Extension Proposal

Group Name: KarHan
Group members: Ng Wei Han, Ong Kar Kei

Behaviour
The Behaviour Interface can be moved into the engine package to reduce dependency in the game package.

Instead of storing a collection of allowable Behaviours in the Actor class, another Behaviour abstract class can be created to store a collection of behaviour for each Actor. The class should also allow the addition and deletion of any behaviours. This allows easy manipulation of behaviours of Actor as with the current implementation, behaviours are hard coded into the Actor class and it can be time consuming to find a behaviour and delete it.

The Behaviour abstract class complies with the Open-Closed Principle as it is easier to extend the Actor class when new Behaviours are added.

Ground
The Ground abstract class did not implement the toString() method unlike the Item class. This can be an issue when we are trying to retrieve and print the name of the ground that the Actor is standing on. Instead of printing the name of the ground, it will print the memory location of the Ground object. With the current implementation, the only solution is to hard code the name which violates the Open-Closed Principle as if in the future, a different ground object is added, the class will need to change. To solve this, add a name attribute and a toString method to retrieve the name of the Ground in the Ground. Thus, the name of the Ground can be obtained and can be used for printing in display or for specific Actions.

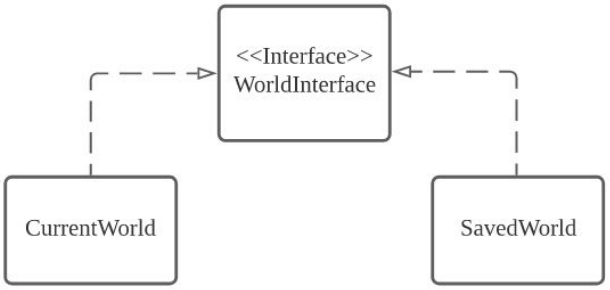
Menu
In the current implementation, Menu is only able to show 26 options for actions and 8 options for moving direction. This may not be an issue for the current system, however, in the future, when more Actions are added, menu may not be able to show all of the options. Therefore, other than using numbers and lowercase alphabets for choosing action, uppercase alphabets and symbols such as %, $ can be used as well. However, this may confuse the user and it is harder to type the option onto the console.

Item & Location
Unlike Ground, Item does not have a canActorEnter method that prevents an Actor from standing on an Item when it is on the ground. Since it is not reasonable to implement this method in the Item class since most items need to be picked up and

dropped while standing on it, the Location class should allow the implementation of limiting access of a location to the Actor through the x and y coordinate of the Location. Thus, this will allow items to be dropped off at a location and the Actor is unable to stand on it.

World

The current game does not allow players to save their progress. Once the game is ended, the entire application is reinitialized and reset. Therefore, we can add a SavedWorld class which copies all attributes from the normal World to the saved one, before creating a new World. Then, we would offer an option in the user interface to load saved progress. When the option is chosen, it would copy back the attributes. In this case, we would create a WorldInterface class in which the CurrentWorld and SavedWorld will implement. This also opens up for extension opportunities to different types of world. Overall, the advantage of this feature is that player can now explore the game more and try out different game features. In contrast, the implementation would only mean only 1 saved progress which is the disadvantage.



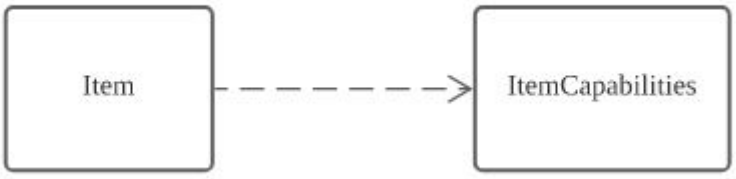
FancyGroundFactory

The current implementation of creating Actors in the map does not follow the Reduce Code Duplication principle and Open/Closed principle. The creation of new actors can only happen when we hardcode the values in the Application class. Instead of doing this, we can create a FancyActorFactory which basically initializes different types of actors in the game. This feature would reduce code repetitiveness in Application class and we can easily add and remove actors to be included in the game.

Item & WeaponItem

WeaponItem is essentially an item which has the capability of damaging actors. However, if we were to create special items for other purposes, we would still use

the Capability class to add a capability to an item. It is not a good design because we are assuming that all types of games have weapons. Instead, we can make the feature more general. We can create an enum class of ItemCapabilities which include different features for items such as weapons, cooking and others. Therefore, this would not imit the Actor interaction with different items. In the Actor class, instead of using getWeapon method, we can use getSpecialItem, in which the method will check for the constant in the ItemCapability class and return the item with the capability. This method follows the Open/Closed principle which allows for more game features regarding items and reduce unnecessary dependencies.



GameMap

Currently, random locations cannot be generated in the Location class. We have to create another class which is associated with the Location class to check for its max and min x and y values. This is redundant because we only have to add a method in the GameMap class to generate location easily. This reduces unnecessary dependencies for GameMap class. Therefore, this allows different game features that can be added such as spawning grass randomly and special items generated from time to time. However, the downside of this implementation is that it might cause different displayed characters to be overwritten by some other character or objects that are not supposed to be at the certain location. This can be easily solved by using a conditional statement and check the validity of a location when it generates it.

Menu & Display

In this game, the user input is only based on Player's action and outputted in the menu. This is not flexible at all because we cannot create other types of user interface from the engine package. Therefore, the solution is that we can create another class called GameActions. GameActions would allow different actions which are related to initialization of the game such as choosing game modes and loading saved progress. In the showmenu method of the Menu class, it will then check for game actions first. Once game actions are run successfully, it will not check for them again. This can be done easily by setting a boolean variable. Therefore, this feature allows the different extension of game user interface which follows the Open/Closed principle.

Probability

In most games, there is always a random factor in them. Therefore, we can include a Probability class to the engine which allows the creation of randomness in different

parts of the game. This can be done by implementing a method called calculateProbability which uses the Random class. Therefore, this implementation allows flexibility and it reduces code duplications.

Inventory

When a player's inventory has multiple of the same item type, it will print out multiple same messages to retrieve the item. This does not look aesthetic to the player at all and the design does not follow Single Responsibility Principle because Player now has an extra responsibility to deal with inventory-related items. Instead, we can have an Inventory class. This class would store a list of items based on only one type such as food, weapon, and others. When duplicated items are added to respective inventory, it will store them in a collection. Furthermore, it will also have a new collection which only contains unique items and its count in the inventory. This would prevent multiple displayed messages of the same item in the menu. The actor class would then have a list of inventories as well to store different types of items.

Actor

Actor class should include a moves field to keep track of their current moves. In that way, we can reduce code duplication as most of the actor's child classes are required to know the current moves to make a decision. This can be done by adding 1 to the moves field each time the playturn method is called.