract class that is being created as the parent class for all types of bullet, reason of creation is the same as **GunWeaponItem.** Currently, we have the **ShotgunBullet** and the **SniperBullet.**

The **ReloadAction** is added to the allowable actions of **ShotgunBullet** and **SniperBullet,** which allows Player to reload when the Player has it. **ReloadAction** takes 2 parameters to ensure that the action can be used to *reload any weapon*. The two parameters are weaponInvolved(String) and Bullet. This means that if the reload action is to reload a shotgun, then the parameter it takes would be

1. “Shotgun”
2. ShotgunBullet object.

Likewise, if the action is to reload a sniper, the parameters would be

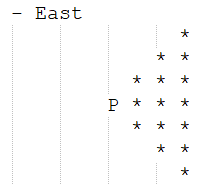
1. “Sniper”
2. SniperBullet object.

There is one edge case in this action. What if the player choose to uses a **SniperBullet** to reload if he does not have a sniper in an inventory. In this case, the **SniperBullet** object will not be removed and a “You don’t have the weapon” message will be returned.

b. Firing the shotgun

A **FireShotgunAction** is created to fire the shotgun. The action is added into the Shotgun allowable action. Once user choose to shoot the shotgun, he will be shown a menu to choose which direction he wants to shoot.

In order to get the locations, say to the East of the Player,



another class called the **LocationGenerator** object is created. This is basically a *data class* that takes the actor x and y coordinate as starting point, then generate the required locations based on direction given by user. I choose this approach instead of a generalised formula (which can reduce a lot of repeated code) because I realise that the generalised formula does not exist for all 8 directions. The generalised formula does exist, however, for 4 directions (North,South,West,East). However, even if this approach is used, the generalised formula is simply to hard to be understood by other programmer (I have tried implementing it).

After generating all the locations, actors standing on these locations have 75% of getting hurt. After they get hurt, there is a probability that they will die and corpse needs to be added. Therefore, I let **FireShotgunAction (**also **FireSniperAction)** to inherit the **AttackAction** class as they require methods that are written in AttackAction. This will reduce repeated code.

1. Firing the sniper

**FireSniperAction** is created for the player to fire the sniper. It is added into Sniper allowable actions. The action will first scan through the map to find possible actors. Then, the name of actors that are of type Zombie or MamboMarie will be displayed for user to choose. After choosing the actor to hurt, Player will have to decide to aim or to shoot. As this action requires a lot of step, I purposely broken down the operation into small methods. For instance, the getZombieFromMap method is specifically used to find the list of actors. And the displayMenu method is used to display all the actors. Doing in this way not only makes the code look cleaner, but also improves encapsulation (each encapsulation only perform one action).

Concentration level of Player is something that relates to the Player class a lot. This is because actions of Player will affect the concentration (switching target, getting hurt,etc). Therefore, a new concentration attribute is placed in the Player’s class, and be retrieved from **FireSniperAction** class via a getter.

I have also used HashTables to store the concentration to probability and concentration to damage table. This will avoid connascence of execution (if an array list to store these instead). It also allows me to expand the program easily (say I want to add one more concentration level).

1. Placing all gun-related class into the gun package

To improve encapsulation, all gun-related classes are moved from the game package to gun package.

2. MamboMarie

As mambo marie has only 5% of appearing each turn, instead of being initialised at the beginning of the game, it needs to rely on a class that ticks every turn to keep track. I decide to create a new type of ground (as ground has a tick method) , called **MamboMarieGrave** which has a 5% chance of spawning a MamboMarie object.

As only one MamboMarie should be on the map, a mamboStop boolean attribute is created in the **MamboMarieGrave** . It will be set to True after a MamboMarie is created to avoid the grave from spawning another MamboMarie in the next turn.

When MamboMarie is created, the grave is being placed in its constructor (dependency injection). MamboMarie will then be associated with **MamboMarieGrave.** This association is required because actions of Mambo Marie will affect whether or not the grave should create another MamboMarie. For instance, if the MamboMarie disappear after 30 turns, it should inform the grave that it can now generate a new MamboMarie. Another example would be if MamboMarie is dead, it would then inform the grave to set itself as Dirt (A kind of magic feeling that I tried to create :) ). No more MamboMarie will be created.

I have created actions for all the things that MamboMarie will do. **RemoveActorAction** will be created when MamboMarie has been on the map for 30 turns. Next, **Chanting Action** will be created if MamboMarie is on the 10th,20th or 30th. In chanting actions, new Zombie will be spawned. A while loop is used to avoid the Zombie being spawned (randomly) in a place where there is an actor standing on it or when the location ground is not a Dirt.

Reason why various classes are created for different actions to improve encapsulation (one class for one action) to prevent MamboMarie class from getting too messy.

3. Going to Town

In Application class, a new ArrayList of String to draw out the layout of the townMap is used.

Continued with adding the map to the world created, then added a Vehicle object ( new class ) which will act as an not portable item that can transport them to town by adding MoveActorAction to Vehicle's allowableAction and created another Vehicle to transport Player back to wildMap(compound map)

4. Ending the Game

Created a NewWorld class which inherits the World class as it is an unmodifiable engine class, Note that it had an int attribute of endGame -- to indicate what kind of ending

* 1 will be player loss
* 2 will be player win
* 3 will be quit game

stillRunning() is a crucial method in World class, to determine whether the game can proceed in the World class they only include if player is dead’s ending and once stillRunning() returns false, the whole game is over hence I make use of this in writing my conditions.

Back to NewWorld class I override the protected method stillRunning() to get super.actorLocations then iterate through,

Excluding actor instanceof Player

* Player win

If there is MamboMarie is killed and All zombies are eliminated

* To check is MamboMarie actually dead and not just not appear on the map, the NewWorld class has an MamboMarieGrave attribute which has a method getDeathInfo - only when MamboMarie died, this will return true
* For zombies, simple iterate through super.actorLocations, check if there is instanceof Zombie, then the boolean zombie will be of true
* Finally, make an if - condition

if(boss && !zombie){

setEndGame(1);

return false;}

* Player loss

Simple loop through actorLocation, check whether there is instanceof Human, if yes then boolean human is true

* Finally make an else if condition
* else if (!human){

setEndGame(2);

return false;

* Quit game

Need an extra EndingGameAction class for this “ option “ to be executed. In EndingGameAction class, there is a NewWorld attribute **world** as this is the targeted world to be ended. Then in the execute method, **world**.setEndGame(3) to end the game.

After this action is executed.

Back to NewWorld class

In the Override stillRunning(), there is an if-condition checking

if (getEndGame()==3){

return false;}

The EndingGameAction was then added to the Player’s playturn’s actions for it to appear in menu

New Functionalities (Bonus)

1. Buying and selling features

A new class called **Shop** is added. It is a non-portable item that has allowable actions: **BuyFromShopAction** and **SellToShopAction**. I decide to split the two as I think the two actions are quite different and splitting them will ensure the **Single Responsibility Principle.**

To allow buying and selling, items that are sellable needs to have a price tag. Therefore, all items that are sellable (food, gun weapon, bullet) will all implement **SellableInterface.** The interface ensures all sellable items have the method getPriceTag.

1. Buying

Essentially a HashTable is used to store the list of items sold. This allows the shop to expand easily (sell other things).

1. Selling

While the buying menu will be constant, the selling one will not. It will loop through the Player’s inventory to see if there is any items that are sellable. Then, the list of items and their price is displayed. Notes that the items will be sold at ¾ of its original price. This increases the game difficulty.

2. Crafting weapons

In the last assignment, ZombieLeg and ZombieArm can be crafted into more powerful weapons. In this new features, the sniper and shotgun can be used to craft the Golden Sniper and Golden Shotgun. To increase the game difficulty, the player cannot craft it himself. He must visit a shop called **CraftShop**. Similar to the **Shop** class, **CraftShop** has allowable actions called **CraftWeaponAtShopAction**.

**CraftWeaponAtShopAction** is a class that inherits from CraftWeaponAction. It is different from its parent because it will need to check if the Player has enough **Gold** to craft the weapon, which **Gold** can be picked up by the Player on the map. **CraftWeaponAtShopAction** will take a weapon and an integer signify number of gold to craft this weapon as parameters.

3. Currency

Wallet class is created and inherited Item class, so that the amount of Wallet can make use of the tick in the item class to increase every time the attribute magicTime hits multiple of 5. This is a way to increase money available.

Player has a Wallet which then will update the private attribute money in Player. Then, it has 2 methods: spendMoney and earnMoney which can change and update wallet.amount. Since player’s money always keeps up to date with the amount of wallet. This also changes the money the player has.

In both SellToShopAction and BuyFromShopAction, there is a use of the player's method to update the amount of wallet after transaction. After getting sure of the price of the item to sell or item to buy, it will either spendMoney(amount spent) or earnMoney(amount earned).