The following are my recommendations to improve the codes in the engine package.

One problem that I faced when I was coding is that I have no control to choose which item I want to retrieve without looping through the whole inventory. If the game were to be expanded in a sense in which the Players can carry a lot of items, the item retrieval process can be inefficient. Therefore, I suggest that instead of using an ArrayList to store the Player’s inventory, use a HashMap with the item's name as key and the item as the value. The advantage of the proposed change is that the retrieval of an item can be done more easily and faster. Instead of looping through the array list, we can just use the get method for HashMap. The disadvantage is that the looping part is still required if we were to check if the player has an item of a certain type (Eg. weaponItem). Besides, the space used for a HashMap is more than that of an ArrayList.

The second problem which I find is that the Actor class is too large and handles too many responsibilities. I, therefore, suggest the inventory-related attributes and methods be placed in another class(eg. ActorInventory) and let that class be an attribute of the Actor. The advantage of this approach is that it allows the more sophisticated implementation of the inventory to be added without complicating the Actor class. For instance, the player's inventory might now be limited in space (a fix- size array or HashMap is used). This means that the methods that allow expansion of inventory will be written to allow Player to expand the size of their inventory. By placing all inventory-related attributes and methods in one class, any changes to the implementation of the inventory will not affect the Actor class. This ensures the Single Responsibility Principle being applied. One disadvantage of this approach is that it will result in longer chain methods. For instance, to add an item to a Player's inventory, the following line will be needed: actor.getActorInventory.addItemToInventory() instead of actor.addItemToInventory().

Another issue I faced is that in the World class, processActorTurn() has created an Actions object which makes it impossible for the children class to inherit. Understand that encapsulation is needed in programming however I personally don't think it is necessary to have this dependency on Actions class as it is used quite often and may need further extension of the system, if any more actions that is not suitable to be implemented in Player’s playTurn should be allowed to modify the **actions** in processActorTurn() by making an Actions attribute ( association is used instead of dependency ) this will avoid more association of other classes and avoid downcasting as well. Another way to actually implement the changes to the **actions**  is to copy the code entirely and override it completely. However we know that this is a bad practice as there is repetition of code. Hence, I suggest there should be an association for actions attribute instead of a dependency for Actions actions in processActorTurn() in World class.

The following are my positive opinion for the codes in the engine package:

Over the course of the three assignments, I find it easy to implement extra functionality based on the code in the engine package. After some thorough analysis, I figure out that the reason why it is easily expandable is that it has adhered the SOLID principles. I will discuss how the code adheres to each principle one by one, with examples.

First, I will start with the Single Responsibility Principle. Each class found in the engine code has a well-defined responsibility. For instance, the WeaponItem class is being separated from the Item class because it has special responsibilities that the Item class does not have (verb and damage). This means that each class is of reasonable size. This improves clarity for other programmers who are using or expanding the code as the responsibility of each class is clearly shown.

Second, the Open-Closed Principle. Various foundational classes have been created in the engine package. These classes include the Actor class, the Action class, the Item class, the Location class and the Ground class. Having these classes as the foundation of the game is very helpful for the game to be expanded as they all have well-defined attributes and methods that allow the game to run properly. New classes that are related to these classes can be created in the other packages very easily through inheritance. For instance, a new type of Actor can be created to have all the characteristics of an Actor by inheriting from the Actor class. This means that by having these foundational classes, the programmer can add an unlimited extension to the game easily without having to change the source code. The creation of these classes is the reason why I can add a lot of features without having to modify the engine code.

Third, the Liskov Substitution Principle. One example of this principle being obeyed is shown in the World class, specifically in the processActorTurn method. The playTurn method of the actor is being executed at the end of the method and an Action object which tells the system what the Actor would do would be returned and be stored in an Action type local variable. However, this local variable turns out to be able to store ANY KIND of actions (run, attack, shout,chant). In other words, if an instance of the base class (Action) is expected, we can substitute it with an instance of a children class without making the program fail. This ensures consistency of the system, in which each actor will have to do something (execute an action) at each turn. This allows expandability of the system as I can now add every action that I want the actor to have in other packages without having to worry about whether it will run or not.

Fourth, the Interface Segregation Principle is also obeyed. Various interfaces have been created such as Capable and Printable instead of having all required methods to be in one interface. The principle used here later inspires me to add special characteristics of the Item instance (make the items that are sellable to implement SellableInteraface). Having various interfaces allows each interface to be specific and avoid classes to implement methods that they might not need. This can shorten the number of code lines and reduce the chance of observing a bug.

Fifth, the Dependency Inversion Principle. One example would be in the ActorLocations class, specifically the locationToActor attributes. The class chooses to associate with the Actor classes (which is an abstract class) instead of a concrete class like the Player and Zombie class. This ensures that when a new Actor is added, there will be no changes required to be done on the ActorLocation class. This again ensures the expandability of the system.