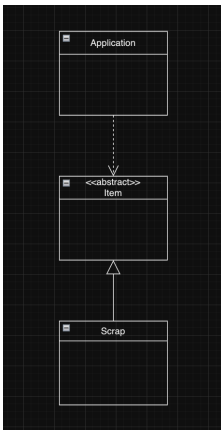
REQ 1

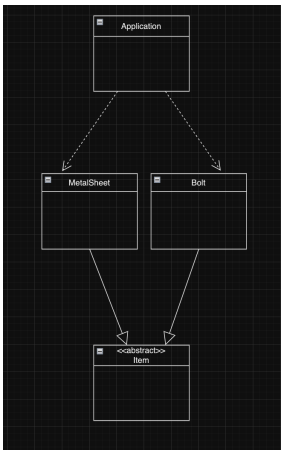
Design 1 : One Scrap class



In this design, a Scrap concrete class is extended from the Item abstract class. The Scrap class is in charge of managing the MetalSheet and Bolt scrap objects and also other scrap objects that will be added in the future . There will be no class added in the system if there are any new items added in. The system will just use the constructor of Scrap class to make different objects.

Although this implementation is simple, the Scrap class has multiple responsibilities which violates the Single Responsibility Principle. To be more specific, if there are any changes for the setting of MetalSheet or Bolt, the Scrap class always needs to be modified. Besides, if there is a method that needs to be performed differently on different items, the method needs to be modified by using “instance of” which will violate the Open-Closed Principle.

Design 2 : Separation of Scrap class into MetalSheet and Bolt class

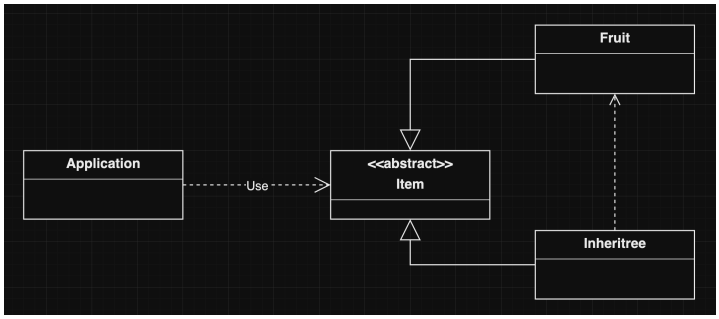


In the design, MetalSheet and Bolt are separated and both extend the Item abstract class. Each class handles one object. In the future, if more items are added, we can just create a new concrete class for it and extend it from the Item class.

The pros of extending from Item abstract class is to avoid repetition of code, which adheres to the DRY principle. With this, more different Items can be added in easily which improves the extendibility of the Item class. Besides, if the setting of an item is changed, we only need to modify the code of the class, which shows the maintainability of the class. Furthermore, each item subclass manages one item, ex, MetalSheet class only needs to handle MetalSheet objects, which comply with the Single Responsibility Principle.

REQ 2

Design 1: One Inheritree One Fruit Class

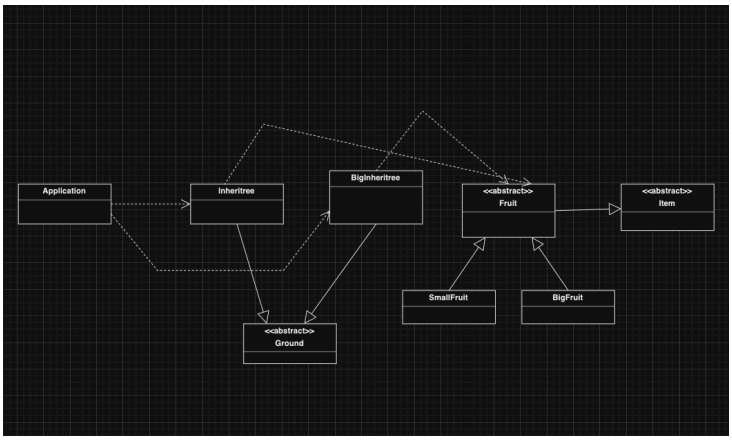


In this design, the inheritree is initially represented by the character ‘t’ and produce smallFruit. When the counter reaches a certain number, it will change its displayChar from ‘t’ to ‘T’ and start producing bigFruit. If there are more types of tree added in, then in the tick method of Inheritree will add the required setDisplayChar method to handle the tree. Also, it will check the character of the tree to produce the correct fruit, for example, if the system checks the tree having a character of ‘T’, then the system will produce Big Fruit.

The pros of this design is that it is straightforward and easy to implement. However, the cons of this design is that everytime when there is some behaviour that needs to be changed, for example, the tree produces different types of fruit or maybe a new tree is introduced, the inheritree class needs to be modified. In this case, it also violates the Open-Closed Principle which states that a class can be extended but not modified. And also, it will be very difficult to maintain in the future if more and more trees or fruit are added in. Hence, it is not very easy to maintain or extend.

Besides, as in the business case, there are 2 types of fruits, bigFruit and smallFruit. In this design, there is only one fruit class , which means its state needs to be changed based on the constructor parameter. Also, the fruit class always needs to check the state of the fruit, whether it is ‘o’ or ‘O’, in order to perform specific actions as different types of fruit might have different requirements. If there are new Fruit types introduced, the class needs to be modified which also violates the Open-Closed Principle.

Design 2: Separate Inheritree and BigInheritree, SmallFruit, BigFruit

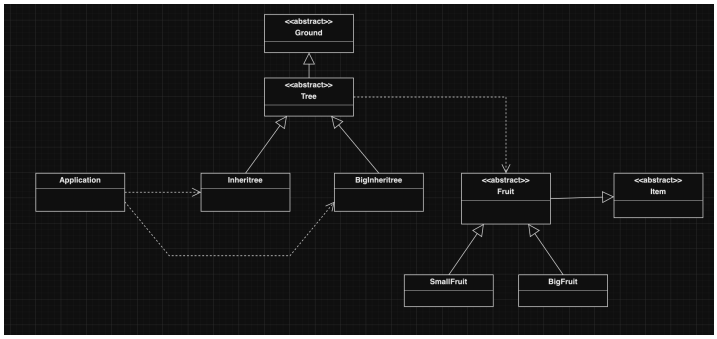


In this design, Inheritree is separated into 2 classes. Each class handle represents a different stage of a tree, ie, Inheritance represents sapling, BigInheritree represents grown-up tree. And both trees extended from Ground, which adheres to the business case requirements.

Besides, the Fruit abstract class provides a common class for SmallFruit and BigFruit to represent different types of fruit. Also, different trees are able to produce a specific type of fruit in each tick, thus tree classes have a dependency with Fruit class.

The pros of this design is that inheritree no longer need to change its displayChar to a different character. In the future, if there is a new type of tree or fruit introduced, they can just extend from the Ground class or the Fruit class, the existing classes are not needed to be modified, which adheres to the Open-Closed Principle. Also, since Inheritree and BigInheritree aren't restricted to specific fruit implementations, they depend on the fruit abstract class instead, so it is easier to expand the fruit types without altering the tree classes which adheres to the Dependency Inversion principle, high and low modules depend on the abstraction.

Design 3: Added Tree abstract class



In this design, a new abstract class, Tree abstract class is introduced. As I noticed that the tick methods of both Inheritree and BigInheritree are sort of similar, where both of them keep track of the counter and either change state or produce fruit, I decided to implement this as a template in the Tree class. So, this avoids the code repeating itself, which follows the DRY principle. Each subclass can then specify particular behaviours, such as the conditions for growth and the type of fruit produced. Furthermore, the tree and fruit classes can be easily extended and maintained if there are new types of tree or fruits introduced, as they just need to override the required methods in the abstract class.

Additionally, as the overridden produceFruit method in both inheritree and bigInheritree perform production of different types of fruits without impairing any functionality that the base class was supposed to carry out. As there is no issue for any operation that works with Tree object to work on its subclasses object, this complies with the Liskov Substitution Principle.

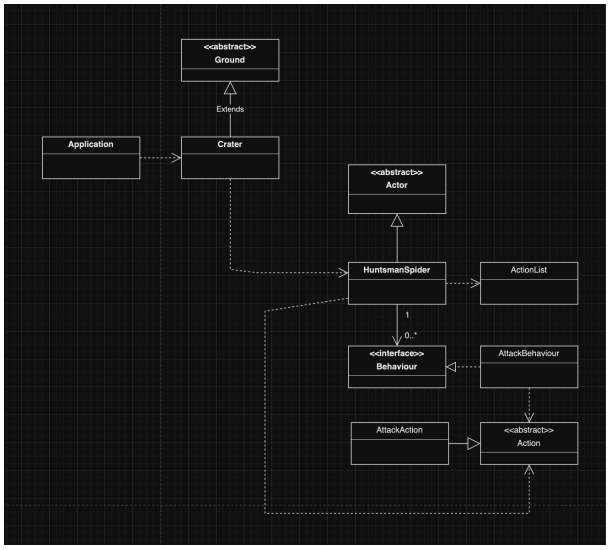
However, there are some limitations in this design, that is, all the trees do need to implement the grow and produce methods. To further improve on this, Grow and ProduceFruit interfaces might be added where different trees can implement if they require these specific capabilities, which allows for

greater flexibility. Also, this would follow the Single Responsibility principle as each class handles growth or fruit production.

However, due to simplicity of the business requirements, I choose to stick to this design as the additional interfaces do not seem necessary at this stage. The current approach strikes for balance in code maintainability and simplicity.

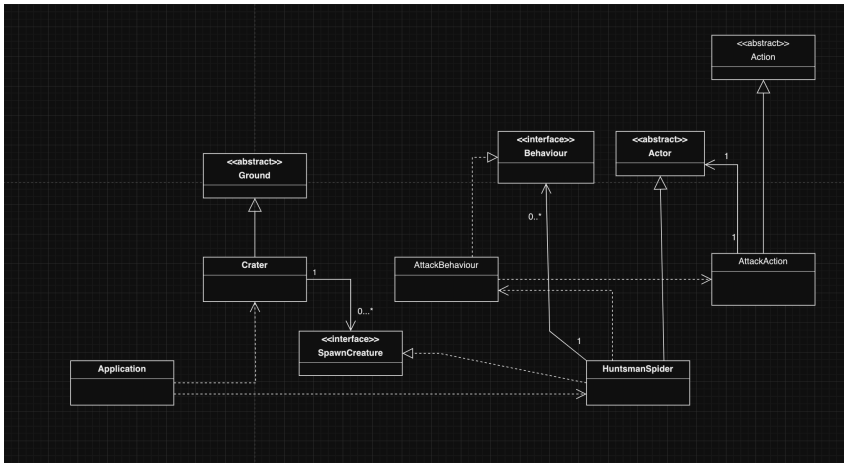
REQ 3

Design 1: Crater has dependency on HuntsmanSpider



In this design, the Crater class extends the Ground abstract class and has dependency on HuntsmanSpider. So, by a certain period, the tick method in the Crater class will spawn a new HuntsmanSpider object. The approach is straightforward, however, it has some limitations. First, the high-module (Crater) directly depends on the low-level module (HuntsmanSpider) which violates the Dependency Inversion Principle. Second, every Crater object can only spawn HuntsmanSpider objects, if there is another creature that has to be spawned by Crater, another dependency relationship needs to be added and the Crater class needs to be changed, which violates Open-Closed Principle. Also, it will add multiple dependencies from the Crater class to other creature classes, which will make the code messy.

Design 2: Add SpawnCreature interface



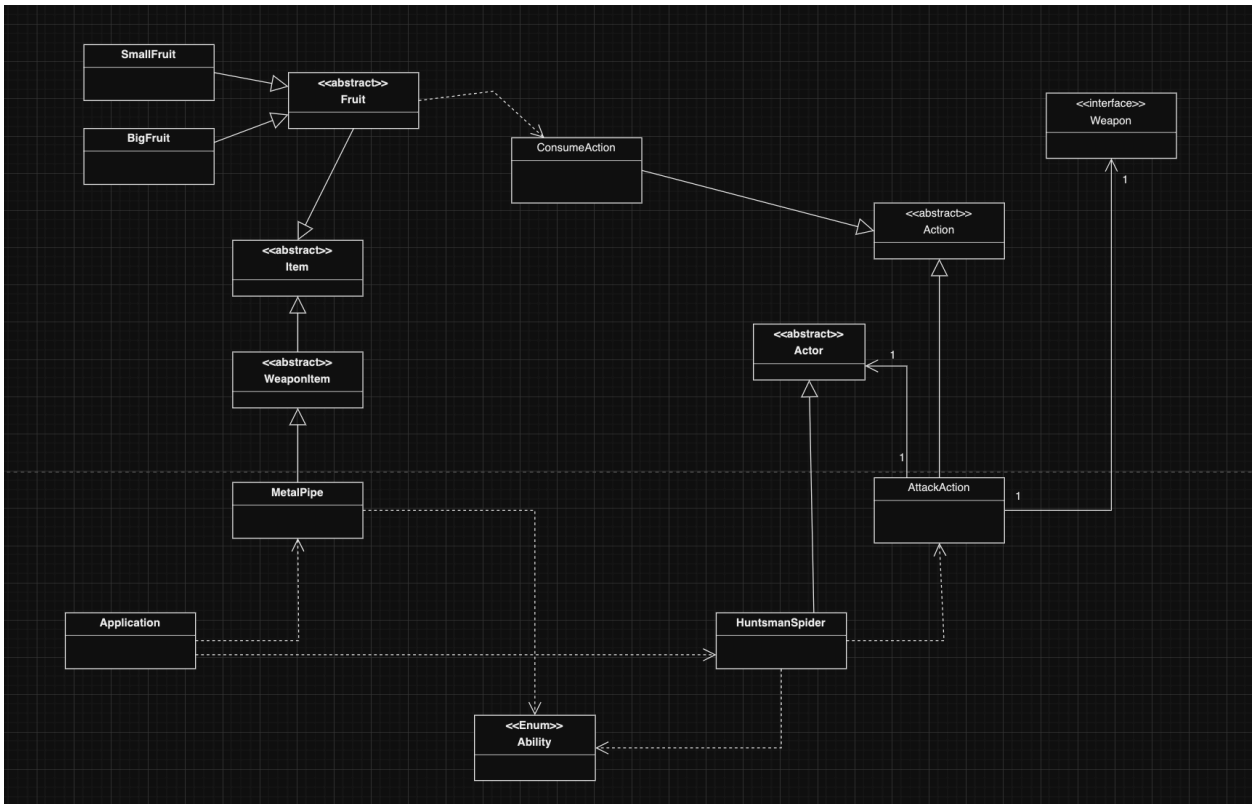
In this design, a SpawnCreature interface is introduced, implemented by the HuntsmanSpider class. In Crater class, we have an attribute, SpawnCreature. In this case, if there is another creature that needs to be spawned by the Crater, it can simply implement the SpawnCreature interface. And also, we do not need to change the Crater class source code in order to spawn the new creature, we just need to define it in the constructor of the Crater class instead. For example, if the Crater is required to spawn Alien, we just need to add " new Crater(new Alien())" in the application. This follows the Open-Closed Principle, as the Crater class does not need to be modified to cater new SpawnCreature.

Furthermore, the HuntsmanSpider also has an association relationship with Behaviour, which is implemented by different behaviours. If there is a new behaviour that the HuntsmanSpider needs to have, it can be easily added by adding the new behaviour class implementing the Behaviour interface. As the high and low modules (HuntsmanSpider and different behaviour classes) both depend on the abstraction, it follows the Dependency Inversion principle.

It is also worth mentioning that each Behaviour class (eg. AttackBehaviour, WanderBehaviour…) is in charge of one action. If there are any changes in certain behaviour, only that specific behaviour class is affected, the HuntsmanSpider or other behaviour class does not need to care about the changes, hence, the class is easily maintained. As each behaviour class only has one job and one reason to change, the behaviour complies with the principle of Single Responsibility.

However, the cons of this design is that a lot of classes will be introduced and might be difficult for the new developers to understand the system.

REQ 4



In this design, a few new concrete and abstract classes have been added. First of all, a MetalPipe concrete class has been added which extends from the WeaponItem class. MetalPipe also has the ability of ‘attack with metal pipe’, so if a player picks up a MetalPipe and adds it to the inventory, the player will also get the ability to ‘attack with metal pipe’.

The pros of doing this is that new abilities can be added to the enum easily without altering how the other abilities are used by the items. For example, if another item called wood allows the actor to attack an enemy with wood, this can be easily achieved by adding a wood object with ‘attack with wood’ abilities. Thus, this complies with the Open-Closed Principle. If these scrap needs to be traded in the future, a Tradable interface with necessary methods, eg. buy or sell method, can to be created which could be easily implemented by the scrap that can be bought or sold.

Besides, the fruits have dependency with the ConsumeAction class, which means that other actors are allowed to consume the fruits and consuming different fruits will heal the player with different points. In this design, only the Fruit abstract class has dependency on the ConsumeAction class, and there is no code repeated in the SmallFruit and BigFruit classes, hence, reduced redundancies and simplifies maintainability adhering to the DRY principle . Besides, ConsumeAction is only responsible for one job which is to consume foods, this comply with the Single Responsibility Principle.

To further improve the design, a Consumable interface could be introduced and food that is consumable can implement it. So, only food that is edible has to add consumable action into the allowed action list, and food that is not edible does not need to care about the consume method. However, due to the simplicity of this business case, I choose to stick to this design, as the all fruits are consumable in and the consumable interface seems unnecessary at this stage. This design strikes a balance between simplicity and maintainability.