Our recommendations for extension to the game engine

A difficulty that we have encountered during the implementation of this game is to increment the eco points of the Player for the following two activities:

- When a ripe fruit is produced by a tree
- When a dinosaur hatches

What we have done in Assignment 2 is to loop through the game map containing the location of the ground to search for the Player via the tick methods of the Tree and Egg classes, respectively. Problem arises in Assignment 3 when the Player is in the second game map and our original implementation is no longer effective.

To solve the above problem, we have traced back the procedure taken to reach the tick method of the Ground class and find that it all begins from the World class to the GameMap.tick(), followed by Location.tick() and finally the Ground.tick(Location). Also, we realize that the World class stores the Player as an attribute.

Combining these two observations, we have come up with an idea, that is to pass the World class object as an additional argument all the way down to the tick method of the Ground class. Then in the World class, create a public getter method for the Player attribute. This way we can easily get the Player and increment the eco points within the tick method of the Ground class.

An advantage of this proposed extension is that the changes required are minimal, which means the reusability of the engine code is maintained, as we believe this is a good and important practice in OOP. However, for the downside, this has somehow introduced dependency between the classes. In particular, this results in the connascence of type and connascence of position.

Other than that, during the implementation of the game driver, the user is prompted to provide the turn limit and eco points target when challenge mode is selected. Since Display class only has method that reads a single character, so we can only allow the user to select a factor of 1-9.

In order to solve the problem, we suggest adding a method that allows the system to read a few more characters or the entire line. One of the main advantages of this is the system will be able to accept more information from the user input, but on the other hand, more information means more processing or checking has to be done to ensure the data the user has provided is correct.

Our positive opinions regarding the game engine

From our experience of using the game engine, we find that it has fulfilled three among the five principles in SOLID.

The first one is Single Responsibility Principle (SRP), which states that a class should have one, and only one, reason to change (Robert C. Martin). This is observed from the World class, which is responsible of running the game. Basically, it needs to continuously process the turns for each actor and every stuff within the game, including the Player as well as the dinosaurs, and also any items and grounds that can experience the passage of time. Particularly in an actor's turn, there are a few actionable options. We see that this is handled exclusively by the processActorTurn method created within the same class. So apparently, the main responsibility of the World class is to keep the game running, and the methods that are contained within it all work together to fulfil the same responsibility. If anything needs to be changed, all the pieces are right there, which means it is a cohesive class. More importantly, the changes are for the same reason. In other words, it is not a God class and this is just in line with the Single Responsibility Principle (SRP).

The second one is Open/Closed Principle (OCP), which states that software entities should be open for extension, but closed for modification (Robert C. Martin). This is observed from the relationship between the Item class, Weapon interface, and WeaponItem class, such that, the Weapon interface supplies the methods needed by weapons, and when an item implements this interface, the item can be used as a weapon, which is a kind of extension. This is exactly what's done by the abstract class WeaponItem which extends Item and implements Weapon. Such brilliant idea of using an interface to add features to an existing program without changing the code and produce a whole new hybrid class is what said to have supported the Open/Closed Principle (OCP).

The third one is Liskov Subtitution Principle (LSP), which essentially means that an instance of a subclass is expected to be able to do what the base class can do. In short, objects should be replaceable by their subtypes. This is observed in the method signatures of many methods within the abstract classes. One of the example is the execute method in the abstract class Action, which specifies the input type of the first parameter as the Actor class, in order to accept any subclasses of the Actor class as an input and perform the same internal logic. This is somewhat emphasized by the Liskov Substitution Principle (LSP).

Finally, most variable in the engine class are made private, with public setters and getters provided. By doing this, encapsulation is achieved thus providing more control on the variables. This prevents privacy leak where anyone is able to access and change the value of the attributes. Furthermore, updating attribute through setters also provide the benefit of data integrity, as it can be modified to only accept valid inputs, make the program more robust. The connascence within the system will also be minimised as only one modification to the system will be required to maintain the overall correctness of the system instead of every usage of the attribute when new changes are added, which is a good practice during software design.