**Group Name: Team Julian  
Members: Julian Yong Hao, Khong Jun Ming**

**Recommendations and Comments on the Game Engine**

Throughout the design and implementation of the Zombie Apocalypse game in Assignment 1 to 3, as the programmers of this game, we had difficulties working with the existing game engine. We would like to address these difficulties in this document as well as suggest changes to resolve these problems. Moreover, we will also take this chance to comment on some good design principles that were implemented in the game engine.

The first problem we’d faced was that we had a hard time understanding the engine in the first place. This is in particular understanding how the game actually starts and ends as well as how it processes every turn. This was somewhat not clearly stated in the design documents and instructions given to us. Hence, we would actually have to go through classes one by one to understand its relationship and interactions with other classes.   
Therefore, we would suggest that to include some simple sequence diagrams to show how things are done in the engine. For example, having a sequence diagram on how an action is executed with the involvement of Application, World, the Actor classes would help us a lot. With the help of sequence diagram, it would definitely help students have a clearer visualization of how the engine and game was intended to start.

The second problem we’d found out was the similarity of classes names used and confusion of class types in the engine. For instance, the class names “Action” and “Actions” are too similar. Both are only in difference with one “s”. This may confuse programmers when they’re using those two classes. In addition, there’s possibility that programmers may mistype and this would probably lead to misuse of another class that has the similar class name. This may also cause further runtime errors. Therefore, we would suggest using class names that’s more unique such as having “Actions” named “ActionList”. By doing this, it would give the programmers a pre-understanding that this class stores multiple Action objects using List. Other than that, the class names in the engine did not hint its type. For example, the class “Ground” and “GroundFactory”,by just looking at the class names of these two classes, we wouldn’t know whether they are abstract classes or interfaces. Despite there is a little icon shown in Eclipse representing what type they are, it still brings confusion and extra time for checking when we as the programmers implement those two classes. Therefore, we would like to suggest that abstract classes and interfaces should be named with words like “abstract” or “interface” at the end of class name. For example, the “Ground” class, we would name it as “GroundAbstract” and “GroundFactory” as “GroundFactoryInterface”. On the other hand, to avoid confusion we could also keep the interfaces in a package as we are less likely to use them. Consequently, This will allow programmers to have an idea on which classes exactly they want to use.

The third problem we’d seen was unable to see the Player’s latest hit points on the game console. This reduces the ability for programmers as well as the Player to keep track of the current hit points the Player has. This haunts us as we as the programmers are difficult to check the functionalities that involves Player’s hit points. For example, the implementation of consuming Food. With just little information that shown by the action descriptions, it’s certainly not enough. We think that it is crucial to provide the Player’s current state information in particular his hit points just like how common it’s in normal fighting games. We think that this can be implemented in the Menu class, in the showMenu() method. We could display the Player’s current hit points right before showing the available actions. By doing this, it also benefits the Player especially in the scenario where he is attacked by some zombies, he could decide if it is more important to heal himself first or attack.

On the other hand, as what we had learnt through this unit, we noticed that the engine followed some good design principles. For example, the principle “Don’t Repeat Yourself” is referred to avoiding repeated codes. Repeated codes are bad because they will appear in multiple places and must be maintained individually. This is troublesome as if there is a change in requirements or there is a bug in the code, the same piece of code must be modified several times. Fortunately, this is not present in the engine. The engine code is implemented with the concept of polymorphism. Hence, classes can be inherited from each other. By doing this, sub-classes that inherit from their superclass can access their superclass’s public methods. The engine is also implemented with abstract classes. Thus, the following sub-classes could inherit and just override certain methods to complete its required functionality. With that, it reduces the cost of maintenance also, for instance, we can simply make changes to the super class and all the sub-classes will get updated too.

Furthermore, we’d noticed that the engine also followed the principle “Avoid excessive use of literals”. The excessive use of literals is bad as it may confuse us as the programmers the hidden meaning behind those literals. In fact, in the engine some functionalities may require the use of literals but the enumeration concept was used to achieve its objective.

Other than that, we also noticed that the engine classes followed the principle “Declare things in the tightest possible scope”. All class members are carefully declared with good access state. This is meant by declaring those class members to private and also using local variables whenever possible. This lowers the risk of other classes or methods to depend on public members in its class. Thus, lowering the possibility of a code failure in the future. Moreover, the classes that works together are also stored in the same package. With the visible clear cut of packages, the engine has well-kept its elements that depend each other in an encapsulation.

Lastly, we would like to say that the engine code has good naming of variables. This has followed the principle “Avoid variables with hidden meaning”. Variables with hidden meaning are indeed a hassle to the programmers. For example, variables like “v1=1” will not mean anything to us and may cost us a hard time figuring them out. Luckily, this was not present in the engine code. Variables in the engine are declared with self-explanatory names which eases the understanding of code. In addition, the comments that were included engine code did help us much too.

To summarise, overall the engine code has followed good design principles in some areas, but they are too changes that can be made which were suggested to improve the understanding and overall functionality of code.