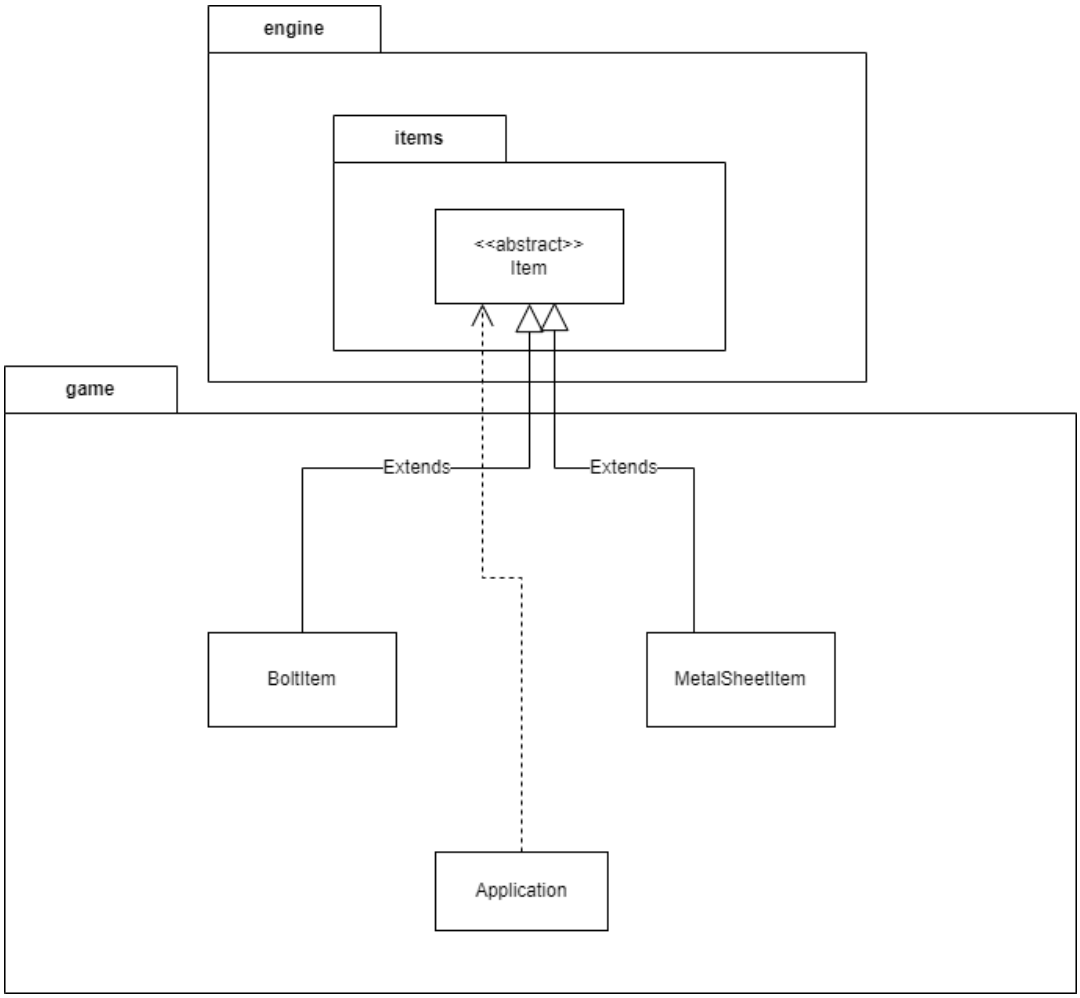
Please note that I have used generative AI for grammar correction.

REQ 1



Implementation:

1.1 Create BoltItem and MetalSheetItem

I have created these classes that inherits from Item, so they can be added to the map and pick up/drop by player

Explanation:

According to my diagram represents an object-oriented system requirement 1, I have create 2 concrete classes which is boltItem and MetalSheetItem and it inherits from Item abstract class. Since the items shared the common attribute they can inherit from Item abstract class to gain the same behavior this follows the principle DRY.

Each class boltItem , MetalSheetItem and Item has a single responsibility which is representing specific types of items. While Item abstract class allow future item to be inherit. This ensures that each class is focused on a specific aspect (Single Responsibility Principle).

Moreover, in future if there is any extension of BoltItem and MetalSheetItem we just need to implement in their respective class. This adherence to the Open-Closed Principle as I able to extend for functionality without modification.

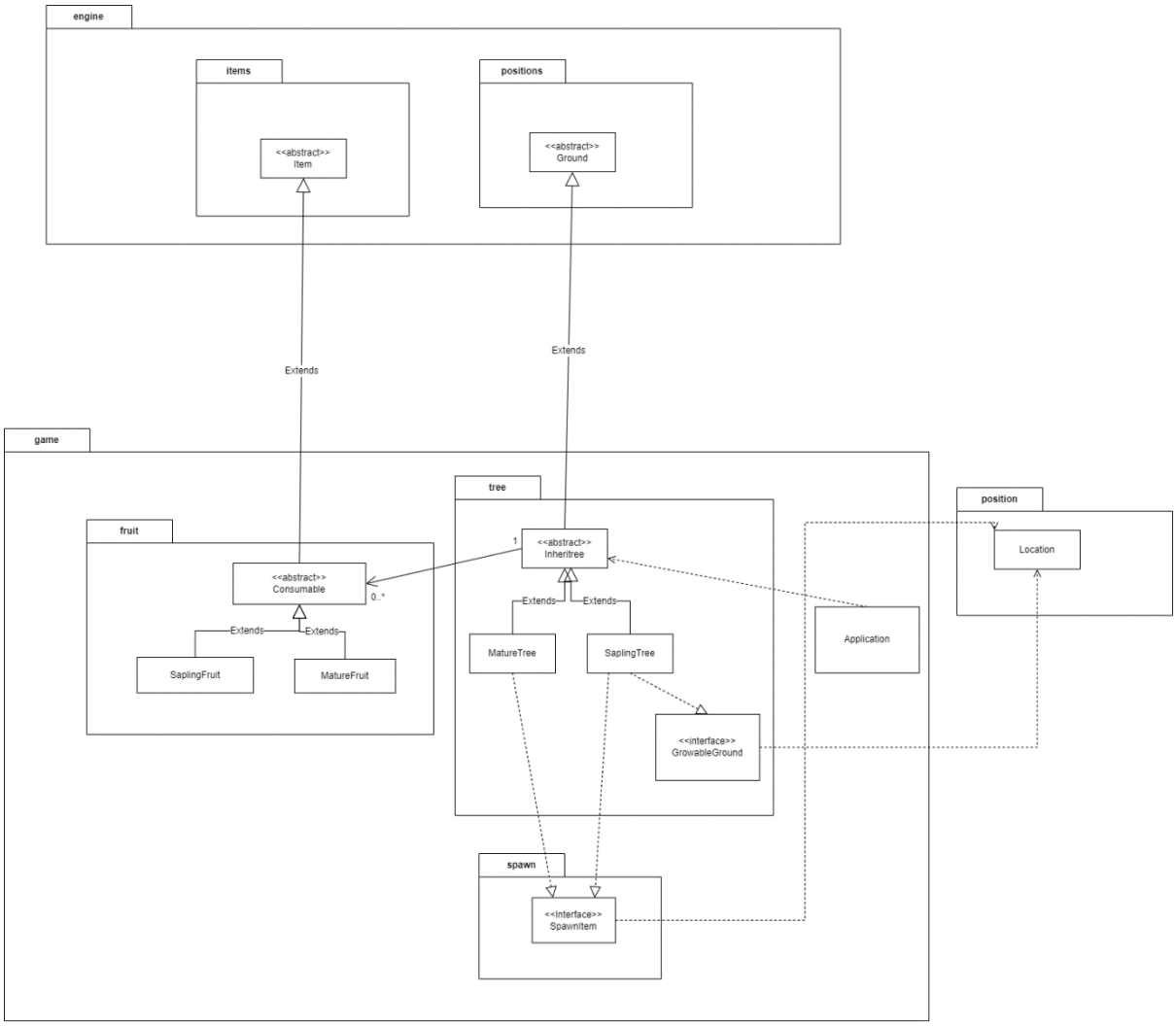
Other than that, since I did not use any downcasting, the behavior of the base class which is item remain consistent and its child class can replace their parent class without changing the behavior of the program. Hence, it follows the Liskov Subsititution Principle.

Besides that, the application class is dependent to Item abstract class instead of each item that added in the application class. This allows to remove dependency from the concrete class. According to Dependency Inversion Principle concrete class should dependent to abstract class instead of other concrete class. So this follows the dependency Inversion principle

The pros of this design are by using inheritance and abstract class, the design promotes code reusability. As all common attributes and behaviors are in the Item abstract class. Moreover, the design allows for easy extension and modification of behavior. New item types can be added by creating additional subclasses of item this promotes flexibility. Furthermore, it is allow for easy maintenance as changes to shared attributes or behaviors can be made in the item abstract class and these changes will automatically reflect in all subclasses. If there is change of behavior of that specific item then we just need to override the method of its parent class.

The cons of this design is when there are many Items added to the game. The hierarchy height will increase and it makes the code harder to track and maintain. Besides that, java only allow one inheritance only so each class can inherit from only one superclass. This restricts the design's flexibility if the subclass wanted to inherit other superclass afterwards.

REQ 2



Implementation:

2.1 Inheritree implementation

An inheritree abstract class is implement by extending Ground. Its child class will have the basic implementation of tree according to inheritree abstract class. The pro of this design is there will be no redundant code as the inheritree provides all the base code for the trees.

2.2 Spawn fruit around trees

For trees that is able to spawn fruit, I have created SpawnItem interface for them to implement. The tree will have its spawn method to produce fruit around its exit each tick. Furthermore, for the tree that is growable, I created a GrowableGround interface for the tree to implement. Thus, the tree can have their own method of growing up. For example sapling tree grow to mature tree after 5 ticks.

The pros of this design are by using interfaces. Each class can implement these interfaces independently. It allows for easier maintenance. Apart from that, the use interfaces allow for flexibility in functionality. Different types of trees can implement the SpawnItem and GrowableGround interfaces based on the tree behaviors. Moreover, the interfaces promote code reusability as multiple classes can implement the same interface.

The cons of the design are by introducing interfaces, when multiple classes implement the interface can add complexity to the codebase, especially if there are numerous classes implementing the same interfaces.

2.3 fruit implementation

A consumable abstract class is created by extending Item. This is for the fruits that is consumable. The subclasses of consumable will be created in the respective trees for it to produce.

The pros of the design will be each fruit that is consumable can inherit from consumable to achieve its base code. By using an abstract class that extends from Item it promotes code reusability. The design ensures consistency as all fruits that are consumable can inherit this abstract class.

The con of the design is when many fruits inherit consumable the code will become complex as managing multiple subclasses of consumable across different trees.

Explanation:

According to my diagram represents an object-oriented system for requirement 2. In this system, I've established a consumable abstract class that inherits from Item. Additionally, I've created two concrete fruit classes that inherit from the consumable abstract class. Furthermore, I've designed an abstract class called inheritree that inherits from Ground and is associated with consumable as 1 inheritree can have 0 or more fruits.

To specify, I have chosen abstract class instead of interfaces because I wanted to include instance variables in both my consumable and Inheritree classes. Additionally, by defining methods with bodies in these abstract classes, I ensured that their child classes wouldn't need to re-implement them, as all tree and consumable exhibit similar behaviors to avoid violation of DRY.

The approach allows to have a consistent behavior across my future consumable items and trees. Any unique behaviors can be implemented in the abstract classes, ensuring that their child classes inherit these functionalities.

Apart from that, I created a SpawnItem interface and GrowableGround interface for classes to implement when they need to spawn item or grow. The choice of interfaces was chosen, as it allows classes that need these features to simply implement the respective interface and inherit all the necessary methods.

According to the Single Responsibility Principle, by giving each class only one responsible. The consumable abstract class's responsibility is to facilitate the creation of new fruit instances when needed, achieved through the creation of concrete fruit classes that inherit from consumable. On the other hand, the inheritree abstract class serves as the blueprint for trees its responsibility is enabling its child classes to have all the base method of tree. Additionally, SpawnItem is for tree to implement if they are able to spawn fruits, where as GrowableGround is for trees that able to grow in the future. So by using different interfaces and classes serve different purposes for better maintainability.

By employing the Inheritree and Consumable abstract classes. Adding new trees or fruits simply involves creating concrete subclasses that inherit from their parent classes, automatically inheriting all required methods without modifying existing code. In my design if the tree can spawn fruit they simply need to implement the SpawnItem interface and since different tree can produce different type fruits. Hence, I have a fruit list in inheritree. If there is any new fruits to be added to the tree just simply call the addFruit method in itemToSpawn method provided by SpawnItem interface to

add new fruit to the existing tree. So with this design I can extend new functionality without modifying the base code this followed the Open-Closed Principle.

In my initial design, I had a Fruit concrete class that all trees depended on, to create their own fruit instances. However this setup violated the Dependency Inversion Principle, which states that concrete classes should depend on abstract classes rather than on other concrete classes. To solve this, I have create a consumable abstract class and all fruit concrete class inherit from consumable abstract class.

With this new design, adding a new fruit in the future simply inheriting from the consumable abstract class. This way, tree will be dependent on the abstract class. This restructuring ensures maintainability and extensibility while following Dependency Inversion Principle.

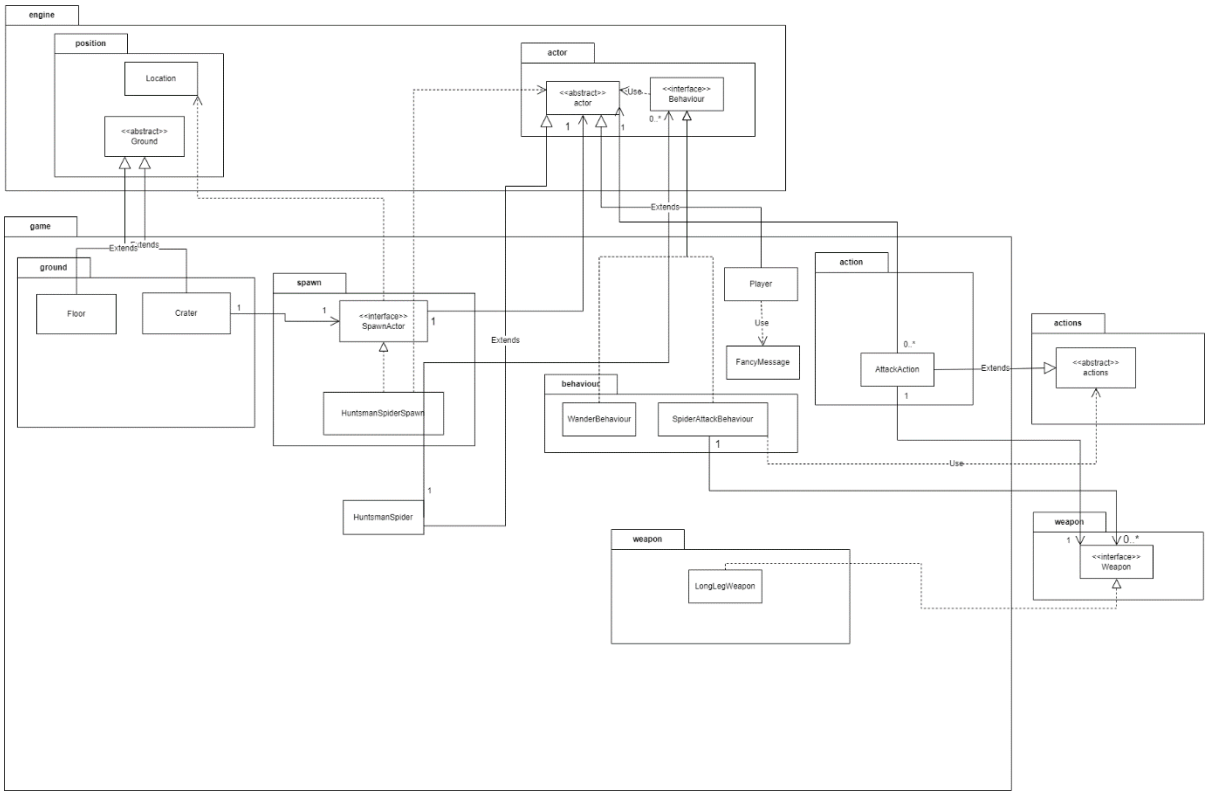
Moreover, I have secured the Liskov Subsitution Principle as I didn't use any downcasting in my code. Hence, the behavior of my base class will not change. In other word, inheritree's child class would be able to replace inheritree without changing the behavior of the program.

The pros of using this design are, it is very easy to understand as in my design all functionality has already been separate to different classes and ensuring one class has only one responsibility. It makes my code easier to understand and maintain. For example, inheritree class act as a blueprint for its subclass, if its subclass can spawn fruits it needs to implement SpawnItem to spawn, if its subclass is growable then it needs to implement GrowableGround. These interfaces act as an add on option for all the trees.

Moreover, it is flexible, the use of abstract class in my design makes it easier for extension in the future. For example, when there is any new function of the tree and fruit I can simply implement in abstract class and their children will have it. So my this design increase in reusability by reducing redundancy. Furthermore, this approach make sure there is consistent behavior across future implementation.

The cons of this design will be, since one class only able to have one parent class. Hence, the tree and fruit wont be able to inherit from other parent class in the future. Furthermore, if there are many trees inherit from inheritree the complexity will increase as the hierarchy height will increase. It will become harder to maintain.

REQ 3



Implementation:

3.1 Crater implementation

I have created a new class Crater that extends from Ground. Pros of this approach is there won't be any redundant code as I inherit from the Ground.

3.2 Changes of Huntsman Spider

I have changed the original implementation of HuntsmanSpider. The map that used to store the behavior was a HashMap, I have change it to TreeMap due to several reasons. The pros of TreeMap are ordered based so the behavior are sorted in ascending order by default while HashMap does not guaranteed the order of the element is consistent across different runs of application. But the cons of TreeMap is it is slower than HashMap and consume memory, however it is consistent and reliable.

3.3 Changes of Floor

I have override the condition for canActorEnter method in floor class. My initial design was put a checking in the wander behavior. If is it floor's display character then the creature won't include the exit. But I found out it would be redundant when I have other players that can enter floor in the future. At that moment I need to override the method or create another wander behavior for that player which is very inefficient.

So I decided to go for another approach which is override the floor's canActorEnter method. So now the floor only available for the actor that has status of HOSTILE_TO_ENEMY to enter. In the future if there are other players I only need to set its status to HOSTILE_TO_ENEMY for them to enter floor. This approach is much cleaner and efficient compare to previous design approach.

The pros of this approach are, the design becomes simpler and straightforward since the floor's behavior is directly tied to the actor's status. So there is no need for additional abstract classes or interfaces. Moreover, directly checking the actor's status, the system avoids unnecessary complexities may come with implementing abstract classes or interfaces. It is more scalable as we can easily modify the canActorEnter method to accommodate future changes.

The con will be if there are many statuses in the future. Managing canActorEnter method may be difficult.

3.4 Show Fancy Message when player is dead

I have override the unconscious method of player by just adding the fancy message.

3.5 Spawning creatures at crater's exit every tick according to the rate

I have created an SpawnActor interface where all creature that wanted to spawn will implement this interface. Then I create a spawner attribute in Crater. So that Crater can take in spawner and spawn the creature every tick based on its percentage.

Pros of the design are I can easily add a new creature in the future for spawning. By simply creating a spawn creature class and implement SpawnActor, then the crater class takes in the spawner and spawn around its exit.

Cons of this design is there will be multiple spawner class implement SpawnActor in the future. This causes complexity to increase.

Explanation:

According to my diagram represents an object-oriented system requirement 3. I have establish an interface class named SpawnActor. I have chosen interface because it allows for flexibility in accommodating different creatures in the future. Each creature may have distinct spawn criteria, but they share a common method called spawn, enabling each creature to define its unique spawn method body.

Additionally, I have created a Crater class that inherits from Ground, a LongLegWeapon class that implements the Weapon interface, and a SpiderAttackBehaviour class that implements the Behavior interface.

I have secure Single Responsibility Principle by assigning single responsibility to each class. The Spawn interface focuses on defining the spawn method for its child classes to implement, ensuring that different creature types must have spawn method. The responsibilities for crater class is to create a spawner for the creature to spawn. Similarly, the SpiderAttackBehaviour is to enable spider to attack. So with single responsibility principle my design able to separate different responsibility to different classes and interface.

Initially, in my design, each creature had to create an instance within the Crater class to enable spawning around the crater. However this approach violated the Open-Closed principle as it required modifying existing code whenever a new creature needed to spawn. To improve this I have followed the Open-Closed Principle. The design improved with a Spawn interface and have a spawner attribute in the Crater class. This modification allows the crater class to associate with the Spawn interface. With this design, whenever I need to spawn a creature. I can create a new creature spawn class that implements the Spawn interface.

Besides that, I override the canActorEnter method in Floor class. Only actor with status of HOSTILE_TO_ENEMY able to be on the floor. So that the creature won't be able to step in the spaceship. This shows that with this design features can be easily modify according to the needs. With all these abstract, interface and inheritance it makes the code easy for extension and maintain.

Additionally, for the spider weapon, when we introduce new weapon in the future. I can create a new weapon class that implements the Weapon interface. This weapon class can be integrated into the SpiderAttackBehaviour and added to the HuntsmanSpider's behavior seamlessly. Moreover, if new player types are introduced in the future, the SpiderAttackBehaviour can still detect and attack all players, this ensure that no modifications are needed in the existing code.

Moreover, SpiderBehaviourAttack class relies on the Actions abstract class instead of directly depending on a specific concrete class. This ensures that the class is flexible and adaptable to changes in the Actions abstract class or future implementations. Additionally, behavior has an interface Behavior so it removes dependency of HuntsmanSpider to other concrete classes like SpiderAttackBehaviour and WandarBehaviour. This follows the Dependency Inversion Principle.

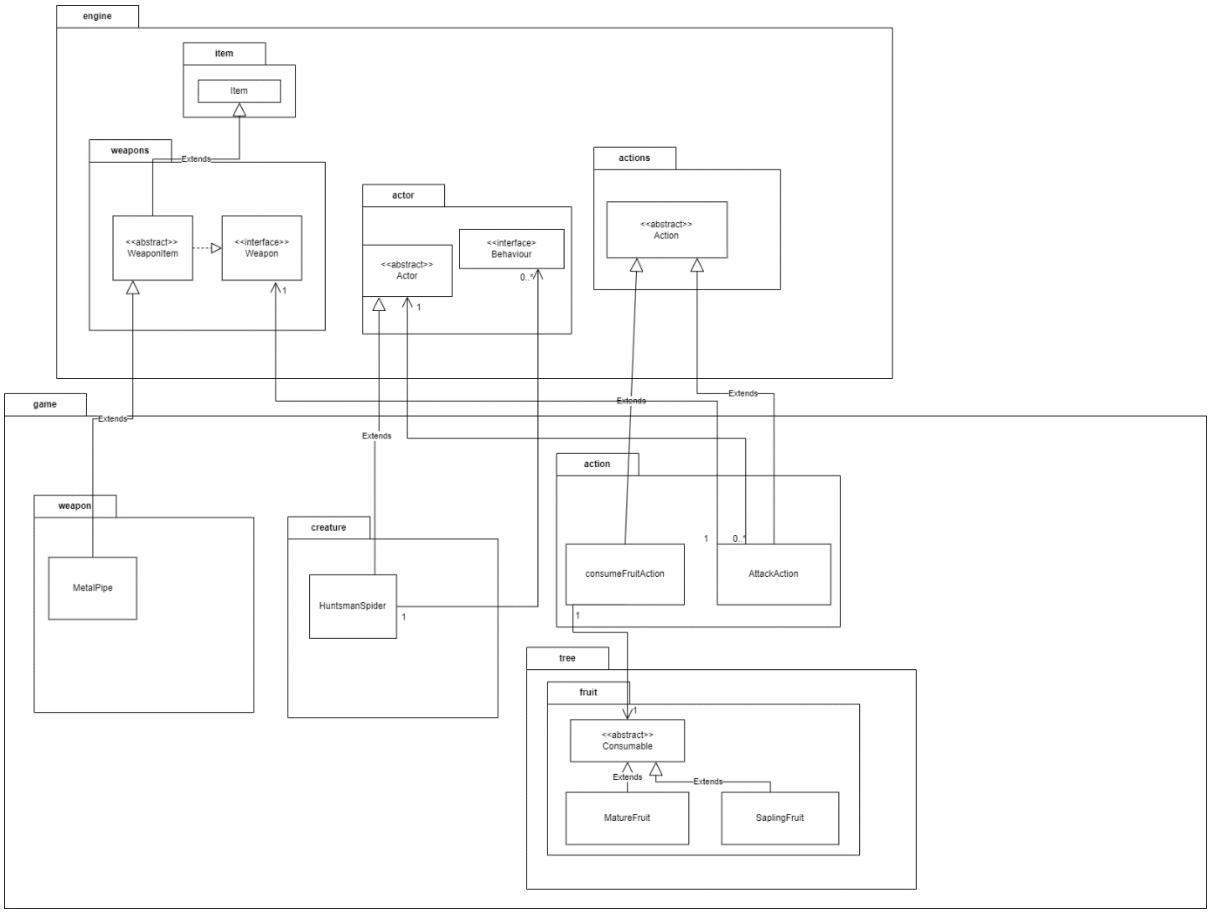
In addition, I have different interfaces for different types of purpose. For example, Spawn interface is implemented for spawning purposes, Weapon interface is for future weapon to implement. I ensured that each class implements only the methods that are directly relevant to its functionality. This followed the Interface Segregation Principle.

Moreover, following Liskov Subsituition Principle in my design I did not use any downcasting. This is to ensure the behavior of the base class remains unchanged and any child class can replace its parent class without changing the behavior.

The pros of utilizing this design are, it is flexible as the design follows Open-Closed Principle and Dependency Inversion Principlem. This design allows for easy extension and modification without requiring extensive changes to existing code. Moreover, the interfaces, abstract classes promote code reusability as common functionalities can be shared among different classes without duplication. Furthermore, by following single responsibility principle, it is making codebase easier to manage and understand

The cons will be when implementing a design with multiple interfaces and abstract classes the inheritance hierarchies can lead to increased complexity of the code.

REQ 4



4.1 Implementation of MetalPipe

I created a MetalPipe class that extend from WeaponItem. I override the allowableAction to add the attackAction when there is creature around by using HOSTILE_TO_PLAYER status to check. The pros is by this approach maintains modularity by having its own behavior within its own class. This makes codebase more organized and easier to maintain. The design is scalable and adaptable to future changes. If there is additional condition required in the future they can be easily added within their classes. Besides that, the approach is efficient as it directly checks the status to determine when to add the attackAction.

The con is when the codebase grows, we should have different abstract class or interface for the weapon special attack to avoid inconsistencies.

4.2 player attack HuntsmanSpider with bare hands

I override the existing getIntrinsicWeapon() method with new define value of hitrate and damage to accommodate the requirement. This shown that I reuses the code and no redundancy.

The cons will be I might have to override each actor for its intrinsicWeapon attack in the future.

4.3 Implementation fruit healing when consume

I override the consumable's allowable action to add the consumeFruitAction to the abstract class. So every fruits inherit from consumable able to have consume action and it can execute based on each fruit properties.

The pros of this approach are, it promotes code reusability as the consume action is defined once in the consumable abstract class and can be reused by all fruit subclasses. This reduces redundancy and improves code maintainability. Furthermore, each fruit subclass can customize its consumeAction based on its specific properties and requirements. This allow for flexibility. Besides that, adding new fruit subclasses in the future is straightforward as they can inherit the consumable abstract class.

The cons of this approach are the consumable will be impacting all the fruit classes. So I need to be careful and considerate when adding new functionalities. Besides that, the complexity of Consumable will increase when many fruits created in the future.

Explanation

According to my diagram represents an object oriented system requirement 4. I have created a MetalPipe concrete class that extends from WeaponItem abstract class. This MetalPipe class is designed to enable the attack functionality against HuntsmanSpiders. Additionally, I have introduced the ConsumeFruitAction class which inherits from the Action class. This action allows actors to consume fruits and heal themselves.

I have chosen this design because it makes my codebase easy to understand. Using different classes for different purposes, for WeaponItem abstract class it specifically responsible for creating a weapon item. It focuses on the characteristics for its child class. So whenever, there is a new weapon in the future just simply inherit WeaponItem class to get all the functionality of a weapon. Besides that, ConsumeFruitAction class is to enable actors to consume fruits and it makes actor gain/lost depends on the fruit types. This followed the single responsibility principle.

In my initial design for weapon to attack creature, I had implemented a loop within the HuntsmanSpider class to check if there was any weapon in the actor's inventory, and if true, I would add an attack action. However, I realized that this approach violated both the Open-Closed principle and the Dependency Inversion Principle, as it required adding a new instance in the HuntsmanSpider class each time a new weapon was introduced leading to inefficiency.

To solve this issue and follow these principles, I improved my code by overriding the allowable action in the WeaponItem class. Specifically, for the MetalPipe, I added an attack action directly within the MetalPipe class. So whenever the actor pick up the metal pipe the actor has an attackAction to the actor that has capability STATUS_HOSTILE_PLAYER. This design is simpler and easier to extend, so in future when we have multiple player other players able to attack creature with this design. Besides that, player won't be able to attack other player with the metalpipe unless their status is STATUS_HOSTILE_PLAYER. Now when introducing a new weapon capable of attacking the spider, I only need to create a new weapon class that implements the WeaponItem interface and add the attackAction based on the status of actor. By using this design I able to achieve Open-Closed principle.

Likewise, in my initial design, I had a loop within the Player class to check the inventory for fruit, adding a consume action to the action list if a fruit was found in the inventory. However this approach violated the Open-Closed Principle and inefficient as it required creating an fruit instance in the Player class for every new fruit, leading to redundancy.

To solve these concerns and adhere to the principles, I enhanced my design by introducing a Fruit abstract class and overriding the allowableAction to return a consumeFruitAction. This ensures that if a new fruit is introduced in the future, we only need to create a fruit class that inherits from the Fruit abstract class. The new fruit class will automatically have the consume option. This design approach eliminates the need to modify previous code and ensures follow the Open-Closed Principle.

Moreover, Liskov Subsitution Principle is secure in my design as I didn't use any downcasting. This ensures that the behavior of the base class remains unchanged, allowing child classes to replace their parent classes without changing the program behavior. This adherence to the Liskov Substitution Principle.

The pros of my design are, for each class has a clear and singular responsibility, which promotes code maintainability and readability. It reduces the risk of bugs when making changes. Furthermore, by using abstraction and inheritance effectively, my design allows for extension without modification of existing code. By using the Dependency Inversion Principle my design ensures that classes depend on abstractions rather than concrete implementation. It removes dependency of multiple concrete class. Thus it enhance my design structure. In my design there is no downcasting involve so it ensures that my program's behavior remain consistent. Other than that, the use of abstract classes it promotes code reusability and reducing redundancy. Not to mention, it is easy for extension, whenever we need to implement new feature to the existing function. We just need to implement in abstract class or inherit from it.

The cons of my design are when there are more weapons or fruit added in the future. The design may become complex, especially with multiple inheritance, abstract classes and interfaces. Which can be hard to understand.