FIT2099 Assignment 3: Recommendations for extensions to the game engine

Problem 1: Difficulty in determining which Action to execute in the playTurn method

Given all valid actions an Actor can execute in one turn, the playTurn method needs to determine which Action that the Actor prioritizes and should execute on that turn. The problem is that the existing game engine does not provide an effective way of doing so.

Recommendation:

- Create a constructor for the Action class (as well as its child classes), that takes in a positive integer representing the priority of the Action. A class level attribute should be created to save the value of the integer, let's call it *priority*.
- Create a public method that returns the priority of the Action.
- Create a class that implements Comparator (similar to the SortHotKeysFirst class in Menu class) to allow Action to be compared with each other and be sorted according to their priority. Since all Actors will use priority as a deciding factor for with Action to execute in the playTurn method, this can be put into a static Utility class.
- In the playTurn method, simply sort all of the valid Action and return the first one.

This works well with classes that implement the Behaviour interface too. Though different Behaviour classes might return the same type of Action, they can create Actions with different priority. For instance, WanderBehaviour can return a DynamicMoveAction of priority 1, while FollowMateBehaviour can return a DynamicMoveAction of priority 3. We will still be able to discern which Action is more important to execute.

Benefits:

- Produces **neat code with little ambiguity, hence easier to maintain**. We do not need convoluted code in the playTurn method, since all we need to do is to sort all the possible Action. We eliminate the need of ever using instanceOf in the playTurn method and **reduce dependencies** effectively.
- As mentioned above, different Behaviours that return the same type of Action can work well with this design, such that we still can tell which is more important. This is not possible with the existing game engine, since we are not allowed to modify it.
- Different Actors can determine the order of Actions that they want to prioritize, for example Stegosaurus might prioritize breeding over feeding, while Allosaur does the opposite. The design greatly **increases the flexibility of the system**.
- Based on the previous two points, future designers of the new game clients will be able to add as many Actors, Actions or Behaviours as they like and make use of the design's flexibility to decide the priority of each Action for each Actor and Behaviour. No changes to the code base would need to be made, other than readjusting the priority values of relevant classes. This **adheres to the Open-Closed Principle**.

Disadvantages:

- Programmers might pass in negative values as the priority by mistake. This can easily be solved by using a static factory method to throw exceptions whenever this occurs in the debugging stage.

- Introduces connascence of values. Programmers must be careful when choosing values to pass a priority to not mistakenly put in the wrong value. However, this disadvantage does not outweigh the benefits the new design provides and is manageable. Being careful of which Action to prioritize would be a problem in the existing design due to the limitation of not changing the engine code anyway.

Problem 2: Inflexibility of the World class in the game engine

The following two recommendations involve improving code in the World class, through making World class an abstract class and creating abstract methods to give future programmers flexibility to control the program flow to suit any kind of game.

Problem 2(i): Inflexibility of the run method

For events to occurs through the entire map, it needs to be in the World class's run method, as that is the method that runs the whole game turn by turn. The problem is that programmers will not be able to add global events without extending the World class, then override that method. The overriding method would eventually be a copy paste of the run method with only a few lines of additional code, which is not ideal.

Recommendation:

- Add an abstract method runGlobalEvent in the beginning of the while loop that runs the whole game in the run method
- Programmers simply need to define that method to add events across the map.

Benefits:

- Prevent copy pasting code, which adheres to the Don't Repeat Yourself principle
- It is a simple solution that will not complicate code unnecessarily

Disadvantages:

- Games that do not need events across the entire map need to implement an extra method. This should not be a big problem since we are only introducing one extra method, which can have an empty body if it is not required for a game.

Problem 2(ii): Inflexibility of the processActorTurn method

The processActorTurn method in the World class is implemented such that the sequence of processing is fixed.

For every Actor, it will

1) Collect each Action that can be done to every item in the Actor's inventory
2) Collect each Action that the Actor can do to each adjacent square Ground or Actor
3) Collect each Action the Actor can do to items on the same square it is standing on
4) Finally, allow the Actor to determine which Action to execute and display the results.

The problem is that programmers might want to change the processing method for different Actors. For instance, dinosaurs should not be able to pick up items, but it will still receive this Action.

Recommendation:

- Decompose processActorTurn into a few smaller public methods, namely separate the three loops that carry out 1), 2) and 3) above into three small methods.
- Make processActorTurn into an abstract method that all classes that extends World class need to implement

Benefits:

- Future programmers can extend this class and override the processActorTurn method to define their own processing sequence. They should be able to achieve this by utilizing the ActorInterface and capabilities to add methods for discern between Actors, then add some if conditions in the new class's processActorTurn method to determine the processing for different Actors. Actors will not have Actions that are not related to them. This design gives the system extra flexibility and utilizes the existing code base.
- Future programmers have access to the decomposed methods in World class readily available to be called if needed. This allows reuse of logic and adheres to the Don't Repeat Yourself Principle.
- The decomposed methods fulfil the Single Responsibility principle since each of them will be in charge of doing one job only.

Disadvantages:

- This recommendation is easy to achieve and does not have any disadvantages to it.