Improvements recommended for the engine.

In this documentation, our group (Tute08Group02) will try to list out our opinions on the game engine that was given to us for our Assignments.

What improvements need to be made:

- Iterating the map to find specific items / grounds : A major problem we faced at the start of the assignment was the lack of method within the engine itself to know who / where each of the specific grounds / items that we are looking for. This forced us to implement the functionality through a double for-loop which loops through the entire map within its x and y boundaries to do simple operations such as finding different bushes / trees (for animals to eat) within the map etc. While this method ultimately works, it might become a problem if we were to expand the project and either add more maps or make the current map much bigger. This would significantly increase the runtime and hampers the overall performance of our game (since this method is used by almost all the major functionalities that we implemented into the map.) thus leading to worse user experiences.

  The fix :

  The fix that we propose is to modify the gameMap class to allow us to store all instances of the specified type (items/ground) along with its location into an BitMap (similar to the actors' one within the ActorLocation class. This will be updated every round to get the latest locations of all the items on the ground/ the grounds itself. The getGround methods can be refactored and made so that they take in a parameter of the ground that we are looking for and return the nearest instance from the BitMap. This prevents us from needing to iterate the map every-time ("**Don't Repeat Yourself**" (**DRY**)) we need to find the items which will significantly reduce the runtime and in turn make the user experience much better. It also helps the program to fail faster which goes in line with the principle of "**Fail fast**"**(FF)** since we would only loop through a small number of items instead of the entire map to find out whether the ground/item that we are looking for exists. An issue with this however is that we might have some problems with implementing it properly if we were to add a new map instead of having just extensions to the map which would mean that in order for this to work, we will need to find a way to combine both list and make sure they are iterated accordingly so that actors from both map are able to access the list and iterate through to find the nearest one to them (since there can be a closer ground even if the actor is on a different map than the ground itself).

- Checking adjacent actors / objects : Another inconvenience we faced when creating the game was the lack of method to check the surroundings of our actor. For example, when we were implementing the Allosaur's attack action. We had to manually code it within the Allosaur class itself to check whether its x-1 (left) or x+1(right) contains a Stegosaur which they can fight. In the context of this assignment, this seems fine since only a limited number of functionalities require us to check the surroundings but for future users of the engine who might design more sophisticated games, they will find it very inconvenient to have to hard code these methods when they require it for more of their functionalities.

  The fix:

  The fix we propose is to add a method within the actor class (and for ground class) which checks each surrounding hex (i.e. (x,y-1),(x+1,y)) etc. This method can be reused by the user each time the player wants to look for the surroundings of the actors / grounds to implement functionalities such as (fighting an adjacent actor, trading with an adjacent merchant) which are fundamental in most games. This will also reduce the amount of repetitive code from the game designer thus making the design cleaner and easier to refactor in the future, A disadvantage of such implementation would be that it would force the actor/ground to check all possible hex when the game designer might just want it to check left and right (player can only fight with enemies beside them but not in other directions) which will then lead us back to the same problem we had which is to implement those methods one by one in each of the classes that requires them.

Which game engine parts are useful and convenient:

- Over the course of doing the assignment, we realised that the capabilities methods coded inside the Actor class which includes **addCapability(), removeCapability()** and **hasCapability()** are extremely intuitive and flexible to apply especially when trying to add new features to the actors. For instance, when a dinosaur in our game becomes pregnant, we need to ensure that it does not breed again until the breeding cooldown has passed. For this, we added a **hasEgg()** capability which comes from the enum Class **CanBreedCapability()** to represent the dinosaur is currently pregnant and cannot be a valid partner for other dinosaurs of the same type to breed with.

<u>Advantage (Reference to design principle):</u>

The implementation of the Capabilities class helps our code to be in line with the “Don’t Repeat Yourself”(DRY) principle since we only need to add or remove the capability we wish to apply on the specific actor and check it using **hasCapability()** that is invoked under certain conditions such as in the **FollowBehaviour()** or **FindItemBehaviour()** class**.** It helps us to avoid from coding the same checking method for every actor by storing the actor’s capabilities under the actor class. Moreover, it ensures we have met the “Fail Fast”(FF) design principle because the decisions will be made based on the Boolean result of hasCapability() for the individual actors.

- Engine class contains a lot of abstract class such as **Action, Actor, Ground, Item, WeaponItem** which provides a foundation for each of the classes. This makes it easier for us to implement new features to the game such as Bush where we can just simply extend the abstract class of Ground and inherit all the methods (or override them if needs be). If we were to add a features such as Bushes and Trees without the abstract classes provided in the engine (along with its Tick method), it would be more time - consuming and challenging to program them.

  <u>Advantage (Reference to design principle):</u>

  In short, the abstract classes provided in the engine covers most of the functionalities we need to add in a game which makes it much easier to start coding from the get-go. If the child class wishes to have their own functionalities that are different from the parent class, they can choose to override the methods in the parent class. By doing so, the RED principle has been met since the dependency relationship only applies for the parent class if we do not override it thus reducing dependency. For example, Player and Dinosaur class extends the Actor abstract class and we can override the playTurn method whereas Tree, Bush, Dirt and Lake class extends the Ground abstract class thus we override the tick method to accomplish different tasks such as creating fruits, creating fish etc.