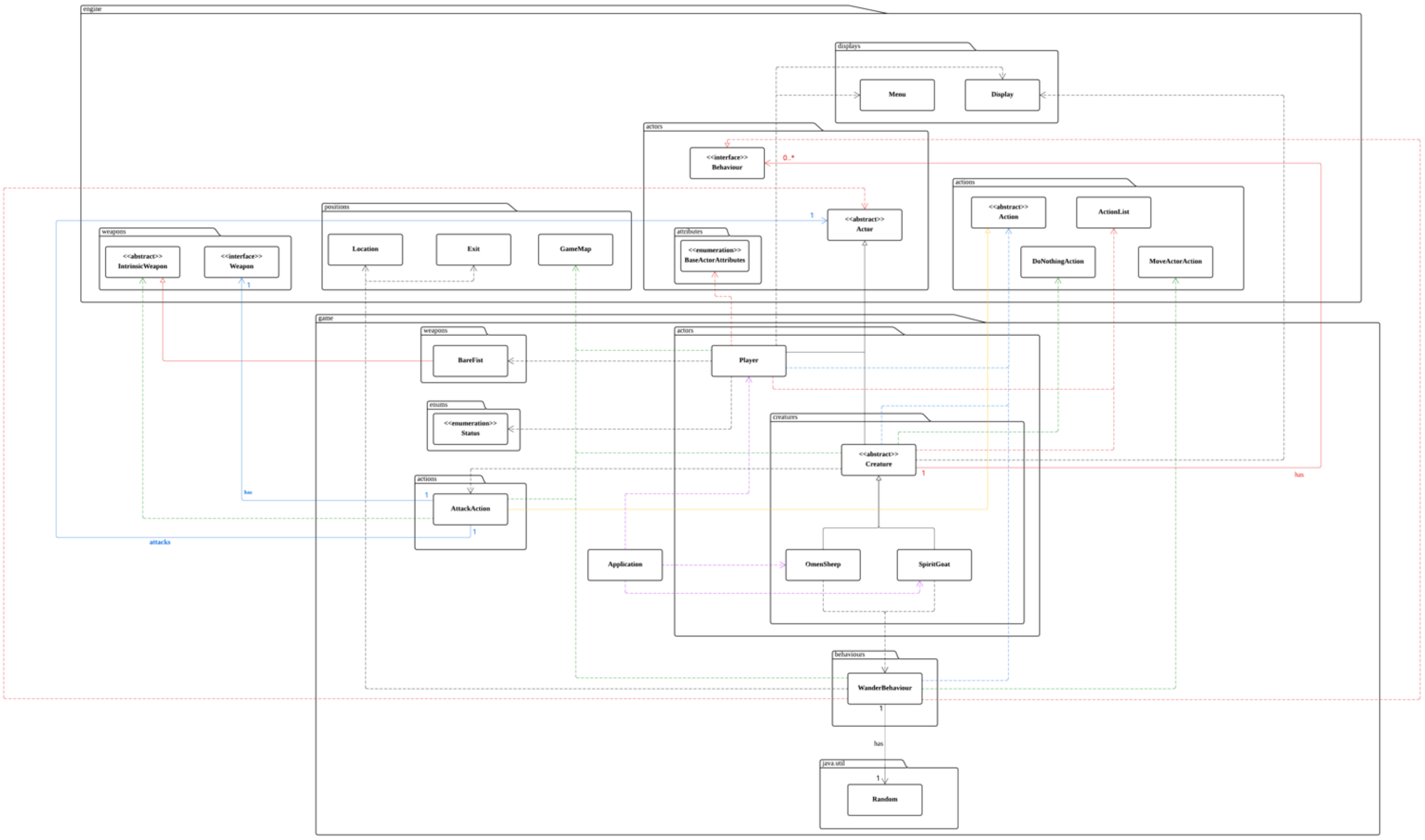
FIT2099 Assignment 1 Design Documentation

Design Diagrams

REQ 1 UML Class Diagram:



Design Rationale

REQ 1:

1. Implement the Player class to represent the farmer by extending from the engine Actor abstract class.

Pros	Cons
It promotes the code reusability as inheriting from abstract Actor class allows the Player class to reuse core functionality. This adhered to the DRY principle.	Extending from the engine's abstract Actor class might introduce tight coupling between Actor and Player class. If the engine's Actor class is modified, Player class might need to change too.
Player class shares common attributes and method with other actors in this	

game. Therefore, by inheriting from Actor class, Player class is grouped with similar responsibilities, promoting high cohesion	

2. Create concrete classes that represents the creatures SpiritGoat and OmenSheep by extending from a newly created abstract Creature class. The abstract Creature class then extended from abstract Actor class.

Pros	Cons
It promotes the code reusability as there are similarities between specific creature classes. Creature class allows specific creature classes to reuse their similar functionality. This adhered to the DRY principle	Introducing the extra abstraction layer (Creature class) increases the depth of the inheritance tree, which might make the debugging more difficult.
It improves cohesion because the specific creature classes have similar function and is grouped in Creature hierarchy, which keeps the system modular and easier to maintain.	Extending from an abstract Creature class might introduce tight coupling between Creature and specific creature classes as modification in Creature class might result in modification in its subclasses.
It enhances the scalability and extensibility as if there are new creatures exists, the new creature can just extend from the abstract Creature class, inheriting the similar functionality. This adhered to the Open Closed Principle	
SpiritGoat and OmenSheep can be used in place of the abstract Creature class since they preserve the same attributes and methods of abstract Creature class, enabling interchangeable use without breaking program correctness. This design adhered to Liskov Substitution Principle.	

3. Add status HOSTILE_TO_PLAYER as there might have creatures that are hostile to player in the future

Pros	Cons
It adds a simple way to mark which creatures are enemies of the player. When there are other new creatures that are hostile towards player, we can simply add capability without	If different type of hostility needed to be handle in unique way, a single status HOSTILE_TO_PLAYER might be too simplistic.

modifying the existing class which adhered to the Open Closed Principle	
It encourages loose coupling, because the system can check the capability of the creature instance instead of checking the specific class type (instanceof). This avoids the tight coupling between the game logic and specific creature classes.	

4. Created a concrete WanderBehaviour class that implements the Behaviour interface. WanderBehaviour is added in specific creature class but not the base creature class because some creature might not able to wander around.

Pros	Cons
Implementing Behaviour interface provides a clear contract of methods that the WanderBehaviour should implement.	We might need to explicitly assign behaviours to each individual creature which may be not efficient when creating many creatures.
It promotes high cohesion as only creatures should wander have the wander behaviour.	
It is easier to extend when there are new behaviour exists. This adhered to the Open Closed Principle	
The WanderBehaviour class only responsible for one responsibility which is to get the wander action from a creature. This design is adhered to Single Responsibility Principle	

5. Use TreeMap to store the behaviour of creatures instead of HashMap

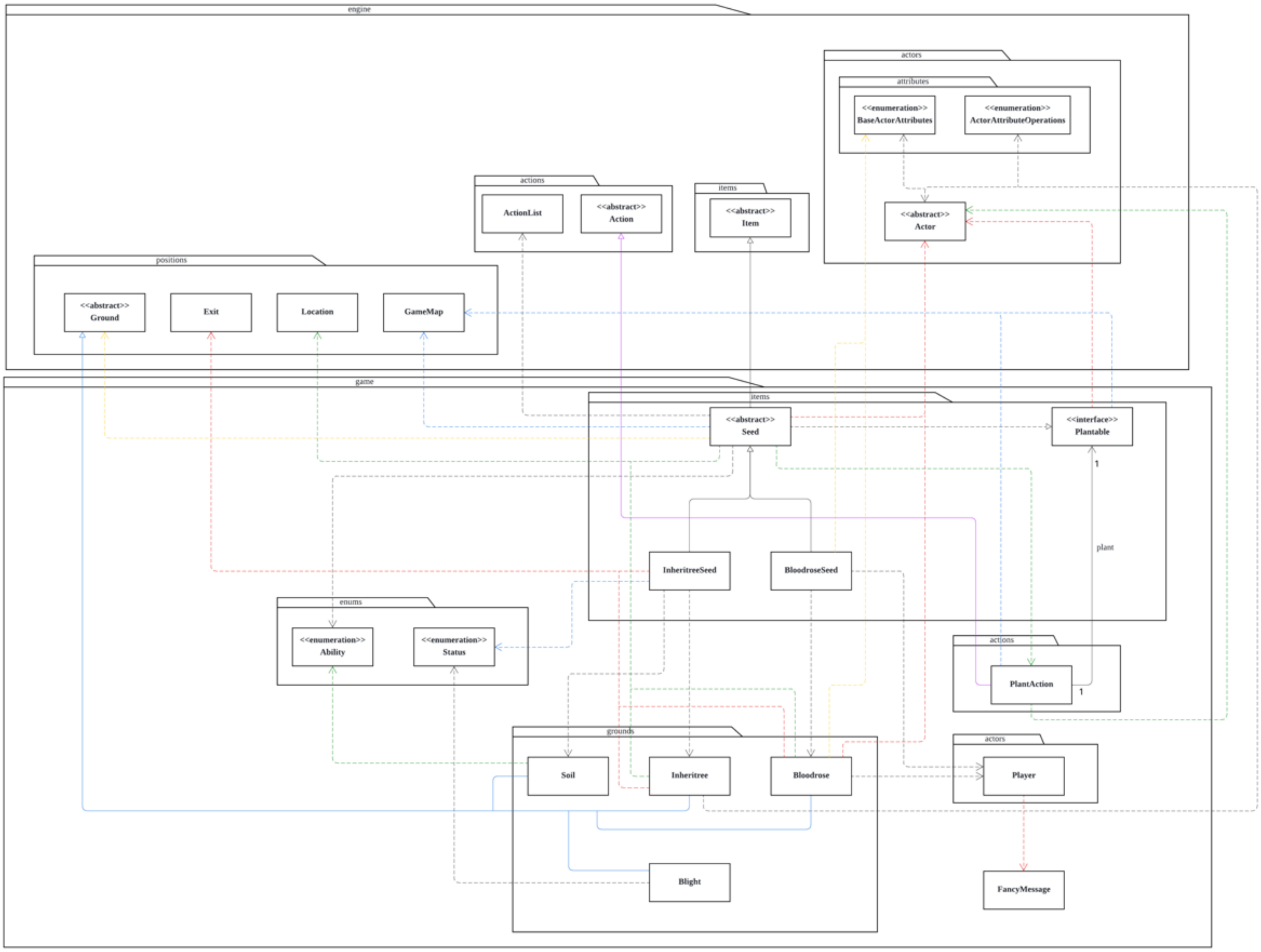
Pros	Cons
TreeMap stores the key in sorted order ensure that high priority behaviour is always execute first.	TreeMap is slightly slower than HashMap and potentially need more memory to store.
TreeMap simplify the logic as it avoids the needs to manually sort the behaviour based on its priority.	
When there are more behaviours in the future, we can add the behaviours into the TreeMap based on its priority which adhered to the Open Closed Principle	

6. Created concrete AttackAction class to represent an attack action which extends from abstract Action class

Pros	Cons
The AttackAction class is only responsible for one responsibility which is executing the attack action on other actor which adhered to the Single Responsibility Principle	AttackAction class is tightly coupled with Weapon and Actor class which a modification on Weapon and Actor class might need to update the AttackAction class.
The AttackAction class can be reused whenever an actor can be attacked without creating multiple methods for each specific case. This improves the code reusability and is adhered to DRY principle.	
This design allows introducing new actions in the future without modifying the existing code which makes the system more maintainable. This design is adhered to Open Closed Principle.	

Design Diagram

REQ 2 UML Class Diagram



Design Rationale

REQ 2:

1. Created concrete Inheritree class to represent an inheritree and concrete Bloodrose class to represent a bloodrose. Both classes extend from abstract Ground class.

Pros	Cons
Both classes extend from abstract Ground class and can reuse core attribute and method from engine. This design adhered to the DRY principle.	Extending from Ground class might introduce tight coupling between the abstract Ground class the its subclasses because a change in Ground class might result in change of all subclasses
Each class only focus on their own responsibility which representing a crop and it is easier to maintain. This design is adhered to the Single Responsibility Principle.	
More ground types can be easier to be added in the future by just extending from the abstract Ground class in which adhered to the Open Closed Principle.	
Both seed subclasses use the superclass constructor to initialize with different values, allowing them to be used interchangeably as Seed without affecting program correctness, which adhered to the Liskov Substitution Principle.	

2. Created concrete InheritreeSeed class to represent an inheritree seed and concrete BloodroseSeed class to represent a bloodrose seed. Both classes extend from a newly created abstract Seed class.

Pros	Cons
It promotes code reusability because there are many similarities between the seed subclasses which the same thing can be placed inside the abstract Seed class. This design is adhered to the DRY principle.	Introducing the abstract Seed class adds another layer of abstraction, which can make the system more complex and harder to understand.
Each concrete seed only responsible on one responsibility which is each seed is only responsible for a specific tree. This design is adhered to the Single Responsibility Principle.	

If more seed types are added, they can simply extend from abstract Seed class, making the system easier to extend when there exists a new feature without modifying the existing code, in which the design adhered to the Open Closed Principle	

3. Created a new Plantable interface, which is implemented by the abstract Seed class to represent the item that can be planted.

Pros	Cons
Easier to extend if there are new plantable item exists. The new plantable item just need to implement the Plantable interface without modifying the existing code. This design is adhered to Open Closed Principle.	The use of Plantable interface could introduce unnecessary complexity for simple system, especially when there are few objects that can be planted.
The Plantable interface defines a clear and concise contract for object that can be planted and it does not force any unrelated methods upon the implementing class which adhered to the Interface Segregation Principle.	If there are more feature added to the plant functionality, the interface may need to be modified which could break the Interface Segregation Principle.
The use of the Plantable interface promote loose coupling, making the system more modularised and easier to maintain.	

4. Created a concrete PlantAction class which extends from abstract Action class to represent a plant action.

Pros	Cons
The PlantAction class is only responsible for executing the plant action. In other word, the class only has a single responsibility which adhered to the Single Responsibility Principle.	The PlantAction class has a tight coupling to the Plantable interface. If the way of planting is different, we may need to update the implementation of Plantable and PlantAction for different cases.
The PlantAction class can be reused whenever there is an object that can be planted. This improves the reusability of the code which adhered to the DRY principle.	

5. Added a new ability PLANTABLE to represent a ground that can be planted.

Pros	Cons
It improves modularity because it decouples the logic from specific ground type and allow multiple ground types to be plantable. Also, it avoids the usage of instanceof which is a bad practice in OOP.	If only one ground is plantable, using a capability might overkill.
We can simply assign this capability to the ground that are plantable in the future without modifying the existing code. This design is adhered to the Open Closed Principle	

6. Implementing the effect of plants when the seed of the plants is planted and also the blooming effect.
   - Inheritree will convert surrounding blight to soil. Therefore, add a status CURSED for the grounds that are cursed and can be cured by inheritree.

Pros	Cons
It encourages loose coupling as the inheritree does not need to check the ground type using instanceof, it only checks for the CURSED capability, which avoided tight coupling to concrete classes.	It requires consistency as developer must remember to add the CURSED status to all cursed ground otherwise may lead to unexpected behaviour
Inheritree can cured ground with CURSED status without modifying the logic when new cursed ground is introduced. This design is adhered to the Open Closed Principle.	

   - Inheritree will convert surrounding blight to soil and bloodrose will sap Farmer's health by 5 points when the seeds are planted. This is done by implementing these logics in the plant() method under both crop's seed classes. The blooming effect of each tree is then implemented in their own tree class's tick() method.

Pros	Cons
Each class only handle its own logic: the seed class handling one-time effect and the crop class handling the blooming effect which adhered to the Single Responsibility Principle.	This might introduce a tight coupling between seed and ground logic.
If there are other plants that could do something different when its	

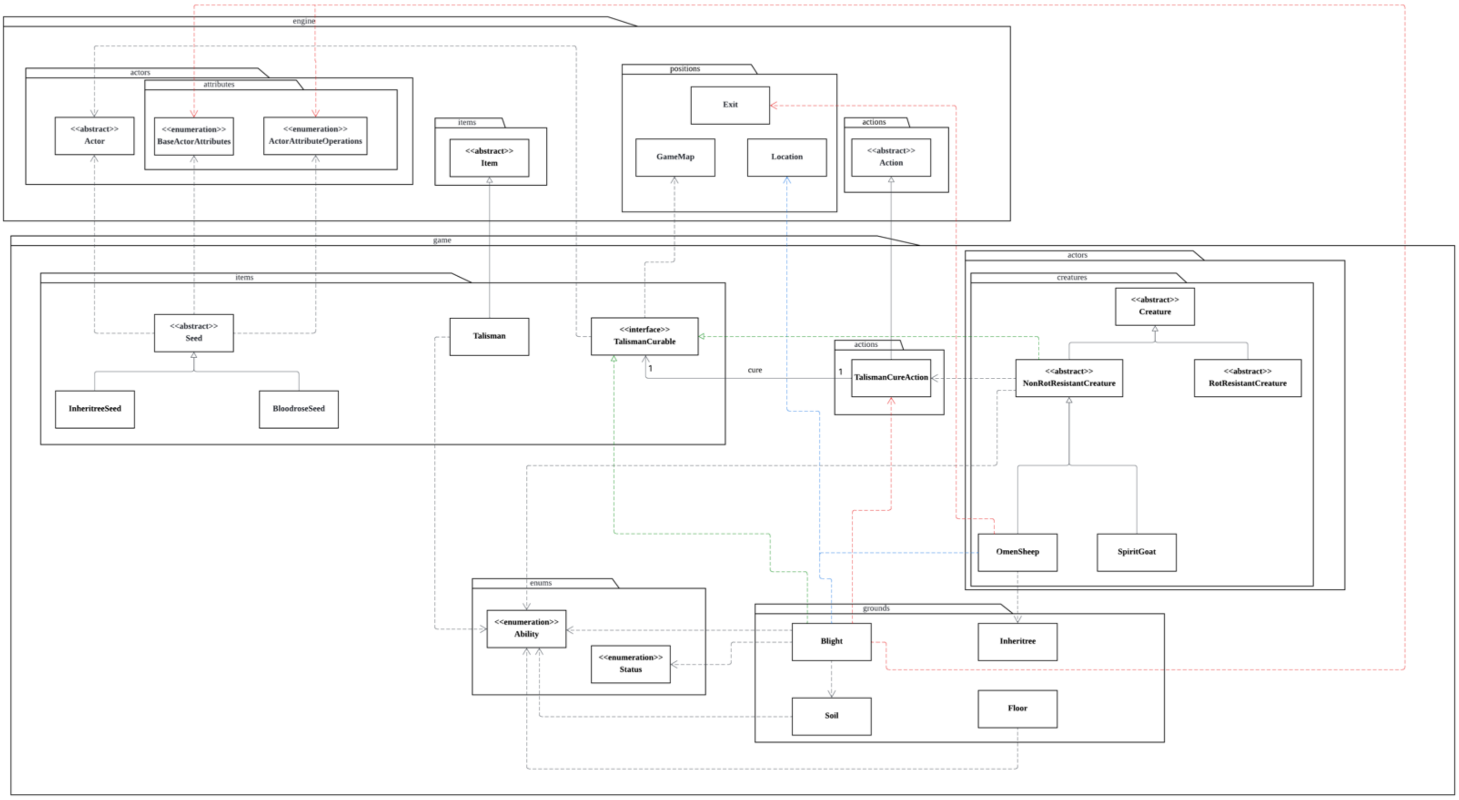
seed is planted, we can just extend from the abstract Seed class and extend from the abstract Ground class and implement the new effect. This adhered to the Open Closed Principle.	

7. Created a new static method under the concrete Player class to show the death message when the Farmer dies. We need to check if the actor has the stamina attribute to differentiate between Player and other actor.

Pros	Cons
Do not need to instantiate a Player instance to call the method	If the future actor (not player) has stamina attribute, we might need to change the checking mechanism.
It promotes the code reusability as we can reuse the method when the Player dies, which adhered to the DRY principle.	

Design Diagram

REQ 3 UML Class Diagram



Design Rationale

REQ 3:

1. Implementing the lose stamina mechanism when the farmer performs an action.
   - The farmer loses stamina when planting a seed. This mechanism is implemented in the abstract Seed class plant() method. It reduces the stamina based on the crops where the value is passed into the attribute plantStaminaConsumed. I differentiate the actor and the farmer by checking the presence of the stamina attribute.

Pros	Cons
This design promotes code reuse because we can reduce the stamina-reduction code in each individual seed class which adhered to the DRY principle.	Since we differentiate the actor and the farmer by checking the presence of the stamina attribute, if a creature with stamina that can also plant seeds is introduced in the future, we might need to modify the code.
This design also enhances scalability and extensibility because we can just extend the Seed class and pass a custom plantStaminaConsumed value when a new seed type exists. This design is adhered to the Open Closed Principle.	

   - The farmer loses stamina when curing the blight to soil using talisman. This mechanism is implemented in the Blight class talismanCure() method. I differentiate the actor and the farmer by checking the presence of the stamina attribute.

Pros	Cons
This approach allows for potential future extension because other actors could lose stamina while curing the blight. This design adhered to the Open Closed Principle.	If other actors gain the stamina attribute in the future, they could unintentionally be treated as farmers unless additional checks are implemented.

2. Implemented the countdown timer for creatures that are not resistant to rot.
   - Created abstract NonRotResistantCreature class to represent the creatures that are not resistant to rot and abstract RotResistantCreature class to represent the creatures that are resistant to rot. Both classes extend from the abstract Creature class. OmenSheep and SpiritGoat classes extend from the NonRotResistantCreature class. I introduced a counter attribute in NonRotResistantCreature class to represent the countdown timer. The

countdown timer mechanism is implemented in NonRotResistanceCreature class playTurn() method. When the counter reached the disappear time, the creature will disappear.

Pros	Cons
By introducing RotResistantCreature and NonRotResistantCreature abstract classes, the logic for rot resistance is neatly encapsulated, which adhered to the Single Responsibility Principle.	Adding another layer of abstraction makes the class hierarchy deeper, which may increase complexity and make the code more difficult to understand.
This design also promotes the code reusability because the similar attributes and method will be inherited which avoid code duplication. This design is adhered to the DRY principle.	
If new creatures are added in the future, it's straightforward to classify them by extending the appropriate class, which adhered to the Open Closed Principle.	

3. Added a new ability CURE_ENTITY for the talisman. When the player picked up the talisman, the player will have the CURE_ENTITY ability.

Pros	Cons
It promotes flexibility and extensibility, as we use the ability instead of hardcoding it into Player or Talisman class.	Slight increase in the complexity as managing ability introduces another layer of logic in the system.
If there are new curing items in the future, we can just assign the ability to the item without modifying the existing code. This design is adhered to the Open Closed Principle.	

4. Created TalismanCurable interface and it is implemented by class that can be cured by Talisman such as the non rot resistance creature (OmenSheep and SpiritGoat) and Blight.
    - The reason for introducing a TalismanCurable interface instead of generic Curable interface is that there might be items that are capable of curing, which may cure different type of object than the talisman does

Pros	Cons
By introducing the TalismanCurable interface, the system can easily be extended to allow the talisman to cure other types of entities in the future. New classes that can be cured by the talisman can simply implement this interface without modifying existing code, which adhered to the Open Closed Principle	It may potentially cause overhead for simple system because for smaller system, introducing specific interface might seem like over-engineering when a single Curable interface could suffice.
Creating a specific interface improves scalability and extensibility. If there are new curing items exists, they can introduce their own curing interface.	This design may increase the complexity of the system when there are more different cure items in the system as these new cure items require their own interface, making the code difficult to understand.
The TalismanCurable interface defines a clear and concise contract for object that can be cured by talisman and it does not force any unrelated method upon the implementing class which adhered to the Interface Segregation Principle.	

5. Created a new concrete TalismanCureAction class to represent the cure action by talisman which extends from abstract Action class.

Pros	Cons
This design encapsulates only the logic related to curing with the talisman, making it easier to maintain. In other word, the class only has a single responsibility which is executing the cure action by talisman which adhered to the Single Responsibility Principle.	The TalismanCureAction has tight coupling to the TalismanCurable interface, which potentially make changes more difficult in the future.
This design enables to reuse the logic whenever the talisman can be used to cure entity, without duplicating code. This adhered to the DRY principle.	
This design also support extensibility as we can create other cure actions in the future by extending from the abstract Action class.	

6. Implementing the effect of curing by talisman.
    - When SpiritGoat is cured, the countdown timer is reset. This is implemented by overriding talismanCure() method from the TalismanCurable interface in SpiritGoat class. In the talismanCure() method, it called the resetCounter() from NonRotResistantCreature class to reset the countdown timer of the SpiritGoat instance.

    - When OmenSheep is cured, a new inheritree grows at each tile surrounding the sheep. This is implemented by overriding the talismanCure() method from the TalismanCurable interface in OmenSheep class. In the talismanCure() method, we set the exits of the OmenSheep instance to grow a new inheritree instance. However, not every ground type can grow a tree, therefore we introduced a new ability GROWABLE which determine whether the ground can grow a tree. By checking the capability, the inheritree will only grow on specific ground type.

    - When a Blight is cured, it reveals the soil underneath it. This is implemented by overriding the talismanCure() method from the TalismanCurable interface in Blight class. In the talismanCure() method, we check for the stamina and set the ground into a new Soil instance and remove its CURSED status.

Pros	Cons
It promotes high cohesion as each class encapsulates its own curing logic, keeping the behaviour localized and easier to understand, maintain.	This design may potentially increase code duplication as there may be similar behaviour repeated across multiple talismanCure() implementation.
Each talismanCurable class implements its own talismanCure() behaviour, making it easy to introduce new curable entities without modifying existing logic which adhered to the Open Closed Principle	