Engine Recommendations

Problem 1

Item class is dependent on Action class. For any new items it had to extend the item class

Explanation:

When creating new items, they all had to extend the Item class. For example starting this assignment many items had different functionalities which were not supported by the engine such as: having nutrition for Food objects, being able to upgrade a **ZombieLeg**/**ZombieArm** object to a **ZombieMace**/**ZombieClub** object and keeping track of how much Ammunition is in a gun or ammunition on the ground. To implement this functionality, we needed to add abstract methods into the **ItemInterface** which lead to down casting these methods into other items that did not need them.

Design Change:

Have the class also iterate through items on every tick so there is less reliant on the Actions class and the expand the Item class to support this change.

Advantages	Disadvantages
• Reduce dependencies as much as possible principle is maintained. • Interface segregation principle is easier to maintain as unneeded methods do not need to be added to the item interface to allow subclasses to work	• This change may be hard for new users to recognise when to create a new item or a new action • Increases runtime of the software as now the items and actions need to be checked for what they can do.

Problem 2

Some of the methods in the engine package are doing too many things which makes it too long. "*FUNCTIONS SHOULD DO ONE THING. THEY SHOULD DO IT WELL. THEY SHOULD DO IT ONLY.*" - Robert C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship pg.35. Following this rule allows better code reusability and maintainability. For example, in the **processActorTurn** in the **World** class is very long so it is most likely doing multiple things. If we have a closer at the body, we can see that there is blocks of code that is:

- Adding actions regarding items
- Adding actions regarding actor to actor interaction
- Adding actions for items at the actor location

So there a few things that **processActorTurn** are doing. Now if a class inherited **World** and want to override the **processActorTurn** method and slightly change it, the whole code would have to be copy and pasted then you can change a few things. This is code duplication and should be avoided. Now if each of the three things outlined above was refactored out of **processActorTurn** and made into its

own method, we can slightly change it and have the refactored method called in the overridden method. This eliminates duplicated code which follows the DRY principle.

In our assignment, we saw this problem when we wanted the result string at the end of the code to only print when the player is in the map. It was a slight modification, just needed an if condition but required the whole code to be copied which is a code smell.

Positive 1:

Classes in the engine can act as superclasses which can be replaced by subclasses without breaking the application.

Explanation:

Classes in the engine all have methods that include parameters required for functionality for later implementation. Thus, the Liskov Substitution Principle is demonstrated withing these classes. The methods within the classes of the engine are very helpful for further implementation of features such as the tick() method in Items which allows that class to virtually keep track of time passing in the system. Furthermore, as these subclasses can be used in place of its parent classes it is much easier to encapsulate certain functionality to only work with specific types of classes without losing the functionality that the parent classes provide.

Positive 2:

The actions in the engine package are well encapsulated and each of them only have one responsibility. This follows the Single Responsibility Principle and makes the classes cohesive. If a change on have an item is dropped, then there is only one class – **DropItemAction** that needs to be modified. This results in the system in easier to maintain and extend.

Positive 3:

Having **Item** as an abstract class allows adding new items easy as the new items must fulfil the contract by implementing the abstract methods set. When the new item implements them, it is guaranteed that higher level classes that are using **Item** objects can use them since they should not care about the fine details of low-level classes. This is the Dependency Inversion Principle and it reduces coupling between classes so a modification in one class wouldn't interfere with another.