Assignment 3 - FIT2099

Recommendations for change to the game engine

As we know, for having a good design, our software or code should functionally correct, performs well enough, be usable, reliable and maintainable.

Obviously, this game designed by a professional and master in Java, so, after considering the game Engine Package, we could not be able to find any issue in the design according to the design principles.

Therefore, I want to mention the positive parts of this game design according to the design principles.

I want to start with the first one which is Do Not Repeat Yourself (DRY), which means don't write duplicate code, instead use Abstraction to abstract common things in one place.I went through all the classes in the Engine package and I couldn't find any code repetition, also I found an appropriate use of abstraction class in Actor

```
public abstract class Actor implements ActorInterface, Skilled, Printable {

    private Skills skills = new Skills();
    protected String name;
    protected char displayChar;
    protected List<Item> inventory = new ArrayList<Item>();
    protected int maxHitPoints;
    protected int hitPoints;
```

Which makes it easier to use and maintain and this class also Implements 3 other classes, instead of having different classes with the same methods or attributes.

The second design principle that I want to mention it is about dependency. As we know, dependency is the biggest issue in the design and it is unavoidable, but we can reduce the dependency and among classes and make them independent. In the engine package we can definitely say, there is a relationship between Actor and

ActorLocations and Map, but if we wanted to depend them to each other, if any bug or issue occurs during the game, it will be harder to find and the whole game can be broken. Another way that we can reduce dependency is to declare the class members private. Like the ActorLocations:

```
 */
public class ActorLocations implements Iterable<Actor> {

    private Map<Location, Actor> locationToActor;
    private Map<Actor, Location> actorToLocation;
    private Actor player;
```

We can also see the Encapsulation in the above examples, we can hide the data by making them private or make them protected and just let their subclasses to use of those methods or attributes. In encapsulation, we can increase the code flexibility, We can make the variables of the class as read-only or write-only depending on our requirement. If we wish to make the variables as read-only then we have to omit the setter methods like setPlayer() or getActorAt().

```
public void setPlayer(Actor player) {
    this.player = player;
}


public Actor getActorAt(Location location) {
    return locationToActor.get(location);
}
```

The other design principle that we can mention here is Liskov Substitution Principle. According to the Liskov Substitution Principle Subtypes must be substitutable for super type. Methods or functions which uses super class type must be able to work with object of subclass without any issue". LSP is closely related to Single responsibility principle and Interface Segregation Principle. If a class has more functionality than subclass might not support some of the functionality and does violated LSP. In order to follow LSP design principle, derived class or subclass must enhance functionality not reducing it. We can see this in many of the classes by using a method or attribute from a superclass without repeating it.

Furthermore, Javadoc is one of the most important parts of any design. With putting comments above each method or any class, we basically give the idea and functionality of anything that we put any use in our design. A code without Javadoc does not make any sense. If a group of people want to work on a piece of code or software, they should explain why they create anything and what does it do, so, according to this, we can see that all the methods and functions have comments and I as a developer can easily find the method or class that I want to use it in other places.

The Last thing that I can mention here, is about using interface classes. As we already know, interfaces define a set of functionality as a rule and when we implement an interface all of these functionality must be implemented in the concrete class. In the engine package, we see that GrandFactory, Printable, Skilled and Weapon are interface classes which all the functionality inside them will be implemented in the concrete classes.

```java
/**
 * An interface for representing a collection of any kind of enum.
 * A practical alternative to type introspection and other problems.
 *
 */
public interface Skilled {
    boolean hasSkill(Enum<?> skill);
    void addSkill(Enum<?> skill);
    void removeSkill(Enum<?> skill);
}
```

```java
public abstract class Actor implements ActorInterface, Skilled, Printable {

    private Skills skills = new Skills();
    protected String name;
    protected char displayChar;
    protected List<Item> inventory = new ArrayList<Item>();
    protected int maxHitPoints;
    protected int hitPoints;
```