StormBreaker 

Assignment 3 - Recommendations for change to the game engine

By Adeline Chew Yao Yi and Tey Kai Ying

Problem perceived when using engine package

- While we were completing the task about “Mambo Marie” in Assignment 3, we faced difficulties to add a new actor into the map like the player. We found out that the **run()** method in **World** class has a while loop, which is basically responsible for processing the whole game. We realised that it is pretty troublesome if we extend the **World** class, as we are not able to change any code in the while loop of **run()** method. This results in duplicated code and low extensibility.

  Therefore, we suggest adding a new method with the code implementation in the while loop in the **World** class. As a result, when we are overriding this method in a new child class, this brings an advantage to us. We are able to give a specific implementation to the inherited method without modifying the parent class method. Furthermore, splitting into functions helps us achieve the design principle “DRY” by invoking the parent class method using super keyword. This principle ensures easy maintenance of code, because when a functionality or a constant change we have to edit the code only in one place.

- We faced the second difficulty when implementing the “Going to town” task that needed to design and add a new **GameMap** into the system. When we added some actors into the new town map and started running the game, we found out that messages displayed for all the **Actor**’s **Action** were mixed and printed together. It is quite confusing and becomes more challenging to read a particular message when the number of actors and game map increases.

  In order to make the UI look cleaner, we suggest making some optimisations for the attributes in the **World** class. The menu descriptions are mixed up as in the **run()** method, it gets the **Actor** from actorLocations which stores the information of all **Actor**s and their **Location** in the game. One of the options is to replace the instance variable actorLocations of type ActorLocations with the ***Map<GameMap, ActorLocations>*** type, **GameMap** as the key and

instance variable **ActorLocations** actorLocations in the **GameMap** as the value. Therefore, when running **processActorTurn()**, we can choose to print the result if the **Actor** is located at the **GameMap** which contains the **Player**.

The advantage of this design is that the User Interface will look neater and shorter. Users are able to identify messages displayed according to maps and increase the understandability of the User Interface. However, we detected Shotgun Surgery code smells in the former system, the responsibility of Actor's locations has been split up among a number of classes, thus the disadvantage of this design is making a new modification required to make many small changes to many different classes and methods. This kind of code smells should be prevented to make the system easy to maintain and better organisation.

Positive opinions on the engine package

However, we did enjoy working with this engine code because of its high understandability and simplicity. By applying the principle of abstraction, this engine is built up with interfaces, abstract classes and subclasses, where each class is classified clearly and implemented for its responsibility. Single responsibility principle is achieved and this will give us the flexibility to make changes in the future. Moreover, these responsibilities are general and suitable for any game.

Besides, most of the methods are short, readable and as simple as possible. Long methods will be very hard to maintain for programmers, and bugs will be harder to find. This might violate the DRY principle as we might keep using the same code in a method instead of implementing the code as a separate method.

We figured out this engine has accomplished encapsulation well by keeping attributes non-public. Encapsulation boundaries are restricted by minimising calls to methods or attributes not in a class. Instead, getter and setter methods are used to access these variables from outside classes. This can avoid privacy leaks and enforce data hiding. Encapsulation has also provided maintainability and flexibility to introduce new code later without causing impact on existing code. For example, in the **Location** class, "x" and "y"

coordinates are declared as private variables and accessors are provided to access these two variables.