Design rationale:

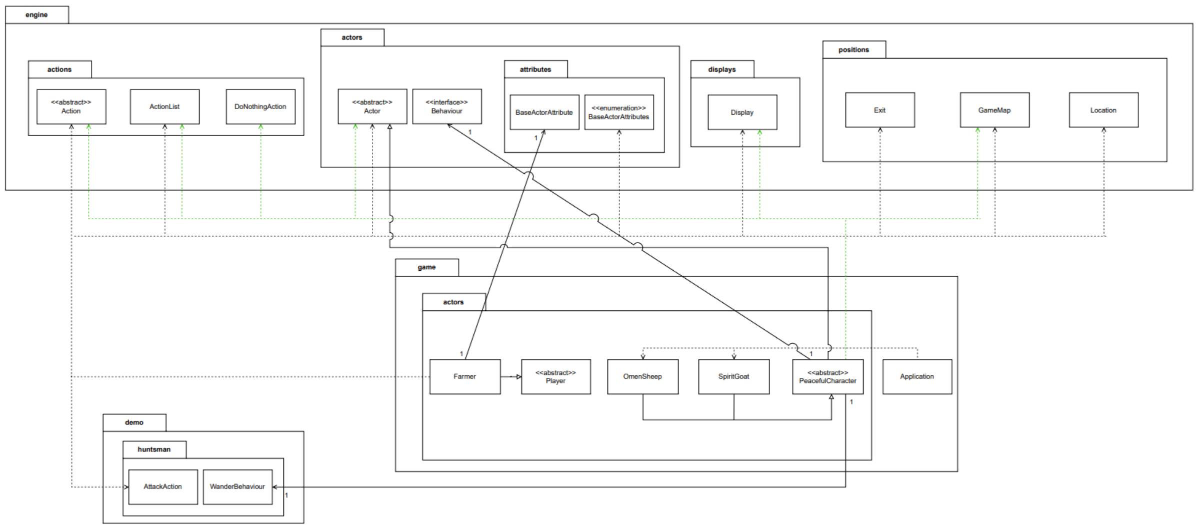
<u>REQ 1:</u>

Design 1: Hardcode each entity to have the wander behaviour

Pros	Cons
Allows for flexibility as someone can simply change the implementation at runtime	Code duplication where each entity would have to have a new behaviour implemented
Simple to implement as it does not require inheritance of external classes	Violation to the open/closed principle as adding new behaviour requires modifying each class
	Not flexible in terms of scalability as more classes would mean that more updates would need to be made to each class

Design 2: Create an abstract class Peaceful Character to implement the shared traits across Spirit Goat and Omen Sheep, Hostile Character for further extension

Pros	Cons
The abstract classes can handle logic that its children classes share	As further extensions take place abstract classes may become "God Classes" and take on too much responsibility
Allows for flexibility as classes can be further extended by modifying the super class	



The diagram above shows the implementation of requirement 1 using design 2, in which abstract classes are used to create logic that its child classes can inherit. This class encapsulates methods such as the behaviours that are taken every turn and the shared string methods, allowing for the adherence to DRY principle as the code will not have to be repeated. This design also aligns with the LSP as both the Spirit Goat and Omen Sheep classes can replace the Peaceful Character class without changing the expected behaviour considering that they are mostly the same. The Farmer class is assumed to be the main player class that the user is taking and therefore the rest of the requirement is coded into the Farmer class. Although the requirement did not specify this, a Hostile Character class was implemented to future-proof the design and enable straightforward extension for hostile creatures. In terms of the Dependency Inversion Principle, the Application depends on the high-level actors while Peaceful Character handles the low-level logic. Furthermore, this approach allows for the Open/Closed principle as new classes of similar type can be added without modifying existing code. This design thus promotes maintainability and scalability of the code where future updates require minimal code changes.

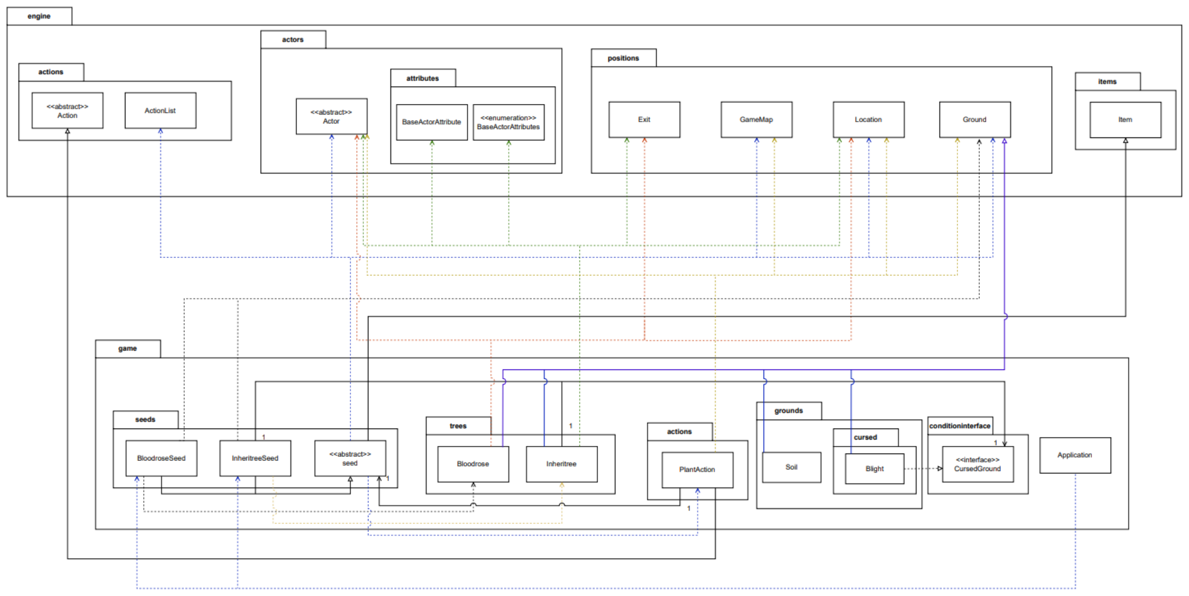
<u>Req 2:</u>

Design 1: Implement the seed type with a type field and if statements to determine if the seed type is either an inheritree seed or a bloodrose seed

Pros	Cons
Easy to implement with a small number of seed types to plant	In terms of scalability this would not hold as the creation of more seeds would force existing code to be modified and increases the chance of bugs
Reduces the number of potential classes that would need to be implemented	In terms of coupling the code is tightly coupled which makes logic harder to be re used
Easy to debug and create modifications as the logic is in one class	

Design 2: Implementing a polymorphic seed hierarchy with abstract classes

Pros	Cons
Each class is responsible for their own logic (PlantAction, the seed types)	The multiple layers of seed types add a layer of complexity where future unique planting logic
New seeds can be added without modifying existing code	
Classes like PlantAction work for a lot of other tasks	



The above diagram is an implementation of requirement 2 which uses design 2.

Both the InheritreeSeed and the BloodroseSeed extend the abstract Seed class which inherits from Item. Through this, the Don't Repeat Yourself principle can be met in which there is shared logic in the seed super class. Override methods like createPlant() and getStaminaCost() can define specific behaviours for better adherence to LSP. The Application class indirectly depends on the abstract seed type by reducing the coupling between high-level models and low-level models. The use of the PlantAction class to encapsulate the planting actions means an improvement in terms of scalability as new seed types can be added without modifying existing code, which also decouples from Farmer. Through the design choices, polymorphism has also been implemented to future-proof the design. For example, InheritreeSeed accepts a type or cursed ground that it can cure such that many unique types of Inheritree variants can be made. This will allow for integration with a variety of cursed grounds without hardcoding dependencies. The seeds were also designed to ensure that each of their effects are self-contained such that the execution was delegated to the tree instances via polymorphism, further aligning with the Open/Closed Principle. This abstraction also allows for future trees to define complex interactions which allows for scalability.

<u>Req 3:</u>

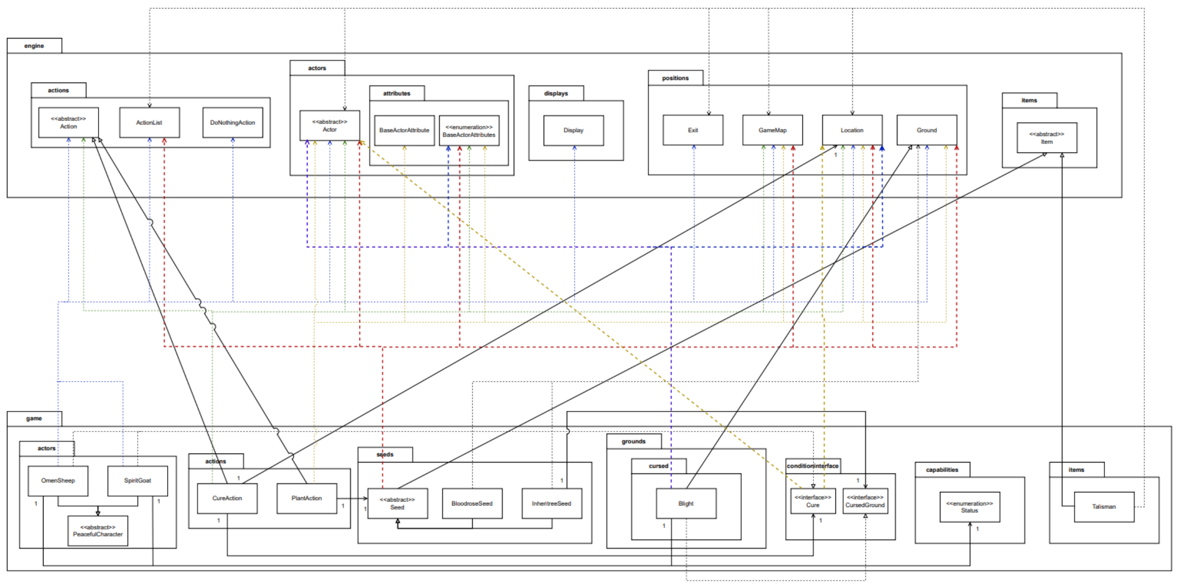
Design 1: Modify the Talisman class directly to implement a centralised curing logic

Pros	Cons
Easier to implement the Talisman class with centralised logic and there is no need for additional abstract classes or interfaces	Scalability issues such that future cure methods would have to be implemented through the direct modification of the Talisman class

Easy to maintain the class for a small amount of cure logic	The Talisman would become tightly coupled to entity classes
	Having all the cure logic in Talisman would mean that there would be conditional logic to handle different targets and undermine OOP

Design 2: Implement a CureAction to handle curing in which classes that can be cured provide their own method of getting cured

Pros	Cons
Future curable entities can be added without modifying existing code as much	Potential misconfiguration as the class might not properly add a capability and lead to silent failures
Decoupled architecture as the Talisman delegates curing logic to targets which removes their dependancy	Diagnosing issues within the classes may lead to navigating through multiple classes in order to find out the problem
Scalability can be better achieved as entities can define their own unique curing logic which allows for better gameplay in the future	



The diagram above shows the final design of requirement 3, which utilises design 2 to implement the cure interface which is implemented by the OmenSheep, SpiritGoat, and Blight in which entities that can be cured will implement this interface. This design ensures that the game's lore driven mechanics are maintainable and scalable. The cure interface mandates two methods that any entity that is curable needs, a method to retrieve the stamina cost of curing the entity and how the entity isi cured. This allows for classes implementing the interface such as OmenSheep, SpiritGoat and Blight to have their own encapsulation of their own curing logic, strongly adhering to the Single Responsibility Principle as behaviour specific responsibilities are isolated to their individual class. The Talisman is the central class for initiating the cure action. In order to avoid hardcoding different behaviours for different entities, the Talisman implements logical checks to identify actors in its surroundings that have the CURABLE status and the ground as well, this lack of hardcoding means that the Open/Closed Principle is better adhered to as new entities that are curable can simply be added by implementing the interface and registering a CURABLE capability. In this design, the use of instanceOf was avoided through the implementation of CURABLE traits eliminating the need for type checking logic. Additionally, the Talisman adheres to the Single Responsibility Principle as encapsulating the logic of classes means that Talisman is unaware of SpiritGoat and Blight, and only invokes the method of them, meaning that no single class becomes overly complex. Through the implementation of these principles the game is more future-proofed and allows for scalability as more curable entities can be introduced without modifying existing ones. This approach overall establishes a foundation for development in the future with the goal of maximising code use, combining polymorphism, interface delegation, all while prioritising gameplay.