Recommendations for extensions to the game engine

The game engine applies the SOLID principles and is designed in such a way that the engine is simple to understand, maintain, modify and extend. The following documentation walks through each of the SOLID principles and examples of how the engine design adheres and applies them, ensuring high engine code quality.

Single Responsibility Principle

The engine applies the single responsibility principle by delegating one specific responsibility to each class. For example, **Location class** is responsible for all the positions available on the map, while the **Actor class** is responsible for the functionalities and attributes of characters in the game.

Besides, taking the **Location addItem()** function, it is implemented such that the function is only responsible for one thing: *adding item to the location*. This makes the engine simple to understand and maintain as one class/method is only responsible for executing one task.

Another application of single responsibility principle can be seen in the classes extended from Action, Each class which was extended from Action had there own unique responsibility such as FeedDinosaurAction was only responsible for allowing actors to feed a dinosaur fruit, and FeedVegKitAction is only responsible to allow the player to feed a dinosaur a Vegetarian Meal Kit and so on.

Open/Closed Principle

The **Action class** clearly illustrates the application of the open/closed principle, which states that code should be open for extension but closed for modification. For example, **Action class** as an abstract class allows children classes to extend its functionality, but the baseline of the extension classes (such as **DropItemAction** and **PickUpItemAction)** must adhere to what is specified (functions and attributes) in the **Action class**.

As a result, the design will ensure backward compatibility as the compiler forces the children classes to implement methods that are required for the engine to work as expected.

The Abstract Class Ground is another great example of Open/Closed Principle. For this assignment we had to implement and edit 4 classes extended from Ground which were Dirt,Bush,Tree and Lake, each of these classes had their own unique probabilities and features that needed to be implemented. This was made easy due to the tick() method inherited from ground class.Since each of the derived classes features were implemented without the editing of Ground Class, it makes it easier for future features for the application to be implemented such as maybe mountains,boulder or even holes/ditches can be implemented into the game easily as the base class ground is untouched.

Interface Segregation Principle

The engine applies the interface segregation principle for the implementation of the **Weapon interface**, **WeaponItem abstract class**. Weapons that are not items (fists,claws of actors) **do not have to implement** methods of weapons that are items, as both use cases implement different abstract classes/interfaces (Weapon, WeaponItem).

Another example would be the **GroundFactory class** and the **FancyGroundFactory class.** The **FancyGroundFactory** contains attributes and methods that our current map needs. However, the methods and attributes are not necessarily compulsory for other maps. With the separation of these two interfaces, there is flexibility to extend a less constrained class (**GroundFactory class)** for a different map, or to write a new class extending **GroundFactory class** which better caters to the new map to be built.

This design makes the codebase cleaner as children classes do not have to implement methods they do not need.

Dependency Inversion

The engine correctly implements this design methodology. By **depending on abstractions over details** whenever a modification on the engine is required, the high-level abstraction class/interface has to be modified first instead of the lower-level classes.
A good example will be the **Capabilities** class vs the **Capable** interface. The method/attribute signature on **Capable interface** will need to be modified before modifying the **Capabilities class**, ensuring that the high-level implementation is prioritised over low-level implementation and the changes propagated throughout the children classes.

This design makes the engine easier to maintain and modify without needing to have much experience with the engine.

Liskov Substitution Principle

In the code there are many instances where the Liskov Substitution Principle can be seen. The Action class and the Ground class are two very good examples of it

When discussing the Ground class. We can see that the classes derived from it such as Dirt,Bush,Tree..etc can be substituted for or can count as an instance of Ground. This can be seen using the Location class. In the location there is a method called getGround() which returns the instance of ground at this location. Even though the method's return type is Ground it was still able to return instances of Dirt,Bush,Tree..etc as they are derived from ground hence can be substituted for it. Due to the fact these classes obey this principle it gives us easy access to the derived classes hence allowing us to call the necessary methods from them without having to implement separate methods to get each type of ground.

Another example of the Liskov Substitution is the classes derived from Action. Let's take the example of the Allosaur class. For our assignment we implemented classes derived from Action such as AttackAction. In the Allosaur class ,the method getAllowableActions returns a list of actions which the actor can perform. We can see that when we added the action to the

list no error was given as it can be substituted as an instance of Action. This allows easy addition of new actions for the actor to perform such as maybe even jumping/sleeping actions if needs be.