**Group Name: Team Julian  
Members: Julian Yong Hao, Khong Jun Ming**

**LoadGunAction**This class is inherited from the Action class. This class loads a gun, either a rifle or a shotgun with a correct ammunition. This action can only be done by the Player if the player is holding both gun and the respective ammunition.

Design Rationale  
This class will require two parameters in order to instantiate which are the gun and the ammunition. When this action is executed, it will remove the capability of ItemCapability.UNLOADED and add the capability to ItemCapability.LOADED. This changes the status of the gun from unloaded to loaded which will enable the Player to shoot. In the same time, we also add two more capabilities which are, ItemCapability.BULLET1 and ItemCapability.BULLET2. By doing this, we can keep track of how many this the gun is used to shoot. A single ammunition can only provide two shots. This class is used for both loading of ammunition for sniper rifle and shotgun as we have considered the principle Don’t Repeat Yourself. This is because, both sniper and shotgun use similar way to load the ammunition. Therefore, there is no need to separate loading ammunition actions for both rifle and shotgun, which reduces duplication of code.

**RifleSelectTargetAction**This class is inherited from the Action class. This class searches the area of 50 range from the location of the Player for any Zombies in range and shows a menu for all actors in range for the Player to choose to target.

Design Rationale  
We use the breadth first search algorithm in order to search through the map. If a Zombie is in range, another action class called TargetedAction will be created. This newly created class will be added to the list of actions which is available for the Player to choose. After the search is completed, we will have a list of targets, another submenu will show up, allowing the Player to choose the targeted Zombie. We used a sub menu so that the console won’t show to much options to the Player which makes it hard to select. The design principle used here is Declare things in the tightest possible scope. This is seen on the methods implementing breadth first search algorithm. Those methods are declared private which reduces the risk of other classes depending on them.

**TargetedAction**This class is inherited from the Action class. This class displays another sub menu to the Player, which gives the Player the option to choose whether to shoot the target or to aim the target for another round.

Design Rationale  
When this class executes, an action class called RifleShootAction is created. This newly created class will be added to the list of actions. The Player will be provided another option which is to aim the target. This is done by return the instance of this class to the user again. If the user has successfully aimed the target for two rounds. On the third round, the Player will be only provided the option to shoot the target. If the Player is distracted or does any other action during the aiming process, Player will lose their target, this is handled in the Player class.

**RifleShootAction**This class inherits from the AttackAction class. This class shoots the targeted Zombie.

Design Rationale  
When this class executes, it will consider how many rounds of aims did the Player spend time doing. The probability of shooting the target is based on the number of aims. We have choose to inherit from the AttackAction class because of the principle, Don’t Repeat Yourself. This is because, the method isTargetConscious is used by AttackAction itself and this class to check whether the targeted actor is still conscious after it is hurt. Therefore, we can reuse the block of code, thus reducing duplication of code. Meanwhile the method updateGunBullet is hosted in AttackAction. This method will update the bullet count after each shot.

**ShotgunDirectionAction**This class inherits from the Action class. This class uses the exits of the Player to show a sub menu for the Player to choose the direction the Player wants to shoot using the shotgun.

Design Rationale  
When this class executes, it will get the exits of the Player and creates a new action class called ShotgunShootAction and adds the action to the list of actions. In the end showing a sub menu for the Player to choose the direction to shoot.

**ShotgunShootAction**This class inherits from the AttackAction class. This class shoots at the targeted direction of the Player.

Design Rationale  
When this class executes, it will search through the targeted direction range of 3 squares. Any actors within the range will be shot with a probability of a successful shot each for 75%. Similarly in RifleShootAction, the principle Don’t Repeat Yourself also applies here. For example is the isTargetConscious method discussed in RifleShootAction. Another example is the updateGunBullet method that is hosted in the AttackAction class. This is because both RifleShootAction and ShotgunShootAction use this block of code. Thus, reducing repeated code.

**QuitGameAction**This class inherits from the Action class. This class provides the Player the option to quit the game anytime.

Design Rationale  
When this class executes, it basically removes the player from the game, which stops the game.

**Player**We have changed code in this class.

Design Rationale  
We have added the respective actions which allow the Player to load and shoot a gun if the respective guns are in the Player’s inventory. This class also tracks the players concentration during the aiming of using the sniper riffle. This is done by using a variable called concentrationAffected which holds a Boolean value. Throughout the implementation of code we have followed a principle which is Avoid variables with hidden meanings. We will take this class as for example, the variable concentrationAffected.

**WeaponGun**This class inherits from the WeaponItem class. This is an abstract class

Design Rationale  
We have declared this class as abstract because we do not want it to be instantiated. Other than that, we considered that both Rifle and Shotgun class uses similar capabilities when being instantiated. Therefore, we make those two class to inherit from this class, reducing repeated code.

**Rifle & Shotgun**Both classes inherits from the WeaponGun class. Both classes represent the sniper rifle and the shotgun respectively.

**RifleAmmunition & ShotgunAmmunition**Both classes inherits from the PortableItem class. Both classes represent the rifle ammunition and the shotgun ammunition respectively.

**NewWorld**This class inherits from the World class.   
  
Design Rationale  
We override only 2 methods from the World class, which reduced the repeated codes. In this class, we initialized MamboMarie directly with some checking conditions. This is because we would like to reduce dependencies between this functionality and other classes. Then, we also implemented the game-ending conditions in this class. We’d used boolean variables to check the conditions instead of literals.   
  
**MamboMarie**This class inherits from the ZombieActor class.

Design Raitonale  
This class shares common characteristics from other actor classes. So again to reduce the repeated codes we inherits it from ZombieActor. We initialized a PortableItem along with the initialization of MamboMarie. The PortableItem is solely used to keep track of turns so MamboMarie will perform certain actions based on it. We keep the PortableItem in the tightest possible scope, as despite being a PortableItem, we’d made it undroppable and will be destroyed when MamboMarie is dead. Just serves as what it meant to be.  
  
**ChantBehaviour**This class implements Behaviour interface.  
  
Design Rationale  
This class has dependencies with 2 Action classes. The classes are ChantAction and VanishAction. We did not make a VanishBehaviour as we would like to group these classes (those that depend on each other) in an encapsulation boundary. This helps in reducing number of classes which result in minimizes of repeated codes.

**ChantAction & VanishAciton**These classes inherit from Action class.  
  
Design Rationale  
As always, to avoid repeated codes, we implemented these classes by inheriting from Action class. Other than that, we did not use any literals in these classes, instead we used enum object which is MamboMarieCapability for condition checking.