Recommendations for change to the game engine

There are several improvements that could be made to the current game engine.

1. Improvement on GameMap class

In the current game engine, the actorLocations attribute in GameMap class refers to the same ActorLocations instance that is instantiated in the World class. In other words, in all the GameMap instances, we can only access a list of actors that appear in all existing maps. There is no way to identify the actors that appear in a specific game map. This is a problem because it hindrances the creation of some game features. For instance, actions that will only affect the actors that are on the same map as the executor will be hard to implement. To demonstrate, in the case where the game has more than one game map, when the player executes a long-ranged attack, only the NPCs that are on the same map as the player will be affected. However, there is no method in the current game engine that can retrieve a list of actors that are on the same map as the player. Therefore, it will be difficult to identify the actors that are hurt.

The problem can be resolved by instantiating a new ActorLocations in the GameMap class instead of making the attribute actorLocations to refer to the same ActorLocations instance that is created in the World class. With this, features such as long-ranged attack will be much easier to be implemented. This design also does not introduce new relationship between classes (the current design already has an association between the GameMap class and ActorLocations).

A potential disadvantage of this proposed design is that it will be more complex to add or remove or move actors in multiple game maps. This is because both the GameMap and World's actorLocations need to be modified accordingly so that the game integrity can be maintained.

2. Improvement on Menu class

The current menu class can only show up to 36 options, because it uses the digits 0-9 and alphabets a-z as the hotkey. Since the digits 0-9 have already been associated with specific move action and quit game action, there are only 26 more options available for the other actions. The current menu class will only display the first 26 actions and ignore the rest. This might be an issue when the menu does not display the option that the player truly wants to choose. This might cause insatisfaction in the player and this is definitely not the best user experience.

To prevent this issue, the Menu class should use an integer counter to display all the actions instead of using alphabets. Since 0-9 has already been occupied by

specific actions, the counter can start from 10. Since the integer range is infinite, the menu can now display all possible actions that the player can carry out in a particular game turn. This way, the option that the player really wants will definitely be shown in the menu.

On the other hand, since the menu can display infinite options, the menu might become too lengthy, which in turn will worsen user experience as well.

3. Improvement on Weapon Interface

In the current game engine design, the IntrinsicWeapon implements a Weapon interface which is also placed in the same engine package. By placing Weapon interface in the engine package, the IntrinsicWeapon becomes less extensible as there is no way to add new methods to IntrinsicWeapon. As a result, new functionalities of IntrinsicWeapon need to be added into a new subclass of IntrinsicWeapon named ZombieIntrinsicWeapon inside the game. Problems occur as downcasting from IntrinsicWeapon to ZombieIntrinsicWeapon becomes inevitable each time we want to use the new IntrinsicWeapon functionality implemented inside the ZombieIntrinsicWeapon class.

This problem can be easily solved by placing the Weapon Interface inside the interfaces packages. By doing so, new methods can be declared in the Weapon interface and the subclasses can override these methods easily (polymorphism). The first advantage of this approach is that new methods can be added to classes that implement Weapon such as IntrinsicWeapon and WeaponItem, without making any changes to the code in the engine package. Secondly, the above-mentioned downcasting references problem can be avoided.Therefore, all weapon-related classes, both IntrinsicWeapon and WeaponItem, become more extensible.

A possible disadvantage is that this approach might violate the Interface Segregation Principle if most of the new methods added into the Weapon Interface are actually meant for IntrinsicWeapon class and are not used by the WeaponItem class. In this case, the WeaponItem class is forced to depend on the methods it does not use. If this scenario happens, splitting the large Weapon interface into two specific and smaller interfaces, namely WeaponInterface and WeaponItemInterface, will be a better approach.

Positive opinion on game engine

1. menuDescription() method in Action class

The menuDescription() method in the Action class adheres to the design principle of **tell-don’t-ask**. The menuDescription() tells the client what it has done by returning the the Action descriptive string which is formed after it performs operations

on the attribute in the Action class. This relieves the client classes from the responsibility to query for required attribute values from Action class and performs logic on the value to form a descriptive string of the Action.

2. Capable and Printable interfaces

The Capable and Printable interfaces adhere to the **Interface Segregation Principle**. It is observed that multiple classes such as Actor and Action implement both of these interfaces. Instead of putting all the capabilities-related methods and printing-related methods in a single interface and making the other classes to implement this single interface, the methods have been encapsulated in different interfaces based on their functionalities. This prevents the necessity for clients that only need capabilities-related methods to implement printing-related methods and vice versa.This in turn helps to reduce repeated code in the engine package.