FIT2099 A1 Design Rationale – All Requirements

Please refer to the relevant requirement UML PDF when reading the rationale.

REQ1: The Intern of the Static factory

- As all scraps are portable, I made an abstract Scrap class extending from Item where the portable attribute defaults to true.
- Scrap extends from Item as the Player needs to be able to pick up and drop scraps, which is part of Item's functionality.

Pros

- I chose to make Scrap an abstract class over an interface as there are no methods yet that pertain to Scrap, just that attribute of being portable. Seeing as abstract classes allow for modification to the constructor, it was the ideal choice.
- This abstraction into a Scrap class follows OCP, as future scraps can just inherit from it (open for extension), rather than manually write portable = true if they were to inherit from Item (closed for modification). This makes the code easily reusable.
- It also allows for future adhesion to LSP as we can assume all objects of type Scrap will have common functionality to them (like their current portability) and thus any subtype can be used where a Scrap is expected without disrupting the behaviour of the program.

Cons/alternatives

- I could have just extended MetalSheet and LargeBolt from Item directly to keep the system simpler (the new class Scrap with just one attribute changed could seem like overengineering). However, there may be Items in the future that are not Scrap, thus it's better to distinguish them.

REQ2: The moon's flora

- Tree is of type Ground as it does not need to be picked up/dropped off.
- The abstract child classes Sapling and Mature allow for different stages of growth to occur.
- InheritreeSapling and InheritreeMature are concrete classes as they hold specific attributes for fallProbability and fallItemDefault.
- SmallFruit and LargeFruit are the fallItemDefault attribute in InheritreeSapling and InheritreeMature respectively and of type Item as they need to be picked up/dropped off.

Pros

- Tree is an abstract class over an interface due to the need to define specific attributes (displaychar, fallProbability and fallItemDefault) and a concrete implementation of producing fruit.
    - Helps achieve OCP due to polymorphism. As all Trees produce an Item (like fruits), its children will inherit this functionality without having to redefine it themselves (also adhering to DRY).
- Sapling and Mature are not interfaces as there are no methods that are unique to them but rather abstract classes as they have unique attributes they need to manage. They

are useful as abstract classes because their children will require specific attributes (displayChar, fallProbability and fallItemDefault) in each stage of growth.
    - They also allow many different types of Trees (like Inheritree) to go through different growth stages. This abstraction helps makes the system extendible.
- The use of Growable as an interface follows the ISP. It allows only the Trees that need to grow to do so, like Sapling.
- The use of the Fallable interface adheres to LSP as any Item of subtype Fallable can be produced by Trees (not just fruits), making the Tree class more reusable.

Cons/alternatives

- Could have gotten rid of the abstract classes Sapling and Mature and had all maturing logic for Inheritree in a child class of Tree to reduce the hierarchy of abstraction. However, this disrupts the OCP, because if more maturing stages were to be added with more fruits, it would not scale well (would have a lot of case-switch/if-else statements).

REQ3: The moon's (hostile) fauna

Pros

- The abstraction of HostileActor helps achieve OCP due to polymorphism and inheritance. The children of HostileActor would have the same base behaviours and ability to be attacked by Player while being able to add more custom behaviours and methods.
- The Spawnable interface follows ISP as only Actors that can spawn (like HunstmanSpider), should.
- Crater's parameter spawnDefault employs the DIP, as rather than crater depending on HunstmanSpider directly, it depends on the abstracted interface Spawnable. Thus, when more Spawnable objects are created, Crater will still depend on the interface, reducing dependencies.
    - HuntsmanSpider being passed as the spawnDefault from the Application class also makes this an example of Dependency Injection through the constructor, where Application is the injector. Thus, can easily replace and test Crater with the different spawnDefaults.
- Rather than manually adding attacking behaviour into HunstmanSpider's class as a method, Behaviours (such as AttackBehaviour), are used to follow SRP. HunstmanSpider's behaviours handle the attacking, instead of HuntsmanSpider itself.

Cons/alternatives

- Could have gotten rid of the Spawnable interface and just had a spawn method in HostileActor, overriding the method in a child class when needed. This would reduce the dependencies as there would no longer be one between Spawnable and Actor. However, as not every HostileActor is spawnable this disrupts the ISP as a class is implementing a method it doesn't need. Thus, I kept the Spawnable interface.

REQ4: Special scraps

Pros

- Rather than manually adding logic of Player using the MetalPipe to attack HuntsmanSpiders as a method in Player, MetalPipe generates an AttackAction that is supplied to Player by the engine to follow SRP (Player is still just responsible for choosing an Action, not enacting it).
- The Consumable interface adheres to ISP, as only objects that can be consumed should implement it
    - It also allows for adherence LSP as seen in ConsumeAction receiving a Consumable in its parameter. This means, any subtype of Consumable can have a ConsumeAction created for it, making the ConsumeAction more extendible.
- ConsumableItem exists as it unites Item and Consumable, introducing a new attribute (change) and allowable actions for it. This couldn't have been achieved with an interface as I introduced a new attribute.
- As LargeFruit and SmallFruit are both able to heal Player, I also made another abstract class HealingConsumableItem with the HEALING ability so I wouldn't repeat the addition of that capability in each of the fruit classes, following the DRY principle.

Cons/alternatives

- Could argue that MetalPipe creating an AttackAction is a disruption of SRP as its not directly responsible for the attacking of one Actor to another
    - Thus, could have had HostileActor create the AttackAction for the MetalPipe weapon in its allowableActions method just as it currently does when no weapon is used. However, this is bad for two reasons. It will involve manually checking if the player has a Weapon using instanceof and once the Weapon is found, it will introduce a new dependency between HostileActor and Weapon which is not ideal. Thus, to follow the Reduce Dependency Principle, we keep the generation of the AttackAction using the MetalPipe in MetalPipe.