**Recommendations for change to the game engine**

1.

**Problem:**

The action that can be done to the actor is only limited to the adjacent Ground, not the Ground where the actor is standing on. In other words, the actions in the allowableActions of a Ground can only be executed when the actor is on the adjacent Ground, not the current Ground the actor is standing on.

This will result in extra code needed for adding a new but same action to the actions to be executed when the actor is standing on the Ground which has special action to be done to the actor, for example in the playTurn method in the Player class.

**Solution:**

In processActorTurn method in the World class, other than getting the actions in the allowableActions of a Ground in the loop *for(Exit exit: here.getExits())*, in which *here* is the location of the actor, we should also get the actions by calling allowableActions method for *here.getGround()* outside of the loop.

**Advantage:**

Reduce repetitive codes to add a new but same action to the actions to be executed in other class (such as in the playTurn method in Player class). With this change, we only need to override the allowableActions method in the subclass of Ground to add the action to be done to the actor when he/she is standing on and adjacent to the subclass of the Ground.

* Follow Do Not Repeat Yourself (DRY) principle

**Disadvantage:**

If the action done to the actor should only be limited to the adjacent Ground of the actor, extra code would be needed to remove the action for the current Ground the actor is standing on. However, logically speaking, it makes sense for the allowableActions of the Ground to include the current Ground the actor is standing on.

**Justification of our positive opinion**

Classes in the engine have followed most of the design principles that we learned. Classes such as Action, Actor, Item, Ground etc has followed the design principle called DRY ( Do not repeat yourself ) as they serve as an abstract class where future classes such as ZombieActor, AttackAction, Tree etc that are related to them can inherit from them. In the engine package,

PickUpItemAction, DropItemAction, DoNothingAction, MoveActorAction are the subclasses of Action whereas WeaponItem is the subclass of Item The codes in these classes clearly shows inheritance helps reducing the repetitive codes, thus following the DRY principle. They also follow the LSP ( Liskov Substitution Principle ), which is one of the design principles from SOLID, as they did not reduce the functionality of their parent class but instead enhance them.

Interfaces such as Capable and Printable are also used in the engine code to provide flexibility for classes as they can be used to provide certain behaviour to the classes even if they are not related. They also follow the ISP ( Interface Segregation Principle ), which is one of the design principles from SOLID, as the classes implement the interfaces that are given to them. For example: Actor implements Capable and Printable. The interfaces in engine code is also easy to implement as each interface has very less functionality, Capable and Printable has only one and GroundFactory has three, which also obeys ISP.

Moreover, the engine code also applies Fail Fast principle. As seen in the engine code, the constructor in the GameMap class throws IllegalArgumentException when certain conditions are not met. This applies the Fail Fast principle because the coders that code classes that create a GameMap object with inaccurate arguments will immediately know where the mistake is when something goes wrong.Hence, coders can find and correct the possible mistakes easily. This will help to ensure the program in the end to be more robust. Other than that, exceptions such as IllegalStateException and IllegalArgumentException are also thrown in the run method in World class and the add method in ActorLocations class respectively.

Next, the engine code is well-encapsulated. The access control modifiers of the attributes in the classes are either private or protected. There are also some methods which are declared as protected instead of public, thus they are only accessible by itself and its subclasses, such as processActorTurn, stillRunning and endGameMessage methods in World class. Here, the code follows the Reduce dependencies as much as possible principle. This is because encapsulation allows information hiding and helps to reduce dependencies. An attribute or a method with restricted access will have lesser risk to have something depending on it, thus there will be less risk that it will cause failure in the code in the future. This also brings out the related principle, Minimize dependencies that cross encapsulation boundaries. Declaring members of a class with the lowest possible visibility is a good practice because as soon as something is visible, programmers can write code that depends on it, and once the access control modifier is changed to be less visible, the systems will break.

Lastly, the engine code avoids privacy leaks. Privacy leaks occur when getters return a reference to a mutable object. Therefore, to avoid this issue, the engine code makes a deepcopy of the collection and returns it. This can be seen at the getInventory method in Actor class and getUnmodifiableActionList method in Actions class. Instead of just returning the collection itself, these getters return an unmodifiable copy of the contents of the collection. Otherwise, the code will lose the benefit of encapsulation as stated above and lose control of connascence.