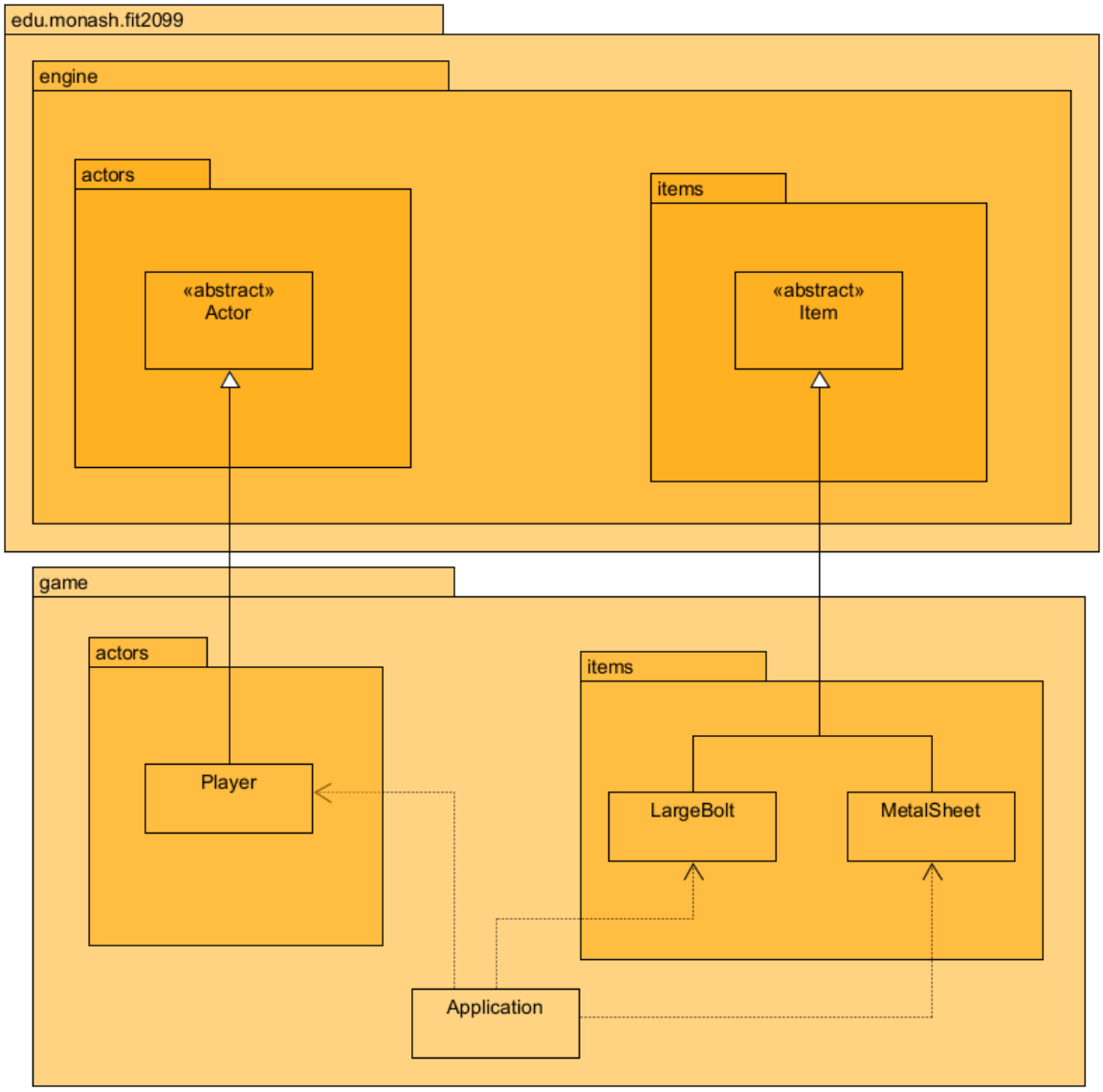
Darryl Teh 34421882 FIT2099 Assignment 1

REQ1: The Intern of the Static factory

UML class diagram of REQ1



Design Rationale for REQ1

The classes that exist in my extended system for requirement 1 are the Player, LargeBolt and MetalSheet class.

The Player class in the actors package represents an Intern whose objective is to collect scraps. For this requirement, the only interactions that the Player performs is picking up and dropping off scraps. The LargeBolt and MetalSheet classes represent two of the scraps that the Player can pick up or drop off.

The Player class extends the abstract Actor class from engine, this is because Player shares some common attributes (name, displayChar, hitPoints) and common methods (playTurn) with the abstract Actor class. Therefore, it is logical for Player to extend Actor, adhering to the DRY principle.

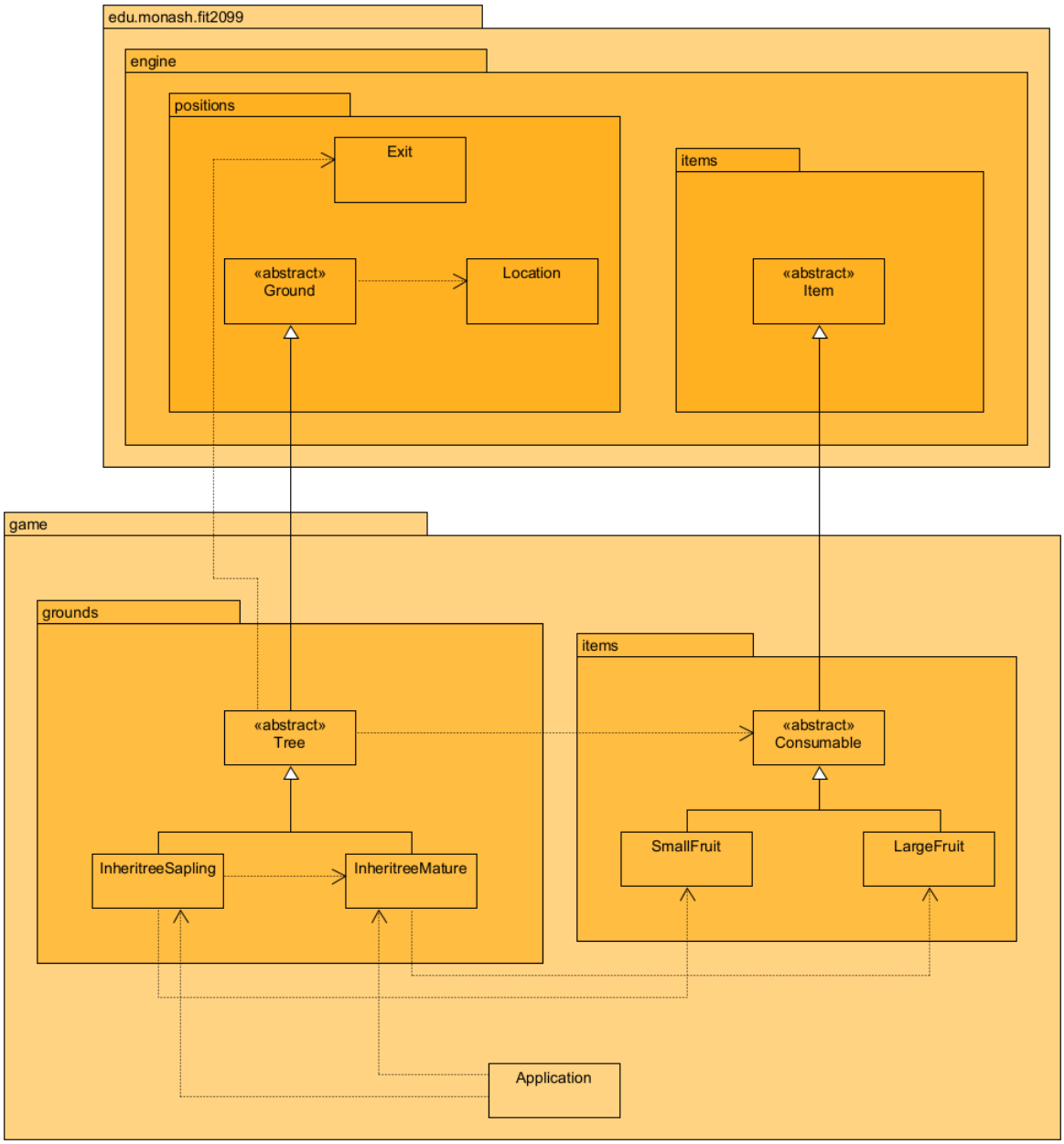
Similarly, the LargeBolt and MetalSheet classes extend the abstract Item class from engine. This is because the Item class already provides common methods and attributes necessary for LargeBolt's and MetalSheet's required functionality such as name, displayChar, portable, getPickUpAction(), getDropAction(). By extending the Item class, LargeBolt and MetalSheet can use its code without duplicating it (DRY).

An advantage of having Player extend Actor and LargeBolt and MetalSheet extend Item is that the code can be easily extended to add new actors (NPCs) or scraps. The Actor and Item abstract classes from engine are closed from modification, but open to extension by creating new items/scraps or actors that extend them without modifying the Actor or Item classes (Open-Closed Principle). Another advantage is that LargeBolt and MetalSheet would be substitutable for Item, and Player for Actor. This adheres to the Liskov Substitution Principle.

The disadvantage to this design is that Player is tightly coupled to Actor and LargeBolt and MetalSheet are tightly coupled to Item. If the Actor and Item abstract classes in the engine were ever modified, it would potentially break the Player, LargeBolt and MetalSheet classes.

REQ2: The moon's flora

UML class diagram of REQ2



Design Rationale for REQ2

The classes that exist in my extended system for requirement 2 are the Tree abstract class, InheritreeSapling and InheritreeMature classes, the Consumable abstract class and the SmallFruit and LargeFruit classes.

The Tree abstract class represents a tree that has the responsibility of dropping fruits and after a certain number of ticks, mature into a different tree, such as InheritreeSapling maturing into InheritreeMature. InheritreeSapling and InheritreeMature extends the Tree abstract class.

The Tree abstract class extends the abstract Grounds class from engine. This is because the Grounds class contains attributes such as displayChar and methods such as tick that the Tree abstract class requires. The Tree abstract class has the methods dropFruit and matureTree, it also overrides the tick method to increment its attribute counter.

The InheritreeSapling and InheritreeMature classes override Tree's tick method, calling the dropFruit and matureTree methods with the appropriate arguments passed into the methods as necessary. For example, if InheritreeSapling drops a small fruit with a 30% chance and matures into InheritreeMature after 5 ticks, InheritreeSapling's tick method would call dropFruit with a chance of 0.3 and an instance of SmallFruit passed in and call matureTree with ticksToMature of 5 and an instance of InheritreeMature passed in.

An alternative implementation of Tree can be seen in the Tree class of the conswaylife package in the demo package. This implementation uses if statements in the tick method of Tree to mature the tree to different stages once the Tree has reached the appropriate age by changing Tree's displayChar. This violates the Open-Closed principle, as if more tree stages were to be added, Tree's tick method would need to be modified to add a new if statement for each new stage, making the Tree class hard to maintain and this design not extensible. This implementation also violates the Single Responsibility Principle, as the Tree class is responsible for tracking the age of the tree and also changing its display character based on its age for each stage that the tree could mature to.

In my design, Tree is an abstract class. One of the advantages of my design is that the abstract Tree class is open to extension without the need to modify it. In order to add a new Tree stage, a new class that extends Tree that overrides the tick method can simply be created. This adheres to the Open-Closed Principle. My design also adheres to the Single Responsibility Principle, as the Tree abstract class is only responsible for common tree behaviour, while the classes that extend it handle their own fruit dropping and maturing behaviour. In my design, any of the tree classes extending Tree will be substitutable for Tree without breaking the application, adhering to the Liskov Substitution Principle.

The disadvantage to having Tree as an abstract class is that child classes of Tree might be forced to have methods that they do not use. For example, if a new tree were added that does not drop fruits, it would still have the dropFruit method. This would be a violation of the Interface Segregation Principle. A solution to this would be implementing an interface for dropping fruits, an interface for maturing the tree, and separate interfaces for any other

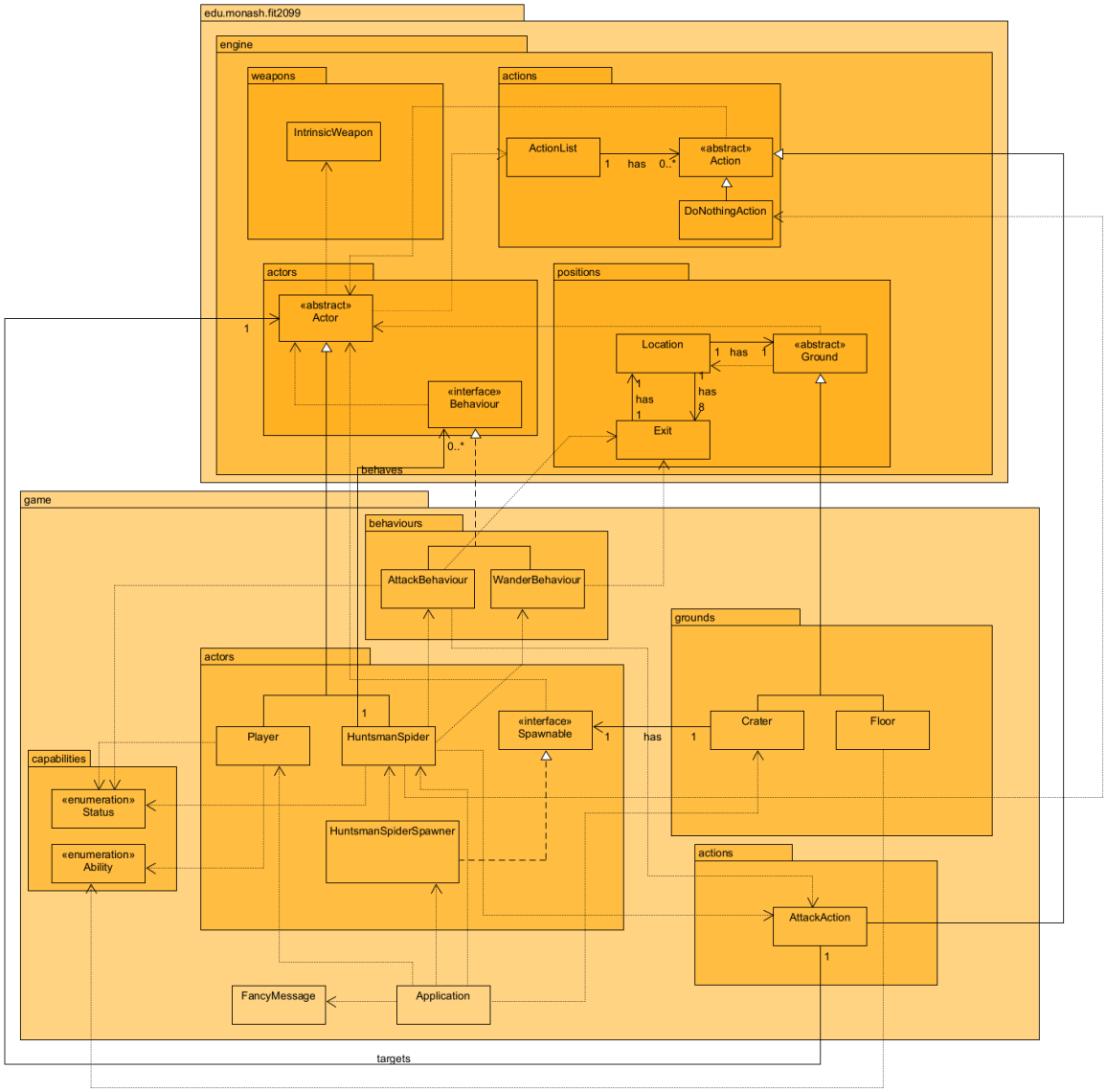
actions that trees could do. However, assuming that all trees will have the ability to drop fruits and mature when the code is extended, this would be unnecessary.

The classes SmallFruit and LargeFruit classes represent the fruits that InheritreeSapling and InheritreeMature respectively drop. As these fruits are able to be consumed later in REQ4, SmallFruit and LargeFruit extend an abstract classed named Consumable.

As of REQ2, the fruits SmallFruit and LargeFruit can only be picked up and dropped off by the Intern. Therefore, the Consumable abstract class extends the Item class from engine for the same reasons that LargeBolt and MetalSheet extend Item in REQ1. The Item class provides common methods and attributes necessary for Consumable's child classes' required functionality such as name, displayChar, portable, getPickUpAction(), getDropAction(). By extending the Item class, Consumable can use Item's code without duplicating it, adhering to the DRY principle.

REQ3: The moon's (hostile) fauna

UML class diagram of REQ3



Design Rationale for REQ3

The classes that exist in my extended system for requirement 3 are the Player class, HuntsmanSpider class, HuntsmanSpiderSpawner class, Spawnable interface, Crater class. Floor class, AttackAction class, AttackBehaviour class, WanderBehaviour class, Status enum, Ability Enum and FancyMessage class.

The HuntsmanSpider class represents a creature with 1 hit point that will wander around and if the Intern is within its surroundings, attack the Intern, dealing 1 damage with 25% accuracy. The HuntsmanSpider cannot enter the Intern's spaceship.

In order to make it so HuntsmanSpider cannot enter the spaceship, my design introduces a new ability in the Ability enum, ENTER_SPACESHIP. Only actors with the ENTER_SPACESHIP capability (such as Player) will be able to move to Floor. This is done by overriding the engine's Ground abstract class's canActorEnter method in the Floor class to return true if an Actor has the ENTER_SPACESHIP capability and false otherwise. An alternative to this would be having Floor's canActorEnter return true if the actor has the status HOSTILE_TO_ENEMY. However, doing so would mean any future actors added when the game is extended that are hostile to enemies would be able to enter the spaceship, which might not be wanted. Hence adding the new ENTER_SPACESHIP ability gives a greater level of control on who can enter the spaceship and makes the game more easily extensible when it comes to adding new actors.

The HuntsmanSpider class extends the abstract Actor class from engine, as it uses methods such as playTurn, allowableActions, getIntrinsicWeapon and attributes such as name, displayChar and hitPoints that the abstract Actor class has. Having HuntsmanSpider extend Actor prevents code duplication and adheres to the DRY principle. The HuntsmanSpider class's wandering and attacking behaviour is implemented using the AttackBehaviour and WanderBehaviour classes. The Huntsman Spider is made to deal 1 damage with 25% accuracy by overriding HuntsmanSpider class's getIntrinsicWeapon method.

The Crater class represents a crater that can spawn a HuntsmanSpider with a 5% chance at every game turn. In my design, I have a Spawnable interface that is implemented by spawner classes such as HuntsmanSpiderSpawner. The Crater class has an instance of a class that implements Spawnable as an attribute. Crater then uses the Spawnable instance's methods getSpawnChance and createSpawn in its tick method to spawn actors such as HuntsmanSpider.

One of the advantages of this design is that the Crater class can easily be extended to spawn different actors, not just HuntsmanSpider. This can be done by creating by creating a new spawner class that implements Spawnable for another actor, and passing it into the Crater class's non-default constructor, without the need to modify the Crater class's code. The Crater class is open for extension and closed for modification, adhering to the Open-Closed Principle. Having Spawnable as an interface helps my design adhere to the Dependency Inversion Principle, as the Crater class depends on the Spawnable abstraction instead of the concrete HuntsmanSpiderSpawner class. This makes the code more flexible. In my design, the Crater class is responsible for spawning creatures, the HuntsmanSpiderSpawner class is

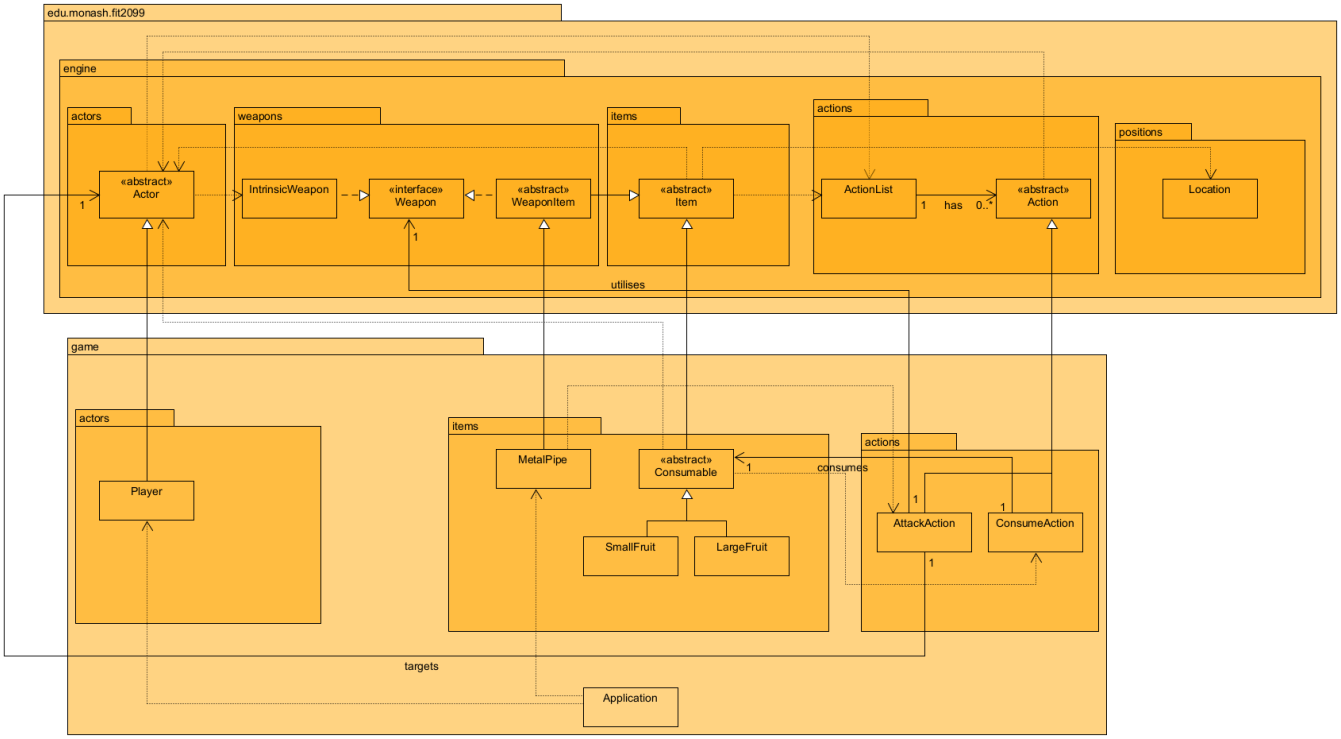
responsible for creating HuntsmanSpider instances and determining their spawn chance, and the HuntsmanSpider class is responsible for its own behaviour. Each class has a single responsibility, adhering to the Single Responsible Principle.

A disadvantage to this design is that spawner classes added when the game is extended might have similar implementation to that of HuntsmanSpiderSpawner. This would lead to code duplication and a violation of DRY. Therefore, making Spawnable an abstract class was considered. However, future actors might also need to spawn differently on the crater when the game is extended. Having Spawnable as an interface would therefore allow any class to implement It and provide its own way of spawning actors.

An alternative design considered was to have the actor classes to be spawned on the crater such as HuntsmanSpider implement the Spawnable interface instead of having spawner classes such as HuntsmanSpiderSpawner. This version of the Spawnable interface would only have the method getSpawnChance and would make the classes that implement it have a spawnChance attribute. The Crater class's non-default constructor would then take in an instance of a class that implemented this interface. This would eliminate the need for spawner classes such as the HuntsmanSpiderSpawner class. However, this would make classes like the HuntsmanSpider class not only responsible for its behaviour, but also its spawning behaviour, leading to a violation of the Single Responsibility Principle.

REQ4: Special scraps

UML class diagram of REQ4



Design Rationale for REQ4

The classes that exist in my extended system for requirement 4 are the Player class, MetalPipe class, Consumable abstract class, SmallFruit class, LargeFruit class, AttackAction class and ConsumeAction class.

The MetalPipe class represents a special scrap that the Intern can pick up or drop off. If the metal pipe is in the player's inventory, the player can deal 1 damage with to hostile creatures in their surroundings with 20% accuracy. If the Intern is not wielding a metal pipe, they deal 1 damage with 5% accuracy.

The MetalPipe class extends the WeaponItem abstract class from engine, which is a child class of the engine's Item abstract class. This is because the MetalPipe requires attributes such as damage, verb and hitrate and methods that the WeaponItem abstract class has. By extending WeaponItem, the MetalPipe class can reuse the code in WeaponItem instead of repeating it. This makes the code easier to maintain and follows the DRY principle. In order for the Intern to attack with the MetalPipe, the MetalPipe class overrides the allowableActions method to include an AttackAction in the ActionList that it returns. For the Intern to deal 1 damage with 5% accuracy when not wielding MetalPipe, the Player class's getIntrinsicWeapon method is overridden.

The SmallFruit and LargeFruit classes first implemented in REQ2 can now be consumed by the player. The Consumable abstract class that SmallFruit and LargeFruit extend has been modified to contain the new attributes consumeEffectValue of type int and and consumeVerb of type String. The methods getConsumeEffectValue, getConsumeVerb, and consumeConsumable are added to the Consumable abstract class. The allowableActions method of Consumable is overridden, to include a ConsumeAction in the ActionList that it returns. ConsumeAction is a class that extends the abstract Action class from engine, it has an instance of Consumable as an attribute and its execute method calls the Consumable class's consumeConsumable method to consume a consumable. The Consumable class's consumeConsumable method decides the logic in how a consumable is consumed. If a new consumable were to be added that hurts the player, it could override Consumable's consumeConsumable method to hurt the player instead of heal.

One of the advantages of this design is that each class has a single responsibility. The Consumable class is responsible for handling the common functionality of consumable items, while the SmallFruit and LargeFruit classes are responsible for handling the specific properties of small fruits and large fruits respectively. This adheres to the Single Responsibility Principle. The design also follows the Open-Closed Principle, as in order to introduce new consumables you can simply create a new class that extends Consumable without the need to modify the Consumable Abstract class. Because SmallFruit and LargeFruit extend Consumable, objects of SmallFruit and LargeFruit can be used wherever Consumable is expected, adhering to the Liskov Substitution Principle. The ConsumeAction class depends on the Consumable abstraction instead of the SmallFruit and LargeFruit classes, making the design more flexible and extensible, adhering to the Dependency Inversion Principle.

A disadvantage with this design is that not all consumables added when the game is extended will necessarily have an effect on the player. A consumable meant to have no effect on the player would still have the consumeEffectValue attribute and the getConsumeEffectValue method, even though they are not used. This is a violation of the Interface Segregation Principle.

An alternative to this design would be to have Consumable as an interface instead of abstract class. This would cause all classes that implement Consumable to provide its own implementation of the getConsumeEffectValue, getConsumeVerb, and consumeConsumable methods, which could lead to code duplication if the consumables are similar. This would be a violation of the DRY principle. Therefore, my design's Consumable is an abstract class.