Extensions for Engine Recommendations

This document represents the conflicts and inconveniences faced during the duration of our Assignment, with major involvement of restricted capabilities from altering the engine class to ease our implementations.

Environment the World and its Actors

This section denotes the recommendations we propose to help improve the accessibility of all engine classes related to the creation of the world and its objects.

World | Getting map instances

During the implementation of adding additional maps into the game, we realised that there were no given getter methods initialized within the World class that could allow us to retrieve the maps. Since the maps are added into a World instance (which is an arraylist) to store and generate objects, access to such maps could have been easily obtained from calling an innate getter method should it had been present. Not to mention that the World class has all its class variables in protected format (for encapsulation purposes), leaving us with no accessible method in retrieving map instances added into World.

We tackled this issue by initializing a dedicated arraylist in the WorldBuilder class and adding the instances of the maps into said variable, which essentially functions the same as the World class arraylist. This is redundant as the same instance of these objects are added into 2 separate arraylists, which could account for additional memory relocation and time complexities. We truly recommend the inclusion of getter and setter for the future of World class for better ease of access and to reduce redundancy.

World | Turn tracking

Within the World class, there are currently no methods that deal with tracking the turns of the game. We suggest that a turn counter be initialized within the class and is implemented in it's run() method. The addition of said counter would give better turn representations for other instances of classes to be utilized for. An example of this implementation is beneficial for the Player class in tracking the current turns before the game ends. Our suggested implications is to shift the game dependencies to more centralized, instead of deploying an independent turn tracking system for every unique class. Although this may generate more dependencies within classes, the benefits of said implementations would outweigh the drawbacks.

Location | Ground types

As we progress through the assignment, implementations we develop that utilizes location fall within a certain pattern of repeating codes. What we are highlighting here is the constant checks we had done throughout the program to determine if the current ground an actor is at is of which type of ground sub-object. The implementation of an additional method within the location class that would deal with checking ground instances would be beneficial to reduce the redundancy of convoluted code within the program as a whole. The suggested method could return a boolean indicator to address if the current ground instance equates its input parameter, that being the ground type needed for.

Actor | Actor types

Similarly to what we suggested above, the implementation of a method that checks for instances of different actor types would be beneficial. As constantly throughout our implementation of the program, we relied on instance checks to determine if the current actor was of which variant. The code written for our program could significantly be reduced, simplified and more visually pleasant had this method been available. On how it should be implemented, as what we've mentioned above, the method could return boolean indicators in a similar fashion.

Positive Opinions

Engine | Item

The Item class has one of the best implementations in the Engine package. For our assignment, it provided almost everything we needed and also allowed us to create functional subclasses such as Food and Egg that inherited its methods. The tick() method proved to be one of the most used functions in our program as it allowed the passage of time for all our food items and egg items. Our subclasses could also add more methods and extensions without modifying the existing Item class, and this applies the Open-Closed principle.

Engine | Actor

As all of our dinosaurs and the player extend the Actor class, we did find a lot of the methods provided to be sufficient for our implementation. The fact that the Actor class had an inventory attribute and the respective getter and setter methods proved to be extremely useful as it allowed the player to pick up items and add them to their inventory, as well as allowing eggs to be added to the dinosaurs. This ensured that we didn't have to code these separately in each dinosaur and player class, and this carries out the Don't Repeat Yourself (DRY) principle. In addition, the Open-Closed principle was also applied as the methods in the Actor class required no modification and none of them interfered with the implementation of any extensions. We also found the playTurn method to be convenient as it allowed the subclasses to decide on an action for every turn.

Engine | PickUpItemAction

This Action class was remarkably easy to make use of and although it appears very simple, its application allowed the Player to pick up almost any item it finds at the location. As it only executes this function and has no other obligations, the Single-Responsibility Principle has been enforced here.

Engine | GameMap

The GameMap class provided several useful functions such as the locationOf, getActorAt, and contains. These methods were prevalent in our behaviour classes as we used them to retrieve various actors and their locations. As it is in charge of the game's map and does not modify or affect other classes, the Single-Responsibility Principle is applied here.