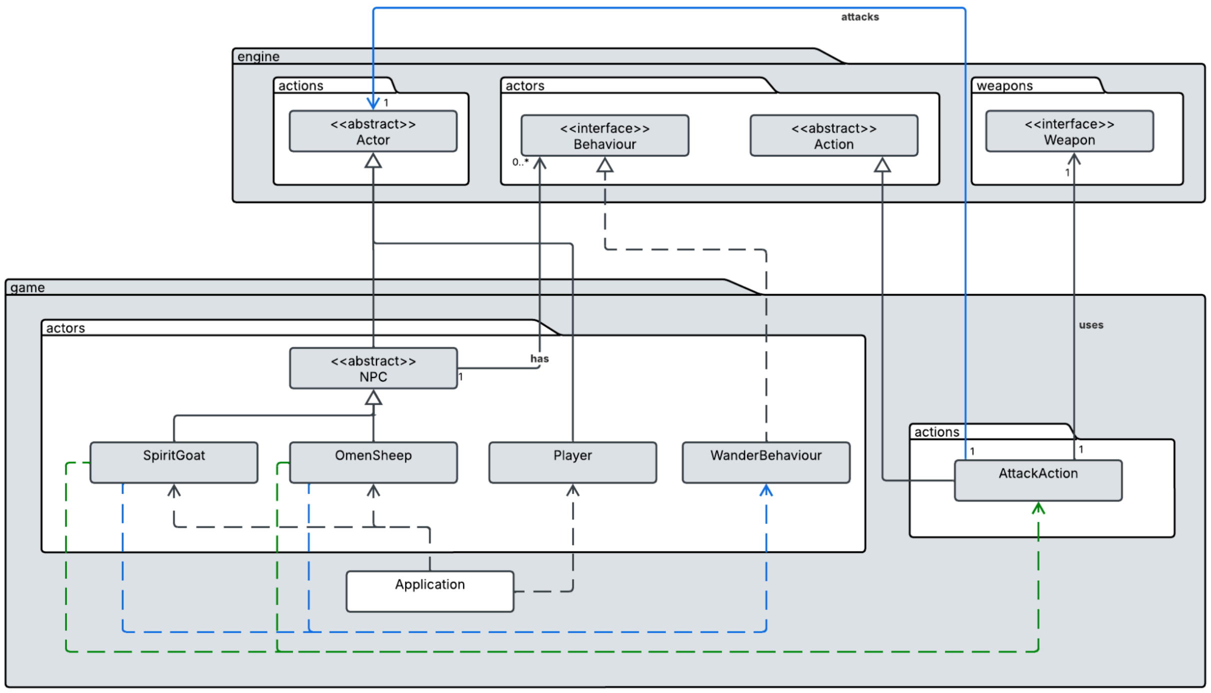
Design Rationale for Requirement 1: The Graceless Farmer (640 words)

Design 1: Hardcode Player(Farmer) and basic creatures separately with manual actions and checks. This means all classes are coded as concrete classes without extending from the abstract actor class. No common abstraction between the classes. Because the creatures do not inherit or have any common abstraction, the attacking of the farmer would be hard coded within the player, basic creatures, and main game loop.

Pros	Cons
Simple and fast to implement	Violates (OCP - Open-Closed Principle), adding new creatures requires modifying the main game loop and having exponentially growing conditions.
Easy to understand initially, as each creature is handled individually	Breaks (SRP - Single Responsibility Principle), because the main game logic would have several responsibilities and conditions for each individual creature behaviour and future interactions.
Minimal upfront design and work	Breaks (DRY - Don't Repeat Yourself), no abstraction means that each creature would have repeated logic for similar behaviours and hostility.
	Not scalable. Adding even a few more new behaviours or creatures would make the maintenance increase exponentially.

Design 1: Introduce a common abstraction from Actor and NPC (non-playable characters) which would handle the behaviour system. Creates a new abstract class NPC which inherits Actor. Actor class has shared fields like hitpoints, whereas the NPC class would have shared fields like behaviours and turn playing. We also introduce a behaviour system (interface) that represents a behaviour an Actor can perform at each game tick, like WanderBehaviour. Each NPC would hold a list of behaviours and allowable actions.

Pros	Cons
OCP - Open-Closed Principle, Adding new NPC actors or Behaviours and actions does not require modifying existing classes.	More upfront work by having to create more abstract classes and interfaces.
SRP - Single Responsibility Principle, Actors are responsible for managing themselves, and the behaviour and action systems are responsible for what the actors do. Each class has one clear purpose and does not overlap on responsibilities.	Seems “over-engineered” initially, as the game is still at a simple state. May be redundant if no future extended modifications are made.
Extensible, Easily able to add new types of creatures and NPCs with different behaviours.	
DIP - Dependency Inversion Principle, Game loop depends on the Actor and NPC abstraction, not concrete classes.	
DRY - Don’t Repeat Yourself, Common logic like wandering and attacking is implemented once and reused.	



The diagram above is the final design used in requirement 1. This design uses Design 2: having a common abstraction and separate classes to manage actions and behaviours.

The actor is an abstract class with shared attributes which is inherited by the NPC abstract class. This was done because actors do not have behaviours, but NPCs would have behaviours since they cannot be controlled by a player. However, it has an abstraction from Actor because the creatures, goat and sheep both have similar attributes, which are health points.

At the moment, we do not have NPC hold WanderBehaviour nor AttackAction, as it is stated that not all NPCs share the same behaviours and actions. This means that although the SpiritGoat and OmenSheep have similar behaviours, future implementations of NPCs such as other players or hostile attackers may not share the same behaviours and actions.

Another thing to note is each behaviour is its own class which inherits from the abstract Behaviour interface. This ensures that each class is responsible for implementing a single type of behaviour and can each be added to NPCs or its child classes more extensively. Same goes for the AttackAction class which inherits from abstract Action class.

By designing it this way, we ensure that adding new behaviours simply involves creating new behaviour classes and not affect the previously implemented behaviours. We also ensure that creating new NPCs would not involve modifying existing logic.

This structure follows SOLID principles tightly and supports long-term scalability and maintainability compared to Design 1.

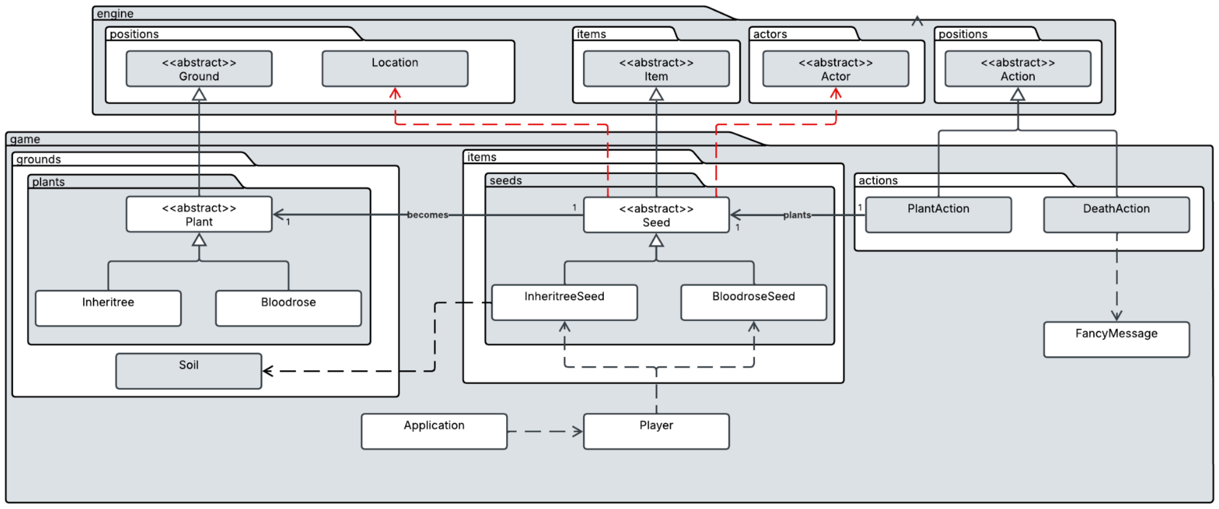
Design Rationale for Requirement 2: The Valley of Inheritree (643 words)

Design 1: Directly extend Ground for each plant without abstraction. Create separate classes, Inheritree and Bloodrose, both directly extending Ground. Each class handles logic and effects individually. They also handle death manually, where actors have to manually check if they are unconscious every tick and print "YOU DIED" when necessary. Seed classes directly modify ground and actor attributes during planting without an action.

Pros	Cons
Very simple and quick to implement.	Violates (OCP - Open-Closed Principle), adding a new plant requires modification of duplicated planting logic and effects.
Easy to understand initially, with a small number of plants and actions.	Breaks (SRP - Single Responsibility Principle), Inheritree and Bloodrose manage both their ground representation and their special behaviours(heal/damage), making them have multiple responsibilities.
Minimal class structure required initially.	Breaks (DRY - Don't Repeat Yourself), Same patterns of checking each surrounding tiles for all plants are duplicated and repeated, making lots of redundancy.
	Violates (DIP - Dependency Inversion Principle), manual death handling and high level game loop termination are being handled by the low level plants without an appropriate action causing inconsistency.
	Low scalability, it is difficult to expand on more complex plants and effects or when new cursed grounds are introduced.

Design 2: Create an abstract class Plant that extends from Ground. Plant defines a general method for applying effects to nearby actors each turn, due to the plants' common functionality. Inheritree and Bloodrose both extend Plant and only override their specific applyEffect() methods where necessary. This design also implements a DeathAction and PlantAction class to consistently handle actor death and planting. Seed items (InheritreeSeed and BloodroseSeed) still handle partial planting logic but rely more on the PlantAction class.

Pros	Cons
Follows OCP, new plants can be added easily by subclassing Plant and overriding the applyEffect() method, without touching existing code.	More upfront design and requires more effort and time to implement.
Adheres to SRP, each class has one focused purpose. Plant manages continuous plant effects, DeathAction manages Death, and PlantAction manages initial planting effects.	More classes that are created initially although only 2 plants exist.
High Cohesion, related logic like healing and damaging is grouped together inside the Plant hierarchy.	Redundant if not scaled and extended.
Follows DIP, other systems like other actors depend on the Plant abstraction and not concrete Inheritree and Bloodrose classes.	
DRY, the reuse of surrounding tile logic check and the actor effects, avoids code duplication.	
Extensibility, easily accommodates new cursed grounds, plants and other seeds with different effects.	



The diagram above is the final design used in requirement 2. This design uses Design 2: having a common Plant abstraction combined with DeathAction and PlantAction classes.

Plant is an abstract class extending from Ground, providing a structure for plants that affect nearby actors each turn or tick. The Inheritree and Bloodrose both extend Plant, overriding their effect applying methods. Inheritree heals nearby actors whereas Bloodrose damages them.

On the other hand, Seed classes (InheritreeSeed and BloodroseSeed) handle initial planting with the help of the PlantAction. The seeds are plantable objects. This PlantAction helps remove the responsibility of removing the seed from the actor and menu descriptions, allowing for the seed to only have 1 responsibility, which are the effects of being planted.

Furthermore, death is now being handled uniformly through DeathAction, instead of having its logic spread out amongst every relevant class. This way, any actor can execute this action which calls unconscious, and depending on if it is a player, will terminate the main game loop, printing “YOU DIED”.

By designing it this way, new plant types can be added in by simply creating a new subclass of Plant. New seeds can be added without any modifications to its existing counterparts through the help of PlantAction. The system remains clean, scalable, and adheres to the SOLID principles for long-term maintainability.

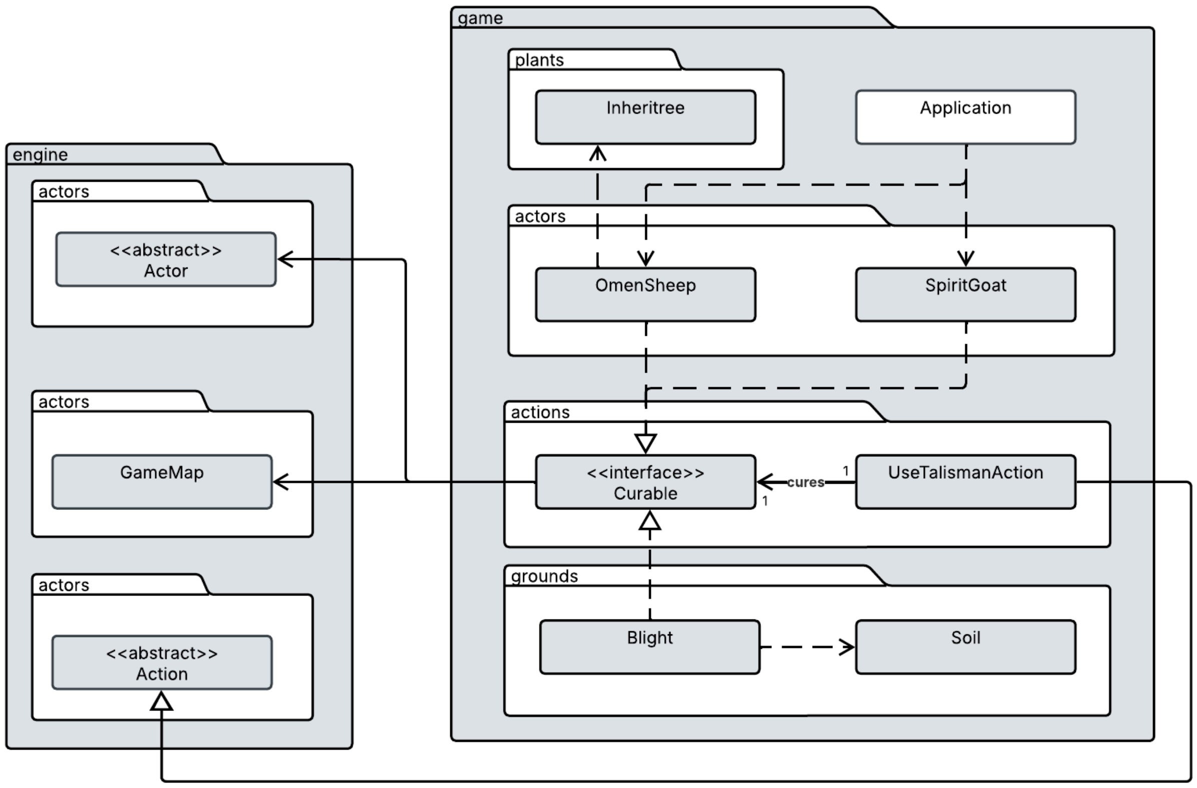
Design Rationale for Requirement 3: Crimson Rot (590 Words)

Design 1: Hardcoding the cure logic directly inside the UseTalismanAction class. This design implements UseTalismanClass to check explicitly the type of target object, for example, the instance of Blight, SpiritGoat, or OmenSheep and apply the correct curing behaviour manually all in the action class. This means that action would manually reset the spirit goat's timer, plant inheritree around omen sheep, or change blight to soil ground.

Pros	Cons
Simple to implement in the beginning	Violates (OCP - Open-Closed Principle), everytime a new curable entity is added, UseTalismanAction would need to be modified, making it difficult to maintain.
No need to create extra interfaces or classes	Breaks (SRP - Single Responsibility Principle), UseTalismanAction would be doing way more than a single responsibility, as it would not only have to decide which object to cure, but also how to cure it.
Easy to hardcode specific behaviours and actions for known creatures or grounds.	Hard to maintain, as the number of curable types grows, UseTalismanAction would start to have a lot of if-else conditional statements.
	Poor Scalability, if we were to extend the rot feature or add new talisman effects and uses, it would require invasive changes.
	Tight Coupling, UseTalismanAction must know about every single possible creature, ground, or future additions that it can cure.

Design 2: Introducing a Curable interface to delegate the curing logic within UseTalismanAction. This is done by creating a Curable interface with a cure() method. All entities which can be cured via UseTalismanAction or even other types of future items will implement this Curable interface and define their specific curing behaviour and logic inside the overwritten logic. UseTalismanAction only checks if the target is an instance of Curable, and if so, simply executes the cure function based on its target's implemented behaviour, reducing case handling.

Pros	Cons
OCP - Open-Closed Principle, Adding new curable entities can be added without modifying UseTalismanAction. Just implement the Curable interface within the new entity.	More initial setup required for creating an interface and have all curable entities implement it.
SRP - Single Responsibility Principle, UseTalsimanAction now only initiates the curing process, while each entity knows how it should be cured and its respective behaviours.	Requires more extensive design and time if more different types of cure actions appear in the future implementations.
Low Coupling, UseTalismanAction does not need to know about the specific types of creatures or grounds.	
High Cohesion, each class handles its own curing behaviour in its respective cure methods.	
Extensible, new curing rules and logic are easier to add without touching existing action logic.	



The diagram above is the final design used in requirement 3. This design uses Design 2: introducing a Curable interface and a UseTalismanAction that delegates the curing process.

Curable is an interface that has an abstract cure() method. Each of the implementations implement Curable and define their own specific curable logic in its own ways. SpiritGoat resetting its rot timer, OmenSheep growing new Inheritrees, and Blight being removed in place of Soil.

UseTalismanAction when executed initializes with a target that is Curable and if so, calls the target's cure() method. This reduces the amount of responsibility needed within the single UseTalismanAction.

By having this design, the new creatures or grounds which choose to implement cure differently are able to implement Curable interface and override the cure() method accordingly. New curable entities in future requirements would not change any existing logic but simply introduce a new implementation. This creates a clean system which remains modular and scalable, aligning with the SOLID design principles.