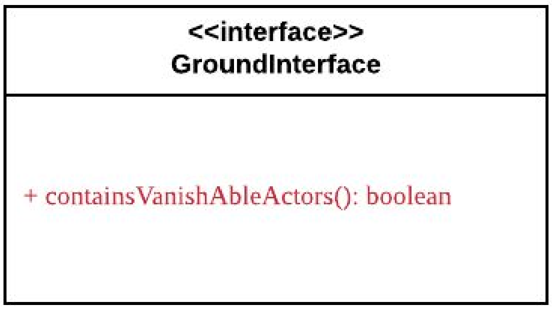
FIT2099 - Update Design Rationale for Assignment 3

Name: Hee Weng Sheng
Student ID: 31226507

Name: Isaac Lee Kian Min
Student ID: 31167462

Old Classes Modified

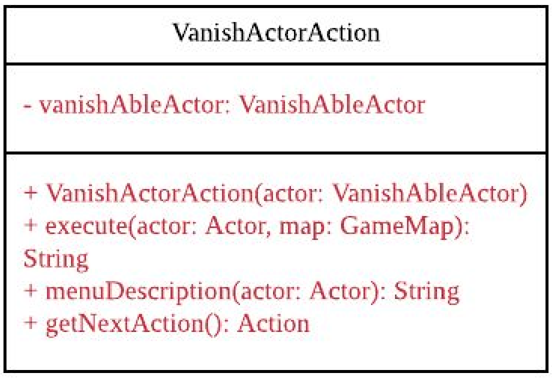
The words in **red** indicate a new variable/method being added into the class, whereas words in **black** indicate extending an already existing method.



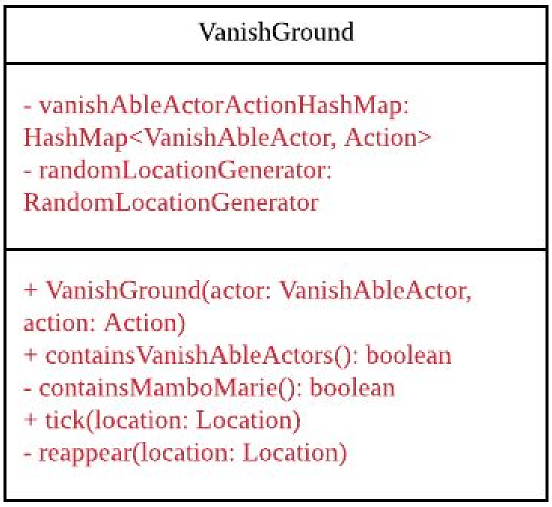
- A **containsVanishAbleActors** method to determine whether the ground houses actors that are able to vanish/disappear.
- Designed it this way because:
    - To reduce downcastings, which causes dependencies between classes
    - Make use of polymorphic code.

New Classes Added/Extended

The words in **red** indicate a new variable/method being added into the class, whereas words in **black** indicate extending an already existing variable/method.

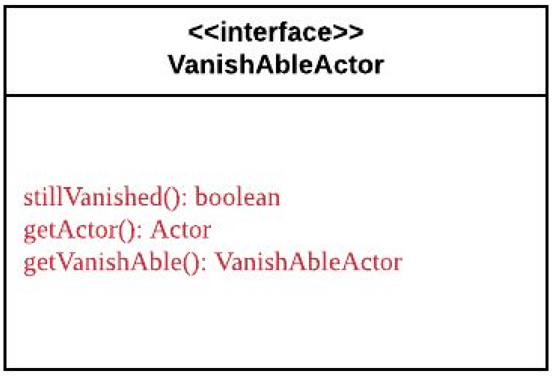

VanishActorAction
- vanishAbleActor: VanishAbleActor
+ VanishActorAction(actor: VanishAbleActor)
+ execute(actor: Actor, map: GameMap): String
+ menuDescription(actor: Actor): String
+ getNextAction(): Action

- A **VanishActorAction** class extends an **Action** class to make an actor that is able to vanish disappear from the current game map.
- A **vanishAbleActor** attribute to determine the actor to be vanished.
- A **VanishActorAction** constructor to create a VanishActorAction object.
- An override **execute** method that makes the vanishAbleActor disappear.
- An override **menuDescription** method that returns a description of the vanished action.
- An override **getNextAction** method to determine whether an actor remains vanished or not.
- Designed it this way because:
    - Extends existing classes to avoid downcastings, which causes dependencies.
    - Encapsulates the action of vanishing into one container class.
    - Use the Single Responsibility Principle as this class's only concern is to cause an actor to be vanished from a game map.

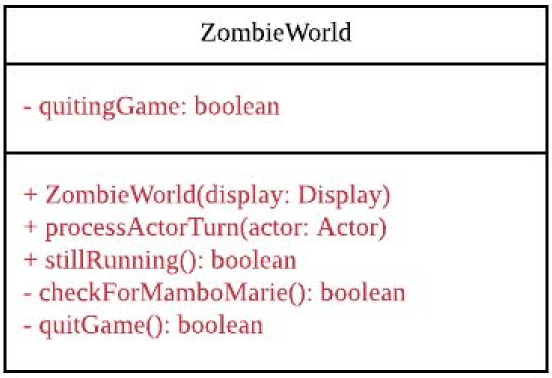

VanishGround
- vanishAbleActorActionHashMap: HashMap<VanishAbleActor, Action>
- randomLocationGenerator: RandomLocationGenerator
+ VanishGround(actor: VanishAbleActor, action: Action)
+ containsVanishAbleActors(): boolean
- containsMamboMarie(): boolean
+ tick(location: Location)
- reappear(location: Location)

- A **VanishGround** class extends a **Dirt** class which houses the actors that are able to vanish from the game map.
- A **vanishAbleActorActionHashMap** attribute to determine the actors that vanished at this location and their last actions.
- A **randomLocationGenerator** attribute to help generate a random location.
- A **VanishGround** constructor to create a VanishGround object.
- An override **containsVanishAbleActors** method to determine whether this ground houses vanished actors.
- A **containsMamboMarie** method to determine whether this ground contains MamboMarie.
- An override **tick** method that gives this ground the flow of time, while also determining whether any vanished actors will reappear on the game map.
- A **reappear** method that determines whether any vanished actors wants to reappear on the game map.
- Designed it this way because:

    - Extension of existing classes to make use of existing mechanisms for accessing a ground, thus reducing possibilities of duplicate codes.
    - Use Open-Closed Principle by overriding superclass methods to implement own behaviours, thus encapsulating this unique behaviour in its own class.



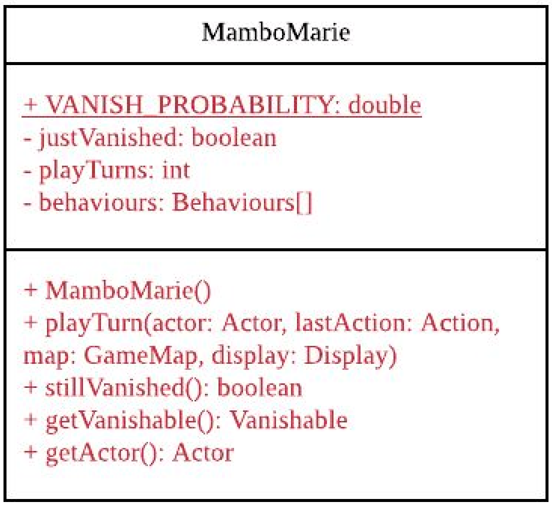
- A **VanishAbleActor** interface that allows an actor in the game to have the ability to vanish from the game map and reappear by implementing it.
- A **stillVanished** method to determine whether the actor remains vanished.
- A **getActor** method to get an Actor reference of this VanishAbleActor.
- A **getVanishAble** method to get a VanishAbleActor reference of the actor.
- Designed it this way because:
    - Provides a blueprint in the form of an interface for all actors to be able to vanish from the game map.
    - Allows for reuse in the case where we create another new character that can also vanish.
    - Provides a layer of abstraction to avoid downcastings by just having a one general reference (e.g. reference a subclass that implements this interface with the superclass), making use of polymorphic code.



ZombieWorld
- quitingGame: boolean
+ ZombieWorld(display: Display)
+ processActorTurn(actor: Actor)
+ stillRunning(): boolean
- checkForMamboMarie(): boolean
- quitGame(): boolean

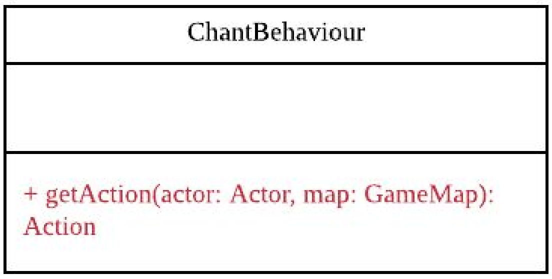
- A **ZombieWorld** class that extends the **World** class to implement certain behaviours.
- A **ZombieWorld** constructor to construct a ZombieWorld object.
- An override **processActorTurn** that checks whether the player wants to quit the game on top of processing the actor's play turn.

- An override **stillRunning** method checks certain conditions to determine the current state of the game (i.e. check whether specific actors are alive) on top of checking whether the player is still alive.
- A **checkForMamboMarie** method that determines whether MamboMarie still exists in the game.
- A **quitGame** method used as a menu to prompt the player on whether it wants to quit the game.
- Designed it this way because:
    - Extends existing class to reduce code duplication by inheriting from parent class.
    - Override to implement own behaviours and implementations, thus encapsulating inside its own class.

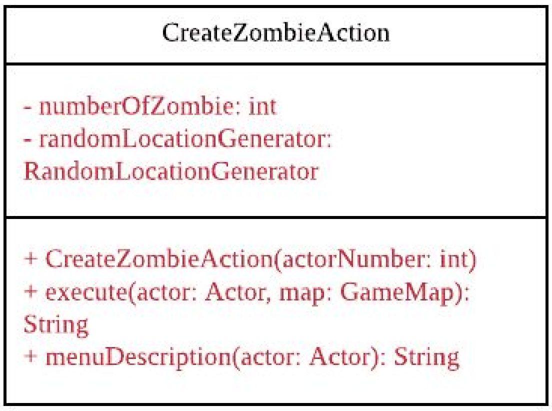

MamboMarie
+ VANISH_PROBABILITY: double
- justVanished: boolean
- playTurns: int
- behaviours: Behaviours[]
+ MamboMarie()
+ playTurn(actor: Actor, lastAction: Action, map: GameMap, display: Display)
+ stillVanished(): boolean
+ getVanishable(): Vanishable
+ getActor(): Actor

- A **MamboMarie** class that extends a **ZombieActor** and implements a **VanishAbleActor** to imitate the Mambo Marie character.
- A **VANISH_PROBABILITY** static attribute to determine the probability of Mambo Marie remaining vanished from the game map.
- A **justVanished** attribute to determine whether it just disappeared from the game map.
- A **playTurns** attribute to keep track of the number of turns played.
- A **behaviours** attribute that determines the behaviours of it.
- A **MamboMarie** constructor to create a MamboMarie object.
- A **playTurn** method to determine the actions played based on its behaviours.
- An implemented **stillVanished** method to determine whether the actor remains vanished.
- An implemented **getActor** method to get an Actor reference of this VanishAbleActor.
- An implemented **getVanishAble** method to get a VanishAbleActor reference of the actor.
- Designed it this way because:

- A static attribute used to remove embedded literals while at the same time provide constant access to all objects of such since its attribute is the same among all instances.
- Extends an existing class to reuse some methods defined in the superclass, reducing duplication of codes.

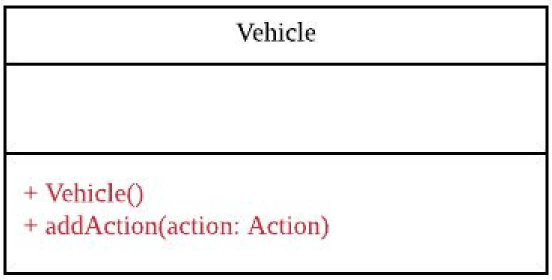


- A **ChantBehaviour** class that implements a **behaviour** and is a nested private class within the **MamboMarie** class.
- An implemented **getAction** method to determine the actions based on its behaviour.
- Designed it this way because:
    - Inner class inside the MamboMarie class because the chanting behaviour of summoning new zombies is unique only to Mambo Marie's chant.
    - Implements an existing interface to avoid downcastings and provide a centralized way to access and invoke its method.

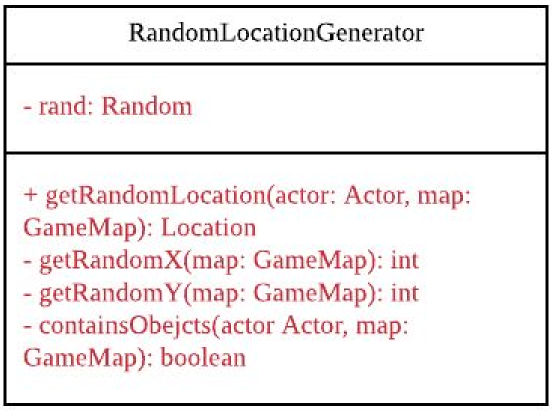

CreateZombieAction
- numberOfZombie: int
- randomLocationGenerator: RandomLocationGenerator
+ CreateZombieAction(actorNumber: int)
+ execute(actor: Actor, map: GameMap): String
+ menuDescription(actor: Actor): String

- A **CreateZombieAction** that extends an **Action** and creates a specified number of **Zombie** actors in the game map.
- A **numberOfZombie** attribute that determines the number of Zombies to be created.
- A **randomLocationGenerator** attribute to help generate a random location.
- A **CreateZombieAction** constructor to create a CreateZombieAction object.
- An override **execute** method to create a specified number of Zombies.
- An override **menuDescription** method to return a description of how many Zombie's created.
- Designed it this way because:
    - Extends existing classes to avoid downcastings, which causes dependencies.
    - Encapsulates the action of creating new zombies into one container class.

    - Use the Single Responsibility Principle as this class's only concern is to create new zombies within the game map.



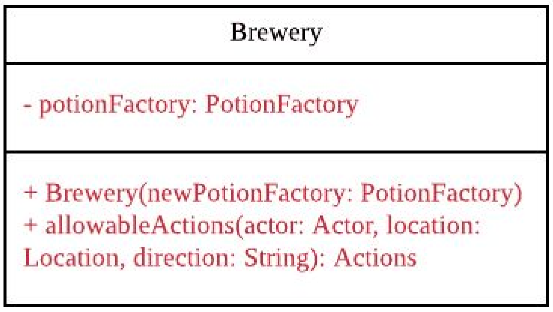
- A **Vehicle** class that extends an **Item** and is used to travel across different game maps.
- A **Vehicle** constructor to create a Vehicle object.
- An **addAction** method that adds a specified action into its allowable actions list.
- Designed it this way because:
    - Extends existing classes to avoid downcastings, which causes dependencies.
    - Use the Single Responsibility Principle as this class's only concern is to mimic the behaviours of a vehicle.


RandomLocationGenerator
- rand: Random
+ getRandomLocation(actor: Actor, map: GameMap): Location
- getRandomX(map: GameMap): int
- getRandomY(map: GameMap): int
- containsObejcts(actor Actor, map: GameMap): boolean

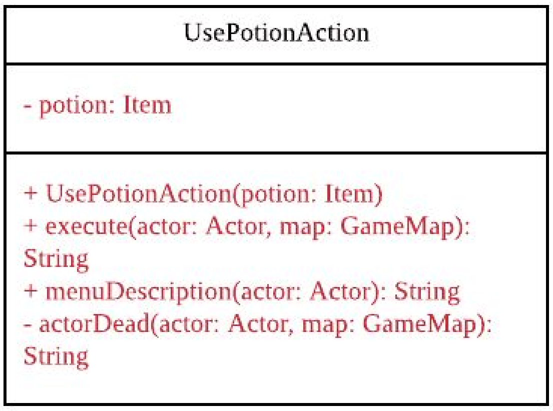
- A **RandomLocationGenerator** class that acts as a helper for getting random locations on a game map.
- A **rand** attribute used as a random number generator.
- A **getRandomLocation** method used to get a random location on a game map.
- A **getRandomX** method used to get a random x coordinate of a location.
- A **getRandomY** method used to get a random y coordinate of a location.
- A **containsObjects** method used to check whether the location contains objects or is impassable.
- Designed it this way because:
    - Use the Single Responsibility Principle as this class's only concern is to find a random location on the game map that is empty and passable.
    - Reduce the need for duplication of code as we can just reuse the class implementations.

        - However, this might lead to dependencies, which may undermine maintainability.

<u>Bonus Mark New Classes</u>

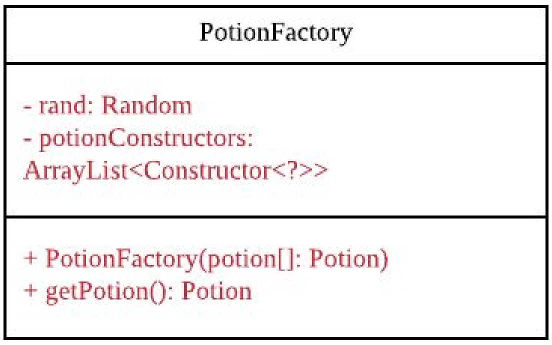

Brewery
- potionFactory: PotionFactory
+ Brewery(newPotionFactory: PotionFactory)
+ allowableActions(actor: Actor, location: Location, direction: String): Actions

- A **Brewery** class created that extends a **Ground** class to model after a shop that sells potions. (Disclaimer: the owner of this brewery is very generous and decides to give free potions)
- A **potionFactory** attribute that allows this Brewery to brew potions.
- A **Brewery** constructor that creates a Brewery object.
- An override **allowableActions** method that allows the Brewery to give out potions to actors for free.
- Designed it this way because:
    - Extends existing classes to avoid downcastings, which causes dependencies.
    - Encapsulates the implementations of it within a class.
    - Use the Single Responsibility Principle as this class's only concern is to provide a place to get potions for actors in the game map.

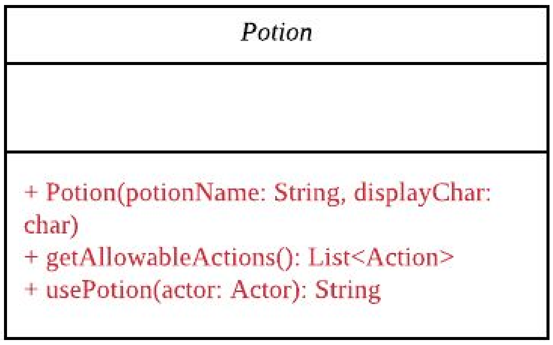

UsePotionAction
- potion: Item
+ UsePotionAction(potion: Item)
+ execute(actor: Actor, map: GameMap): String
+ menuDescription(actor: Actor): String
- actorDead(actor: Actor, map: GameMap): String

- A **UsePotionAction** class that extends the **Action** class.
- A **potion** attribute that specifies the potion used.
- A **UsePotionAction** constructor to create a UsePotionAction object.
- An implemented **execute** method that allows the actor with this potion to use it.
- An implemented **menuDescription** method that returns a description of using the potion.

- An **actorDead** method to determine whether an actor is still conscious after using the potion.
- Designed it this way because:
    - Extends existing classes to avoid downcastings, which causes dependencies.
    - Use the Single Responsibility Principle as this class's only concern is to execute the action of using a potion.
    - Encapsulates the action of using a potion into one container class.

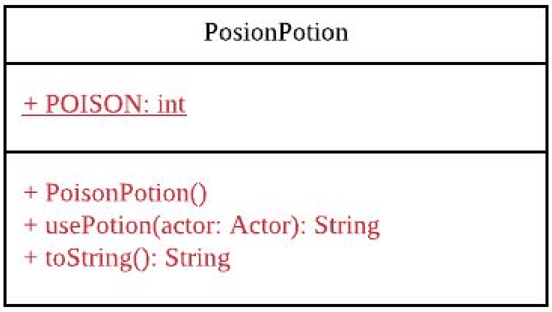

PotionFactory
- rand: Random
- potionConstructors: ArrayList<Constructor<?>>
+ PotionFactory(potion[]: Potion)
+ getPotion(): Potion

- A **PotionFactory** class acts as a factory for creating different types of potions.
- A **rand** attribute used as a random number generator.
- A **potionConstructors** attribute to store the type of potions this factory brews.
- A **PotionFactory** constructor that creates a PotionFactory object.
- A **getPotion** method that gets a randomly brewed potion.
- Designed it this way because:
    - Provide a centralized mechanism for creating potions, reducing possibilities for dependencies in the forms of constructing new objects.
    - Uses the Dependency Injection Principle in its constructor which allows the specification of the potions this factory can brew.
    - Use the Single Responsibility Principle as this class's only concern is to provide brewed potions.
    - However, there is a dependency link with the Potion class.

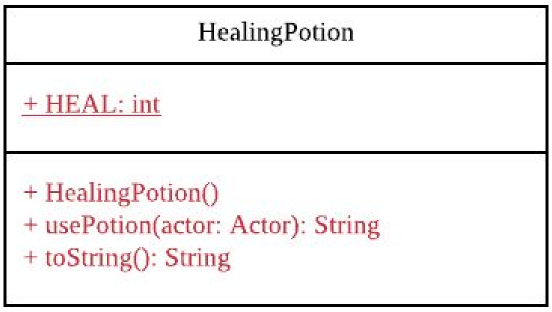

Potion
+ Potion(potionName: String, displayChar: char)
+ getAllowableActions(): List<Action>
+ usePotion(actor: Actor): String

- A **Potion** abstract class that extends an **Item** class and defines what a potion is.
- A **Potion** constructor that creates a Potion object.
- An override **getAllowableActions** method that allows an actor to get this potion since potions are not portable, i.e. they break once dropped on the ground.

- An abstract **usePotion** method that when implemented, executes the effects of using it, which may benefit or cause harm to the user.
- Designed it this way because:
    - Provides a layer of abstraction, which is useful in avoiding downcastings through polymorphic code.
    - Use the Liskov Substitution Principle as we can substitute the superclass with subclasses. In this case, Potion is defined as abstract.
    - Extends existing classes to avoid downcastings, which causes dependencies.

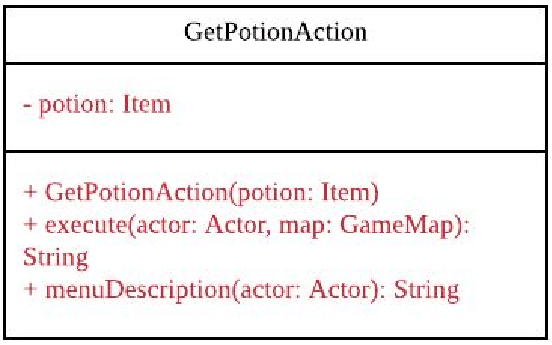


- A **PoisonPotion** class that extends a **Potion** class that models a potion that poisons.
- A **POISON** static attribute that determines the damage this poison does.
- A **PoisonPotion** constructor that creates a PoisonPotion object.
- An implemented **usePotion** method that causes the actor to get poisoned.
- An overridden **toString** method that returns a name of the poison potion.
- Designed it this way because:
    - Extends the Potion class to make use of the abstraction used in Potion class and make use of inheritance.
    - Use the Single Responsibility Principle as this class's only concern is to model after a PoisonPotion.
    - Encapsulates its implementations and behaviours within a container class.



- A **HealingPotion** class that extends a **Potion** class that models a potion that heals.
- A **HEAL** static attribute that determines the damage this poison does.
- A **HeaingPotion** constructor that creates a PoisonPotion object.
- An implemented **usePotion** method that heals the actor using it.
- An overridden **toString** method that returns a name of the healing potion.
- Designed it this way because:

    - Extends the Potion class to make use of the abstraction used in Potion class and make use of inheritance.
    - Use the Single Responsibility Principle as this class's only concern is to model after a HealingPotion.
    - Encapsulates its implementations and behaviours within a container class.


GetPotionAction
- potion: Item
+ GetPotionAction(potion: Item)
+ execute(actor: Actor, map: GameMap): String
+ menuDescription(actor: Actor): String

- A **GetPotionAction** class that extends the **Action** class.
- A **potion** attribute that specifies the potion to get.
- A **GetPotionAction** constructor to create a GetPotionAction object.
- An implemented **execute** method that allows the actor to get the potion.
- An implemented **menuDescription** method that returns a description of getting the potion.
- Designed it this way because:
    - Extends existing classes to avoid downcastings, which causes dependencies.
    - Use the Single Responsibility Principle as this class's only concern is to execute the action of getting a potion.
    - Encapsulates the action of getting a potion into one container class.

How are these classes used to implement the features described in the Assignment:

From the Going To Town question,

1) You must implement a town level. Place a vehicle somewhere on your existing map. The vehicle should provide the player with the option to move to a town map.
    a) Create a new class Vehicle that extends an Item abstract class. Create a new addAction method in it that adds a specified Action into its allowableActions variable.
    b) In the Application class, create a new instance of Vehicle and add a new MoveActorAction that moves to a different Location on a different GameMap.
    c) Designed it this way because:
        i) Extended this class with Item to reduce downcastings through polymorphic code and implement own behaviours.
        ii) Make use of the existing class (e.g. MoveActorAction) to reduce duplicate codes.

From the Mambo Marie question,

1) Mambo Marie is a Voodoo priestess and the source of the local zombie epidemic. If she is not currently on the map, she has a 5% chance per turn of appearing. She starts at the edge of the map and wanders randomly. Every 10 turns, she will stop and spend a turn chanting. This will cause five new zombies to appear in random locations on the map.
    a) Create a new MamboMarie class that extends the ZombieActor class.
    b) MamboMarie has a playTurns attribute to keep track of the number of turns played.
    c) Create an inner private class inside the MamboMarie class called ChantBehaviour, which implements a Behaviour interface.
    d) Create a CreateZombieAction class that extends the Action class.
    e) ChantBehaviour checks whether MamboMarie has played 10 turns. If so, it returns a CreateZombieAction which will create 5 new Zombies and place them on different Locations on the same GameMap as MamboMarie.
    f) In the MamboMarie class, add a behaviour attribute that stores an array of behaviours, which are WanderBehaviour and ChantBehaviour in order.
    g) Access these behaviours by implementing the abstract playTurn method.
    h) Designed it this way because:
        i) ChantBehaviour implemented as an inner class to encapsulate the chant behaviour of MamboMarie, which results in the creation of new zombies.
        ii) CreateZombieAction encapsulates its complexities and behaviours within a class. It also applies the Single Responsibility Principle as it has the purpose of just creating new zombies, which also allows for easy maintainability and reusability.
        iii) However, the behaviours attribute may have some level of connascence of execution (but since it's already implemented beforehand, no need worry)

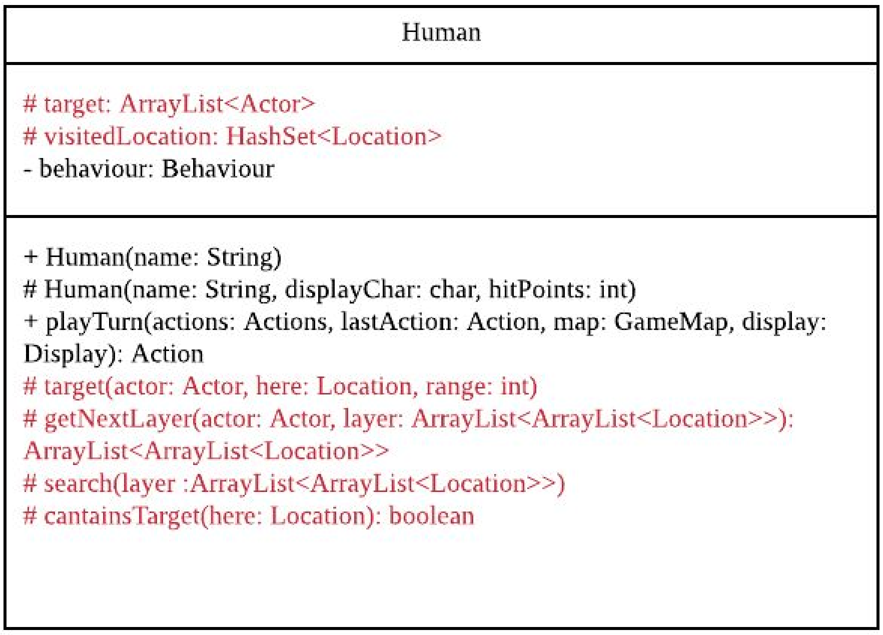
2) If she is not killed, she will vanish after 30 turns. Mambo Marie will keep coming back until she is killed.
    a) Create a new VanishAbleActor interface.
    b) Create a new VanishActorAction class that extends the Action class.
    c) Create a new VanishGround class that extends the Dirt class.
    d) Group these 3 classes into a vanishcapabilities package.
    e) MamboMarie implements a VanishAbleActor to allow MamboMarie to vanish after 30 turns.
    f) Inside the overridden playTurn method, check whether the playTurns attribute is more than 30. If so, create a new VanishActorAction instance which will remove MamboMarie from the current GameMap and place it in a new VanishGround instance. Set the ground of the Location MamboMarie was last seen using this VanishGround instance.
    g) Inside the VanishGround class, override the tick method to check whether MamboMarie wants to remain vanished by reusing the VanishActorAction, which overrides the getNextAction method to check whether the VanishAbleActor wants to remain vanished.

h) If not, she will reappear at a random Location of the GameMap it was in before it vanished.
i) Designed it this way because:
    i) VanishAbleActor interface allows for reusability as any Actor that wants to vanish from the GameMap simply needs to implement this interface (just as how the Weapon interface is used in the game).
    ii) VanishActorAction class encapsulates the complexities of removing the VanishAbleActor from the GameMap. Besides that, it also reuses the action to determine whether the actor wants to remain vanished. This also reduces downcastings by inheriting existing class and implementing own behaviours while making use of polymorphic code.
    iii) VanishGround class again encapsulates its own behaviours within the class, taking care of the complexities of making an actor vanish to and reappearing. It also reuses the Dirt class to reduce duplicate code and make use of polymorphic code by overriding parent implementations with its own.
    iv) Encapsulated these 3 classes into the vanishcapbilities package to allow for easy access and maintainability.

3) For the Bonus Mark question, we decided to implement a Brewery, which is a shop that brews potions and gives these potions freely to any actor within close proximity to it. We made changes to the initial design suggested, which was to only brew and give out potions based on their rarity to instead just brew and give out a random potion.
    a) Create a new class called Brewery that models after a potion shop.
    b) In it, define a PotionFactory attribute that allows the Brewery to brew potions.
    c) The Brewery will provide potions to any actor within one space of it through the allowableActions method.
    d) Within this allowableActions method, a new potion is brewed by calling the getPotion method in the PotionFactory class.
    e) After which, we include this newly brewed potion as an argument while instantiating a new GetPotionAction object.
    f) In the event an actor has a potion in its inventory, it can choose to use it. This will invoke the getAllowableActions method, which creates a new UsePotionAction object that allows the actor to use the potion.
    g) Designed it this way because:
        i) Placing these new created classes inside its own package allows for better maintainability.
        ii) Each class is designed to have only one responsibility, making it easier to extend or maintain the source code. This also helps reusability as we won't be required to redefine a new class, reducing duplication of codes.

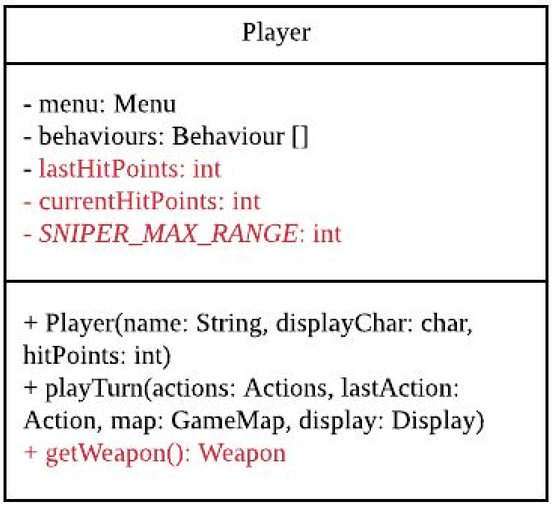
Old Classes Modified

The words in red indicate a new variable/method being added into the class, whereas words in black indicate extending an already existing variable/method.



```
Human
---------------------------------------------
# target: ArrayList<Actor>
# visitedLocation: HashSet<Location>
- behaviour: Behaviour
---------------------------------------------
+ Human(name: String)
# Human(name: String, displayChar: char, hitPoints: int)
+ playTurn(actions: Actions, lastAction: Action, map: GameMap, display:
Display): Action
# target(actor: Actor, here: Location, range: int)
# getNextLayer(actor: Actor, layer: ArrayList<ArrayList<Location>>):
ArrayList<ArrayList<Location>>
# search(layer :ArrayList<ArrayList<Location>>)
# cantainsTarget(here: Location): boolean
```

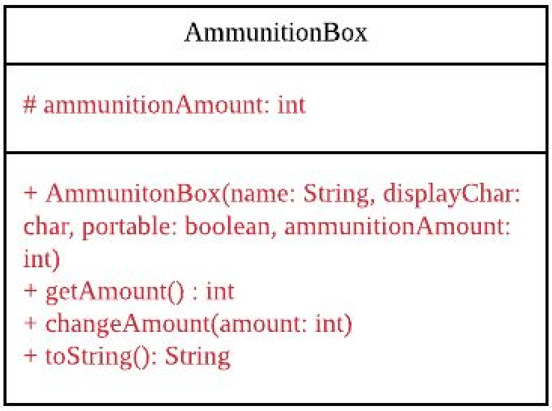
- A **target** attribute which indicates the target a specific human actor wants to find.
- A **visitedLocaiton** attribute which indicates all the location within a range.
- A **target** method that search for the target base on the action of the specific human actor is performing.
- A **getNextLayer** method that search for the next location.
- A **search** method that is used to search through all the location whether it contains the target.
- A **containsTarget** method that return whether the location contains the target.


Player
- menu: Menu
- behaviours: Behaviour []
- lastHitPoints: int
- currentHitPoints: int
- *SNIPER_MAX_RANGE*: int
+ Player(name: String, displayChar: char, hitPoints: int)
+ playTurn(actions: Actions, lastAction: Action, map: GameMap, display: Display)
+ getWeapon(): Weapon

- **lastHitPoints** and **currentHitPoints** attributes are used to track the hitPoints of current and last turn of the player.
- **SNIPER_MAX_RANGE** attribute is defined as a constant for the sniper range.

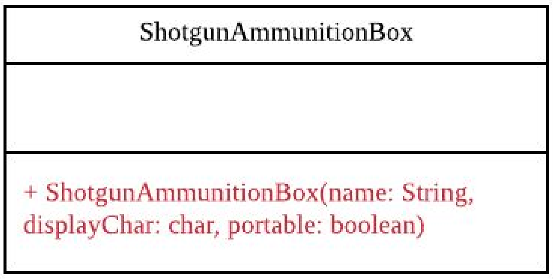
New Classes Added/ Extended

The words in red indicate a new variable/method being added into the class, whereas words in black indicate extending an already existing variable/method.

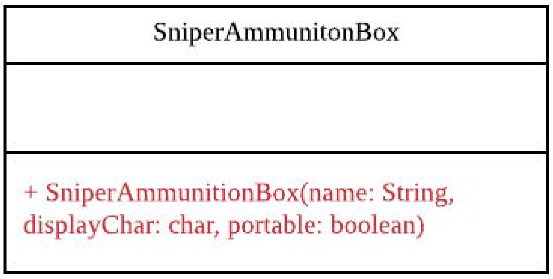

AmmunitionBox
# ammunitionAmount: int
+ AmmunitonBox(name: String, displayChar: char, portable: boolean, ammunitionAmount: int)
+ getAmount() : int
+ changeAmount(amount: int)
+ toString(): String

- An **AmmunitionBox** extends an **Item**.
- An **AmmunitionAmount** attribute keeps track of the number of ammunitions left in the box.
- An **AmmunitionBox** constructor that takes in the name of the AmmunitionBox, the character to be display, whether it is portable and the amount of ammunition in the box to creates an AmmunitionBox object.
- A **getAmount** method that returns the amount of ammunition left in the box.

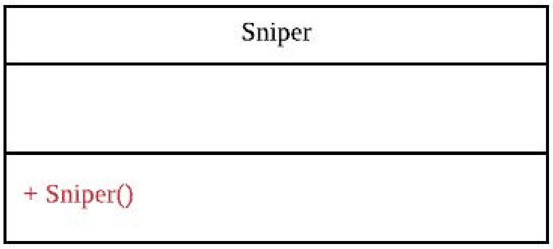
- A **changeAmount** method that takes in the amount to be change.
- A **toString** method to display the amount of ammunition left in the box.


ShotgunAmmunitionBox
+ ShotgunAmmunitionBox(name: String, displayChar: char, portable: boolean)

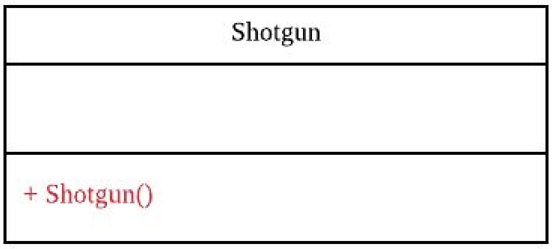
- A **ShotgunAmmunitionBox** extends a **AmmunitionBox**.
- A **ShotgunAmmunitionBox constructor** method that takes in the name of the ammunition box, char to be display and whether the ammunition box is portable to creates a ShotgunAmmunitionBox object..


SniperAmmunitonBox
+ SniperAmmunitionBox(name: String, displayChar: char, portable: boolean)

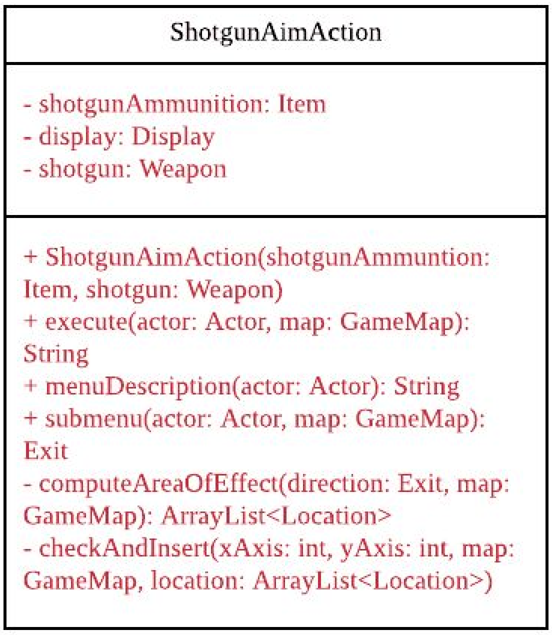
- A **SniperAmmunitionBox** extends a **AmmunitionBox**.
- A **SniperAmmunitonBox constructor** method that takes in the name of the ammunition box, char to be display and whether the ammunition box is portable to create a SniperAmmunitionBox object.



- A **Sniper** class that extends **WeaponItem** class.
- A **Sniper** constructor method to create a Sniper object.

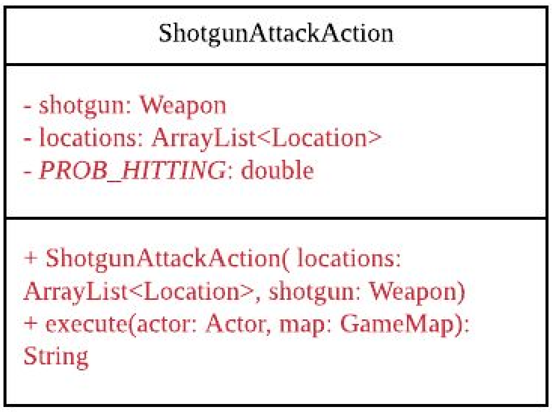


- A **Shotgun** class that extends **WeaponItem** class.
- A **Shotgun** constructor to creates shotgun object.


ShotgunAimAction
- shotgunAmmunition: Item
- display: Display
- shotgun: Weapon
+ ShotgunAimAction(shotgunAmmuntion: Item, shotgun: Weapon)
+ execute(actor: Actor, map: GameMap): String
+ menuDescription(actor: Actor): String
+ submenu(actor: Actor, map: GameMap): Exit
- computeAreaOfEffect(direction: Exit, map: GameMap): ArrayList<Location>
- checkAndInsert(xAxis: int, yAxis: int, map: GameMap, location: ArrayList<Location>)

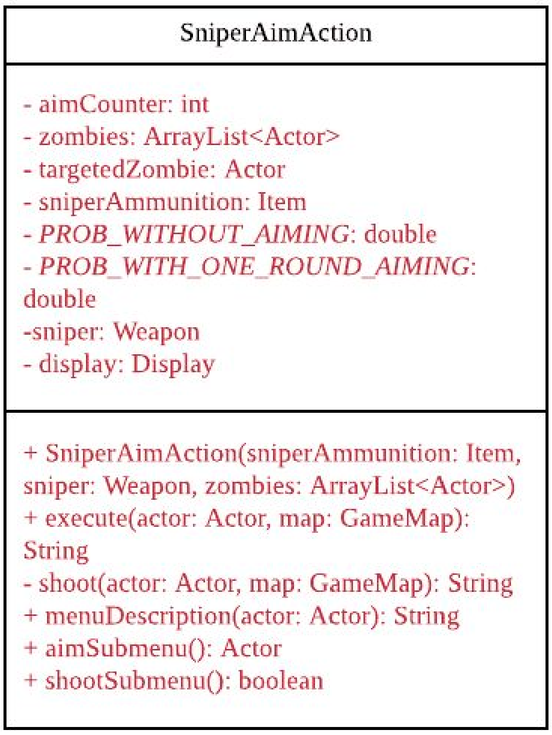
- A **ShotgunAimAction** class that extends **Action**.
- A **shotgunAmmunition** attribute that keep track of which ammunition the shotgun is using.
- A **display** attribute to display or get user input.
- A **shotgun** attribute to keep track of which shotgun the user is using.
- A **ShotgunAimAction** constructor that takes in the shotgunAmmunition and shotgun to create ShotgunAimAction object.
- An **execute** override method which read user input on which direction they want to shoot at and fire at that direction.
- A **menuDescription** override method which return a description of the user uses a shotgun
- A **submenu** method that display the possible direction to shoot and prompt the user for the direction the user wants to shoot at.
- A **computeAreaOfEffect** method that map out the locations that will be fired at.

- A **checkAndInsert** method is used to check whether the location is on the map and insert it into an ArrayList of Location if true.



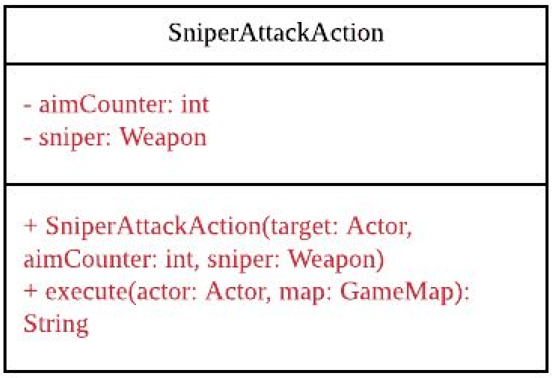
ShotgunAttackAction
- shotgun: Weapon
- locations: ArrayList<Location>
- PROB_HITTING: double
+ ShotgunAttackAction(locations: ArrayList<Location>, shotgun: Weapon)
+ execute(actor: Actor, map: GameMap): String

- A **ShotgunAttackAction** class that extends **AttackAction** class.
- A **shotgun** attribute that keep track of the shotgun the user is using.
- A **locations** attribute that store the locations to be fired at.
- A **PROB_HITTING** attribute is a constant that defines the probability of hitting an actor.
- A **ShotgunAttackAction** constructor takes in locations and shotgun to create ShotgunAttackAction object.
- A **execute** override method that deal damage to the actor who are in range and return a description who got damaged.

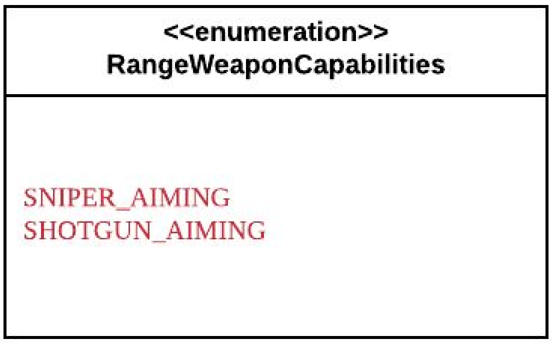

SniperAimAction
- aimCounter: int
- zombies: ArrayList<Actor>
- targetedZombie: Actor
- sniperAmmunition: Item
- PROB_WITHOUT_AIMING: double
- PROB_WITH_ONE_ROUND_AIMING: double
-sniper: Weapon
- display: Display
+ SniperAimAction(sniperAmmunition: Item, sniper: Weapon, zombies: ArrayList<Actor>)
+ execute(actor: Actor, map: GameMap): String
- shoot(actor: Actor, map: GameMap): String
+ menuDescription(actor: Actor): String
+ aimSubmenu(): Actor
+ shootSubmenu(): boolean

- A **SniperAimAction** class that extends Action class.
- An **aimCounter** attribute to keep track of the number of rounds the user aim at the target.
- A **zombies** attribute that store the zombie that are in range of the sniper.
- A **targetedZombie** attribute that keep tracks of which zombie the user is aiming at.
- A **sniperAmmunition** attribute that keep tracks of which ammunition box the user is using.
- A **PROB_WITHOUT_AIMING** and **PROB_WITH_ONE_ROUND_AIMING** attributes that define the probability of successfully shooting the zombie.
- A **sniper** attribute that keep tracks which sniper is the user using.
- A **display** attribute that enables the user to enter input and display text.
- A **SniperAimAction** constructor that takes in sniperAmmunition, sniper and zomebies to create SniperAimAction object.
- An **execute** override method that prompt user which target the user want to aim and prompt user got another input whether the user want to fire or continue aiming the zombie. It returns a description of the actor aims which zombie for how many turns.
- A **shoot** method that create a new SniperAttackAction if the user wants to shoot the target and return a description of the damage dealt to the target.
- A **menuDexcription** override method that return a description of user uses the sniper.

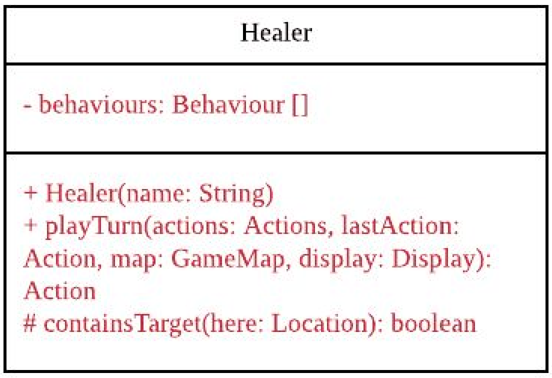
- An **aimSubmenu** method that display the available target and prompt the user for the target the user wants to aim.
- A **shootSubmenu** method that prompt user whether the user wants to shoot or continue aiming.


SniperAttackAction
- aimCounter: int
- sniper: Weapon
+ SniperAttackAction(target: Actor, aimCounter: int, sniper: Weapon)
+ execute(actor: Actor, map: GameMap): String

- A **SniperAttackAction** class that extends **AttackAction** class.
- An **aimCounter** attribute that stores the number of turns the user used in aiming the target.
- A sniper attribute that keeps track of which sniper is the user using.
- A **SniperAttackAction constructor** that takes in the target to be shoot at, the number of turns takes to aim the target and the sniper the user is using to create SniperAttackAction.
- An execute override method that deal damage to the actor who are in range and return a description who got damaged.

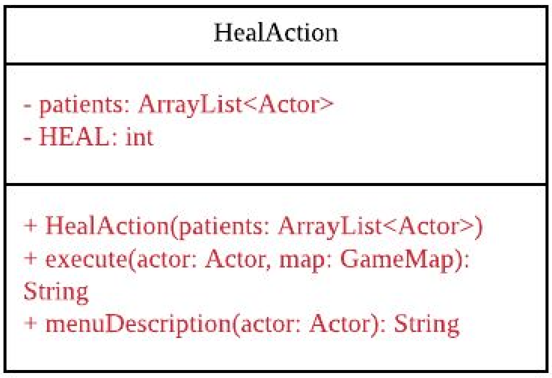


- A **RangeWeaponCapabilities** enum class that defines the different capabilities of aiming each RangeWeapon has.
- Currently defines **SNIPER_AIMING**, **SHOTGUN_AIMING** as different capabilities of each RangeWeapon has.



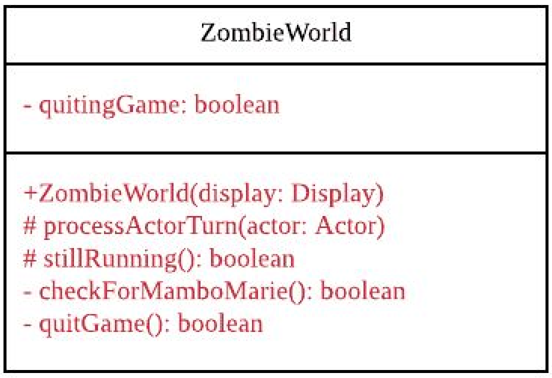
```
Healer
-----------------------------------------
- behaviours: Behaviour []
-----------------------------------------
+ Healer(name: String)
+ playTurn(actions: Actions, lastAction:
Action, map: GameMap, display: Display):
Action
# containsTarget(here: Location): boolean
```

- **Healer** class that extends **Human** class.
- A **behaviours** attribute that stores the different type of Behaviour the healer has.
- A **Healer** constructor that takes in the name of the healer to create Healer object.
- A **playTurn** override method that gets an Action to perform when is the Healer turn.
- A **containsTarget** method that check whether the location contains target.



```
HealAction
-----------------------------------------
- patients: ArrayList<Actor>
- HEAL: int
-----------------------------------------
+ HealAction(patients: ArrayList<Actor>)
+ execute(actor: Actor, map: GameMap):
String
+ menuDescription(actor: Actor): String
```

- A **HealAction** class that extends Action class.
- A **patients** attribute that store the Actor to be heal.
- A **HEAL** attribute which is a constant that defines the number of health the patients will gain.
- A **HealAction** constructor that takes in ArrayList of patients to be heal to create HealAction object.
- An **execute** override method that heal all the patients and return a description of which patients is heal.
- A **menuDescription** override method that return a description of the patients who is healed.


ZombieWorld
- quitingGame: boolean
+ZombieWorld(display: Display)
# processActorTurn(actor: Actor)
# stillRunning(): boolean
- checkForMamboMarie(): boolean
- quitGame(): boolean

- A **ZombieWorld** class extends World class
- A **quittingGame** attributes which a boolean value whether the user want to quit the game.
- A **ZombieWorld** constructor method that takes in display to create ZombieWorld objec.
- A **processActorTurn** overrides method to check whether the user wants to quit the game.
- A **stillRunning** overrides method to check whether the player, human, farmer, zombies, and mambo marie is still alive.
- A **checkForMamboMarie** method to check whether the mambo marie is still on the compound map even though it is temporarily vanished.
- A **quitGame** that prompt user whether the user wants to quit the game or continue on with the game.

How are these classes used to implement the features described in the Assignment 3:

From the New weapons: shotgun and sniper rifle question,

1) Ranged weapons use ammunition. Place boxes of shotgun and rifle ammunition on the town and compound maps in the locations of your choice.
    a) Creates a new abstraction class called AmmunitionBox class which extends the Item class.
    b) This class contains getAmount() which allow the user to keep track of the ammunition amount in the box. It also contains changeAmount() which enables user to increase or decrease the number of ammunition.
    c) Creates two new classes that represent two different kinds of AmmunitionBox which are called SniperAmmunitionBox and ShotgunAmmunitionBox.
    d) These two new classes both extends the AmmunitionBox class.
    e) Designed it this way because:
        i) To prevent duplicate codes. So created an abstraction class which contains the methods that both subclasses have.
        ii) This shows a good practice of design principle which is “Don’t Repeat Yourself” (DRY).

iii) Create two different types of AmmunitionBox as different range weapons have different kind of ammunition.

2) The **shotgun** has a short range, but sends a 90◦ cone of pellets out that can hit more than one target – that is, it does area effect damage. Rather than being fired at a target, the shotgun is fired in a direction. Its range is three squares. So, if the shotgun is fired north, it can hit anything in the three squares north of the shooter, northeast of the shooter, northwest of the shooter, or anything in between. It has a 75% chance of hitting any Actor within its area of effect.

a. Creates Shotgun class which extends WeaponItem class.

b. Creates ShotgunAimAction class which extends Action class.

    i. Create new method submenu which will display a submenu that shows the direction (Eg, North, North-West ...) the user can shoot at and prompt user for the direction the user wants.

    ii. Create new method computeAreaEffect which will return an ArrayList of Location which are the locations the pellets are going to shoot at.

c. Creates ShotgunAttackAction class which extends AttackAction class.

    i. Attribute of PROB_HITTING which is a constant that defines the probability of hitting any actor is 75%.

    ii. Overrides the execute method which now will check for every location I have compute, if there's an actor, it has a 75% chance to hit the target.

d. Designed it this way because:

    i. I divided into two class so each class has its own functionality instead having both Action in one class and form a big chunk of class which is hard to understand.

    ii. Provides a submenu when the user want to shoot instead of giving all the direction in the main menu. This implementation can make space for other actions or else it will almost always take up 8 slots just for the direction to shoot at.

3) When the player fires your sniper rifle, they should be presented with a submenu allowing them to choose a target. Once they have selected a target, they have the option of shooting straight away or spending a round aiming.

   a. Creates a Sniper class which extends WeaponItem class.

   b. Creates an attribute called aimCounter which tracks the number of rounds the actor spends on aiming the target.

   c. Creates a SniperAimAction class which extends Action class.

      i. Creates a new method which is aimSubmenu. This will display all the possible targets which are within the range of the snipers. Then it will prompt users for the zombies they want to aim at.

      ii. Creates a new method which is shootSubmenu. This will display to the user whether they want to continue aiming for a turn or shoot. Then it will prompt the user for the decision of shooting or aiming. If the user chooses to shoot it will call the shoot method. Else it will increment the aimCounter by 1.

      iii. Creates new method which is shoot. This method will determine the probability and damage of hitting base on the numbers of turn the user spend to aim on the target. If it is within the probability it will create a new SniperAttackAction which perform the shooting part.

   d. Creates a SniperAttackAction class which extends AttackAction.

      i. Override the execute method so it will has its own way to deal damage to the target with a sniper.

   e. Design in this way because:

      i. Divided into two different class of SniperAimAction and SniperAttackAction instead of having a big chunk of class that store all the functionality of a sniper.

4) If the shooter takes any action other than aiming or firing during this process, their concentration is broken and they lose their target. They will also lose their target if they take any damage.

a. To check player take any damage while aiming. In Player class playTurn method, I will keep track of the player hitPoints if the player has both sniper and sniper ammunition in his inventory. When the player lastAction is SniperAimAction, I will have a check on the currentHitPoints of the player and the previous turn hitPoints. If the hitPoints are not equally it will not add the last SniperAimAction instead it will add a new SniperAimAction which restart the aimCounter.

b. To check player takes any action other than aiming during the aiming process. In Player class playTurn method, if the lastAction is SniperAimAction then I will add the last SniperAimAction. Else, it means the player did other action other than aiming on the target therefore it will add a new SniperAimAction which restart the aimCounter.

c. To check player change to another target. In SniperAimAction, it will check whether the player choose another zombie to continue the aiming. If the zombie is different it will set the aimCounter back to 0.

d. Designed it this way because:

    i. The result of the number of turns spend on aiming should only be dealt in SniperAimAction. The SniperAimAction should only deal with how to shoot the target.

From the Ending the Game question:

1) A “quit game” option in the menu.

    a. In the newly created zombieWorld, overrides the processActorTurn. In this method, if the actor is Player, it will prompt the player whether the player want to quit the game or not by calling the newly created method called quitGame.

    b. Design it this way because:

        i. It is a simple way to check and do not need to create any new class like QuitAction.

        ii. It will prompt the player whether the player wants to continue playing the game or quit every turn.

2) A “player loses” ending for when the player is killed, or all the other humans in the compound are killed.

A “player wins” ending for when the zombies and mambo marie have been wiped out and the compound is safe.

a. In the newly created ZombieWorld, overrides the stillRunning method.

b. In this stillRunning overrides method, first check whether the player is alive. If the player is alive, then check whether humans or farmers are alive. It also check whether zombies and mambo marie is still alive.

c. In ZombieWorld, newly added method called checkMamboMarie. This will check whether the mambo marie is still in the compound map even though she is temporarily vanished from the map.

d. Design it in this way because:

   i. Mambo Marie may be vanished when all zombies are killed. Therefore, we will added a method to check whether the mambo marie is still on the compound map.

   ii. Instead of running the old world class that only checks whether the player is dead, we create a new class that extends it. This allows us to replace the world with ZombieWorld as the subclass without breaking the application. This design principle is one of the S.O.L.I.D principle which is the Liskov substitution principle.

For the bonus features:

1) New Healer actor that will heal the allies(Human, Farmer and Player) for 10 health every turn within 3 squares.

a. Creates a Healer class that extends Human class.

b. In the Healer playTurn override method, check whether there are any allies within the 3 squares range. If there is, the healer will perform the HealAction. Else, the healer will wander around.

c. Creates new class which is HealAction. This class is used to heal the allies within the range for 10 health.

d. Design it this way because:

   i. Choose to extends the Human class to prevent duplicate codes. In human class it contains the target method can be used in healer to find the healer allies. Therefore this can prevent duplicates code which is the DRY principle.