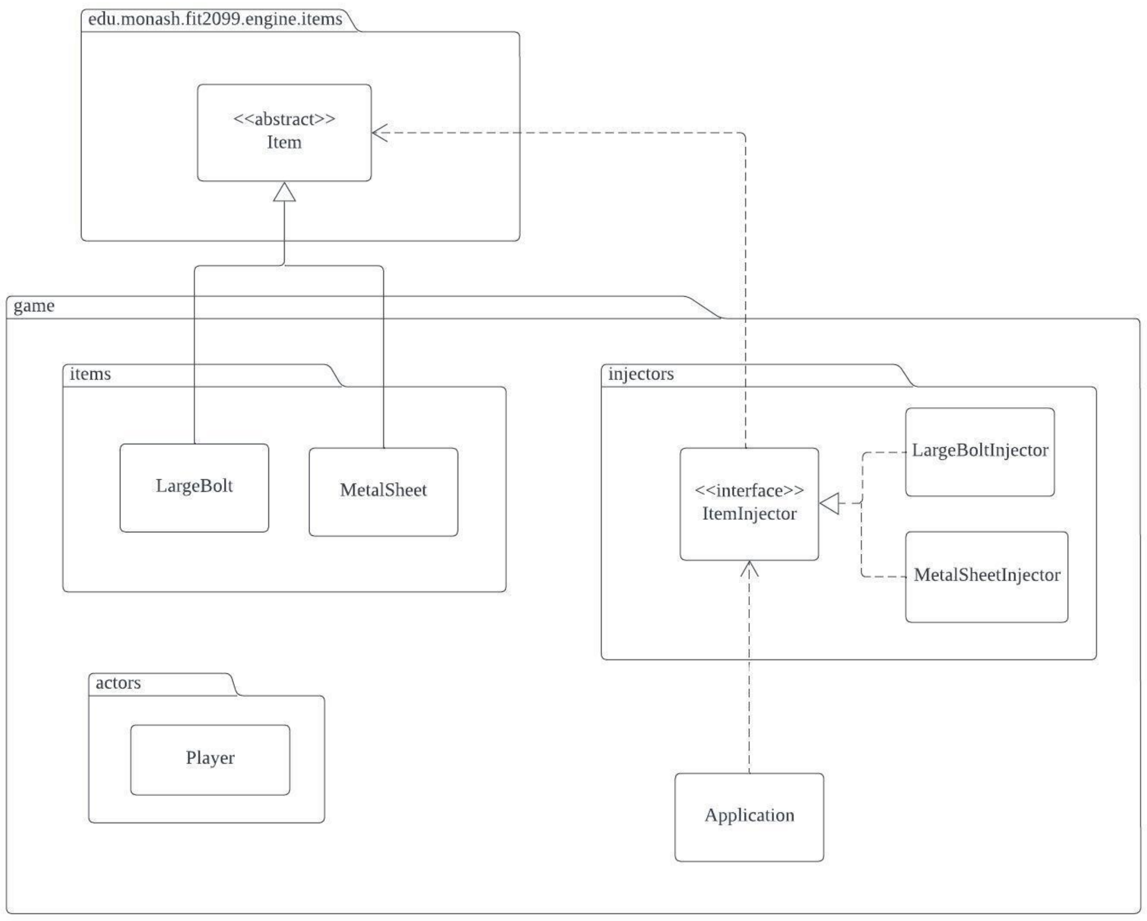
Requirement 1 - The Intern of the Static factory

UML Diagram



Rationale for Requirement

LargeBolt and MetalSheet extend the abstract Item class within the engine package. LargeBolt and MetalSheet can be picked and dropped by the actor. The interaction of these two items is the same as the default property with the abstract Item class in the engine, so I just make them inherit the Item abstract class to prevent repeated implementation (DRY) LargeBolt and MetalSheet are separated classes to represent different item (SRP). The benefit of extending the Item abstract class, ensures these two items have the property of the item in the game. In the future, if we want to add more property on these two items, we just need to provide their method or attributes. To make these items provide different interactions in the game, we can just overwrite the method in LargeBolt and MetalSheet.

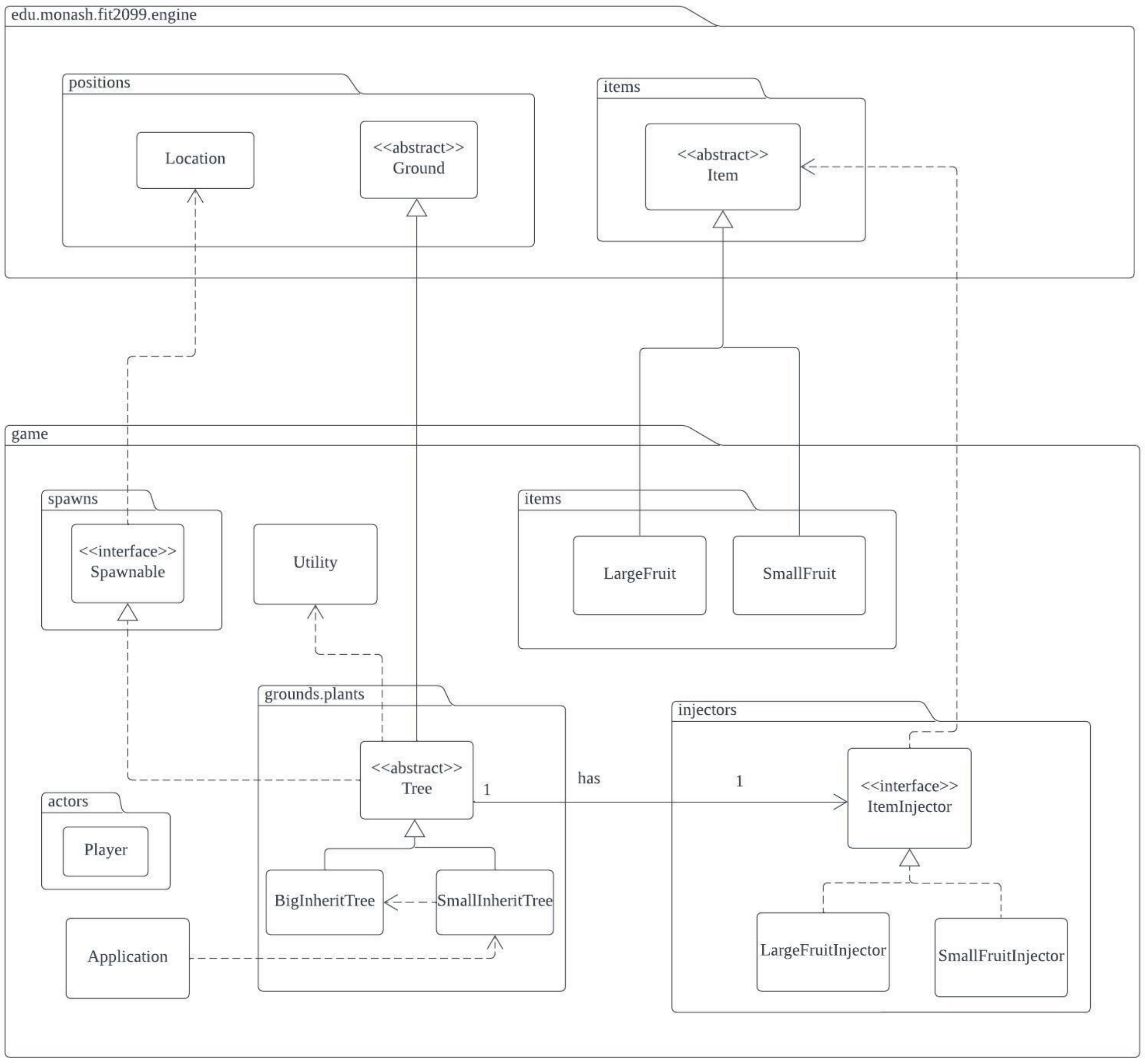
ItemInjector which is an interface that provide a method to generate an Item object. Since this interface only generate Item object so it follows the Integration Segregation Principle (ISP). The advantage is that it ensures the class implements this interface must be able to generate an item since this interface is only related to the Item. The disadvantage is we need to create more and more class to implement this method as the variety of item increased in the game. The alternative way is that we do not need this interface, but to generate the item in the game, the application can instantiate the item involved in the game in the Application class.

LargeBoltInjector and MetalSheetInjector are created to implement the ItemInjector interface to generate the specific item in the game. LargeBoltInjector can generate LargeBolt and MetalSheetInjector can generate MetalSheet. ItemInjector will be used in the Application class so LargeBoltInjector and MetalSheetInjector can replace the ItemInjector and complete the actions so it follows the Liskov substitution principle (LSP). The benefit is that we reduce the application dependency on the ItemInjector subclass. We only need to declare ItemInjector in the application and we can use every subclass of it. It was useful when the type of items increased in the future.

In the engine code, the item will give the option to pick up the Item class when the player is standing on the ground which contains the Item. For the player's turn, the item in the inventory will provide an option for the player to drop the item. Due to the abstraction layer of the Item, LargeBolt and MetalSheet can be picked and dropped by the player.

Requirement 2 - The Intern of the Static factory

UML Diagram



Rationale for Requirement

Two classes are created to represent LargeFruit and SmallFruit. These two classes are concrete as they will be instantiated in the game. At this stage, they are considered to be picked up and dropped off by the player so they fulfil the property of the Item. We can make them extend the Item abstract class to prevent redundant implementation (DRY). LargeFruit and SmallFruit are separated into two different classes because they are various types of items and they are possible to have different functions in the future (SRP). The advantage is we do not need to repeat the implementation by using inheritance and adding their feature separately instead of just using one class to represent them. The alternative design decision is using a god class to represent these two fruits but when adding features we need to use a conditional statement to recognize both fruits. The code will be lengthy if more and more features are added.

A spawnable interface is created to provide a method that spawns an object at the location of the gameMap only and it follows the Interface Segregation Principle (ISP). This interface will be implemented by a class to generate objects like ground spawn items and more. In this requirement, the tree will use this interface to spawn fruits. The advantage is that any classes that can spawn objects in the future extension can reuse this spawnable interface. The disadvantage is that when using this interface, we must provide its implementation, sometimes some classes will have a similar implementation.

LargeFruitInjector and SmallFruitInjector classes implement the ItemInjector I created in the previous requirement. LargeFruitInjector creates a new instance of LargeFruit and SmallFruitInjector creates a new instance of SmallFruit. As we can see the Tree abstract class has ItemInjector as an attribute. Therefore, LargeFruitInjector and SmallFruitInjector can be used by the Tree classes and its subclasses (LSP). The tree only depends on the ItemInjextor and does not depend on the subclass of ItemInjector anymore (DIP). The advantage is that we reuse the interface that was previously created. When we want to use a variety of types of ItemInjector in the Tree class, we do not need to declare any subclasses of ItemInjector in the Tree classes. The disadvantage is that we still need to instantiate LargeFruitInjector and SmallFruitInjector in the subclasses of the Tree abstract class.

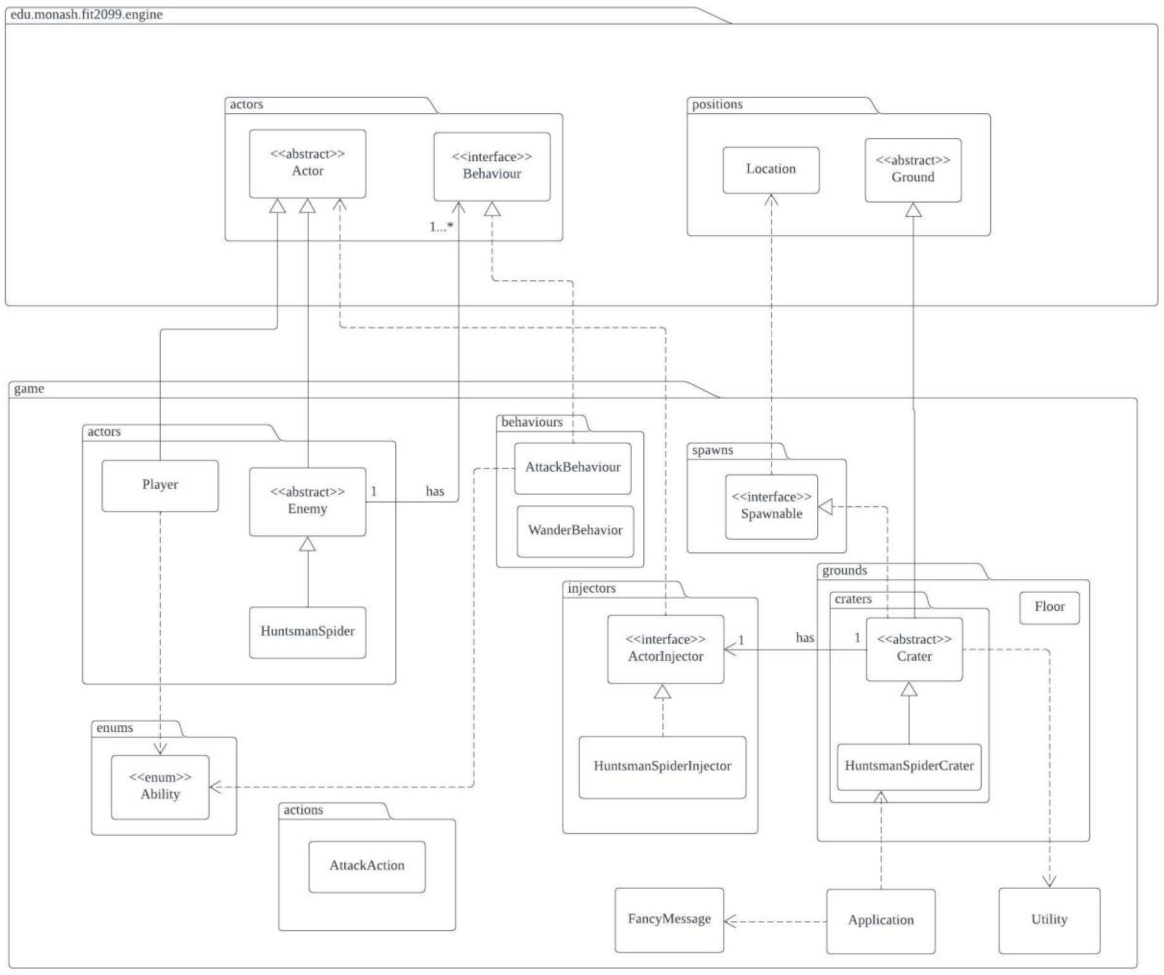
Abstract tree class is created by extending the Ground abstract class in the engine. Tree class is abstract as it will not be instantiated in the game. The tree has the property that a ground has so I extend the Ground abstract class to prevent repeating the same implementation (DRY). Tree abstract class has the common property of a tree that can drop fruits. Inheritree can drop different types of fruits so I use BigInheritTree class to represent the mature inheritree and SmallInheritTree to represent the immature inheritree (SRP). These two classes extend the Tree abstract class as they can drop fruit. We do not need to modify the code of Tree abstract classes and create different types of trees by extending it and this follows the Open-closed Principle (OCP). The advantage of the Tree abstract class is we can add more trees that drop different fruits easily by extending it. Separating the inheritree into two classes makes each class have only one responsible for dropping one type of fruit. It can prevent using conditional statements to distinguish the fruit that should be generated in the code. The disadvantage is we need to instantiate the class in the tick method when a tree can grow up.

An alternative way to design the inheritree is we do not need the abstract class by creating the inheritree. We can directly extend the Ground abstract class. But this design is limited and we should always repeat the implementation if more trees are added into the game.

The tree is added to the application class and it will drop the fruit due to the implementation of the tick method. Fruits can be picked and dropped by the player as they are items.

Requirement 3 - The moon's (hostile) fauna

UML Diagram



Rationale for Requirement

ActorInjector is an interface created to provide the functionality to generate a new instance of actor in the game. This interface only focuses on making the Actor objects (ISP). The advantage is this class can increase the flexibility of switching to different injectors to generate different subclasses of actor in the game. The disadvantage is that the number of classes will be increased if we need different implementations for different actors.

Crater is an abstract class which extends the abstract Ground in the engine. The Crater is abstract because it is a template for a crater to spawn an actor. It also implements the Spawnable interface as it can spawn actors. As the diagram shows, the HuntsnmanSpiderCrater is the Crater that will spawn HuntsmanSpider. In the future, we can generate different craters by extending the abstract Crater class without modifying the code (OCP). The advantage is that we can easily add new features to the different craters without altering the existing code. The disadvantage is that at this stage, I only have the requirement of one crater and the implementation of the abstract Crater class may not be general.

The Crater has an associative line with the ActorInjector. It will only depend on the ActorInjector and does not depend on the lower-level module to generate an actor (DIP). The advantage is that the subclass of crater can use any implementation of ActorInjector to spawn different actor in the game. The disadvantage is that it must be an association to prevent the dependency on the subclass of Crater with the classes implement the ActorInjector.

HuntsmanSpiderCrater extends the abstract Crater class to prevent the repeated chunk of code (DRY). HuntsmanSpiderCrater can only generate the HuntsmanSpider at the exit of its location (SRP). The advantage is that the code is easy to manage and clean as HuntsmanSpiderCrater only focuses on the HuntsmanSpider. The disadvantage of separating craters to generate different actors is that the number of classes will grow if we have many different craters in the future.

AttackBehavior implements the Behavior interface in the engine. This class implements the behaviour of an NPC to generate AttackAction for the player and it can be added to the behaviour of any NPC to replace the behaviour object to execute the code (LSP). The Ability enum class is used to recognize the actor can be attacked. The advantage is that it can be reused by any NPC to attack the player in the game. The disadvantage is that we need to consistently incorporate this feature into the NPC's behaviour if it can attack.

The Enemy class is modified using the existing code in the HuntsmanSpider provided in the base code. I make it to an abstract class and rename it to the Enemy to represent the enemy of the player. I assume that Enemy is an NPC so it can have many behaviours. The common behaviour is wandering around. It will not be instantiated in the game so it is abstract. It provides the common implementation of an enemy in the game so we can create a new subclass by extending it without modification (OCP). The advantage is that we can customize the enemy using it and add new functionality on the classes extending it. The disadvantage is that the number of its subclass will be increased if we have more types of enemies in the future.

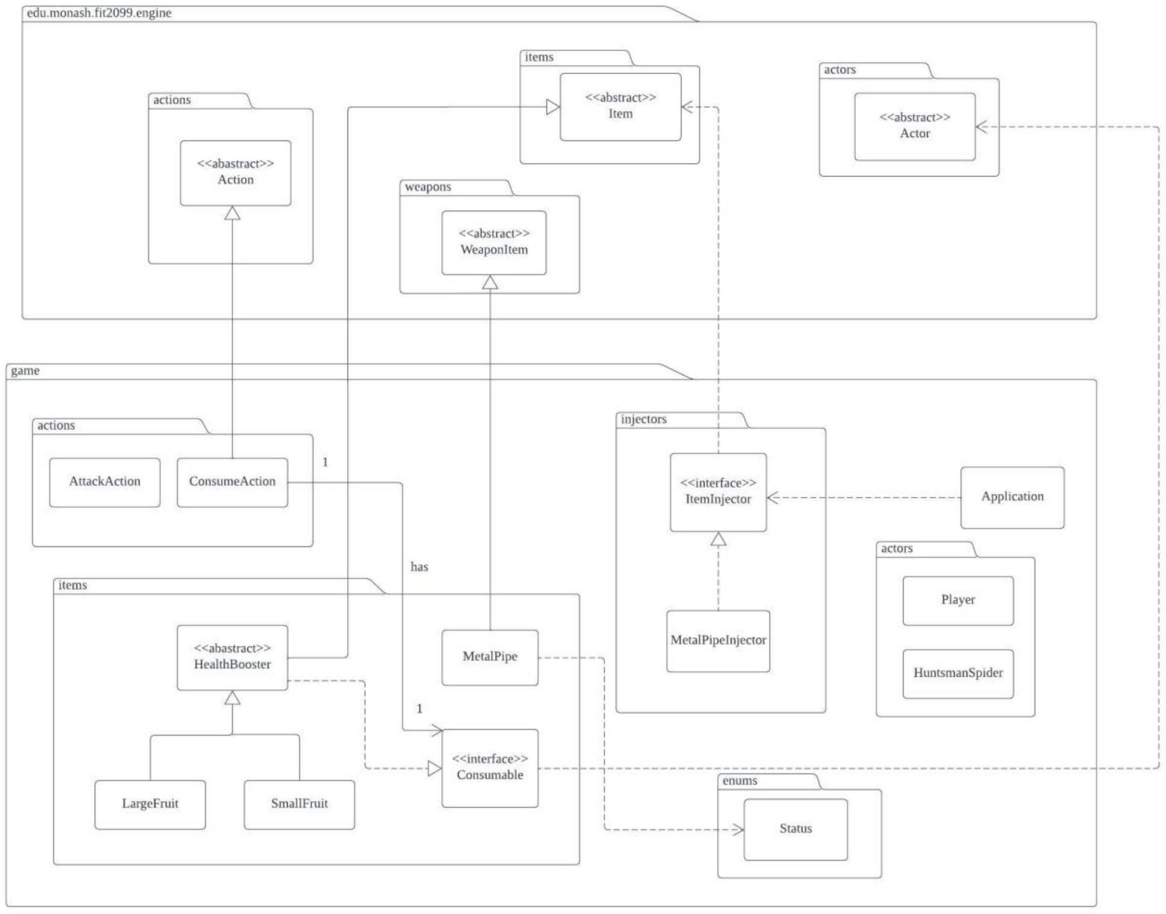
HuntsmanSpider extends the abstract Enemy class so we prevent the repeating implementation (DRY). The method of getIntrinsicWeapon is overwritten to make it can kick the player with damage equal to one and hitRate equal to 25%. The advantage is that we can provide specific functionality to the HuntsmanSpider such as the IntrinsicWeapon of HuntsmanSpider.

HuntsmanSpiderCrater is instantiated in the Application to make it spawn HuntsmanSpider in the game. The HuntsmanSpider is an NPC so it will make decisions based on the behaviour. To only allow the player to enter the floor, ENTER_FLOOR ability is added to the Player and the Floor is modified by the allowed actor entered based on this enum. The hitpoint of the player is printed by adding a chunk of code in the playTurn function before printing the menu. The Fancy message about the game over is used in the Application to print out the message. Utility class is used to handle all the probability in the game.

Another design is to keep the HuntsmanSpider the same and only create a Crater that spawns HuntsmanSpider without using the injector. I do not choose this design because the crater may spawn another actor in the future so we must modify the crater class to achieve this extension. It violated the open-closed principle. We will repeat the implementation for another creature without the abstract Enemy class.

Requirement 4 – Special scraps

UML Diagram



Rationale for Requirement

MetalPipe class extends the abstract WeaponItem class in the engine. MetalPipe is a weapon that the actor can pick up and attack another actor in the game. The properties of metal pipe are similar to the WeaponItem. Extending the WeaponItem class can prevent the repeating implementation (DRY). This MetalPipe is only used to represent metal pipe (SRP). The advantage is that we only need to provide the detail of this weapon by passing it to its parent's constructor and overwriting the allowableActions to generate an AttackAction using this weapon based on the status of actors at the exits of the owner. The disadvantage is that as the variety of weapons grows, the number of classes will increase due to our adherence to the Single Responsibility Principle.

MetalPipeInjector is created by implementing the ItemInjector which I created in the previous requirement. It can be used in the Application class to add the instance of MetalPipe by replacing the ItemInjector object into the game without causing an error (LSP). The advantage is that we reuse the interface to add the MetalItem into the game without directly instantiating the MetalPipe in the Application class and prevent the Application depend on the MetalPipe. The disadvantage is that we still need to instantiate the MetalPipeInjector when adding MetalPipe to the Application class.

The Consumable interface is designed to provide the functionality of an object that can be consumed by an actor. It only focuses on the effect of the object when consuming the object by the actor (ISP). The advantage is that this interface can be reused by objects in the game that can be consumed by an actor. The disadvantage is that we need to provide implementation when using this interface and some implementation will be the same.

ConsumeAction is a class that extends the abstract Action class in the engine. It can replace the Action object in the engine when executing the action chosen by the actor without causing an error (LSP). ConsumeAction needs to have the Consumable to execute the function of the object that can be consumed by the actor. It depends on the Consumable interface to complete this action (DIP). The advantage is that ConsumeAction can accept any objects that implement the Consumable interface and apply its effect to the actor who consumes the object.

HealthBooster is an abstract class that extends the abstract Item class in the engine and implements the Consumable interface. It is used to provide the template for an item that can be consumed by actors to increase their hitpoints. It can be extended without modification to achieve this purpose (OCP). To allow user to consume, we just need to overwrite the allowableActions to return a consumeAction by passing itself. The advantage is that we can easily create consumable items that increase the hitpoints of actors and add specific functionality in the classes to extend it. The only disadvantage is that the subclass can only be an item.

I modified the LargeFruit and SmallFruit to extend the HealthBooster abstract class. LargeFruit and SmallFruit are now can be consumed by the actor and we do not need to repeat similar implementations for these two fruits by extending the HealthBooster abstract class (DRY). The advantage is that we only need to pass the details of these two items to the constructor of their parent to create these two items. We can add more specific functionality for these two fruit separately.

In this requirement, metal pipe needed added to the application using the MetalPipeInjector. Since it is also an item, it can be picked and dropped by the player and also provide AttackAction based on the description above. For the LargeFruit and SmallFruit, since we already added them in the previous requirement, we just need to modify them as above to make them can be consumed by the player to heal the player. The player is involved in the game and HuntsmanSpider can be attacked by the player.

An alternative design for LargeFruit and SmallFruit is that do not use the consumable interface. We can just adding a HealingAction and make them return the the HealingAction in the allowableAction. This design can only provide healing action and when actor can consume another object we need to create another action it is not generic so I do not choose this design.