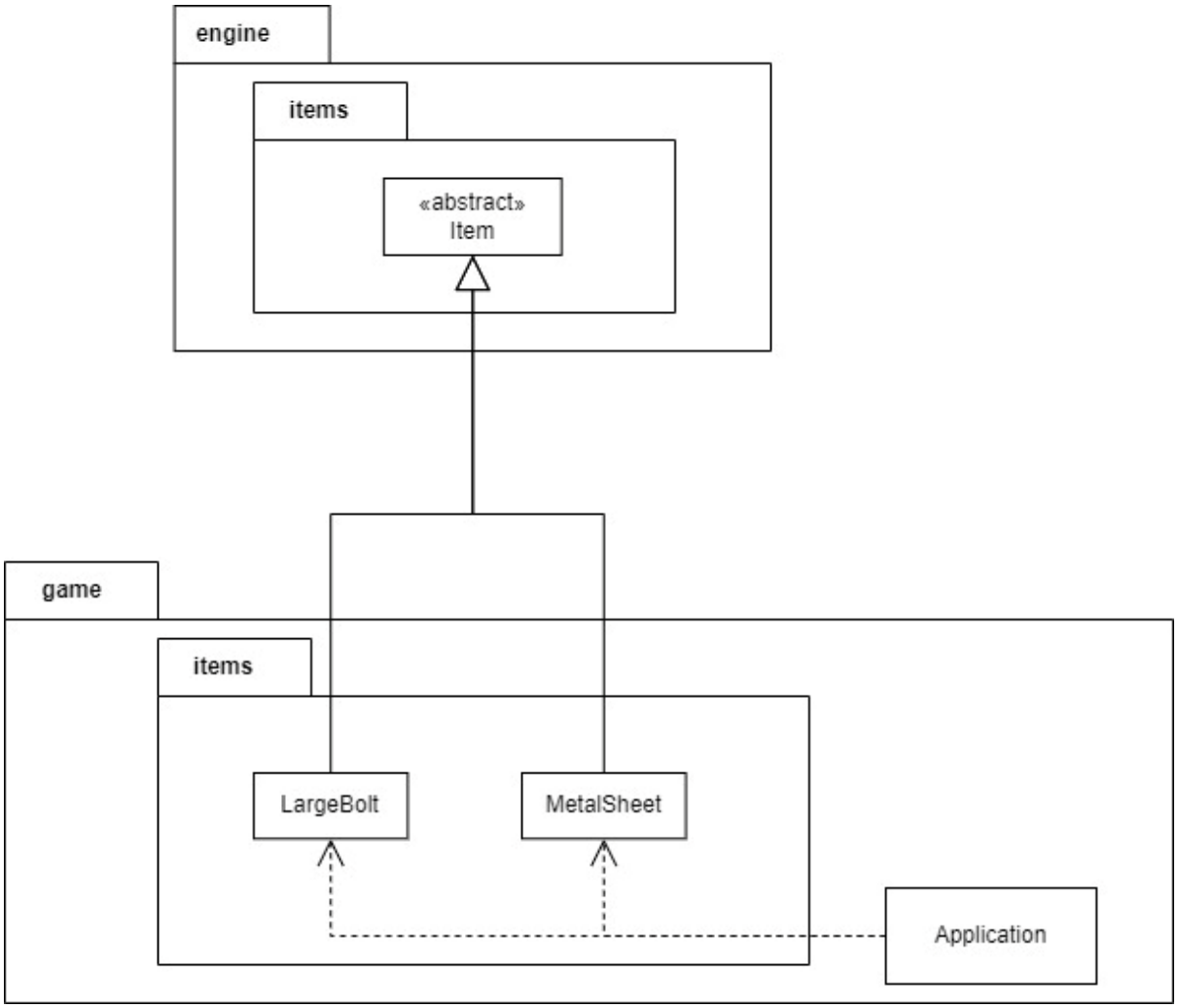
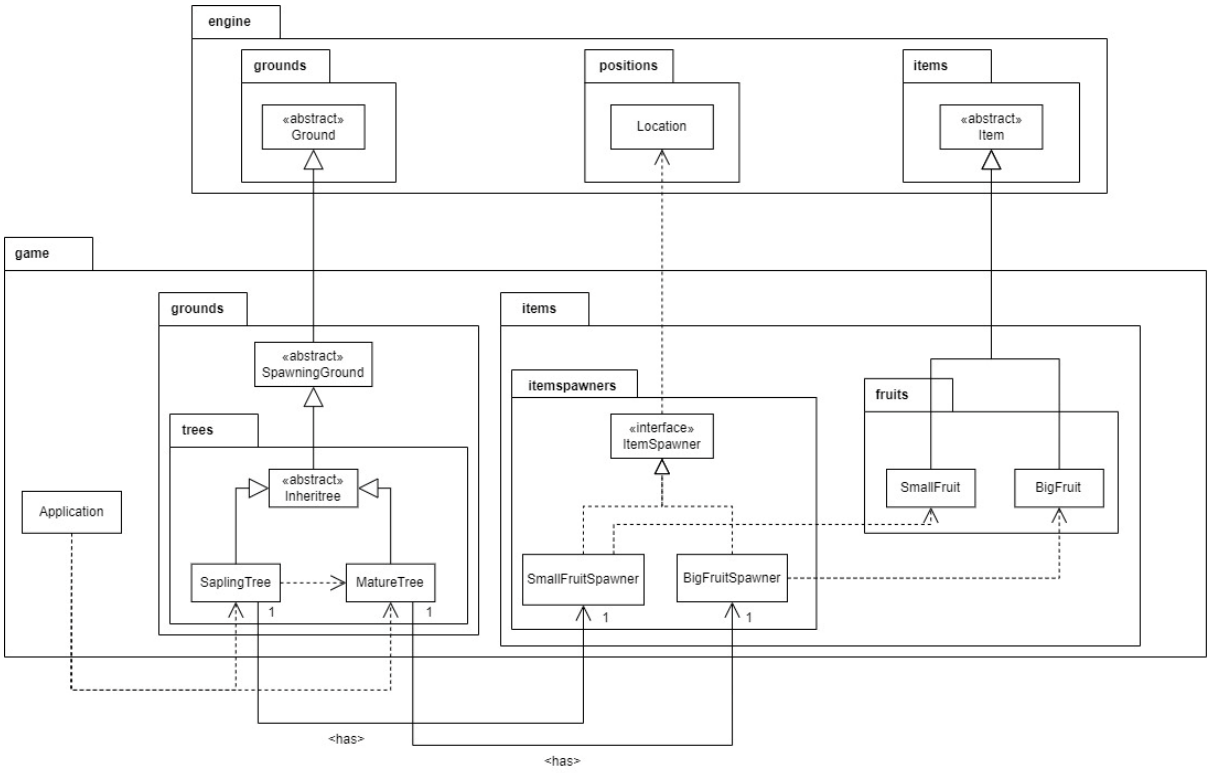
Requirement 1: The Intern of the Static factory



I added LargeBolt and MetalSheet classes by extending the Item abstract class and organised them in a package items. These classes inherit the pickup and droppoff functionality from the Item parent class, avoid redundant code and promote reuse (Don't Repeat Yourself). By implementing LargeBolt and MetalSheet as two distinct classes, this approach ensures clear differentiation between scrap types and enables specific interaction unique to each item in the future, as each class encapsulates its own behaviour and properties. They have dependencies on Application because I initialized the LargeBolt and MetalSheet object using the addItem function of Location in Application class.

Requirement 2: The moon's flora



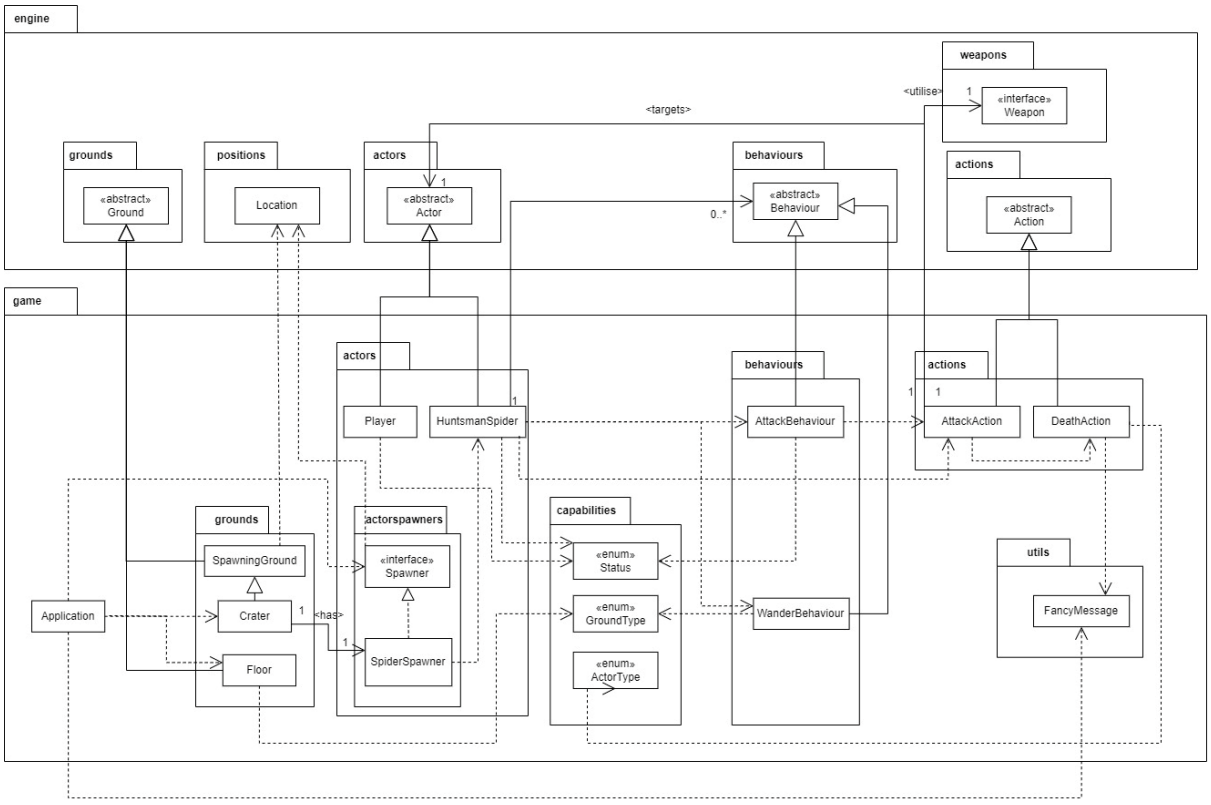
To tackle events where the ground can produce things, I added a SpawningGround abstract class that extends Ground class.

As sapling tree and mature tree can spawn different things, I made a SaplingTree and MatureTree class that extends from Inheritree abstract class. The Inheritree abstract class contains common logic of SaplingTree and MatureTree, avoid repetition in the children classes (Don't Repeat Yourself Principle). The Inheritree abstract class extends from the SpawningGround abstract class as it can produce fruits. There is also a SmallFruit and BigFruit class that represents fruits produced by SaplingTree and MatureTree, respectively.

To handle the functionality of producing fruit, the ItemSpawner interface was added. This interface is implemented by SmallFruitSpawner and BigFruitSpawner classes, which are responsible for initializing SmallFruit and BigFruit objects respectively (Interface Segregation Principle). It allows the SmallFruitSpawner and BigFruitSpawner objects to be passed into the tree classes, rather than having the tree classes directly initialize the fruit object (Dependency Injection Principle). The SmallFruit and BigFruit classes extends from the Item abstract class, representing the fruits produced by the trees. Therefore, the SmallFruitSpawner and BigFruitSpawner classes only has one responsibility, which is to spawn they respective fruits (Single Responsibility Principle). The alternative design would be to initialise SmallFruit and BigFruit item directly in SaplingTree and MatureTree class, however it violates Dependency Injection Principle. Other way is to initialise SmallFruit and Bigruit in the class itself and let SaplingTree and

MatureTree to have SmallFruit and BigFruit object, however it will overload SmallFruit and BigFruit as they are handling more than one responsibility at a time, violating Single Responsibility Principle. However, this approach may lead to long codes if there are many items to be spawn in the future since each object will have their own spawner class.

Requirement 3: The moon's (hostile) fauna



To make a crater spawn a HuntsmanSpider, we have a Crater class that extends SpawningGround. I added a HuntsmanSpiderSpawner class that implements ActorSpawner interface. This class initialise new HuntsmanSpider that has been spawn. The Crater class will then accept a ActorSpawner object as parameter. This makes the code easily extensible as in the future, if the crater can spawn other type of creature, it can still pass into the Crater parameter as ActorSpawner object (Liskov Substitutable Principle). Furthermore, the Crater class depends on the ActorSpawner abstraction, not on the HuntsmanSpiderSpawner concretion (Dependency Inversion Principle). By making a HuntsmanSpiderSpawner instead of making HuntsmanSpider object to implements ActorSpawner, this makes one class responsible of one thing only (Single Responsibility Principle). However, this approach will lead to many classes if there are many creatures to be spawned because each creatures have their own spawner class. An alternative design would be to initialise HuntsmanSpider object in Crater class, however this approach lacks extensibility to handle case when a Crater can spawn other creatures in the future.

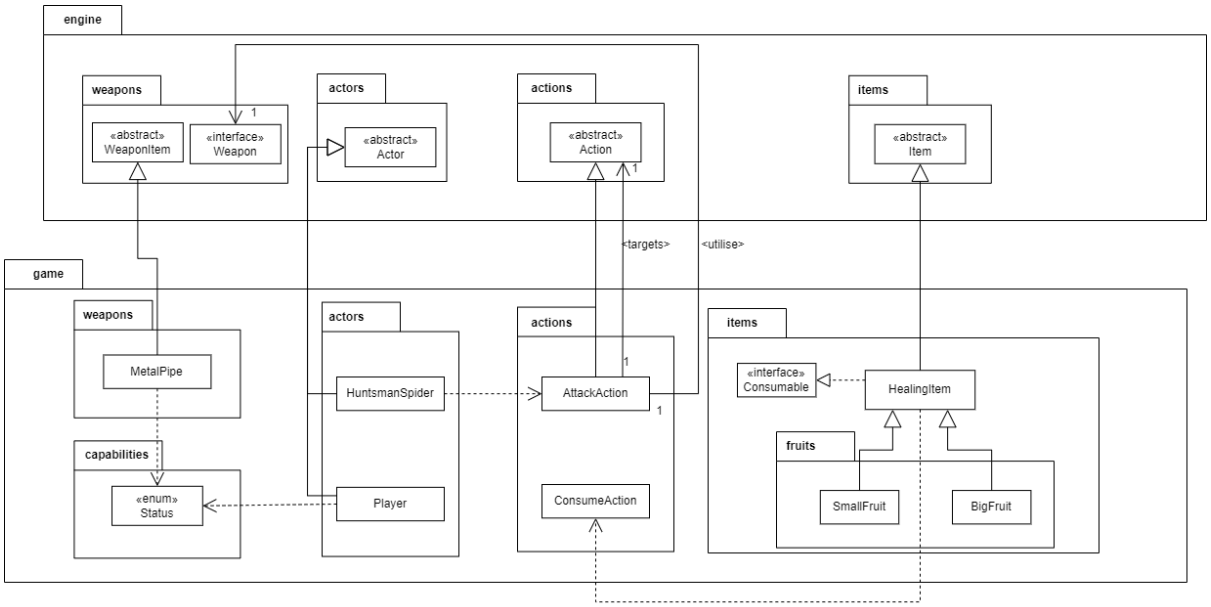
Then, to make the HuntsmanSpider attack, we add AttackBehaviour class that extends Behaviour. This class is called in HuntsmanSpider class with a 0 key to prioritize this behaviour compared to WanderBehaviour. To make it attack creatures that are hostile to the enemy, we utilize the Status enum to check the status of actors in the AttackBehaviour class and call AttackAction to attack Intern. This approach promotes

extensibility since any other creatures can add AttackBehaviour if they can attack (Open Closed Principle). Another approach is to initialise AttackAction in HuntsmanSpider's allowableAction, however that violates Open Closed Principle because other creatures that can attack would need to implements their own AttackAction too.

To prevent the HuntsmanSpider from entering the spaceship, a GroundType enum was created and a RESTRICTED GroundType capability was added to the Floor class. This design choice allows for easy checking of exits and restricts the HuntsmanSpider from wandering into the spaceship. An alternative design would be to hardcode the coordinates in the WanderBehaviour to check if exits are not within those coordinates. However, this approach is not easily extensible since we will have to change the coordinates if new spaceship ground is added, and it would be difficult to code if the spaceship is not rectangular. By making a GroundType enum, we make the code easy to extend as we do not need to change anything if a new spaceship area is added in the future. Furthermore, if other GroundType has other capabilities in the future, we can easily extend it (Open Closed Principle).

The DeathAction class was added to handle the death of the actors. This class is called by the AttackAction class if an actor dies from an attack. Here, we introduced the enum ActorType so that DeathAction can check if the ActorType is a player and show the Fancy Message if it is a player. This is easily extensible because if an actor can die due to several other reasons in the future, we just need to call the DeathAction without stating the reason and showing a fancy message every time (Don't Repeat Yourself Principle). In addition, any death will be handled by the DeathAction, thus we do not overload AttackAction with extra functionalities (Single Responsibility Principle). An alternative design is handle death in the action that caused death itself, however that may lead to repetitive code.

Requirement 4: Special scraps



To make MetalPipe a special scrap that can be used to attack, I created a MetalPipe class that inherits WeaponItem, which allows it to utilize the pickup and drop-off functionalities of the WeaponItem class (Don't Repeat Yourself Principle). The allowableAction method of the MetalPipe class is overridden to check the Status of the actor and attack those who are hostile to the player. This design is extensible because if new actors are added in the future, their Status just needs to be defined for the player to interact with them using the MetalPipe. An alternative approach would be to set a Status.HIT_WITH_METALPIPE when the MetalPipe is picked up, and every time a hostile creature is around, loop through the item inventory of the player to find if there is an item with that Status, then add a new AttackAction with that item. However, this is not extensible since if a new weapon is added, it would need to have a new Status too. Also, this approach requires downcasting of the item into a weapon, which is not a good approach.

To implement the feature where the Intern can punch hostile creatures with bare hands, I override the getIntrinsicWeapon method to add a new IntrinsicWeapon for the Player, and add a new AttackAction in the HuntsmanSpider's allowableAction so that the Intern will attack the HuntsmanSpider with the IntrinsicWeapon. However, the disadvantage of this approach is that if there are many creatures for the Intern to attack with the IntrinsicWeapon, we need to add them to that creature's allowableAction.

To represent item that can be used to heal, I implemented a HealingItem abstract class that extend Item. HealingItem implements Consumable Interface, which means items that can be consumed (Interface Segregation Principle). The SmallFruit and BigFruit class then extends HealingItem. The consumed function contains the logic when player consumes fruit and is only implemented once in HealingItem rather than separately in

both children class (Don’t Repeat Yourself). This is extensible because if other items except from fruits have healing functionalities in the future, they can directly extend from HealingItem class (Open Closed Principle). Then in the HealingItem class, we call ConsumeAction to consume (Single Responsibility Principle). However, this approach assumes that all healing functionalities will be get by ConsumeAction, which will lead to confusion if there are other healing functionalities in the future that do not need ConsumeAction. Another way of doing it is to put the healing logic in ConsumeAction class, thus can shorten code in HealingItem, however that overload the ConsumeAction to have more than one responsibility and make it hard to extend if there are ConsumeAction that do not have healing functionalities in the future.