Design Rationale - Assignment 1

Requirement 1

Design Goal

In requirement 1, new items need to be added into the game which are large bolts and metal sheets. To adhere to the Open-Closed Principle, I decided to create separate classes for these two game items and extend them from the already existing Item abstract class that was provided in the engine files. By doing so, I have followed the rules to be open for extension and closed for modification where I did not modify the Item abstract class to add the new game items but instead added on by creating new classes to extend from them.

I added the game items to the game map using the Application class using the methods from the Location and GameMap classes provided in the engine file.

Pros

- Maintenance in the future will be easy
    - Issues with a game object means an individual can look for the specific game item class and debug it from there
    - If items are to get special properties, they can be done separately from the abstract class where other items may not have these special properties
- Easy to create game items and add them into the game map

Cons

- Many items may have similar properties that may result in duplicate codes
    - The only difference between each item are its display character
- Directly adding code in the Application class to add game items into the game map will make it tougher to maintain in the future
    - Application class will become large and will be a hindrance to detect bugs under multiple lines of code

Alternative

Creating a separate method or separate class to handle adding items to the game map may help ease the readability of Application class as more items get added in the future

Requirement 2

Design Goals

In requirement 2, Inheritrees are added into the game where they act as Ground objects that spawn SmallFruit and LargeFruit Items depending on if the Inheritree is a SmallTree or a BigTree respectively. A SmallTree will mature into a BigTree after 5 game ticks.

From the requirements, I had changed the Ground type at certain locations on the game map to become a Tree, where it is similar to the other Ground types given such as Dirt and Puddle. Tree ground will be the location for where Inheritrees will grow. Inheritree is made to be an interface where the classes SmallTree and BigTree will implement the methods required to define an Inheritree. The Inheritree children will have to increment their age every time they are asked to grow fruits each round and the age will be checked to see if the tree matures from a SmallTree to a BigTree. With an interface, SmallTree and BigTree can be confirmed to have all methods to represent an Inheritree's behaviour which follows Liskov's Substitution Principle.

The current age of the tree is passed on to the next mature tree phase where the age continues to increment. As they are two separate classes to identify a SmallTree and a BigTree, it adheres to the Open-Closed Principle as a new object will be responsible for the continuation of the tree's ageing process. This will allow for extension of other types of trees to add on to the cycle.

Inheritree interface has a growFruit method where its children classes will have to implement. For this, I have created a SpawnerBehaviour class where it checks for the exits of a certain location and returns a randomly picked exit location, which works similarly to the findExitFromHere method inside the Actor class. I implemented the method to search for exits based on the radius given. This will allow an extensive use of SpawnerBehaviour to allow for other spawner type Grounds to utilise the class to mimic the spawning mechanic.

Pros

- Extensive use of SpawnerBehaviour for future spawner type Ground
    - Future spawners can utilise SpawnerBehaviour to look for exits
- Inheritree can have more tree growth phases which can be easily implemented

Cons

- Slight refactoring is required to link the previous Inheritree growth phase to the next Inheritree growth phase

Requirement 3

Design Goals

In requirement 3, HuntsmanSpiders are given more characteristics with a spawner to add more HuntsmanSpider into the map. A crater is used to represent the spawning location of the HuntsmanSpider which utilises the SpawnerBehaviour that was implemented in the previous requirement.

SpawnerBehaviour would need to use the same approach to find an exit but different conditions as only one Actor can exist at a location unlike Items that can have infinite amount at one location. SpawnerBehaviour would have to check for the areas that are not only unwalkable but also areas that have Actors existing before providing a randomly chosen location for the HuntsmanSpider to spawn. By implementing this method into SpawnerBehaviour, this will follow the Open-Closed Principle as it will allow for future Actor spawners to utilise the method.

HuntsmanSpider will also have to inherit a new AttackBehaviour where it has to prioritise it before other behaviours. This would mean that it has a choice between attacking or moving, it would attack every time. An AttackBehaviour will be required to search for other actors within the immediate surrounding of the attacker before it can create an AttackAction against them. AttackBehaviour is implemented where it will check for actors in the surrounding that have the capability of being hostile to enemies. This capability is currently only available to the Player actor which means the HuntsmanSpider will only attack the Player class and not other HuntsmanSpider. This would allow for extensive use of AttackBehaviour for other enemies that would be added into the game as it will allow them to not attack other enemies and only the Player actor.

Since enemy Actors are introduced to the game, other enemies could be added. Therefore, an Enemy abstract class was implemented where it extends from the Actor abstract class and HuntsmanSpider extends from the Enemy abstract class instead. By doing this, Crater will be able to spawn other types of Enemy objects in the future, should there be more enemies added into the game. This complies with the Liskov Substitution Principle where I can be sure that all Enemy classes will have the same method that can be used by the Crater class to spawn them.

Pros

- Enemy type classes can be easily added into the Crater's spawn mechanic
    - Crater only requires the type of enemy to spawn from the Crater
- SpawnerBehaviour gains a new override method for spawning Actors now
    - Allows extensive use of the new method for future Actor spawning mechanic
- AttackBehaviour can be used by other Enemy classes that can attack Player classes without the need to reimplement the logic for it again

Cons

- Crater may have a fixed way of spawning enemies, which would make SpawnerBehaviour a redundant class

Requirement 4

Design Goals

In requirement 4, special scraps are added into the game which are Metal Pipe, SmallFruit and LargeFruit. MetalPipe acts as an Item that can be used as a Weapon, hence it extends from the WeaponItem abstract class. SmallFruit and LargeFruit can be used as healing items to heal the actor that consumes them.

To implement these to have special actions that can be performed with the items when they are in the Player's inventory, overriding was done to the allowableActions class to ensure it can return a list of actions that the item can perform. A MetalPipe can perform an AttackAction to enemy actors while the fruits can perform a HealAction to heal the actor's health points.

Both items override the same method name but they require different parameters to be passed. Metal pipe uses an allowableActions method that requires an Actor and a Location parameters whereas the fruit classes use an allowableActions method that require only an Actor parameter.

Due to this, I have avoided creating a SpecialScrap interface that would have both types of allowableActions inside that will require the special scraps to implement them where only one of the methods will return actions while the other returns a blank action list. This is due to the fact that it may violate the Interface Segregation Principle where an interface is implemented by all special scraps; however, not all the special scraps that implement the interface will have the same fully implemented methods.

Pros

- Prevents repetitive code as special scraps need to override different allowableActions with different required parameters
- HealAction can be used for future healing items in the future

Cons

- With no way to identify special scraps, it would be difficult to add new special scraps in the future with the current implmentation