DESIGN RATIONALE - ASSIGNMENT 3

Going to town

- Vehicle class extends Item

Vehicle has to be an Item in order to be displayed on a GameMap and to impart an Action to the Actor using it. A Vehicle has two GameMap instance variables `start` and `destination` which respectively represent the departure and destination map of the vehicle. A vehicle at town map will have a reversed set of `start` and `destination` from a vehicle at compound map.

Vehicle overrides `getAllowableAction` method to return its only allowable action ChangeMapAction which enables an Actor to make a switch from the current map to destination map. By using polymorphism on Item, an Actor can invoke `getAllowableAction` method on Vehicle and retrieve a ChangeMapAction.

Design principle: DRY, Polymorphism

- ChangeMapAction extends Action

ChangeMapAction also has two GameMap instance variables representing the departure and destination point.

The first reason for creating a new Action class for map switching is to take advantage of Action's interface with which the rest of the game engine interacts. As a result, the action can be added to Player's set of options and have a displayable menu description.

The second reason is to increase encapsulation. Player does not have access to underlying implementation details of the map-transferring process and hence cannot have dependencies on these details in future operations.

Design principle: DRY, Encapsulation, Dependency Inversion Principle

Shotgun and sniper rifle.

- **new methods; reload, getRounds, fire in ItemInterface**

Default methods to be overridden when necessary to avoid the need to downcast.

Design principle: Polymorphism

- killTarget() in AttackAction

This method kills a target and removes the target from the map. If the target was holding any items in their inventory, they will be dropped to the ground at the target's location. This method was pulled up from `ShotgunShootingAction` and `SniperShootingAction` to prevent duplication of code.

Design principle: DRY

- ReloadAction extends Action

Special weapons require ammunition. This class allows the player, if he is carrying any ammo, to reload his special weapon. In both Shotgun and SniperRifle, overriding

`getAllowableActions` enables `ReloadAction` to be performed if the right condition is met. Using polymorphism, the weapon's owner (Player) will be prompted for the option of reloading.
Shotgun can only be reloaded with ShotgunAmmo and SniperRifle can only be reloaded with SniperRifleAmmo. During execution, we loop through the `Actor's` inventory to check if there is ammunition of the right type. ReloadAction checks for the right ammunition type using a parameter provided at its instantiation so that it doesn't have to know the concrete ammo class (be it SniperRifleAmmo, ShotgunAmmo or other types).
Once an ammo is found, the weapon is reloaded using the weapon's `reload` method. Once reloaded, the item is dropped and deleted from the map.

Design principle: Dependency Inversion Principle, Polymorphism

Specification: *Implement new kind of ranged weapon, shotgun and a sniper rifle*
- new Classes; Shotgun, SniperRifle extends WeaponItem
Shotgun and Sniper both extend WeaponItem in order to allow it to be used as a weapon (i.e: deal damage on a target using WeaponItem's `damage` method).
Both Shotgun and SniperRifle have new instance variables, `clipSize` and `DAMAGE`. `clipSize` is the number of rounds or ammunition left to fire the shotgun and `DAMAGE` is the damage dealt to an Actor by the shotgun.
Shotgun and Sniper override methods `fire` and `reload` from `ItemInterface`. `fire` method decreases the `clipSize` by 1. `reload` refills ammunition by increasing the clipSize.

Design principle: Polymorphism, Classes should be responsible for their own properties

Specification: *Range weapons use ammunition.*
- new Classes; ShotgunAmmo, SniperRifleAmmo extends PortableItem
ShotgunAmmo, SniperRifleAmmo extend `PortableItem` to enable the player to have a PickUpAction on them. They override `getRounds` method from `ItemInterface` to get the number of rounds refilled in a weapon. This method is called from `ReloadAction` when the weapon is reloaded, avoiding the dependency between ReloadAction and concrete Ammo types.

Design principle: Polymorphism

Specification: *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. The shotgun is fired in a direction. Its range is three squares. Has a 75% chance of hitting any Actor within its area of effect.*
- ShotgunShootingAction extends AttackAction
ShotgunAction extends AttackAction instead of Action primarily to reuse the `killTarget` method. ShotgunShootingAction itself has inner classes that handle different functions required to produce the overall output which is shooting any Actors present within the 90°, 3 square range area of damage. During execution of the ShotgunShootingAction, the Actor will be presented with a menu of FireAction for each direction.

Design principle: DRY

- private inner class FireAction extends Action

FireAction is responsible for identifying `x` and `y` coordinates of the shotgun's damage area based on the chosen direction and executes the firing. If the direction is not cardinal, `shootingXY` method is called to retrieve the area, else, `shootingCardinal` method is used. This is because the algorithm to create the area of damage is different for both cardinal and non - cardinal positions. Both these methods return the method `executeFiring` that identifies all `Actors` in the damage area and inflict damage accordingly.

FireAction is declared with private visibility and can be accessed from inside ShotgunShootingAction because it is executed only during ShotgunShootingAction's course of action. It is very unlikely for outside classes to have the need to call or have dependency on the class.

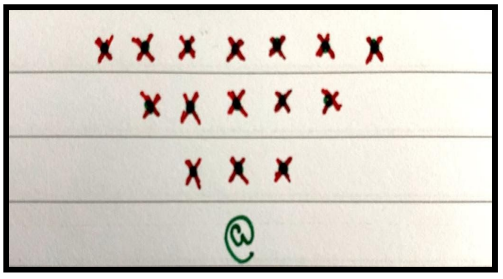
Design principle: Single Responsibility Principle, *Declare things at the tightest possible scope*

- private method shootingXY, shootingCardinal in FireAction

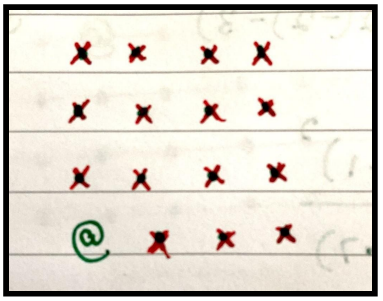
Both internal methods use the `Player's location` to retrieve the Player's `x` and `y` positions on the map. We identified that when fired in the four main directions (North, South, East, and West), the damage area is in the shape of a triangle. Thus we implemented an algorithm that uses the player's position to generate x and y coordinates of a 90° triangle with the length of 3 squares for the direction selected by the player

For cardinal directions we identified that the damage area resembled a square. Hence using the player's x and y coordinates, we implemented a similar but slightly altered algorithm to calculate the coordinates of the square area for the direction chosen.

The calculated `xRange` and `yRange` is then used as parameters in `executeFiring` method.



XY Directions



Cardinal Directions

- private method executeFiring in FireAction

This method executes `fireXYDirection` if input the parameter `cardinal` is not true, else executes `fireCardinalDirection`. Both above mentioned methods return a list of `Actors` that were hurt are in the shotgun shooting area. For these actors in the list, `killTarget` method of `AttackAction` is used to kill any actors with zero health and return a string `output`.

Design principle: Single Responsibility Principle

- private method firingXYDirection, firingCardinalDirection in FireAction
For cardinal directions, `firingCardinalDirection` is used otherwise for any other direction, `firingXYDirection` is used. The two methods simply scan through the `xRange` and `yRange` combinations to check if an `Actor` is present in a given `x,y` coordinate. If so, the method `addTarget` is called that deals damage to the `Actor` (this happens with a 75% success rate) and adds the `Actor` to the `hurtActors` list which is returned at the end of both firing methods.

Design principle: Single Responsibility Principle

- private method addTarget in FireAction
A simple method called in `firingXYDirection` and `firingCardinalDirection`. Damage is dealt to the target (`Actor`) with a success rate of 75% and adds the hurt target to the `hurtActors` list.

Design principle: Declare things at the tightest possible scope

- ShotgunSubMenu extends Menu
A submenu designed to display a list of directions the `Player` could choose to fire at when using the shotgun.
`ShotgunSubMenu` extends `Menu` and allows its client code (ShotgunShootingAction) to use polymorphism to operate menu functions and replace dependency to a concrete class with a dependency to a high-level class (Menu).

Design principle: Dependency Inversion Principle

- private enum class Direction
`Direction` is a `private enum class` that contains all the directions the player is allowed to fire in, i.e North, North East, East, South East, South, South West, West and North West.

Design principle: Avoid excessive use of literals

Specification: *When the player fires your sniper rifle, they should be presented with a submenu allowing them to choose a target.*
- SniperSubMenu extends Menu

(Follows the same design rationale mentioned in `ShotgunSubMenu`., allowing SniperShootingAction to operate on the interface of Menu instead of the concrete Menu class)
The method displays a list of zombies in the player's field of view, allowing the player to select which zombie to snipe and returns the `Actor(zombie)` associated with the player's selection. The overridden `showMenu` method shows the player his options on what to do with the selected target. i.e the player could fire, aim or retreat. Once selected, it returns the action associated with the selected option.

Design principle: Dependency Inversion Principle

Specification: *Once they have selected a target, they have the option of shooting straight away or spending a round aiming. Spending time improves aiming as follows:*
No aim : 75% chance to hit, standard damage
One round aiming: 90% chance to hit, double damage
Two rounds aimin: 100% chance to hit, instakill
- SniperShootingAction extends AttackAction
(Follows the same design rationale mentioned in `ShotgunShootingAction`.)

- overriding getNextAction in Action
As the specification didn't carry details on how big the player's field of vision was or if he is able to snipe between walls, we decided to allow the player to choose zombies in the half the map the player is in. For example, if the player's `x` coordinate is less than the `x` coordinate of the centre of the map, the player is shown all the zombies in the left half of the map.
The method is designed to run in a loop if the player chooses to aim in order to make the game more realistic, since you cannot run away from zombies and aim at the same time.
To allow this feature, `Action's getNextAction` method was overridden to return `SniperShootingAction` if `aim` is greater than zero. Therefore the player can only continue to aim, fire or retreat if the player had selected to aim in their previous turn. All the other features of the game run normally.

Design principle: Declare things at the tightest possible scope, Single Responsibility Principle, *Polymorphism*

- private class FireSniperAction, AimAction, RetreatAction extends Action
The whole process of firing the sniper rifle is split into 3 classes to ensure that each class handles one responsibility.

> **FireSniperAction** deals damage to the selected zombie based on the specification rules. Internal method `inflictDamage` hurts the actor. If aim is greater than 2, killTarget method is used to remove the Actor
>
> **AimAction** simply increments Player's concentration using `setConcentration` of Player.

RetreatAction loses concentration and target. Player's `deleteZombieTarget` and `setConcentration` is used to set concentration to zero and set Player's class variable `target` to null.

These classes are declared with private visibility because they are only executed during SniperShootingAction's course of action. It is very unlikely for outside classes to have the need to call or have dependency on the class, or it will risk going against game rules. Hence, its access is highly restricted.

Design principle: Declare things at the tightest possible scope, Single Responsibility Principle

- new default methods setZombieTarget, getZombieTarget, deleteZombieTarget, setConcentration and getConcentration in ActorInterface and override methods from ActorInterface in Player.

- Introducing new variables `target` and `concentration` to save and retrieve the target when the player uses the sniper rifle and to keep track of the number of turns the player has been aiming.

Overrides setters (setZombieTarget, setConcentration, deleteZombieTarget) and getters (getZombieTarget, getConcentration) from `ActorInterface` to modify and return `target` and `concentration`.

Overriding hurt from Actor to set the player's target to null and resets the concentration to zero when the player is dealt damage from a zombie.

Design principle: Polymorphism

MamboMarie

- Mambo extends ZombieActor

Mambo inherits ZombieActor as it has all the characteristics of an Actor including but not limited to the ability to experience time passing, execute actions and be inflicted from an attack action, etc. As a subclass of Actor, Mambo can interact with other game elements such as items, actions or other actors using Actor's public interface with the help of polymorphism, instead of enforcing direct dependency to itself.

Design principle: DRY, Polymorphism, *Dependency Inversion Principle*

- MamboLocation extends Location

Mambo needs to occupy a Location in order to experience time passing (i.e: to have `tick()` invoked). However, reusing an existing Location on the Map class would cause Mambo to be displayed the moment right after it is added to the Map, while Mambo is not allowed to appear until certain conditions are met. It also means Mambo has to occupy a whole position, which doesn't make sense as Mambo should not obstruct other Actor's movement prior to its appearance on the Map.

New class MamboLocation is created to address the above problems. It is located outside of the map's range so that it won't be displayed and won't affect the moving space of other Actors.

(As a side note, since Mambo does not appear on game's map from the beginning by default, it won't use the same mechanism to enter the map like other Actors. Instead, a Mambo will be initialized with a GameMap parameter.)
Design principle: DRY

- classes AppearAtMapEdgeAction, ChantAction, VanishAction extend Action
Mambo's action is split into 3 distinct classes since each of these classes is in charge of executing only 1 out of many actions. Choosing which action to take, which is a completely different task that involves querying the current state of the game, is the job of MamboBehaviour.
Not only does this implementation abide to Command-Query Separation and Single Responsibility Principle, it also creates more modularized classes in much smaller size, which in turn reduces certain code smells and enhances program's overall maintainability.

Design principle: Command-Query Separation, Single Responsibility Principle

- MamboBehaviour class implements Behaviour interface
At each play turn, Mambo has its Action by invoking `getAction` on MamboBehaviour, where the deliberation process of which action to take next happens. The reason why this process is not incorporated into Mambo's `playTurn` is to ensure a clear separation of processing turns and selecting actions responsibility. Since `playTurn` method, as its name suggests, has already taken charge of processing Actor's turns, adding the extra task (that would involve dependencies to 3 different classes) would simply risk it being a "god method".

Design principle: Single Responsibility Principle

Ending the game
Specification: *A "quit game" option in the menu*
-private inner class QuitGameAction extends Action
The decision to encapsulate QuitGameAction under Player is to limit access and avoid any dependency from other classes. Other Actors or game elements should not be able to invoke a QuitGameAction, hence it is unnecessary for them to know about the existence of the class.

Design principle: Declare things at the tightest possible scope, *Encapsulation*

Specification: *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*
- private static variable `population` and public static method `getPopulation` of class Zombie, Human, and Mambo
Zombie, Human and Mambo each have their own static variable int population which keeps track of how many instances of Zombie/ Human/ Mambo has been created. The constructor will increase `population` of their class by 1. As soon as the actor becomes unconscious (i.e: `isConscious()` returns `false`), `population` decreases by 1.

Access to manipulate `population` is restricted to methods within the class. The only operation allowed for external classes is to query current `population` data using `getPopulation`.

Design principle: Declare things at the tightest possible scope, Classes should be responsible for their own properties

- Player's inner enum class Result

Result enum class has two Result constants: Result.WIN and Result.LOSE defined with their own string representation *"Player wins"* and *"Player loses*". EndGame's constructor requires a parameter of type Result whose string representation will be displayed at its execution. This way of implementation allows game outcome to be determined at Player class instead of EndGame class and excuses EndGame from dependencies with game details, which eventually enhances EndGame's reusability in future implementations.

Design principle: Classes should be responsible for their own properties (for Player class)

- Player's private inner class EndGame extends Action

The reason ending the game is made into a separate Action class (as opposed to being done in Player's `playTurn`) was to preserve the single responsibility of the `playTurn` method. This implementation could ensure playTurn is solely in charge of processing the turn and EndGame is solely in charge of ending the game.

The reason why EndGame is made an inner class or Player was to limit its visibility from outside the class (similar to QuitGameAction)

Design principle: Single Responsibility Responsibility, Declare things at the tightest possible scope