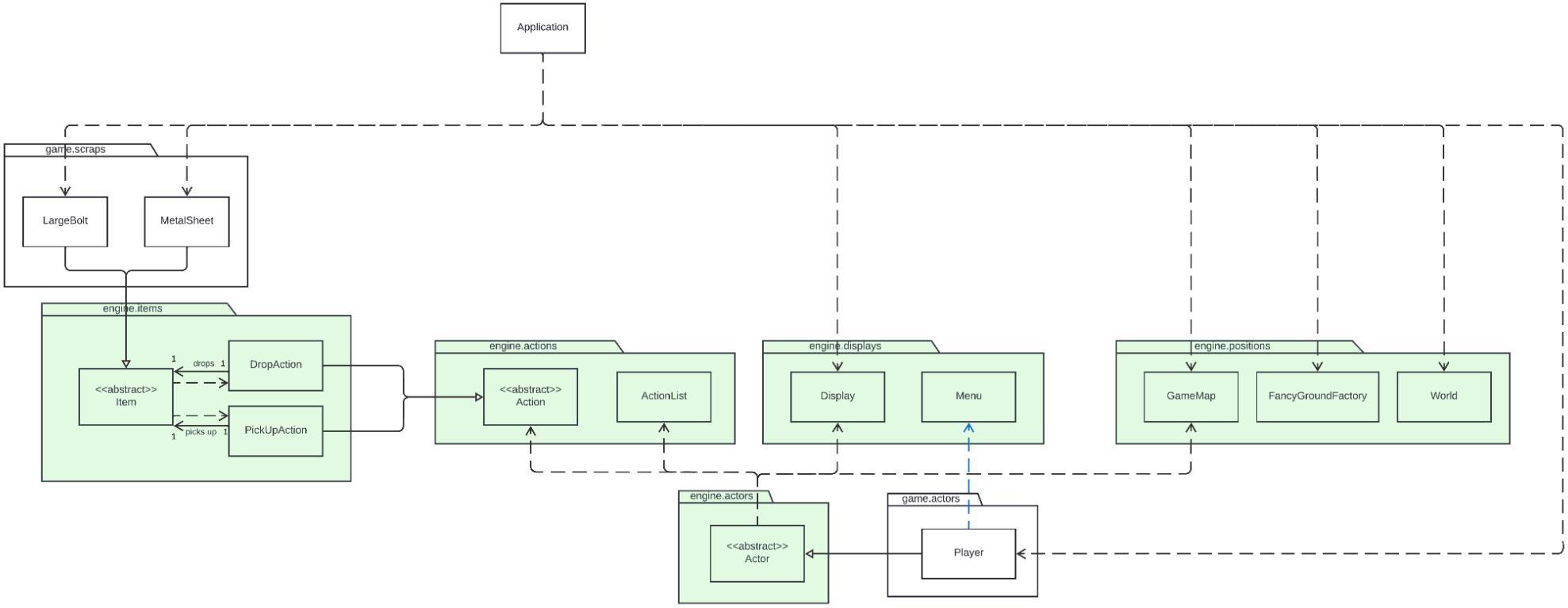
Design Rationale

Design Goal

The design aims to break down the system into smaller, more manageable classes to promote modularity and encapsulation. In this system, each class will have a clear and distinct responsibility so it is easier to understand and maintain. By adhering to the SOLID principle and DRY principle, the system should allow for easy extension and promote reusability. This involves identifying common functionalities and abstracting them into reusable components as well as anticipating potential changes in order to enhance code flexibility and adaptability to changing requirements and future extensions.

Req 1 - The Intern of the Static Factory



Req 1 UML Diagram

<table>
<tr><th>Classes Modified / Created</th><th>Roles and Responsibilities</th><th>Rationale</th></tr>
<tr><td>LargeBolt
(extends Item)</td><td>Class representing large bolt.

Relationships:
1. It extends the Item class to achieve portable functionality.</td><td rowspan="2">Alternate Solution:
Create a concrete class Scrap and implement all the details of the large bolt and metal sheets in this class.

Finalised Solution:
While introducing different scraps to the map, the details of the scraps are implemented in the two newly created concrete classes inheriting Item class instead of being directly implemented inside the Item class.

Reasons for Decision:
1. It is more scalable and will not introduce complexity to the classes by doing non-relevance responsibilities or even breaking the original code when abundant new scraps are introduced. (Single Responsibility Principle)

2. It improves code maintainability as modification on the existing classes is not required when more scraps are introduced instead only an extension of the Item class by the new scraps is needed for the new feature. (Open-closed Principle)

3. While iterating with the item (eg. in World class), multiple statements, each for different types of instances is not required. Instead, they are replaceable by Item without breaking their expected behaviour. (Liskov substitution Principle)

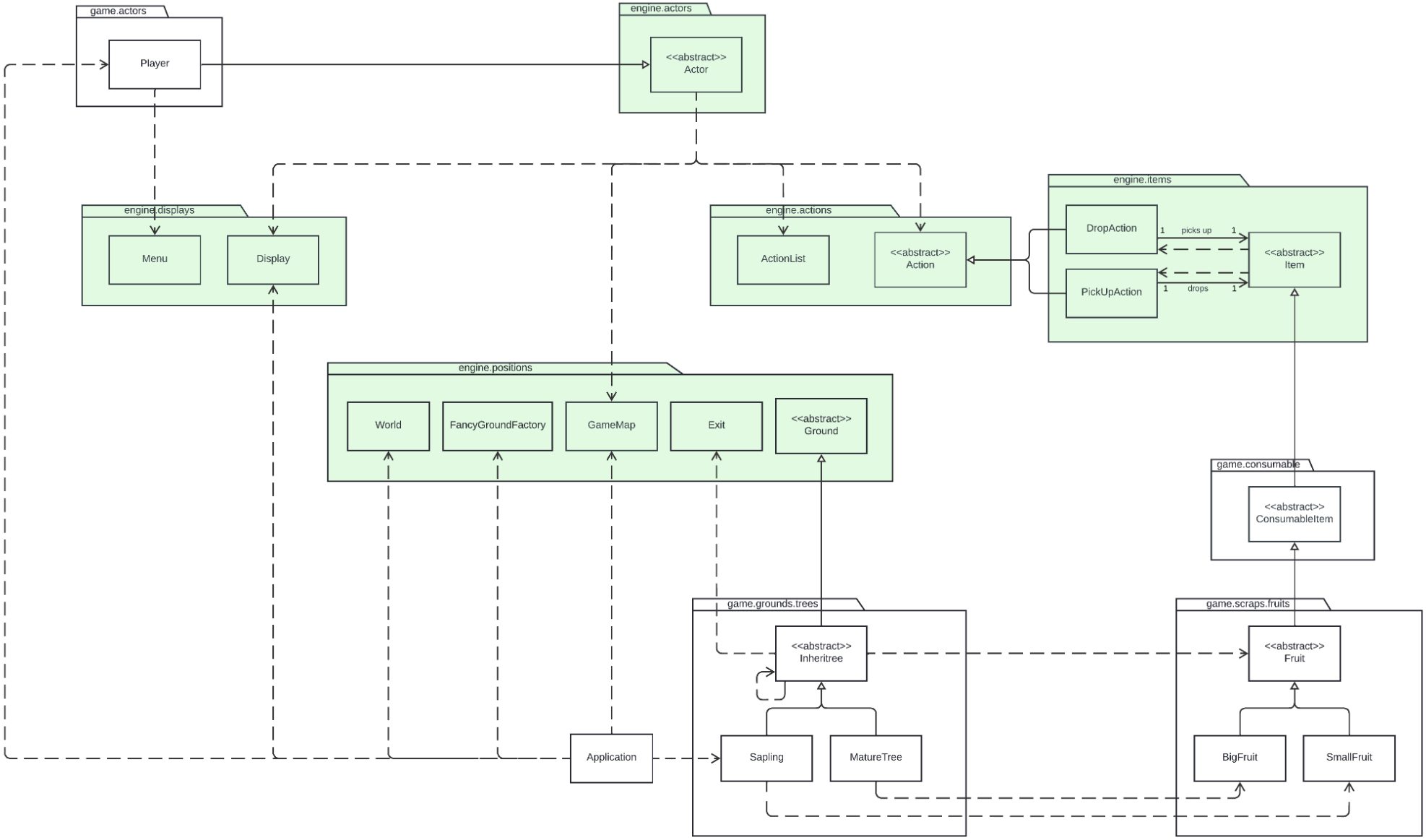
4. The large bolt and metal sheets are not forced to implement each other’s method they don’t need to. This makes the code more focused and easier to understand. (Interface Segregation Principle)

5. The high-level modules (eg. World class) can directly depend on the abstract class Item which helps prevent it from having a dependency on each and every type of scraps so that it is unaffected by the changes in all of the scraps (low-level modules). (Dependency Inversion Principle)</td></tr>
<tr><td>MetalSheet
(extends Item)</td><td>Class representing metal sheet.

Relationships:
1. It extends the Item class to achieve portable functionality.</td></tr>
</table>

		6. As the Item class already did the work for pick up and drop action, LargeBolt and MetalSheets extend from it to prevent repeated code. *(DRY Principle)* **Limitations and Tradeoffs:** 1. More classes will be created when more scraps are introduced. This resulted in code scattered everywhere and might make it hard to trace the flow of the implementation logic. However, this can be improved by having a sequence diagram without compromising with the pros. 2. Introducing separate classes for LargeBolt and MetalSheet might overengineering a simple item which might create unnecessary overhead in terms of memory usage and development effort if these classes don't offer significant additional functionality compared to a more generic approach.

Req 2 - The moon's flora



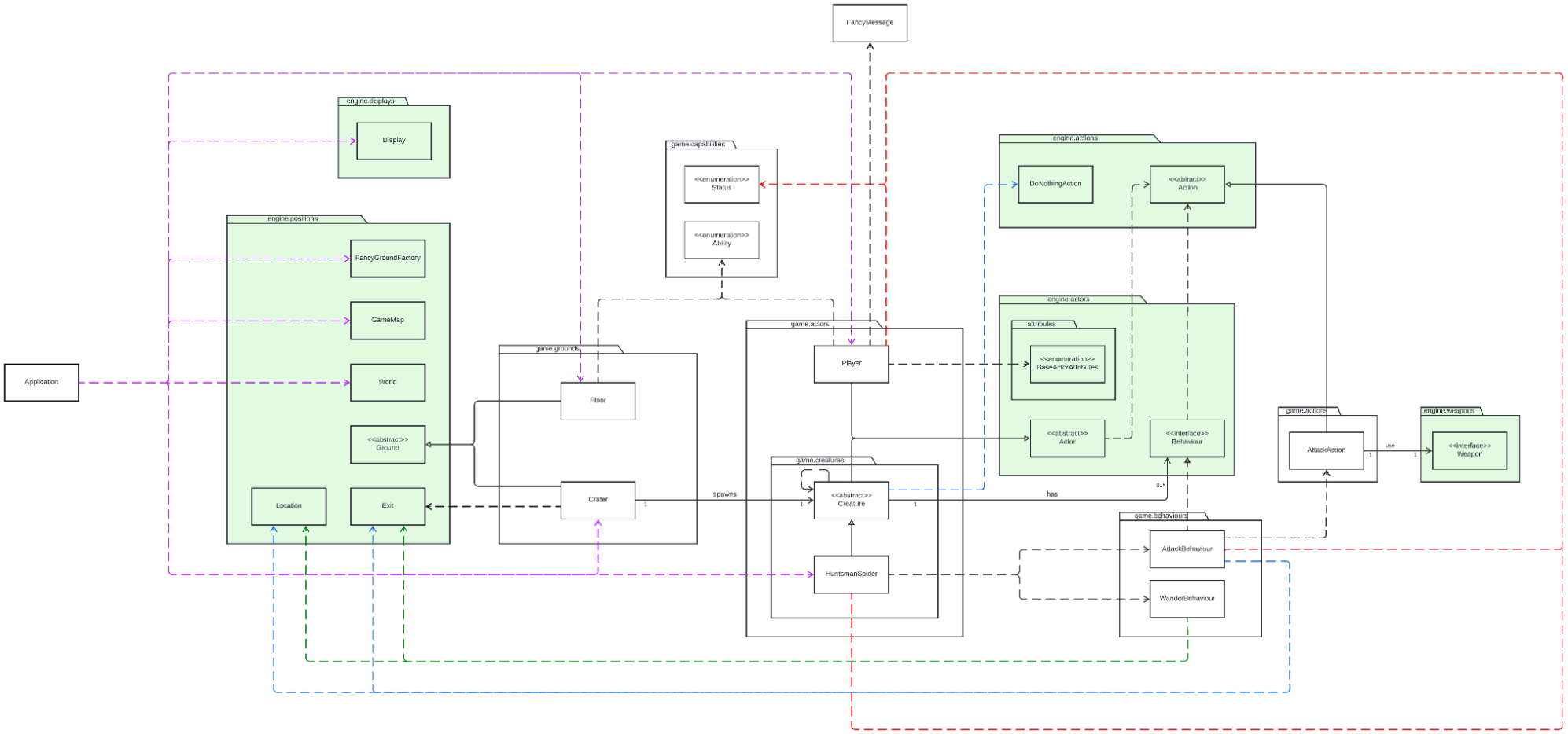
Req 2 UML Diagram

Classes Modified / Created	Roles and Responsibilities	Rationale
Inheritree (extends Ground)	An abstract class representing inheritree. Relationships: 1. It extends the Ground class to achieve the growing and fruit-producing features by overriding the tick() method. 2. It depends on Fruit to promise each stage of the inheritree (eg. Sapling and MatureTree) can produce specific fruits. 3. It depends on itself to determine which stages to grow into.	**Alternate Solution:** Implement all the characteristics and behaviours in different stages in an Inheritree concrete class. **Finalised Solution:** Introduce an Inheritree abstract class extending Ground class and create concrete classes for all stages of inheritree that inherit from it. On top of this, introduce a Fruit abstract class that inherits ConsumableItem abstract class and create concrete classes for all types of fruits which inherit from it. ConsumableItem class that inherits the Item class is created for later requirements. The Inheritree class and Fruit class are designed as an abstract class instead of a concrete class to avoid instantiation as they are just blueprints which help in creating consistent interfaces each for all stage classes and fruit classes respectively. To achieve this, an abstract method growFruit() is declared in Inheritree class to promise all the stages of the inheritree to have the characteristic to produce specific fruit. For the abstract class Fruit, the SmallFruit class and BigFruit class that inherit it promise to have the characteristic dropping with a specific probability (ie. drop() method). Furthermore, interface is not considered for both the Inheritree class and Fruit class since all stage subclasses and fruit subclasses themselves have the same attribute (eg. ticksBeforeGrow, droppingProbability) and implementation of methods (eg. tick(), drop()). It is declared as an abstract class to enhance code reusability and avoid repeated code. *(DRY Principle)* **Reasons for Decision:** 1. The details of each stage of the inheritree (eg. display character, next state to grow into, ticks needed for growing, and fruit to be produced in current state) and the details of the specific fruits (eg. display character, probability of dropping) are implemented in their own classes instead of being directly
Sapling (extends Inheritree)	Class representing the sapling state of the inheritree. Relationships: 1. It extends the Inheritree class to promise to produce a fruit at each tick. 2. It depends on SmallFruit to produce SmallFruit. 3. It depends on Inheritree and MatureTree to grow to the next state (ie. the mature state).	
MatureTree (extends Inheritree)	Class representing the mature state of the inheritree.	

	Relationships: 1. It extends the Inheritree class to promise to produce a fruit at each tick. 2. It depends on BigFruit to produce BigFruit.	implemented inside the Inheritree class since it might introduce complexity to the Inheritree class by doing non-relevance responsibilities or even breaking the original code when more states and types of fruits are introduced. *(Single Responsibility Principle)* 2. It improves code maintainability as multiple if statements to handle different growing stages can be avoided since it might make the class hard to maintain. Instead, we can decide whether to grow to the next stage with only a single if condition by checking whether the number of ticks reached the ticks needed for different stages to grow into which are specified in each stage class. Each time a new state is introduced, it will implement the Inheritree class and will specify the next stage to grow into with the ticks needed for growing itself so we don't need to modify the if statement in the tick() method of the Inheritree class again. *(Open-closed Principle)* 3. Different types of fruit can be replaced with only Fruit for the return type of the growFruit() method in the Inheritree class without breaking the expected behaviour. *(Liskov Substitution Principle)* 4. Multiple instanceOf checks and if statements to handle different dropping probabilities can be avoided so that the Inheritree class will not depend on each and every fruit class, instead it only depends on a Fruit abstract class. *(Dependency Inversed Principle)* Hence, we can decide whether to add the fruit to the map with only a single if condition by calling the drop() method inherited by all the subclasses. When a new type of fruit is introduced, it will inherits from the Fruit class while having its droppingProbability and we don't need to modify the if statement in the tick() method of the Inheritree class again. *(Open-closed Principle)* 5. The tree and the fruit classes are created separately so that they are not forced to implement non-relevance methods (eg. the fruits should not implement methods to grow to the next stages). *(Interface Segregation Principle)*
Fruit (extends ConsumableItem)	An abstract class representing fruit produced by inheritree. Relationships: 1. It extends the ConsumableItem class which extends the Item class to achieve the portable functionality.	
SmallFruit (extends Fruit)	Class representing the small fruit produced by the inheritree at sapling state. Relationships: 1. It extends the Fruit class which extends the Fruit class to drop itself with a 20% probability.	
BigFruit (extends Fruit)	Class representing the small fruit produced by the inheritree at mature stage. 1. It extends the Fruit class which extends the Fruit class to drop itself with a 30% probability.	

		Limitations and Tradeoffs: 1. Introducing abstract classes and multiple stages and fruits classes may increase the complexity of the code as it adds layers of abstraction which might require additional time and effort for developers to locate specific functionality and navigate the codebase. 2. With multiple abstract and concrete classes for inheritree stages and fruit types, the class hierarchy may become deeper and more complex, making it harder to navigate and understand the relationships between classes.

Req 3 - The moon's (hostile) fauna



Req 2 UML Diagram

<table>
<tr><th>Classes Modified / Created</th><th>Roles and Responsibilities</th><th>Rationale</th></tr>
<tr><td>Player
(extends Actor)</td><td>Class representing the player.

Relationships:
1. It depends on the enum class Ability to indicate itself having the ability to enter the spaceship.
2. It depends on the enum class Status to indicate itself is hostile to the enemy.
3. It depends on the FancyMessage and BaseActorAttributes to display its health points and unconscious message.</td><td rowspan="2">Alternate Solution:
Overriding the canActorEnter() method by using instanceOf() to do multiple checks to ensure the instance of the actor is allowed to enter the spaceship.

Finalised Solution:
Add the enum Ability.ENTER_SPACESHIP in the Ability class as the qualifications for entering the spaceship. In the Floor class, canActorEnter() method is overridden to check the actors having the capability Ability.ENTER_SPACESHIP.

Reasons for Decision:
1. As there might be multiple actors allowed to enter the spaceship, modification on the canActorEnter() method each time a new actor is introduced can be prevented which reduces the risk of introducing new bugs and breaks the existing code. (Open-closed Principle)

2. Dependency of the Floor class on each actor class can be avoided to improve code maintainability. (Dependency Inversion Principle)

Limitations and Tradeoffs:
1. If new criteria or conditions are introduced, it may require significant modifications to the existing code which may lead to the code being harder to maintain.
2. Using the enum value to represent the qualifications for entering the spaceship may make the code less readable and understandable compared to explicit checks that clearly express the conditions for entry.</td></tr>
<tr><td>Floor
(extends Ground)</td><td>Class representing the floor of the spaceship.

Relationships:
1. It extends the Ground class to restrict the accessibility of actors onto it.
2. It depends on the enum class Ability to check the ability of the actor to enter the spaceship.</td></tr>
<tr><td>Crater
(extends Ground)</td><td>Class representing the crater that can spawn creatures.

Relationships:
1. It extends the Ground class to</td><td>Alternate Solution:
Implement the details of the crater directly into a generic ground class and directly spawn a new HuntsmanSpider in every tick in this class.</td></tr>
</table>

<table>
<tr>
<td></td>
<td>spawn a creature at every tick.
2. It depends on the Creature class to store a specific type of creature to be spawned.
3. It depends on the Exit to spawn the creature at the exit.</td>
<td rowspan="3">Finalised Solution:
Create a Crater concrete class which inherits the Ground abstract class. To implement the spawning feature for the crater, we introduce a Creature abstract class and the HuntsmanSpider class inherits from it. The Creature class has an abstract method spawn() to ensure that all the subclass of the Creature class (eg. HuntsmanSpider class) has the ability to spawn themselves.

The Creature class are designed as an abstract class instead of a concrete class to avoid instantiation as they are just blueprints which help in creating a consistent interface for all types of creature. Furthermore, interface is not considered for the Creature class since all types of creatures have the same attribute (eg., a list of behaviours) and implementation of methods (eg. playTurn() method). It is declared as an abstract class to enhance code reusability and avoid repeated code. (DRY Principle)

Reasons for Decision:
1. The details of the crater are implemented in its own classes instead of all being directly implemented inside a generic ground class since it might introduce complexity to the class by doing non-relevance responsibilities or even breaking the original code when more creatures are introduced. (Single Responsibility Principle)

2. As there might be multiple craters each spawning different creatures in the game, the crater has an attribute to store the type of creature it can spawn. With the introduction of the abstract class Creature, the subclass creatures can be replaced with Creature in the attribute of the Crater class for storing the type of creatures it can spawn without breaking the expected behaviour. (Liskov Substitution Principle)

3. The Crater calls the spawn() method in the specific creature class it will spawn to get the new creature object. By doing this, multiple instanceOf checks and if statements to handle different creatures spawning can be avoided so that the Crater class no longer required to depend on each and every creature class and</td>
</tr>
<tr>
<td>Creature
(extends Actor)</td>
<td>Class representing the creature that can be spawned by the crater.

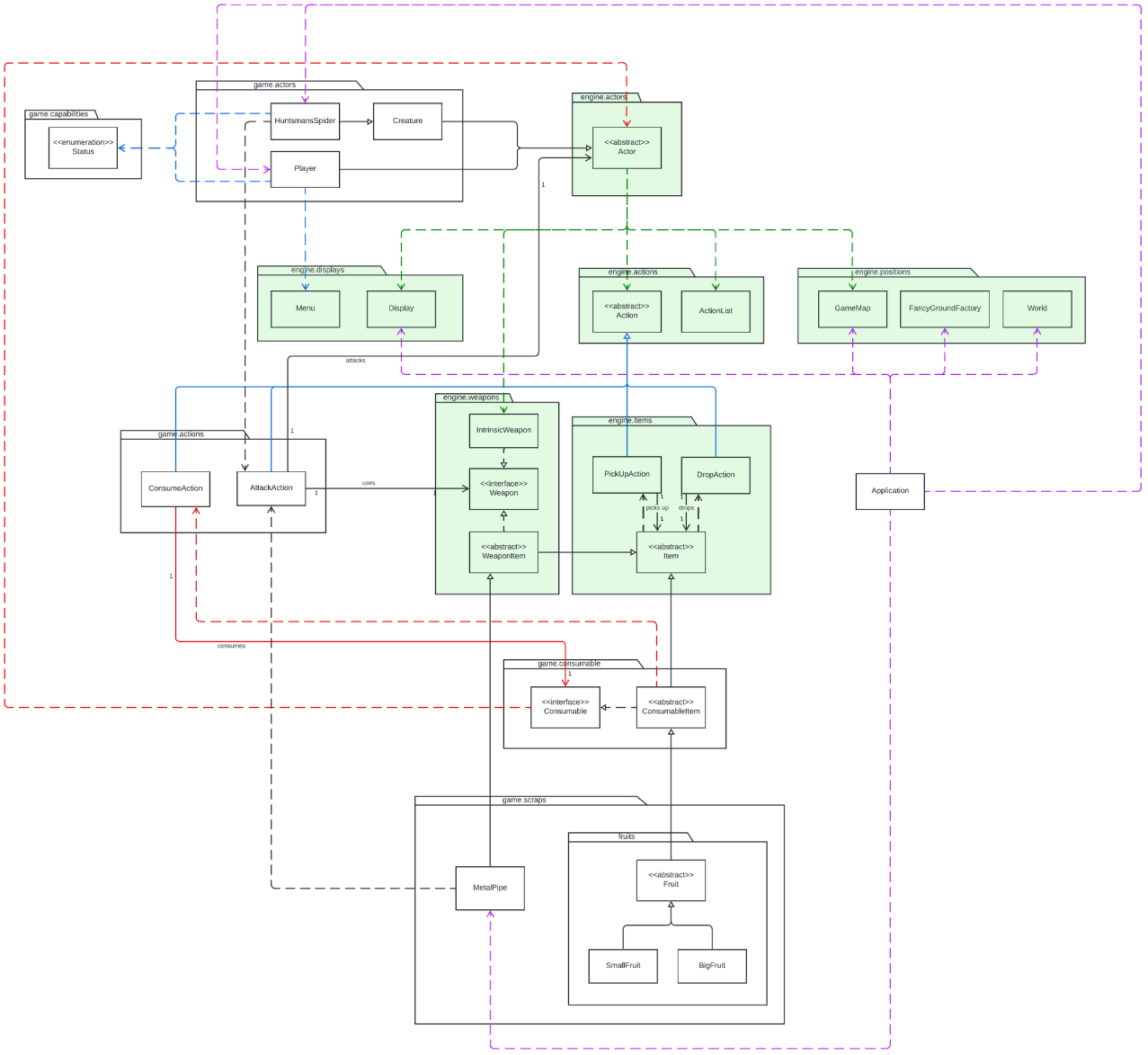
Relationships:
1. It extends the Actor class as it is a game entity with hitpoints that can perform an action in each turn.
2. It is associated with some Behaviours to perform actions.
3. It depends on DoNothingAction to do nothing if they have no special behaviour that can be performed.</td>
</tr>
<tr>
<td>HuntsmanSpider
(extends Creature)</td>
<td>Class representing the large and hostile huntsman spider.

Relationships:
1. It extends Creature class to be spawned by the crater.
2. It depends on AttackBehaviour class and WanderBehaviour class to perform an attack and wander around.
3. It depends on enum class Status to indicate itself is hostile to</td>
</tr>
</table>

	intern.	thus it is more maintainable as it is unaffected by the low-level module classes. *(Dependency Inversed Principle)* 4. When a new type of creature is introduced, it will implement the Creature class and we don't need to modify the if statement in the tick() method of the Crater class every time. *(Open-closed Principle)* 5. The crater and the creature are created separately so they are not forced to implement non-relevance methods (eg. the crater should not implement methods that perform actions). *(Interface Segregation Principle)* **Limitations and Tradeoffs:** 1. Introducing abstract classes and multiple creature classes may increase the complexity of the code as it adds layers of abstraction which might require additional time and effort for developers to locate specific functionality and navigate the codebase.
AttackBehaviour (extends Behaviour)	Class representing the attack behaviour of an actor which can generate an attack action. Relationships: 1. It extends Behaviour class to generate an action. 2. It depends on AttackAction class to generate an attack action. 3. It depends on Location and Exit to generate attack action on the location of the exit. 4. It depends on enum class Status to ensure that it cannot attack creatures that are hostile to the intern.	**Alternative Solution:** Combine all the behaviours of the huntsman spider (ie. attack and wander) in a single class by implementing the details of the attack behaviour in the same class with wander behaviour. **Finalised Solution:** Create a new class for AttackBehaviour which inherits from the abstract class Behaviour and all the details of the attack behaviour are implemented in this class. Instead of implementing the action logic in the same class, it depends on the AttackAction class to generate attack actions. **Reasons for Decision:** 1. Separating attack behaviour into its own class keeps each behaviour class focused on a single responsibility which allows easier extension and modification of attack-related functionality without affecting other behaviours. *(Single Responsibility Principle)*

		2. As the creature may have various behaviours, extending the Behaviour abstract class allows creating only a single TreeMap of type Behaviour to store all the behaviours the creature has. Thus, the HuntsmanSpider concrete class will not associated with each and every behaviour class which helps in code maintenance. *(Dependency Inversion Principle)* 3. The addition of new behaviour can be implemented through inheritance and polymorphism without altering the existing code which promotes code maintainability and reduces the risk of introducing bugs when extending the new behaviour. *(Open-closed Principle)* 4. The subclasses of the Behaviour class can be substituted for the base class without affecting the expected behaviours of the creature. This flexibility enables different types of behaviour to be performed interchangeably within the Creature class. *(Liskov Substitution Principle)* 5. The Behaviour class is split from the Action class to ensure the behaviour classes only implement the methods they need. *(Interface Segregation Principle)* **Limitations and Tradeoffs:** 1. Having various behaviour classes might increase dependency management overhead which might lead to challenges in tracking and managing dependencies and result in a fragile codebase.

Req 4 - Special scraps



Req 4 UML Diagram

Classes Modified / Created	Roles and Responsibilities	Rationale
Player (extends Actor)	Class representing the player (intern) in the game. Relationships: 1. It extends Actor class which has an IntrinsicWeapon to attack.	**Alternate Solution:** Implement the details of metal pipe in a generic special scraps class which is responsible for handling all special scrap types and this class inherits from the Item abstract class. To implement the attack function for the metal pipe, override the allowableActions() method in the Item class by checking each type of special scraps to execute its specific behaviour. **Finalised Solution:** Create a concrete class MetalPipe which inherits from the abstract class WeaponItem. By overriding the allowableActions() method, it can interact with the actor to attack using metal pipe. **Reasons for Decision:** 1. By creating a new class for MetalPipe and separating it from other special scrap and weapon classes, we can ensure that each special scrap class has only a single responsibility to handle a single type of special scrap. This improves the code maintainability and extendability as changes to metal pipe specific behaviour will not affect other types of special scraps and weapons. *(Single Responsibility Principle)* 2. Extending the abstract class WeaponItem allows for easy extension to add new weapon types without altering the existing code. Modification on the checking in the allowableActions() method is not required each time a new type of special scrap is introduced. This promotes code stability and reduces the risk of breaking the existing code due to frequent modification. *(Open-closed Principle)* 3. Implement a separate abstract class (ie. WeaponItem class) ensures that metal pipes and other special scraps are not forced to implement the methods they are not needed. For example, metal pipe should not have the capability to drop
MetalPipe (extends WeaponItem)	Class representing the metal pipe. Relationships: 1. It extends WeaponItem to achieve portable and weapon functionality. 2. It depends on AttackAction to generate an attack action using a metal pipe.	

		from the tree and fruit should not have the capability to be used as a weapon. *(Interface Segregation Principle)* 4. As there may be various types of weapon items in the future game, extending the WeaponItem abstract class allows us to create an attribute to store the weapon used in the AttackAction class by replacing the weapon item subclass type with type Weapon without breaking their expected behaviour. *(Liskov Substitution Principle)* Thus, the AttackAction concrete class will not associated with each and every weapon item class which facilitates code reuse and modularity. *(Dependency Inversion Principle)* **Limitations and Tradeoffs:** 1. Introducing multiple abstract classes for special scraps may increase the complexity of the code. While this design promotes flexibility, it also adds complexity to codebase navigation and makes it more difficult to understand the relationships between different classes. 2. Introducing a new class for MetalPipe might overengineering a simple item which might create unnecessary overhead in terms of memory usage and development effort if the MetalPipe class doesn't offer significant additional functionality compared to more generic approach.
Fruit (extends ConsumableItem)	An abstract class representing fruit produced by inheritree. Relationships: 1. It extends the ConsumableItem class to represent that it can be consumed by an actor.	**Alternate Solutions:** 1. In each class of the item that is consumable, override the allowableActions() in the Item class by adding the ConsumeAction into the ActionList. 2. Create a ConsumableItem abstract class and all the items that is consumable inherit from it. a. Implement the method consumedBy() in the abstract class to avoid repeated code in each consumable item class. b. Declare an abstract method consumedBy() in the abstract class for the details of the consume action. **Finalised Solution:**
SmallFruit (extends Fruit)	Class representing the small fruit produced by the inheritree at sapling state.	

	Relationships: 1. It extends the Fruit class to represent it can be consumed by an actor and heal the actor with 1 point.	Create an interface Consumable which defines the single method consumedBy() for the implementation of consuming. Then, create an abstract class ConsumableItem which inherits from Item class and all item classes that can be consumable extend it. Lastly, create a concrete class ConsumeAction which inherits from the Action class with an attribute to store the consumable and override the execute() method by invoking the consumedBy() method implemented by the consumable. **Reasons for Decision:** 1. By separating the consume action details from the Consumable class, we ensure that each class focus on a single responsibility which makes it easier to understand, extend and modify each class independently. *(Single Responsibility Principle)* 2. By providing abstract classes ConsumableItem, repeated code can be avoided by directly overriding the allowableActions() method to add the ConsumeAction so that we are not required to repeat overriding in the same way in every item class that is consumable. *(DRY Principle)* 3. As there may be things other than items that are consumable (eg. Ground), an interface Consumable is introduced for the promise of implementing the code logic of consuming instead of declaring an abstract method directly in the ConsumableItem class. Thus, the actual consumable subclass can be substituted with the Consumable type in the attribute of the ConsumeAction class without breaking their expected behaviour. *(Liskov Substitution Principle)* 4. Using the interface Consumable, multiple checks for the consumable type are not required as all the consumption details are implemented by the subclasses. Thus, the ConsumeAction class is no longer required to depend on every consumable subclass which decouples the ConsumeAction class from the details of specific consumable implementations. *(Dependency Inversion Principle)* When a new consumable is introduced with a different consumption effect, it only needs to invoke the consumedBy() method in the
BigFruit (extends Fruit)	Class representing the small fruit produced by the inheritree at mature stage. Relationships: 1. It extends the Fruit class to represent it can be consumed by an actor and heal the actor with 2 points.	
Consumable	An interface for something that can be consumed by an actor. Relationships: 1. It depends on Actor class to know which actor consumed the consumable.	
ConsumableItem (extends Item)	An abstract class representing item that can be consumed by an actor. Relationships: 1. It extends the Item class as it represents an item that can be consumed. 2. It depends on ConsumeAction class to generate a consume action from the consumable item	

	for an actor to consume it.	consumable class without any modification in the ConsumeAction class. This reduces the risk of introducing bugs. *(Open-closed Principle)* 5. The Consumable interface is small enough and specialized to have a single method for consuming details. Subclasses that implement it are not forced to implement methods they are not required to. This enhanced the clarity and maintainability of code compared to having a large interface for all special scraps (eg. including trading, attacking and consuming methods). *(Interface Segregation Principle)* **Limitations and Tradeoffs:** 1. Using an interface for Consumable instead of an abstract class might introduce duplication of code across multiple subclasses which violates DRY principle as there might exist similar implementations of consuming effect for several consumable substances. 2. With both abstract class and interface for consumable substances and items, the class hierarchy may become deeper and more complex, making it harder to navigate and understand the relationships between classes.
ConsumeAction (extends Action)	Class representing the consume action. Relationships: 1. It extends the Action class to perform an action. 2. It is associated with the Consumable interface to perform the consume action on the consumable substance.	