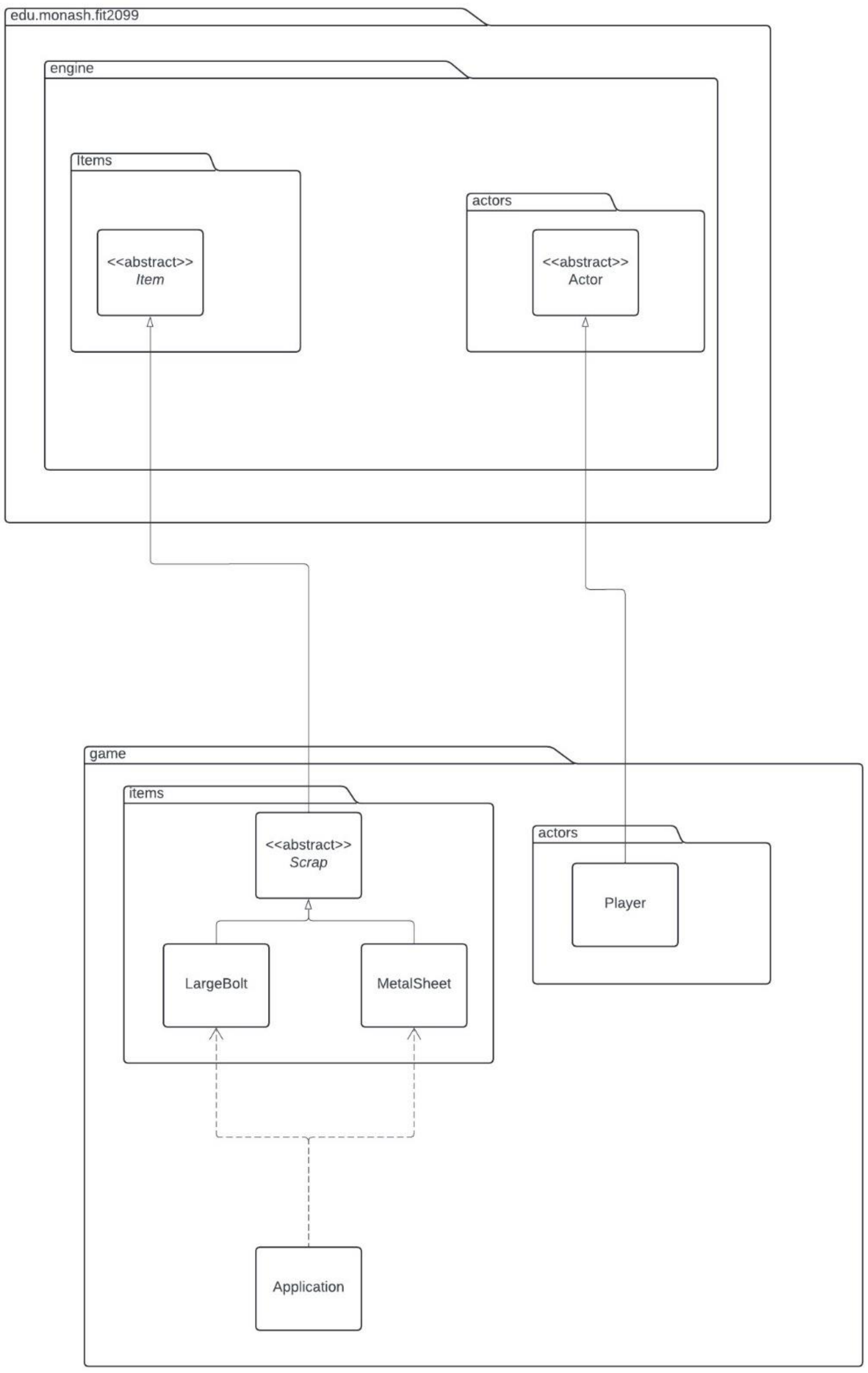
Assignment 1 Report – UML Diagrams & Design Rationale's – Dean Mascitti – 33110360

REQ1

REQ1 UML Diagram:



Design Goals

The goal of this requirement was to add functionality in that creates 2 different scraps being metal sheets and large bolts and make it so that the intern can pick up these scraps and drop them off when prompted.

The goals in designing this functionality were to ensure that the design was extensible for future extension in upcoming assignments, as well as repeated code was limited throughout the project, abiding by the do not repeat yourself (DRY) principle. The use of abstraction is a goal of this requirement to ensure that both these principles are abided by.

Design & What OOP Principles Were Applied

The main design idea for this requirement was to make use of a Scrap abstract class, this scrap abstract class can then be inherited by MetalSheet and LargeBolt classes. In the future if other scraps are added to the game if appropriate and depending on the context.

The Scrap abstract class also inherits from the Item abstract class in the engine code, ensuring it adopts the functionality made in the engine code for items that can be picked up and dropped off.

In terms of functionality the Items were placed in the game map in the Application class for the Intern to interact with.

In my design for this requirement the following OOP design principles were used: extensibility, open closed principle, abstraction and DRY.

How They Were Applied & Why in This Way

- The use of the abstraction of use of the Scrap class was implemented because if there is any similar functionality to be done by scraps in the future it can be written in the Scrap abstract class therefore applying extensibility. In addition, this ensures that any similar functionality is written within the Scrap abstract class and does not need to be repeated in other subclasses (scraps) applying the do not repeat yourself (DRY) principle.
- The open-closed principle is present in this design due to the fact that if we need to add any more scraps to the project, we can simply create new classes and make them inherit Scrap, therefore not needing to edit any existing code. This was done so in further requirements it is easy to add features and does not require refactoring.

Design Pros and Cons

Pros

- Use of abstraction
- Easily extensible for future applications and features
- No repeated code within the program

Cons

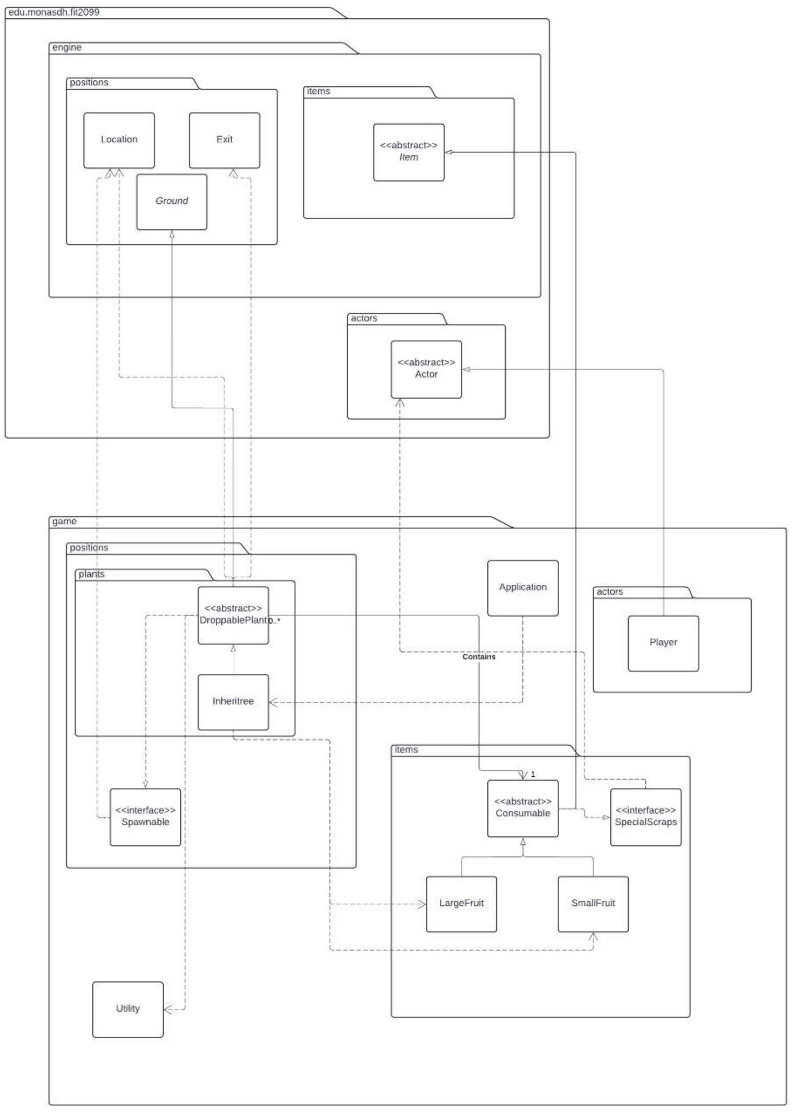
- Double abstraction (Scraps inherit from Scrap and Scrap inherits from Item)
- Other scraps may have completely different functionality to MetalSheet and LargeBolt and it may not make sense for them to both be subclasses of Scrap
- Most functionality is done in engine code (PickUp and Drop actions) so Scrap abstract class does not have a lot of responsibility at this point in time.

Alternative Designs

An alternative design would have just been to create a MetalSheet and LargeBolt class each and have them inherit from Item, however the Item class is very generic and there is expected to be many different types of items, and scraps are one of them. We can expect that scraps may potentially have similar functionality in the future, therefore the reason I chose to make this abstract class is because it is a place where the functionality can be written so that each scrap subclass does not need to repeat themselves.

REQ2

REQ2 UML Diagram:



Design Goals

The aim of this requirement is to implement the functionality of Inheritree in that its ability to drop small and large fruits every tick, and its ability to mature after a certain number of ticks and therefore start dropping large fruits as opposed to small fruits.

The goal in the design for this requirement is to ensure that the program is extensible in the sense that if more plants are added to the program, they are easy to implement and don't require a lot of refactoring or repeated code. The use of abstraction therefore is a goal of this requirement through use of interfaces as well as abstract classes.

Design & What OOP Principles Were Applied

In terms of the Inheritree implementation the main strategy of the design was to create an abstract class called DroppablePlant, in which plants who are able to drop fruits, or any other consumable would inherit. In addition, the use of a Spawnable interface was utilized to give DroppablePlant the ability to drop items and to group them with other object/classes/grounds that can also spawn/drop things. DroppablePlant inherits from Ground as since plants are not mobile it would suit the functionality of the Ground, in which is programmed in the engine code.

The implementation of the Inheritree functionality of changing its display character, dropping fruits depending on a drop rate and maturing was done primarily in the DroppablePlant abstract class, as you would expect these functionalities would be similar for all plants in which can drop consumables.

In terms of the implementation of the fruits themselves I made a Consumable abstract class in which every item that can be consumed inherits from 9used in later requirements more so), therefore SmallFruit and LargeFruit classes inherited from this class. Consumable inherits from the engine code's Item abstract class due to the fact that consumables adapt the baseline functionality of items represented in the engine code.

In my design for this requirement the following OOP design principles were applied: extensibility, DRY, abstraction, open-closed principle, dependency inversion and single responsibility principle.

How They Were Applied & Why in This Way

- The reason the DroppablePlant abstract class was made and used for the Inheritree to inherit is because there would be a lot of common functionality between Inheritree and other plants that drop consumables, such as applying the drop rate, having a consumable it drops and dropping the actual consumable, therefore all of this code can be written in the DroppablePlant abstract class. Therefore, this code is not to be repeated in other Inheritree and other future plants that drop consumables, so the DRY principle was achieved. In addition, as plants which drop consumables are extremely easy to add and can be added without changing existing code this design decision also adheres to the extensibility and open-closed principles. This class also allows for Inheritree to handle the ticking function only, and not the dropping off its fruits and the inner functionality of that (drop rates/ maturity status etc.), therefore adhering to the single responsibility principle.
- The reason why a Spawnable interface was made/used is because there may be multiple things in the game in future that will be able to spawn things that aren't plants, therefore the Spawnable interface acts as an easy way to give these classes the ability to do this spawn functionality and groups all of these classes together for future dependency inversion applications in the future.

- The addition of the Consumable abstract class for which the large and small fruit classes inherited was added because there is likely to be many other fruits/vegetables that may be added to the game and may have the similar capabilities, such as being dropped from a plant and be consumed. Therefore, having this abstract class allows for this functionality to be written in this class and not repeated again abiding by DRY. This also allows for dependency inversion in the DroppablePlant class as we do not have to specify what exact subclass or consumable a plant is dropping, we can just set this attribute as a Consumable, therefore reducing dependencies.

Design Pros and Cons

Pros

- High level of abstraction
- Extensible code – easy to add to for future features
- When similar classes are added there will not have to be much refactoring of existing code
- Polymorphism capabilities
- Reduction of dependencies

Cons

- Spawnable for dependency inversion is not used at this point
- Multi-layer abstraction (SmallFruit -> Consumable -> Item, for example), which at times may be confusing or may cause issues in overriding methods.

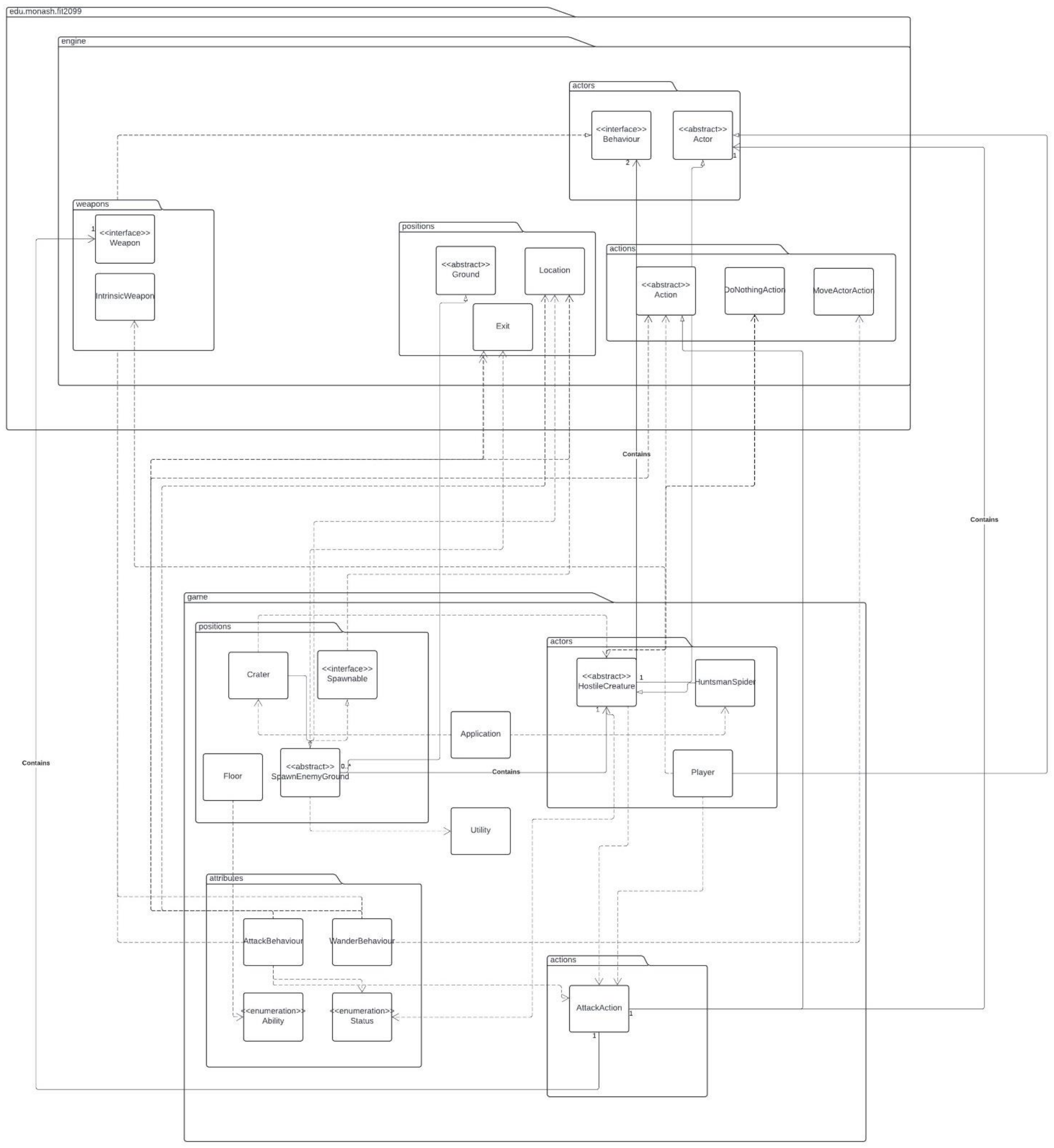
Alternative Designs

An alternative design would be not to use abstraction and just make the e classes as there description, however this would result in code that is not extensible at all, and a lot of repeated code for future applications as you would have to remake very similar classes and re write these similar functionalities.

An idea I had was just to make a droppable abstract class for every class that can drop something, but this didn’t end up eventuating because it seemed too generic and I didn’t want to make too many assumptions in that other classes would have the same rules of dropping as plants (maturity, drop rates etc.), therefore I decided against this.

REQ3

REQ3 UML Diagram:



Design Goals

The aim of this requirement was to implement craters and their ability to spawn huntsman spiders based off a 5% drop rate each tick. I addition the creature is expected to wonder around the map until n the vicinity of the intern then it will attack the intern with its long legs.

The goals of my design were to ensure that my program is extensible in that in implementing the craters ensuring that they could handle dropping other creatures in the future easily. Use of abstraction and restricting repeating code are also goals of this design.

Design & What OOP Principles Were Applied

A SpawnEnemyGround abstract class was made in which the crater inherited from, this class is expected to be inherited from any ground types In which can spawn an enemy including craters.

This SpawnEnemyGround class implemented the Spawnable interface mentioned in REQ2 that allows any class to spawn or drop something, grouping all of these classes together.

In terms of the huntsman spider implementation an abstract class was made called HostileCreature as it is expected that all hostile creatures would have the same functionality.

In terms of implementing huntsman spider attack intern functionality a new behaviour was added in which checks if the hostile creature is in the vicinity of an actor that isn't hostile to the intern and then prompts the creature to attack that actor.

The design OOP principles used in this requirement include: dependency inversion, single responsibility, open-closed, DRY, abstraction and extensibility.

How They Were Applied & Why in This Way

- The reason the SpawnEnemyGround abstract class was made is because you would expect that any ground type that spawns an enemy will have similar functionality, therefore by adding this functionality in this abstract class, subclasses like crater can simply inherit this and use the functionality without writing the code in its own class, therefore DRY and open-closed principles are applied as when more ground types that can spawn enemies are added no original code will need to be refactored, and the program will therefore be extensible.
- The reason the HostileCreature class was added is because it is assumed that all hostile creatures will attack players when in the vicinity, wander otherwise and have the ability to attack. Therefore, all of this functionality can be made in the abstract class and when other hostile creatures are added in the future little code will need to be refactored and subclasses will not need to repeat each other; achieving extensibility, abstraction, DRY and open-closed principles. In addition, this abstract class allows for dependency inversion in the SpawnEnemyGround class due to the fact that we can specify its attribute as a HostileCreature rather than the specific subclass, therefore reducing dependencies.
- The reason why the AttackBehaviour class was made it may be possible that other actors may have the same behaviour, therefore could be used in future applications elsewhere in addition to for HostileCreature, promoting extensibility and single responsibility, as other classes won't be responsible for this behaviour.

Design Pros and Cons

Pros

- Extensible for future enemy spawners, hostile creatures and attacking behaved actors.

- Detailed use of abstraction
- Reduced dependencies
- Polymorphism capabilities
- Lack of repeated code

Cons

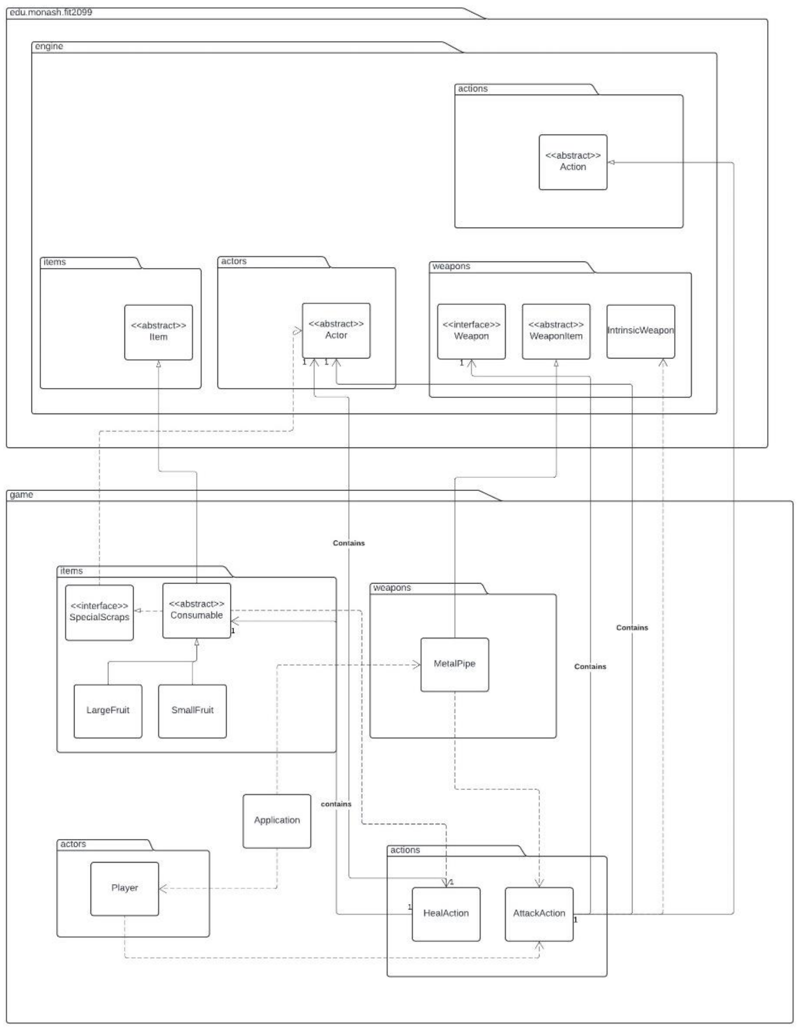
- Potentially could be abstracted broader (instead of hostile creature could be hostile actor, however this would come with other issues).
- Use of double abstractions

Alternative Designs

An alternative design I initially thought of was to not have the SpawnEnemyGround class omitted, and therefore have the crater class customizable based on drop rate and what creature it spawns, however that ignores the fact there may be ground types other than craters in which can spawn creatures, therefore I chose not to implement that design.

REQ4

REQ4 UML Diagram:



Design Goals

The aim of this requirement is to implement a metal pipe functionality that the player can pick up put in their inventory and then use to attack hostile creatures, in addition the intern should be able to punch enemies as well. The other part is to implement functionality for the intern to be able to consume the fruits from req2 and regain health through consumption.

The design goals of this requirement include ensuring that future extensibility is though through and accounted for, as well as implementing classes in which are easily extensible and adhere to good OOP practice, such as not repeating code, making code easily added on without refactoring and use of abstraction where appropriate.

Design & What OOP Principles Were Applied

The metal pipe class was added and inherited the WeaponItem abstract class from the engine code provided, the WeaponItem class has much expected functionality of items used to attack other actors. Also added a SpecialScrap class which gives special scraps the ability to return the list of their special abilities and MetalPipe inherits this class.

In terms of creating heal options for intern with fruits a HealAction class as created which allows the player to have the option of consuming the fruit and healing themselves, and then executing this action. This action is added to the allowableActions method for the Consumable class as we are under the assumption all consumables can be consumed and potentially heal the player if in their inventory. Also added a SpecialScrap class which gives special scraps the ability to return the list of their special abilities and Consumable inherits this class.

The ability of the player to also attack hostile creatures with their bare hands was added as the player's intrinsic weapon, as it is assumed intrinsic weapon refers to the actors fundamental/given weapon/way of attack.

The design OOP principles used in this requirement include: open-closed, DRY, extensibility, abstraction and single responsibility.

How They Were Applied & Why in This Way

- The reason that the Metal pipe class inherited the WeaponItem abstract class is because this class from the engine already had a lot of attributes and functionalities that the metal pipe required, therefore using ready made classes rather than making my own very similar class was the better option under the DRY principles. This also adheres to the abstraction and open-closed principles as through the use of WeaponItem abstraction and the adding of the MetalPipe class functionality without changing any existing code.
- The reason why a HealAction was made is because it is a relatively generic action in that it does not refer to a certain type of healing or doesn't restrict any classes from using it, therefore promoting extensibility for other actors to be healed by a consumable, therefore if this is the case where other actors can in the future consume something and be healed this action is ready made and would not need any other implementation or repetition of code, therefore abiding by DRY and open-closed principles. Also taking the responsibility away from actors and consumables in doing the heal action adhering to the single responsibility principle.

Design Pros and Cons

Pros

- Largely extensible for more healing applications and special items that can be used as a weapon
- Polymorphism capabilities
- Adding new features requires little refactoring original code
- Effective use of provided engine code

Cons

- Special scraps are somewhat not related functionality wise (they can have different functionalities)
- Double abstraction – can possibly lead to overriding method issues and lack data leaks

Alternative Designs

An obvious alternative would have been to make a functionality connection between all future and current special scraps; however, I found many issues with the prospect of this. Firstly, weapon scraps already have an abstract class in the engine code being WeaponItem, so for other special scraps like ones who can heal it does not make much sense to create an abstraction between the two because the only difference they hold is there allowable actions, and this method is already abstracted in the Item class. Therefore, instead I chose to make an interface for all special scraps named SpecialScrap, so that they can be grouped together and used in upcoming assignments to be traded and sold.