Design Rationale

The UML diagram only contains classes that are modified, new classes and inheritance for better readability. In all the new functionality implemented, downcasting is explicitly avoided by creating interface methods and implementing them in appropriate classes.

Going to town

Classes

- Vehicle : A subclass of Item which represents a vehicle for Player to move across maps.
- Application : Added a new map , a vehicle to move across maps ,some weapons, some zombies and humans in the new map under main.

Class Interaction

- When running the application, a new map will be created and new humans , zombies and weapons like shotgun and sniper will be placed on the new map. Player can move to the new map by a vehicle in this case it's a train. Player can move from either the old map to new map or the new map to old map. By doing this player can pick up the new weapon like shotgun and sniper from the new map and kill the zombie in the old map.

Design principles and rationale

- Inheritance: Inheritance is used for the new class like vehicle so that DRY principle is achieved.

Ammunition

Classes

- ShotgunAmmunitionBox: A subclass of PortableItem which represents a shotgun ammunition box.
- SniperAmmunitionBox: A subclass of PortableItem which represents a sniper ammunition box.

- Human: Added shotgun_ammo and sniper_ammo which represents the number of sniper and shotgun ammunition is carrying. Setter and getter methods for the ammunitions are also implemented.

Class Interaction

- When a player picks up a ShotgunAmmunitionBox or SniperAmmunitionBox, it is first placed in the inventory. The tick(Location currentLocation, Actor actor) which handles the passage of time in player inventory will increment shotgun_ammo or sniper_ammo depending on the ammunition box picked up. After that, the ammunition box is discarded from the inventory.

Design principles and rationale

- Inheritance: Inheritance is used for all the new classes so that DRY principle is achieved.
- Encapsulation: shotgun_ammo and sniper_ammo are made private and setter and getter methods are implemented for both instance variables.
- Maintainability: shotgun_ammo and sniper_ammo are placed in Human class instead of Player class so that in future development Humans can use guns the methods don't have to be implemented again.
- An integer instance variable is used to represent the number of ammunition the player is carrying instead of creating individual Ammunition items and placing it in inventory to reduce code complexity and increase game performance. As an example, if individual ammunition is placed in the inventory, when the getter for ammunition count is called, a for loop of the inventory will be required and a counter will be required. For the integer instance variable only the instance variable is returned to get the ammunition count.

Shotgun

Classes

- Shotgun: A subclass of WeaponItem which represents a shotgun
- BlastMenuAction: A subclass of Action which will bring up a submenu for player to choose fire direction.
- BlastAction: A subclass of Action that fires the shotgun in the given direction.

Class Interaction

- When a player chooses to fire the shotgun, they will be presented with a submenu created by BlastMenuAction to choose the direction of fire. The direction of fire is based on the exits available to the player, for example if the player cannot exit to south he/she cannot fire at south direction. When the player has chosen a direction, An instance BlastAction will be created by BlastMenuAction and the shotgun will fire in the chosen direction.
- Shotgun range for cardinal directions are as follow, A is the actor firing the shotgun:
  North:

  x x x x x

  x x x

  x

  A

- Shotgun range for intercardinal directions, A is the actor firing the shotgun:
  North-East:

  x x x x

  x x x x

  x x x x

  A x x x

Design principles and rationale

- Inheritance: Inheritance is used for all the new classes so that DRY principle is achieved.
- Encapsulation: All instance variables of shotgun are made private and setters and getters are implemented when appropriate, thus data hiding is achieved.
- Maintainability: RANGE constant is used when calculating the range of pellets, when the range needs to be changed in the future only RANGE will need to be modified and it will not affect the rest of the system.
- Single Responsibility Principle: Instead of clumping the direction choosing and firing together, they are separated into BlastActionMenu which handles the menu for choosing fire direction and BlastAction which handles the firing of the shotgun, thus Single Responsibility Principle is adhered.

- BlastAction is used to handle firing of a shotgun and it will create a new AttackAction with the actor each time an actor is in range, because shotguns are fairly inaccurate so the chance of dropping limbs is high.

SniperRifle

Classes

- SniperRifle: A subclass of WeaponItem which represents a sniper rifle.
- TargetMenuAction: A subclass of Action which will bring up a menu for the player to choose a target, will create an instance of ShootAimMenu once a target is chosen.
- ShootAimMenu: A subclass of Action which will bring up a menu that allows the player to aim or fire the sniper rifle, will create an instance of AimAction or SnipeAction depending on the option chosen.
- AimAction: A subclass of Action which represents the action of aiming with a sniper rifle.
- SnipeAction: A subclass of Action which represents the action of firing the sniper rifle.

Class Interaction

- When a player is holding a sniper rifle, a new instance of TargetMenuAction will be created and added to allowableActions of SniperRifle so that the player can choose a target. If the player choose to target an enemy with the sniper rifle, TargetMenuAction will loop through every location of the map to find out valid targets and a new instance of ShootAimMenu for the target is created and added to actions which will then be added to a new Menu and the menu will be displayed, allowing the player to select which target they wish to aim or fire at. When the player chooses a target, another submenu will be displayed and they can choose to aim or fire at the target, and depending on the option they chose AimAction or SnipeAction will be created.
- Player concentration is handled in the tick(Location currentLocation, Actor actor). Concentration will be lost if the player takes damage or takes any other action other than aim and fire.

Design principles and rationale

- Inheritance: Inheritance is used for all the new classes so that DRY principle is achieved.

- Single Responsibility Principle: Distinct classes are used to display target submenu, submenu for aiming and firing instead of clumping them into one class. Each new class also performs a single functionality and no more.
- Polymorphism: tick(Location currentLocation, Actor actor) is overridden in SniperRifle to handle concentration checks, because it is called every round.
- SnipeAction is used to handle firing of sniper rifle instead of AttackAction to avoid interface segregation. Besides that, sniper rifles are used to target vital areas instead of extremities such as limbs, hence AttackAction is not used.

Mambo Marie

Classes

- ZombieMap: A subclass of GameMap which controls the spawning and vanishing of Mambo Marie.
- MamboMarie: A subclass of ZombieActor which represents Mambo Marie.
- ChantBehaviour: A subclass of Behaviour which controls when Mambo Marie stops to perform a chant that creates zombies.
- ChantAction: A subclass of Action which represents the action of chanting to create 5 new zombies.

Class Interaction

- When Mambo Marie is not in ZombieMap, there is a 5 percent chance that she will spawn in the top left corner of the map. ZombieMap will automatically remove Mambo Marie when she appeared on the map for 30 turns. ZombieMap will also cease to spawn Mambo Marie in the map if she is killed.
- MamboMarie has 2 behaviours, WanderBehaviour and ChantBehaviour. She will wander around if she is not chanting and will stop to chant every 10 turns which is handled by ChantBehaviour.
- ChantAction will create 5 new Zombies across random locations on the map if there are no actors on that location or the location allows an actor to enter.

Design principles and rationale

- Inheritance: Inheritance is used for all the new classes so that DRY principle is achieved.
- Polymorphism and DRY principle: To achieve the functionality of GameMap and Mambo Marie, the tick() method from GameMap is overridden, and super.tick() is called before adding statements to achieve the new functionality to avoid repeating code while maintaining functionality of GameMap.

- ZombieMap is created as a Subclass of GameMap so that a GameMap with its original functionality is retained and Mambo Marie functionality can be added.
- Mambo Marie is a separate ZombieActor instead of a Zombie because in the requirements it is not stated that she could hunt and attack humans. However, her ZombieCapability is set to UNDEAD so that humans and players can target her with attacks and kill her.

Ending the game

Classes

- endGame : A subclass of World which checks the conditions of the ending game.
- QuitAction : A subclass of Action which represents the action of quitting the game.

Class Interaction

- Under endGame check the conditions of the ending game such that if game over if all human dead, player dead or player won if all zombie is dead, if match the condition then end the game by printing their corresponding endGameMessage. QuitAction is added to the actions in Player playTurn() method where it will be printed to the menu. Once Player chooses QuitAction, Player will be removed from the map which will end the game.

Design principles and rationale

- Inheritance: Inheritance is used for the new class like QuitAction so that DRY principle is achieved.
- Polymorphism: stillRunning() is overridden in endGame to check the conditions of ending the game.