Recommendations for extensions to the game engine

1) **An editable global class**
   One of the more prominent problems we've encountered when coding this game is the lack of ability to edit the World class to implement global functions like rain, or simply per turn prints like eco points or dinosaur statuses. Our recommendation for a solution would be a sort of global controller class that is in some way editable. It could be called during each turn by World as part of the run() method, or perhaps as part of the GameMap's tick() method. Either way, it would let developers implement global events more cohesively, and avoid implementing events like rain in the Player class or the like. This would of course increase the complexity of the engine, and might have limited functionality outside of global events. However, depending on the game, it might become even more useful, such as for spawning actors or items throughout the map without reliance on items or grounds like eggs or dirt.

2) **Display being passed into tick() methods**
   One of the other limitations of the current engine is the fact that the Display class instance isn't being passed into tick() methods. This limits the ways in which we can display non actor related events like rain, what happens in Ground objects etc. The global class mentioned above could help, but simply passing a Display instance into more per turn methods like tick() could also help alleviate the problem.

3) **Ground instances knowing their location**
   Another limitation that we encountered is the fact that instances of the Ground class do not know what their location is. This is not the case for instances of Location, which know what map they're on, and isn't hard to implement, so it doesn't make that much sense. The Ground instance's location is passed into some of its methods as well, meaning that it is expected to be used. A simple change to the constructor and Ground class would solve this issue. It would make it so that developers didn't only get to access the Ground instance's location in certain predetermined methods, and allow for more flexibility.

4) **Display allowing to read inputs other than char**
   Display being the Class that manages I/O for the system suggests that it should be used to read/write to the console. But the methods it currently has only allows for reading char. For cases like needing the player to input the number of moves or Eco Points for the Game Mode Challenge we would need to use Scanner outside the Display class. It is possible to extend the class itself, but it wouldn't make much sense to make a child class just to add a

method to read integers from the console when it can be added in the Display class. Plus doing so will bring the Single Responsibility Principle (SRP) too far by dividing the classes too small. The advantages of this would be that only one Scanner is needed for each display and there won't be confusion of scanner and display being used simultaneously for the same purpose.

Justification of positive opinion of game engine

1) **It adheres to the SOLID principles**
   The game engine follows the SOLID principles of Object-Oriented Design which keeps the design clean and easily extendable. All classes follow the Single Responsibility Principle (SRP) so the code was easy to understand and to use. There weren't any God classes with too many responsibilities.

   Other than that, being unable to modify code in the engine and it being designed so that the game can be extended without modifying the engine code forces us to follow the Open/Closed Principle (OCP). Additional interfaces such as ActionInterface, are also provided in support of this. Advantages are being able to extend the game without risk of breaking old features.

   The Liskov Substitution Principle (LSP) was also used so that extensions can be made to classes and still be able to function with previous code. For example, the in the playturn function can be applied to actor class and all of it's child classes. This can also be seen as a form of polymorphism. With this, we can add subclasses without the need to change the base code; it allows greater flexibility and convenience to extend the code.

2) **Encapsulation and ReD**
   The engine code has a good amount of information hiding to provide a simple interface for each class. Other than making the code easy to use, it also prevents unnecessary connascence which reduces dependencies. An example of this would be in the GameMap class. Methods and attributes have had their access controlled using modifiers like for the method createMapFromStrings() that has the modifier private because it's use is only for that class. The attributes of the Location class is also private to prevent outside access and modification. Their access is then controlled by getter methods which can ensure no privacy leaks occur (e.g. getExits will return unmodifiable list of exits).