Classes Responsibilities & Design Rationale

Going to Town

Functionality: Player can travel between the compound and town maps using vehicle

New classes	Class Responsibility	Design Rationale
ZombieGameMapFactory	Abstract class that defines the models (size, terrain, actors etc.) of the maps in the Zombie game world. Factory for creating maps of the game world. - Has **private** attributes: 1. map: the map terrain expressed as list of strings 2. humansOnMap: a list of names of the humans that appear on the map 3. zombiesOnMap: a HashMap that stores the names and location coordinates of all zombies on the map 4. farmersOnMap: a HashMap that stores the names and location coordinates of all farmers on the map 5. mapName: the name of the map - **protected** default methods: 1. setMapString(String[] mapString): takes in a string that describes the map terrain and assigns it to map.	This class is created because there are now two different maps in the game system. Although the two maps are different, fundamentally they share the same things: map shape, terrain, actors and items. As such this class is introduced and it has a few private attributes that store the map terrain and actors that are on the map. These attributes (map, humansOnMap, zombiesOnMap, farmersOnMap and mapName) are declared as private to **minimise dependencies across encapsulation boundaries**. This class also has multiple default methods to remove duplicate codes in CompoundMapFactory and TownMapFactory. This follows the design principle: **Don't Repeat Yourself**. Besides, in all the default setter methods, a new instance of the argument will be created before the value is being assigned into the private attribute. In the default getter methods, the values returned are made to be unmodifiable. These prevent the value of the argument from being modified other than calling methods in this class and hence **prevents privacy leak issues**.

	2. setHumans(String[] humanNames): takes in a list of human names and assigns it to humansOnMap. 3. setZombies(HashMap<String, Integer[]> zombies): takes in a HashMap that contains zombie information and assigns the HashMap to the private attribute, zombiesOnMap. 4. setFarmers(HashMap<String, Integer[]> farmers): takes in a HashMap that contains farmer information and assigns the HashMap to the private attribute, farmersOnMap. 5. setMapName(String name): takes in the name of the map and assigns it to the private attribute, mapName. 6. getMapString(): returns an unmodifiable list of strings that represents the map terrain 7. getHumans(): returns an unmodifiable list of strings that represent the human names 8. getZombies(): returns an unmodifiable Map object that contains all the zombies' names and their location coordinates 9. getFarmers(): returns an unmodifiable Map object that contains all the farmers’ names and their location coordinates	

	10. getMapName(): returns the name of the map	
CompoundMapFactory extends ZombieGameMapFactory	Class that defines the compound map in the game world. Factory for creating the compound map. - Constructor: 1. Call setter methods in the ZombieGameMap class to define the terrain of the map as well as actors that appear on the map. 2. Name of the map is “compound”	This class only defines the details of map terrain and actors that will appear on the map and does not directly create the ZombieGameMap, Human, Farmer and Zombie instances so that the dependencies between classes can be reduced. This follows the design principle: **Reduce Dependencies**.
TownMapFactory extends ZombieGameMapFactory	Class that defines the town map in the game world. Factory for creating the town map. - Constructor: 1. Call setter methods in the ZombieGameMap class to define the terrain of the map as well as actors that appear on the map 2. Name of the map is “town”	This class only defines the details of map terrain and actors that will appear on the map and does not directly create the ZombieGameMap, Human, Farmer and Zombie instances so that the dependencies between classes can be reduced. This follows the design principle: **Reduce Dependencies**.
ZombieGameMap extends GameMap	Class that represents the maps in the Zombie game world. - Has **private** attributes: 1. mapName: the name of the map, e.g. compound, town 2. actorsOnMap: A list of actors that are currently on the map - 3 different constructors based on the constructors in	This class is created so that methods in the GameMap class can be overloaded. Attributes like mapName and actorsOnMap are declared as private attributes to **minimise dependencies across encapsulation boundaries**. Besides, getActorList() returns an unmodifiable actorsOnMap to **prevent privacy leak**.

	GameMap class - Override methods: 1. addActor(Actor actor, Location location): Add a new Actor at the given location and also add the Actor into actorsOnMap 2. removeActor(Actor actor): Remove the given Actor from the system and remove Actor from actorsOnMap as well. - Getter methods: 1. getMapName(): allows other classes to retrieve the name of the map 2. getActorList(): allows other classes to retrieve a list of actors that are on the map	
Vehicle extends Item	Represents the vehicle that is used for Player to travel between different maps. - Has **private** attributes: 1. destinationMap: the map that player will reach if he uses this vehicle - Constructor takes in the following parameter: 1. destination: a ZombieGameMap instance, the map that Player will reach after travelling using this vehicle - Override method: 1. getAllowableActions(): create a new Actions	The attribute destinationMap is set as a private attribute to **minimize dependencies across encapsulation boundaries**. In our current game design, there is a vehicle on each map. The vehicle on compound map will transport the Player to town map whereas the vehicle on town map will transport the Player to compound map.

	object that contains TravelAction in it, and assigns this Actions object as the vehicle's allowableActions, then returns an unmodifiable version of allowableActions.	
TravelAction extends Action	Represents the Player's action to travel between the compound map and town map. - Has **private** attributes: 1. destinationMap: the map that player will reach after travelling 2. destinationName: the name of the destination map - Constructor takes in the following parameters: 1. destination: a ZombieGameMap instance, the map that Player will reach after travelling using this vehicle 2. The constructor will then call the method getMapName() to retrieve the name of destination map and stores the string into destinationName - Has methods: 1. execute(Actor actor,GameMap actorCurrentMap): Creates a SuitableLocation instance and finds a suitable location, i.e. location that does not contain any other actor and has a passable terrain. Remove the actor	In the previous design, there is no class that is responsible to move an actor across different maps. Following the design principle: **Single Responsibility Principle**, this responsibility should not be added to other classes. Hence, this class is created. The attributes destinationMap and destinationName are set as private to **minimize dependencies across encapsulation boundaries**.

	(Player) from the current map and add the actor onto the destination map at the found suitable location. 2. menuDescription(Actor actor): returns a description string for the action, e.g. travels to town map	
SuitableLocation	This class is responsible for finding a suitable location to add an Actor onto a map. - Has **private** attributes: 1. destinationMap: the map which the actor will be added to 2. expectedNewLoc: the new location that the actor is expected to be at, initialized as null 3. actorToAdd: the actor which needs to be added onto the map - Constructor takes in the following parameters: 1. map: the map where the actor will be added onto, stored in private attribute, destinationMap 2. newLocation: the location that the actor is expected to be at when actor is added onto destinationMap, stored in private attribute, expectedNewLoc	It is noticed that several classes in the game system will need to repeatedly find a location that is suitable for an actor to be added onto map. These classes are HumanCorpse and TravelAction. Finding a suitable location for an actor to be added is inevitable because sometimes, the expected new location of the actor has already been occupied by the other actor. Under such circumstances, the nearest location where the actor can be added needs to be determined. However, implementing this “finding suitable location” functionality in multiple classes will result in repeated code. Hence, following the design principle: **Don’t Repeat Yourself**, this class is created to refactor duplicated code in these classes. Besides, the attributes (destinationMap, exepctedNewLoc, actorToAdd) and methods (withinMapRange, findNewLocation, computeNewLocation, suitableLocAroundExpectedLoc) are declared as private attributes/methods to **minimize**

	3. actor: the actor which needs to be added onto the map, stored in private attribute, actorToAdd - Has **public** method: 1. findSuitableLocation(): Determines a suitable location for the actor to be added onto the map. It will call different private methods to compute the suitable location depending on the value of expectedNewLoc. A suitable location is a location that does not contain any other actor and has a passable terrain. In this case, since expectedNewLoc will be set (not null), the method suitableLocAroundExpectedLoc() will be called. - Has **private** methods: 1. suitableLocAroundExpectedLoc(): Determines whether the actor can be added to the map at expectedNewLoc. If the given location is not suitable, the method will determine and return the nearest suitable location for the actor to be added onto the map. To do this, it calls the method findNewLocation. 2. findNewLocation(Location startingLoc, int[][] nearbyCoords): This method is called when the initial location where the actor is expected to be at is not suitable for the actor to be added. This method will continuously find the next nearest neighbour location that is	**dependencies across encapsulation boundaries**. In the cases where the actor cannot be added onto a map at the given location, adjacent locations need to be identified. By instinct, one might think of using the method getExits() and getDestination() to get these adjacent locations. However, if no suitable location can be found in the first layer of exits, the function will need to recursively find the adjacent exits of each of the Exit objects until a suitable location is found. It is found that this implementation will make the code ridiculously lengthy and incomprehensible. Hence, for the code to be easier to be understood, it is decided to use simple mathematical calculation to obtain the adjacent locations. Nevertheless, the complexity of finding a suitable location for the actor to be added is **encapsulated** in this class. Users will be able to get a suitable location to add an actor through this class easily without understanding the logic behind the code.

	suitable for the actor to be added. To do this, it calls the method computeNewLocation. 3. computeNewLocation(Location startingLoc, int[] coordPair): Computes the coordinates of neighbour location and determines if the location is suitable for the actor to be added. 4. withinMapRange(int x, int y): Determine if a location is within the X, Y range of the destination map. Returns true if the coordinates (x, y) is within the map boundary range.	

Modified classes	Class Responsibility	Design Rationale
Application	The driver of the whole game. - Modified constructor to create two different maps and call its own private methods to create and add actors/items onto the maps - Creates ZombieGameMap instances instead of GameMap instances - Add **private** static methods: 1. addHumansOntoMap(GameMap gameMap, List<String> humans): creates and adds human instances to gameMap using the name	Creating ZombieGameMap instances instead of GameMap instances does not break other part of codes that expect a GameMap instance as an input because ZombieGameMap is a GameMap subclass and hence a ZombieGameMap instance can be used to replace the GameMap instance due to **Liskov Substitution Principle**. Besides, it is decided to create all maps and actors instances in this driver class so that dependencies between map factory classes and other classes can be minimised. This follows the design principle: **Reduce dependencies**.

	listed in the list humans. 2. addZombiesOntoMap(GameMap gameMap, Map<String, Integer[]> zombies): creates and adds zombie instances to gameMap at the location coordinates specified in HashMap zombies. 3. addFarmersOntoMap(GameMap gameMap, Map<String, Integer[]> farmers): creates and adds farmer instances to gameMap at the location coordinates specified in HashMap farmers.	
HumanCorpse extends PortableItem	Represents the human corpse in the Zombie game world. - Modified method: 1. revive(Location corpseLoc): create a SuitableLocation instance to get the location where the revived corpse is to be added onto the map.	The code in revive() that is used to find a suitable location for the revived Zombie to be added has been removed, instead a SuitableLocation instance is used to do this job. This is to prevent duplicated code because there are multiple classes that need to repeatedly find a suitable location for an actor to be added. This follows the design principle: **Don't Repeat Yourself**.

Mambo Marie

Functionality: Mambo Marie has 5% to appear at map border (if she has not appeared on any map)

New classes	Class Responsibility	Design Rationale
Priestess extends ZombieActor	Represents the priestess, which is the source of the local zombie epidemic in the zombie game world.	In our current game design, there is only one priestess, that is Mambo Marie. Mambo Marie cannot attack humans, she only knows chanting and creating new

	- Constructor: 1. The name of priestess is fixed as “Mambo Marie”, because there is currently only one priestess in the game. - Has **private** attribute: 1. appearTurn: keeps track of the number of turns that has passed since Mambo Marie's has appeared in the map 2. SPAWN_PROBABILITY: a constant that represents the probability for the Mambo Marie to spawn on the map every turn, which is 5% - Has capabilities: 1. UNDEAD: priestess is on the same team as Zombie 2. PRIESTESS: to indicate that this actor is the priestess - Override method: 1. getAllowableActions(Actor otherActor, String direction, GameMap map): returns an empty Actions instance. 2. playTurn(Actions actions, Action lastAction, GameMap map, Display display): increments appearTurn by 1 every time this method is executed. Depending on the value of appearTurn, the priestess will carry out different actions.	zombies. Therefore, the method getAllowableActions() in ZombieActor class is overridden because Priestess does not have any AttackAction. The attribute appearTurn is declared as private to **minimize dependencies across encapsulation boundaries**. Besides, there is a 5% chance for Priestess (Mambo Marie) to spawn on the map every turn (provided that it has yet to appear in any map). This value is being stored in the constant SPAWN_PROBABILITY instead of being hardcoded in methods to **reduce implicit dependencies on literals**. The method spawnLocation() is added into the Priestess class, following the design principle: **Tell, Don’t Ask**. This design makes the Priestess object to tell the ZombieWorld where it should appear on the map and hence, ZombieWorld class will not need to bother about how to find a suitable location to add the Priestess instance.

	- New methods: 1. spawn(): Generate a random Double value and convert it into percentage. If this value is less than or equal to SPAWNING_PROBABILITY, then the method will return true. Otherwise, it returns false. 2. spawnLocation(GameMap destinationMap): Returns the location where the Priestess (Mambo Marie) will be added on the destination map. This location can be determined using a SuitableLocation instance.	
SuitableLocation	This class is responsible for finding a suitable location to add an Actor onto a map. - Another constructor that takes in the following parameters: 1. map: the map where the actor will be added onto and it will be stored in the private attribute, destinationMap 2. actor: the actor which needs to be added onto the map, will be stored in the private attribute - Has method: 1. findSuitableLocation(): Determines a suitable location for the actor to be added. In the case where expectedNewLoc is not defined (null), the actor is expected to be added at map border. Hence, the method suitableLocAlongBorder() is called.	Priestess (Mambo Marie) also needs to be added onto the map at a suitable location, but adding Mambo Marie onto the map has one more constraint, that is, the Priestess can only appear at the edge of the map. However, despite this part, the part of finding new suitable locations is almost the same as “finding suitable location” in the SuitableLocation class. Hence it is decided that the SuitableLocation class should be extended so that it can determine a suitable location for Mambo Marie to be added onto the map as well. This follows the design principles **Single Responsibility Principle** and **Don’t Repeat Yourself**. Therefore, another constructor that takes in two locations is created. This constructor should be used when the actor is expected to be added at a location along the map edge/border, just like the case of Priestess (Mambo Marie).

	- New **private** methods: 1. suitableLoc AlongBorder(): Determines whether the actor can be added to the map at the top left or bottom right corner of the map. If the actor cannot be added to any of these two locations, this method will find and return a location along the map border which is suitable for the actor to be added. A suitable location is a location that does not contain any other actor and has a passable terrain.	Besides, another private method, suitableLocAlongBorder() has also been added into the class. The complexity of finding a suitable location along the map edges is all **encapsulated** within this method. Users can easily obtain the suitable location by using this class and the method findSuitableLocation() without understanding the code logic.
ZombieWorld extends World	Class that represents the whole game world. - Has **private** attribute: 1. mamboMarie: stores the single Priestess instance - Override method: 1. run(): At the beginning of the game (before the while loop), the method will create an instance of Priestess.In each while loop iteration, the method will check whether Mambo Marie has already been added on any map. If Mambo Marie has not yet appeared in any map and Mambo Marie is still alive and has a chance to spawn, this method will add the priestess onto the map player is currently on at a suitable location. This suitable location can be determined by calling the spawnLocation() in the Priestess class.	In our game design, Mambo Marie can spawn on any map, i.e. it can appear in both compound and town maps. However, it can only be at either one of the maps at one time. As such, we implement the system such that the Mambo Marie will spawn on the map that the Player is currently on, provided that it has not appeared in any of the maps yet.

Modified classes	Class Responsibility	Design Rationale
ZombieCapabaility	Stores all the enumeration values that are used in the Zombie world. - New enum: 1. PRIESTESS: indicate that the actor that has this capability is the priestess	This capability is added so that we can decide whether the DEAD capability should be added to the target when it is killed in HumanAttackAction class. DEAD capability should only be added to Priestess instance when it is killed.

Functionality: Mambo Marie chants and creates 5 new Zombie every 10 turns

New classes	Class Responsibility	Design Rationale
Priestess extends ZombieActor	Represents the priestess, which is the source of the local zombie epidemic in the zombie game world. - Has **private** attribute: 1. CHANT_INTERVAL: a constant that represents the turn interval for Mambo Marie to chant and create more zombies, which is 10 turns. - Override method: 1. playTurn(Actions actions, Action lastAction, GameMap map, Display display): This method will check if Mambo Marie's appearTurn has reached 10/20/30 every time this method got executed. If it is the 10th, 20th and 30th turn, a ChantingAction instance will be returned. Else, the parent class's playTurn() will be called and either a MoveActorAction or DoNothingAction	The priestess will chant every 10 turns.This value is being stored in the constant CHANT_INTERVAL instead of being hardcoded in methods to **reduce implicit dependencies on literals**.

	will be returned.	
ChantingAction extends Action	Represents the chanting action of priestess, Mambo Marie. - Has **private** attributes: 1. rand: stores a Random object which will be used to generate random number in the methods within this class. 2. ZOMBIE_QUANTITY: a constant that stores the number of zombies that will be created every time the chanting action is carried out. The value is fixed at 5 now. - Has methods: 1. execute(Actor actor, GameMap map): creates 5 new Zombie instances and adds them onto map at suitable random locations. Suitable locations are locations that are within map boundaries and do not contain any other actor. 2. menuDescription(Actor actor): returns the string "Mambo Marie chanted spells and five Zombie is created! Your doomday is near!" - Has **private** methods: 1. generateZombieName(): Generates random string to be used as the name of the new Zombie. The length of random string generated will be within 5 to 10 characters.	5 new zombies will be created every time chanting action is being carried out. The number of zombies is being stored in the constant ZOMBIE_QUANTITY instead of being hardcoded in methods to **reduce implicit dependencies on literals**. The methods generateZombieName() and generateZombieLoc() will only be used within this class, hence they are declared as private methods. This **minimizes dependencies across encapsulation boundaries**.

	2. generateZombieLoc(GameMap map): Generates random location where the new Zombie can be added onto the map.	

Functionality: Mambo Marie disappears after 30 turns

New classes	Class Responsibility	Design Rationale
Priestess extends ZombieActor	Represents the priestess, which is the source of the local zombie epidemic in the zombie game world. - Has **private** attributes: 1. VANISH_INTERVAL: a constant that represents the turn interval for Mambo Marie to vanish, which is 30 turns. - Has method: 1. getAppearTurn(): returns the value of private attribute, appearTurn. 2. resetAppearTurn(): resets the value of appearTurn back to its initial value, 0. 3. getVanishInterval(): returns the value of the constant VANISH_INTERVAL.	The Priestess (Mambo Marie) will disappear after 30 turns, ie. vanish at the 31th turn. This value is being stored in the constant VANISH_INTERVAL instead of being hardcoded in methods to **reduce implicit dependencies on literals**.
ZombieWorld extends World	Class that represents the whole game world. - Override method: 1. run(): In each while loop iteration, the method	After 30 turns, Mambo MArie will be removed from the map that she is currently on. Then, depending on Mambo Marie's alive status, the system will either continue to spawn her onto maps (if she is still alive) or

	will check if Mambo Marie has already been added onto any map. If Mambo Marie has already appeared, this method will check whether Mambo Marie's appearTurn is equal or greater than her VANISH_INTERVAL. If it has, the method will remove Mambo Marie from the map she is currently on and reset the Mambo Marie's appearTurn value.	stop spawning her (if she is dead already).

Functionality: Mambo Marie will not reappear anymore once it's been killed

New class	Class Responsibility	Design Rationale
ZombieWorld extends World	Class that represents the whole game world. - Override method: 1. run(): In each while loop iteration, the method will check whether Mambo Marie is still alive. If Mambo Marie is still alive, then the method will continue to spawn Mambo Marie on map. However, if Mambo Marie has already been killed, then the method will make sure not to spawn Mambo Marie on map anymore.The method can determine Mambo Marie's alive status by using the capability DEAD.	In our current game design, Mambo Marie will stop appearing on the map once she has been killed.

Modified classes	Class Responsibility	Design Rationale
ZombieCapability	Stores all the enumeration values that are used in the Zombie world.	This capability is added so that we can know whether the Priestess has been killed already or not. This helps

	- New enum: 1. DEAD: to indicate that the actor that has this capability is already dead.	in deciding whether to continue spawning the Priestess onto map.
HumanAttackAction extends AttackAction	Represents the actions that will be taken by Human to attack Zombie or Mambo Marie. - Modified method: 1. killTarget(): checks whether the target killed is a priestess or not using the capability PRIESTESS. If it is, then add the DEAD capability to the target instance.	Adding the DEAD capability to Priestess when it is killed is important so that the system can know that it should stop spawning Mambo Marie.

Ending the Game

Functionality: Player is able to quit the game voluntarily

New classes	Class Responsibility	Design Rationale
QuitGameAction extends Action	Represents the action to quit game. - Methods: 1. execute(Actor actor, GameMap map): Add a capability QUIT_GAME to the actor (which is the Player), and calls menuDescription() to return a string. 2. menuDescription(Actor actor): return “quit game”	In the previous game design, there’s no class that enables the Player to end the game voluntarily. Therefore, it was decided to create a new class for this quit game action so that it follows the design principle: **Single Responsibility Principle**.

	3. hotKey(): associates the key “0” with the quit game action.	
ZombieWorld extends World	Class that represents the whole game world. - Override method: 1. stillRunning(): check if the Player has ZombieCapability.QUIT_GAME. If Player has the capability, it means that the player has chosen to quit the game. Hence, the method will return false.	Since we cannot change the code in the engine package directly, inheritance and method overriding is the only way for us to modify the method in classes that are in the engine package. Therefore, this class is created. Following the design principle **Don’t Repeat Yourself**, we only override necessary methods to fulfil the required game functionality. To illustrate, the checking of whether the player has chosen to quit game or not can also be done in the run() method, but doing so will lead to many duplicated code, hence we decide to only override stillRunning() and do the checking in this method.

Modified classes	Class Responsibility	Design Rationale
ZombieCapability	Stores all the enumeration values that are used in the Zombie world. - Add enum: 1. QUIT_GAME: used to determine if the Player has chosen to quit the game	All the constants stored in ZombieCapability act as tags which are used to identify specific Objects throughout the game system. Thus, can be used to compare and determine actions to be performed onto the specific objects and **reduces the implicit dependency on literals.**
Player	Represents the player in the game. - Modified method: 1. playTurn(Actions actions, Action lastAction,	SInce quitting the game is an action that can only be carried out by the Player, it is being created inside the Player class.

	GameMap map, Display display): Added QuitGameAction into the actions list, so that “quit game” can be displayed as an option in the menu for the Player to choose.	
Application	The driver of the whole game. - Modified constructor: 1. Create ZombieWorld instance instead of World	Creating ZombieWorld instance instead of World instance does not break other part of codes that expect a World instance as an input because ZombieWorld is a World subclass and hence a ZombieWorld instance can be used to replace the World instance due to **Liskov Substitution Principle**.

Functionality: Player loses and wins

New classes	Class Responsibility	Design Rationale
ZombieWorld extends World	Class that represents the whole game world. - Has **private** attributes: 1. playerDead: a Boolean flag that is used to indicate whether the player has died. Initialized to false. 2. gotHuman: a Boolean flag that is used to indicate whether there's any human (excluding Player) left in the game. 3. gotZombie: a Boolean flag that is used to indicate whether there's any Zombie or Mambo Marie left in the game.	The attributes playerDead, gotHuman and got Zombie are declared as private, which is the tightest visibility modifier, so that it follows the design principle: **Minimise dependencies that cross encapsulation boundaries**. In our game design, the game will only end under the following four conditions: 1. The Player chooses to quit the game voluntarily 2. There’s no Human (excluding Player) in the compound map. Player loses in this case. 3. Player dies. Player loses in this case. 4. There’s no Zombie in the compound map and Mambo Marie has been killed. Player wins in this case.

	- Override method: 1. stillRunning(): Checks if the game world still contains Player, if no, then set playerDead to true and return false. If the Player is still around, the method will check if there's still any human (excluding player) or zombie in the compound map. This can be done by looping through actorsOnMap list of the compound map and checking the capability of each actor in the list. If there's no other Human in the game, gotHuman will be set to false and method will return false. If there's no Zombie remaining in the world, gotZombie will be set to false and the method will return false. 2. endGameMessage(): Returns a message depending on how the game was ended. If the player is dead or no Human is left in the game, it will return "player loses\nGame Over". If there's no Zombie left in the game and Mambo Marie is dead, it will return "player wins\nGame Over". The method can know if the player is dead or alive and if there's still anyHuman or Zombie around by checking the three boolean flags: playerDead, gotHuman and gotZombie. It can determine whether Mambo Marie has been killed or not by checking its DEAD capability. - New **private** method: 1. findCompoundMap(): determines and returns	

	the compound object from the gameMaps list by checking the name of the map.	

New weapons: Shotgun and Sniper Rifle

Modification for action filtering

Modified classes	Class Responsibility	Design Rationale
ItemInterface	An interface which provides the ability to add methods to Item, without modifying code in the engine, or downcasting references in the game. - Add **default** methods: 1. getAllowableActions(Actor actor, GameMap map): A default method that returns an unmodifiable list of Action. By default returns an empty list of Action. Can be overridden in subclasses which need to perform condition checking using Actor or GameMap before returning the list of actions allowable for the item. 2. isInPlayerInventory(Actor actor, GameMap map): A default method that checks if the item is in a Player's inventory.	Based on **Single Responsibility Principle**, the new default method getAllowableActions with 2 additional arguments are added into ItemInterface, to allow Item itself to be able to check the specific condition and return the allowable actions back to its client. This removes the extra responsibility for the client class of Item to check the condition and filter out the allowable actions for the Item. The isInPlayerInventory default method is added as most of the Items only provide actions on them to their clients (in this case Player), if the items are in the Player inventory. Thus, based on the **Don't Repeat Yourself** principle, this avoids duplicated code in Item subclasses such as Limb and Food to check if the item is in the Player's inventory to return specific allowable actions. The initial less ideal approach which makes Action to have capabilities to identify specific Action returned from Item and remove them in other classes are removed and replaced by a better approach which makes the Item to be responsible for their own allowable actions.
Limb extends PortableItem	A class which represents the limb that drops off from the Zombie - Override method: 1. getAllowableActions(Actor actor, GameMap	

	map): Return a list consisting of CraftWeaponAction only if the actor is Player and has the limb in his inventory. This is to make sure only Player can perform CraftWeaponAction on the limb and the limb must be picked up before it is able to be crafted.	
Food extends PortableItem	Represents food in the Zombie game world. - Override method: 1. getAllowableActions(Actor actor, GameMap map): Return a list consisting of EatingAction only if the actor is Player and has the food in his inventory. This is to make sure only Player can perform EatingAction for the food and the food must be picked up before it is able to be eaten.	
ActionInterface	An interface that provides the ability to add methods to Action, without modifying code in the engine, or downcasting references in the game. - Remove default methods: 1. hasCapability(Enum<?> capability) 2. addCapability(Enum<?> capability) 3. removeCapability(Enum<?> capability)	
CraftWeaponAction	A class which represents action for crafting zombie arm or leg into club or mace.	

	- Remove overridden methods; 1. hasCapability(Enum<?> capability) 2. addCapability(Enum<?> capability) 3. removeCapability(Enum<?> capability)	
EatingAction extends Action	A class which represents an action to eat food. - Remove overridden methods; 1. hasCapability(Enum<?> capability) 2. addCapability(Enum<?> capability) 3. removeCapability(Enum<?> capability)	

New feature: Ammunition Box

New classes	Class Responsibility	Design Rationale
AmmunitionBox extends PortableItem	A class which represents the ammunition box. - Has **private** attribute: 1. bulletLeft: An integer representing the number of bullet left - Constructor: 1. AmmunitionBox(): Create an ammunition box which is also a portable item, has 20 bullets in it initially and has ZombieCapability.AMMUNITION_BOX - New methods: 1. isEmpty(): Check if the ammunition box is	Based on the **minimize dependencies that cross encapsulation boundaries** principle, the bulletLeft attribute in AmmunitionBox is set as private.

	empty. 2. useBullet(): Use one bullet in the ammunition box. Decrement bullet number left by one. 3. getBulletLeft(): Get the number of bullets left in the ammunition box.	

Modified classes	Class Responsibility	Design Rationale
ActorInterface	This interface provides the ability to add methods to Actor, without modifying code in the engine, or downcasting references in the game. - Add **default** method: 1. getAmmunitionBoxInInventory(): Get the first ammunition box in actor inventory if there is any.By default, this method returns null.	The default method getAmmunitionInInventory is added into ActorInterface to avoid downcasting from Actor to ZombieActor in order to make ZombieActor to be able to get the first ammunition box in inventory if there is any.
ZombieActor extends Actor	Base class for Actors in the Zombie World - Override method: 1. getAmmunitionBoxInInventory(): Get the first ammunition box in the actor's inventory if there is any.	
ZombieCapability	Class that handles all enumerations used in the game.	In order to **reduce implicit dependency on literals** and **avoid excessive use of literals,** a constant

	- New enum: 1. AMMUNITION_BOX: to identify ammunition box	ZombieCapability.AMMUNITION_BOX is used to represent the AmmunitionBox instead of using the string name of the AmmunitionBox.

New features: Long-ranged weapon and short-ranged weapon

New classes	Class Responsibility	Design Rationale
LongRangedWeapon extends WeaponItem	An abstract base class for long-ranged weapons. - Has **private** attribute: 1. ammunitionBox: An AmmunitionBox which contains bullets for the LongRangedWeapon - Constructor: 1. LongRangedWeapon(String name, char displayChar, int damage, String verb): Create a WeaponItem with capability of LONG_RANGED_WEAPON. - New methods: 1. shoot(): Shoot using the long-ranged weapon and return a string reporting the number of bullets left. Use one bullet in the ammunition box of the long-ranged weapon 2. refillAmmunitionBox(): Refill the ammunitionBox of the long-ranged weapon. Assign a new AmmunitionBox instance to the ammunitionBox attribute of the long-ranged weapon.	There are 2 groups of WeaponItem, long-ranged weapons and short-ranged weapons. The long-ranged weapon has some additional functionalities that are not available for short-ranged weapons such as having ammunitionBox, able to be used for shooting and to be refilled with bullets. Thus, new abstract classes LongRangedWeapon and ShortRangedWeapon are created as base classes for these 2 types of weapon. Abstract classes are used so that the base classes cannot be instantiated. Based on the **minimize dependencies that cross encapsulation boundaries** principle, the ammunitionBox attribute in LongRangedWeapon is set as private. Based on the **Don't Repeat Yourself** principle, functionalities that are common to all long-ranged weapons are declared in the LongRangedWeapon base class such as shoot(), refillAmmunitionBox() and hasBullet(). This avoids duplicated code in all the LongRangedWeapon subclasses.

	3. hasBullet(): Check if the long-ranged weapon has bullets left to use.	In order to **reduce implicit dependency on literals** and **avoid excessive use of literals,** constant ZombieCapability.LONG_RANGED_WEAPON and ZombieCapability.SHORT_RANGED_WEAPON are used to represent all long-ranged weapons and short-ranged weapon instead of using their string name.
ShortRangedWeapon extends WeaponItem	An abstract base class for short-ranged weapons. - Constructor: 1. ShortRangedWeapon(String name, char displayChar, int damage, String verb): Create a WeaponItem with capability of SHORT_RANGED_WEAPON.	

Modified classes	Class Responsibility	Design Rationale
ZombieActor extends Actor	Base class for Actors in the Zombie World. - Override method: 1. getWeapon(): Get the weapon this Actor is using.If the current Actor is carrying weapons, returns the first short-ranged weapon in the inventory. Otherwise, returns the Actor's natural fighting equipment e.g.fists.	The getWeapon method of ZombieActor is overridden to never return a long-ranged weapon in order to restrict all ZombieActor from using long-ranged weapons as short-ranged weapons to attack the adjacent target.
Club extends ShortRangedWeapon	A class which represents a club crafted from a zombie's dropped arm. - Modified constructor: 1. Club(): Create a club using ShortRangedWeapon constructor with required	Club, Mace and Plank have the functionalities as a short-ranged weapon. Based on **Liskov Substitution Principle**, by making Club, Mace and Plank subclasses of ShortRangedWeapon, Club, Mace and Plank acquire all the functionalities of ShortRangedWeapon and the methods can be invoked on a reference of type

	arguments name of "club", displayChar of '~', damage of 30 and verb to use the mace which is "uses club to hit".	LongRangedWeapon.
Mace extends ShortRangedWeapon	A class which represents a mace crafted from a zombie's dropped leg. - Modified constructor: 1. Mace(): Create a mace using ShortRangedWeapon constructor with required arguments name of "mace", displayChar of '$', damage of 40 and verb to use the mace which is "use mace to hit".	
Plank extends ShortRangedWeapon	A primitive weapon. - Modified constructor: 1. Plank(): Create a Plank using ShortRangedWeapon constructor with required arguments name of "plank", displayChar of ')', damage of 20 and verb to use the mace which is "whacks".	
ZombieCapability	Class that handles all enumerations used in the game. - New enum: 1. LONG_RANGED_WEAPON: to identify long-ranged weapon 2. SHORT_RANGED_WEAPON: to identify	These 2 capabilities are added to differentiate between long-ranged weapons and short-ranged weapons.

	short-ranged weapon	

New features: shotgun and sniper rifle

New classes	Class Responsibility	Design Rationale
Shotgun extends LongRangedWeapon	A class which represents a shotgun. - Constructor: 1. Shotgun(): Create a shotgun weapon item and has capability of LONG_RANGED_WEAPON. - Override method: 1. getAllowableActions(Actor actor, GameMap map): Return a list consisting of ShootingAction only if the actor is Player and has the shotgun which still has bullets in his inventory. This is to make sure only Player can perform ShootingAction using shotgun which still has bullet and the shotgun must be picked up before it is able to be used.Use parent class LongRangedWeapon.getAllowableActions to get RefillBulletAction if the shotgun in the actor's inventory does not have any bullet left and there is at least one ammunition box in the actor's inventory.	Shotgun and Sniper Rifle have the functionalities as a long-ranged weapon. Based on **Liskov Substitution Principle**, by making Shotgun and SniperRifle subclasses of LongRangedWeapon, Shotgun and SniperRifle acquire all the functionalities of LongRangedWeapon and the methods can be invoked on a reference of type LongRangedWeapon. Based on the **Single Responsibility principle**, getAllowableActions methods are overridden in Shotgun and SniperRifle to make them responsible for checking specific conditions before returning their allowable actions to their clients.
SniperRifle extended LongRangedWeapon	A class which represents a sniper rifle. - Constructor:	

	1. SniperRifle(): Create a sniper rifle weapon item and has capability of LONG_RANGED_WEAPON. - Override method: 1. getAllowableActions(Actor actor, GameMap map): Return a list consisting of UseSniperAction only if the actor is Player, has the sniperRifle which still has bullets in his inventory and is not aiming at a target. This is to make sure only Player can use sniperRifle which still has bullet and the sniperRifle must be picked up before it is able to be used and to make sure the UseSniperAction is only available if the player is not aiming at any target.	

Functionality: Refill bullet into long-ranged weapon

New classes	Class Responsibility	Design Rationale
RefillBulletAction extends Action	A class which represents an action to refill bullets of a weapon. - Has **private** attributes: 1. weapon: a LongRangedWeapon object to refill its bullet 2. newAmmunitionBox: an Item representing ammunition box to be used to refill the weapon	Based on **Liskov Substitution Principle**, by making RefillBulletAction a subclass of Action, RefillBulletAction acquires all the functionalities of Action and the methods can be invoked on a reference of type Action. Based on the **minimize dependencies that cross encapsulation boundaries** principle, the weapon and newAmmunitionBox attributes in RefillBulletAction are set as private.

	- Constructor: 1. RefillBulletAction(LongRangedWeapon weapon, AmmunitionBox newAmmunitionBox): Specify the weapon and new ammunition box for the RefillBulletAction. - Override methods: 1. execute(Actor actor, GameMap map): Refill bullets in the weapon and remove the new ammunition box from the actor inventory. 2. menuDescription(Actor actor): Return a descriptive string of the RefillBulletAction. (e.g. "Player refills bullets into shotgun")	

Functionality: Perform shooting action using shotgun

New classes	Class Responsibility	Design Rationale
ShootingAction extends Action	A class which represents shooting action which allows the player to choose shooting direction and results in area effect damage to more than one target based on specific miss chance. - Has **private** attributes: 1. shotgun: a LongRangedWeapon object used as weapon in shooting action 2. shootingDirection: an Exit object used to represent direction of shooting action 3. SHOOTING_MISS_CHANCE: a constant	Based on **Liskov Substitution Principle**, by making ShootingAction a subclass of Action, ShootingAction acquires all the functionalities of Action and the methods can be invoked on a reference of type Action. As ShootingAction is only available for LongRangedWeapon, thus the input type of the shotgun is set as LongRangedWeapon, so that only a subclass object of LongRangedWeapon can be used to construct the ShootingAction. Based on the **minimize dependencies that cross**

	integer of 25 which represents the miss chance of shooting action - Has 2 constructors: 1. ShootingAction(LongRangedWeapon shotgun): specify the shotgun used in the ShootingAction instance. 2. ShootingAction(LongRangedWeapon shotgun, Exit shootingDirection): specify both the shotgun and shooting direction used in the ShootingAction instance. - Override methods: 1. execute(Actor actor, GameMap map): Perform the shooting action. If the player chooses to shoot, a sub menu will be displayed to allow Player to choose the shooting direction, and all actors in the affected area of the shooting action will have a chance to be attacked.Each time shooting, a bullet in the shotgun will be used up. 2. menuDescription(Actor actor): Show general description of shooting action if the shooting direction is not yet chosen, else also include the shooting direction in description. - New **private** methods: 1. getAffectedDirections(List<Exit> allPossibleDirection, Exit direction): Get the	**encapsulation boundaries** principle, the shotgun, shootingDirection and SHOOTING_MISS_CHANCE attributes as well as some of the methods only accessible by the ShootingAction itself are set as private. In order to **avoid excessive use of literals,** the constant SHOOTING_MISS_CHANCE is used.

	directions that will form the area affected by the shooting action based on the shooter current locations and the shooting direction chosen. (eg: if a player chose to shoot in North and if the current exits from the player location include North, North West and North East, this method will return a list of Exits which include these 3 exits. Else if the current exits from the player location do not include North East, this method will return a list of Exits which include North and North West only.) 2. getAffectedArea(List<Exit> affectedDirection, Exit mainDirection, Location actorLocation): Get the areas affected by the shooting action performed by the shooter in a specific direction. 3. getBranchInDirectionOf(Location startLocation, Exit direction, int length): Get a path of specific length in specific direction starting from the location given (excluded start location). 4. getLocationInDirectionOf(Location location, Exit direction): Get a location in the specific direction from the location given.	

Modified classes	Class Responsibility	Design Rationale
AttackAction extends	Special Action for attacking other Actors.	New protected attributes weapon and missChance are

Action	- Add **protected** attribute: 1. weapon: A Weapon object used to attack the target 2. missChance: An integer representing the miss chance of the attack action - Modified method: 1. execute(Actor actor, GameMap map): Perform the Action. Only get the first weapon from the actor inventory which must always either be a short-ranged weapon or intrinsic weapon, if the weapon is not specified in the AttackAction. If miss chance of attack action is specified, use the miss chance. Else, calculate miss chance based on operation in missAttack method.	added into AttackAction to allow these 2 attributes to be specified by client when instantiating any subclass of AttackAction.
HumanAttackAction extends AttackAction	Special class which represents action taken by alive actors (Human/Player) to attack undead actor or priestess (Zombie/Mambo Marie). - Add new constructor: 1. HumanAttackAction(Actor target, Weapon weapon, int missChance): Specify the target, weapon and miss chance to be used in the HumanAttackAction. - Modified **protected** method: 1. missAttack(Weapon weapon): Return Boolean to indicate if the actor misses the target based	A new constructor with additional missChance argument is added into both HumanAttackAction and ZombieAttackAction. This is to allow the missChance of HumanAttackAction and ZombieAttackActionto be specified instead of using the default missChance of the weapon.

	on miss chance if it is specified in the attack action, else based on 50% chance.	
ZombieAttackAction extends AttackAction	Special class which represents action taken by undead actor or priestess (Zombie/Mambo Marie) to attack alive actors (Human/Player) . - Add new constructor: 1. ZombieAttackAction(Actor target, Weapon weapon, int missChance): Specify the target, weapon and miss chance to be used in the ZombieAttackAction . - Modified **protected** method: 1. missAttack(Weapon weapon): Return a Boolean that indicates whether Zombie will miss the attack based on the miss chance if specified in the attack action, else based on miss chance of different type of weapon being used in that attack.	

Functionality: perform aiming and snipping action using sniper rifle

New classes	Class Responsibility	Design Rationale
UseSniperAction extends Action	A class which represents action for an actor to use a sniper rifle, by providing a sub menu for actor to choose a target from actor's current map and after choosing a target, provide another sub menu for actor to choose action to perform on the target, either aiming	Based on **Liskov Substitution Principle**, by making UseSniperAction a subclass of Action, UseSniperAction acquires all the functionalities of Action and the methods can be invoked on a reference of type Action.

	at target or shoot at target. This action is only available if the actor is not focusing on a target. - Has **private** attributes: 1. target: An Actor which acts as target of UseSniperAction 2. sniperRifle: a LongRangedWeapon used in UseSniperAction 3. lastSubmenuAction: an Action which represents the action chosen in the sub menu to perform on the target - constructors: 1. UseSniperAction(LongRangedWeapon sniperRifle): Specify the sniperRifle used in the UseSniperAction instance. 2. UseSniperAction(LongRangedWeapon sniperRifle, Actor target): Specify the sniperRifle used and the target in the UseSniperAction instance. - Override methods: 1. execute(Actor actor, GameMap map): Perform the UseSniperAction. Show a sub menu consists of all the other actors in the actor's current map for the actor to choose a target. Once a target is chosen, show another sub menu which provides options for the actor to choose action to perform on the target (eg: aim	As UseSniperAction is only available for LongRangedWeapon, thus the input type of the sniperRifle is set as LongRangedWeapon, so that only a subclass object of LongRangedWeapon can be used to construct the ShootingAction. Based on the **minimize dependencies that cross encapsulation boundaries** principle, the target, sniperRifle and lastSubmenuAction attributes are set as private.

	at target snipe at target). Once an action is chosen, the action will be executed and a description string describing this process and the consequence of the action will be returned. 2. menuDescription(Actor actor): Show general description of UseSniperAction. (eg: Player uses sniper rifle). Also show the target of the action if it is specified. 3. getNextActions(): Return the next actions of the last action chosen from the sub menu created in UseSniperAction.	
AimingAction extends Action	A class which represents action to aim at a target. - Has **private** attributes: 1. target: an Actor acts as target for aiming 2. sniperRifle: a LongRangedWeapon used to aim at target 3. aimNumber: an integer representing number of times of consecutive aiming - Constructor: 1. AimingAction(Actor target, LongRangedWeapon sniperRifle): Specify the target and weapon used in AimingAction. aimNumber by default set as 1.	Based on **Liskov Substitution Principle**, by making AimingAction a subclass of Action, AimingAction acquires all the functionalities of Action and the methods can be invoked on a reference of type Action. As AimingAction is only available for LongRangedWeapon, thus the input type of the sniperRifle is set as LongRangedWeapon, so that only a subclass object of LongRangedWeapon can be used to construct the ShootingAction. Based on the **minimize dependencies that cross encapsulation boundaries** principle, the target, sniperRifle and aimNumer attributes are set as private.

	2. AimingAction(Actor target, LongRangedWeapon sniperRifle, int aimNumber): Specify the target, weapon and aimNumber for the AimingAction. - Override methods: 1. execute(Actor actor, GameMap map): Set the actor attribute to indicate it is aiming at a target. 2. menuDescription(Actor actor): Get a description string which describes the number of consecutive aiming at target. 3. getNextActions(): Based on the aim number, return the next corresponding AimingAction which has current aim number+1 and also SnippingAction which has specific miss chance and damage and return null if the aim number is zero or greater than 2	
SnippingAction extends Action	A class which represents sniping action which perform attack to specific target using specific miss chance and damage. - Has **private** attributes: 1. target: An Actor which acts as target of SnippingAction 2. sniperRifle: a LongRangedWeapon used in SnippingAction	Based on **Liskov Substitution Principle**, by making SnippingAction a subclass of Action, SnippingAction acquires all the functionalities of Action and the methods can be invoked on a reference of type Action. As SnippingAction is only available for LongRangedWeapon, thus the input type of the sniperRifle is set as LongRangedWeapon, so that only a subclass object of LongRangedWeapon can be used to construct the ShootingAction.

	3. aimNumber: an integer representing the number of consecutive aiming before SnippingAction 4. damage: an array of integer represent different damage of SnippingAction after different number of consecutive aiming corresponding to the value of index 5. MISS_CHANCE: a constant array of integer represent different miss chance of SnippingAction after different number of consecutive aiming corresponding to the value of index - Constructor; 1. SnippingAction(Actor target, LongRangedWeapon sniperRifle): Specify the target and weapon used in SnippingAction. aimNumber set as 0 initially.Initialize value for damage array based on the target and sniperRifle. 2. SnippingAction(Actor target, LongRangedWeapon sniperRifle, int aimNumber): Specify the target, weapon and aimNumber used in SnippingAction. - Has **private** method: 1. createDamagesLevel(Actor target, LongRangedWeapon sniperRifle): Initialize value for damage of SnippingAction based on	Based on the **minimize dependencies that cross encapsulation boundaries** principle, all attributes and the createDamagesLevel method which is only accessible to SnippingAction are set as private. In order to **avoid excessive use of literals,** the constant MISS_CHANCE is used.

	target and sniperRifle for each index which correspond to different number of consecutive aiming before the SnippingAction. - Override methods: 1. execute(Actor actor, GameMap map): Perform attack to the target with different damage and miss chance based on the number of consecutive aiming before the SnippingAction. Each execution of SnippingAction uses up one bullet in the SniperRifle and the actor loses concentration at the target after shooting. 2. menuDescription(Actor actor); Return description string which describe the level of damage of the SnippingAction.(eg: "Player snipe at Groan for double damage", "Player instant kill Groan")	

Modified classes	Class Responsibility	Design Rationale
ActionInterface	This interface provides the ability to add methods to Action, without modifying code in the engine, or downcasting references in the game. - Add **default** method: 1. getNextActions(): Get a list of next actions that can be performed followed by current action. By default, it returns null which indicate no	The default method getNextActions is added into ActionInterface to avoid downcasting from Action to UseSniperAction or AimingAction in order for these two classes to be able to return their next Action list based on certain conditions.

	specific Action that can be performed after this Action.	
ActorInterface	This interface provides the ability to add methods to Actor, without modifying code in the engine, or downcasting references in the game. - Add **default** method: 1. aimingAtTarget(): Check if the actor is aiming at a target 2. setAimingAtTarget(Boolean aim): Set the actor to indicate that it is aiming at a target or not	The default methods aimingAtTarget and setAimingAtTarget are added into ActorInterface to avoid downcasting from Actor to ZombieActor in order for Player to be able to aim at a target.
ZombieActor extends Actor	Base class for Actors in the Zombie World. - Add **protected** attribute: 1. aiming: a Boolean attribute to keep track if the actor is aiming at its target, initialized as false - Override methods: 1. aimingAtTarget(): Check if the actor is aiming at target. 2. setAimingAtTarget(Boolean aim): Set the actor to indicate that it is aiming at a target or not	Based on the **minimize dependencies that cross encapsulation boundaries** principle, the aiming attribute is set as protected to be accessible to the ZombieActor subclasses.
Player extends Human	Class representing the Player. - Modified methods: 1. playTurn(Actions actions, Action lastAction, GameMap map, Display display): Select and	Player’s hurt method is overridden to make the player lose concentration at target when it is being hurt during the aiming process.

	return an action to perform on the current turn. Check if the player is concentrating at his target in the current turn and add suitable actions into the menu to be chosen by the player to perform for the current turn. 2. hurt(int points): Do some damage to the Player. The player loses his concentration to the target when being hurt.	Inside the Player’s playTurn method, there will be a method call of getAllowableAction(this, map) to each item in the player’s inventory to get the allowable actions that have been checked and filtered in the Item class itself.
HumanAttackAction extends AttackAction	Special class which represents action taken by alive actors (Human/Player) to attack undead actor or priestess (Zombie/Mambo Marie). - Add new constructor: 1. HumanAttackAction(Actor target, Weapon weapon, int damage, int missChance); Specify the target, weapon, damage and miss chance to be used in the HumanAttackAction.	A new constructor with additional missChance and damage arguments are added into both HumanAttackAction and ZombieAttackAction. This is to allow the missChance and damage point of HumanAttackAction and ZombieAttackActionto be specified instead of using the default missChance and damage of the weapon.
ZombieAttackAction extends AttackAction	Special class which represents action taken by undead actor or priestess (Zombie/Mambo Marie) to attack alive actors (Human/Player) . - Add new constructor: 1. ZombieAttackAction(Actor target, Weapon weapon, int damage, int missChance): Specify the target, weapon, damage and miss chance to be used in the ZombieAttackAction.	