Recommendation for changes to the game design for Assignment 3

The engine class and its base codes are introduced to us when assignment 1 of FIT 2099 was released. There were many new classes introduced which seemed intimidating for us as we had to understand what every class did before we could start implementing out solution for the assignment and develop the game as per required by each assignment. As we explored the code, UML diagrams and understood the purpose of every classes, we realized and reached an agreement that it was a very well written code following good design implementations.

One of the specific classes that we enjoyed working on were the gameMap and level class. It provides us with customizing layouts like the town map that we have implemented and the compound map that was provided to us from the base code. Utilizing the groundFactory class, it allowed us to check the ground of the game map and to interact with it. Initially, we thought it was not possible for an actor to stand above the ground as the ground would be replaced by the actor, we came to realize it did not work as what we expected as we implemented sowing of crops and farmer. Besides that, it also allowed us to have a coordinate system, we could easily initialize actors and items onto the map by utilizing its xRange and yRange (min and max) method to get the length and width of the map. In terms of design principles, it would allow us to create customizable layers of map by simply refactoring the code that was provided in the engine and create a dependency within application class to create a level in the game.

Some other classes that we enjoyed working on is the action class. The action class has allowed us to create custom dialogues for an actor within the game s well as creating special actions for specific actors. With the help of Display object created that is passed into World class objects, when each actor’s turn is processed, a string would be displayed from the past turn to indicate the actions that were performed by every actor on the game. However, during an interview for assignment 2, we found out that it doesn’t display the player’s effective health points after being attacked by a zombie. This could be easily solved by passing the Human object when creating the ZombieAttackAction dependency to calculate the human’s effective health after it is damaged by the Zombie. In regard to design principle, by extending the engine’s action class, we were able to specially make actions for certain actors such as sowing which is exclusive to the farmer only.

A different class that we sub classed multiple times was the Items class. This includes portableItem and WeaponItem class. We were able to make food, craftable weapons, vehicle and guns. By sub classing portableItem, it allows us to prevent repeated code for having object set to true on portability option every time we would like to make a new portable item. Whereas the item class has portability set to false on default, this allows us to decide which class to extend from easily. With the current level of customizability on items, we could implement a loot system for zombies where they would drop coins and player could buy stuff from the shop in town. Furthermore, by extending item and portableItem, we could use the methods that were implemented which averts us from entirely designing and repeat coding similar methods for the Food class that was implemented.

Another class that was widely used within all assignments was the actor class. We would normally inherit the ZombieActor and Human class as it contains most of the method that is required for the assignments. From assignment 1, we had to implement a Food object which is only obtainable by harvesting ripe crops, adhering to the design principles of DrY (Do not repeat yourself), we have implemented 2 new methods, existFood and healActor within the Human class. By doing so, any new classes that would extend the current Human class would have the ability to utilize the methods mentioned. To prevent other actors from being able to use it, we have chosen to have it encapsulated with the ‘protected’ keyword. In every new class that extended ZombieActor and Human, we have made sure that its class attributes are private and for reducing dependencies, some class attributes contain an object from another class which allows reusability of the object without creating a new object each time. An example would be the MamboMarie class, we have made a class attribute with the variable name rand which is a Random object which we would use to recalculate the probability on whether Mambo should remain vanished or discovered within her playTurn method.

Moving on to the problem that we perceived as we were coding on the engine. We were a little worried on creating too many actions which would overwhelm the current Menu system for the Player. As we browsed through the Menu class, we found out that there was only a limit of 26 options available. This means that if the player hoards on many items that could be introduced within a bonus feature, such as having a teleportation scroll between the current compound and town map, having 2 craftable weapons that could be further crafted into an ultimate weapon or having too many new food within the player’s inventory, it would just clog up the menu interface and the Player would have to scroll through and find the desired action. On the topic of actions, each time a Food object is added into the player’s inventory it would return an action. This means that if the player is currently having 10 watermelon, there would be 10 healAction objects in the menu. Instead of having multiple actions, the engine could have introduced a counter system to count the number of similar actions available which would significantly reduce the quantity of actions on the Player’s display menu. Another alternative solution to overcome the limitation of 26 options available is to utilize capital A-Z to allow up to 52 options to the Player.

In conclusion, the overall engine code was well designed, and we had an overall positive opinion in relation to design principles. On our end, we had mostly utilized the following design principles when we were producing our solutions. They include don’t repeat yourself (DrY), encapsulation and privacy (private, protected or public), reduce dependency (ReD) and liskov substitution principle (lsp) when performing inheritance.