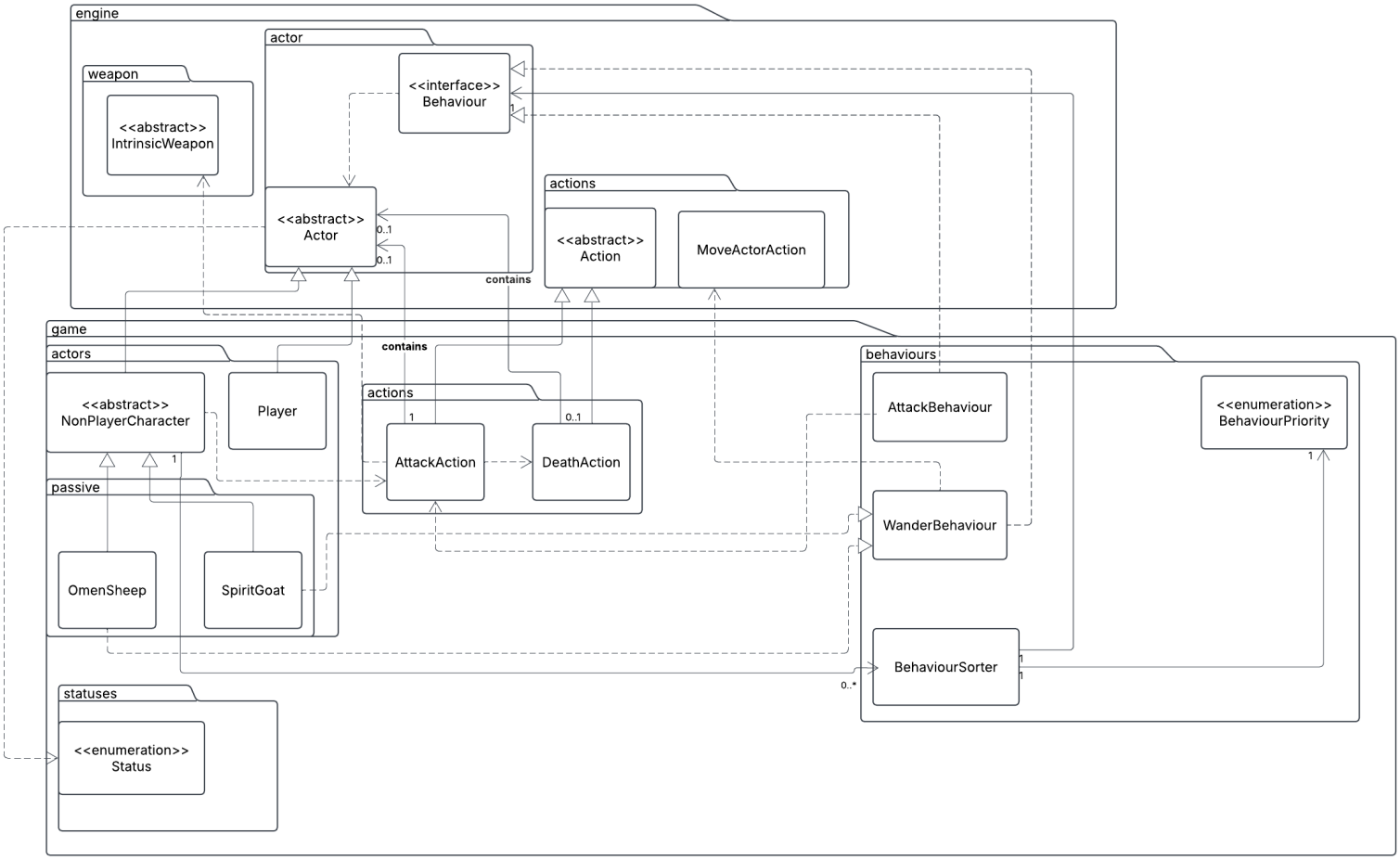
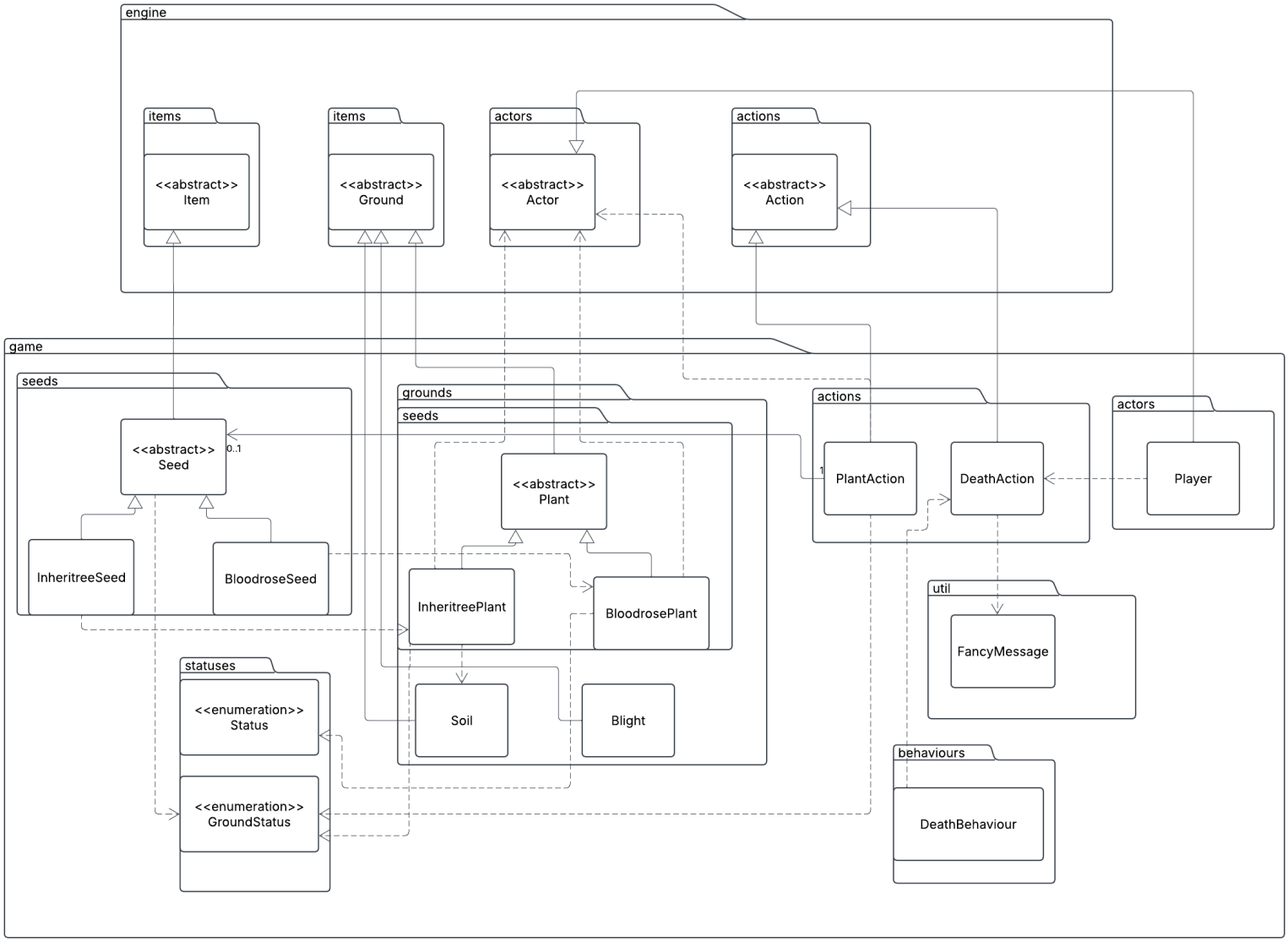
FIT2099 Assignment 1 Design Diagrams and Rationale

UML Class Diagrams

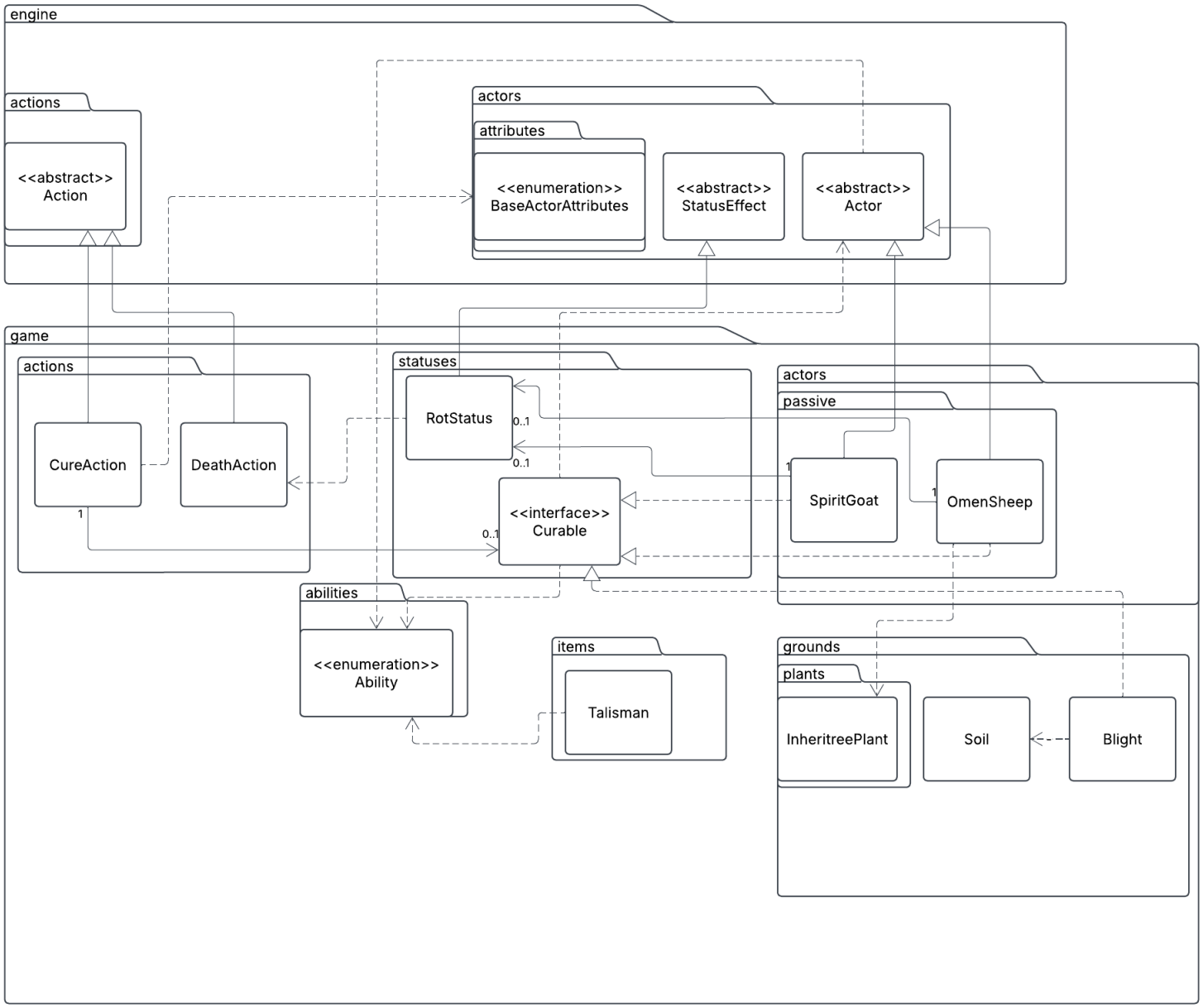
Requirement 1



Requirement 2



Requirement 3



Design Rationale

Requirement 1

Implementation of SpiritGoat and OmenSheep (and future NPCs)

In order to represent the SpiritGoat and OmenSheep, a new abstract class, extending Actor, is created, called NonPlayableCharacter and exists to represent Creatures which are not controlled by the user, in this case the two passive creatures in this requirement. As all NPCs will have certain actions dictated by certain Behaviours, the use of the NPC abstract class allows for every NPC to perform actions based on a list of behaviours, adhering to the DRY principle, while the individual behaviours are added to each class. This also adheres to OCP, as we can add new NPCs easily without modifying the existing base code for every NPC. This also adheres to LSP as since NonPlayableCharacter extends Actor, any subclass will also work in the engine's code when it's expecting Actor.

To ensure each NPC can have Behaviours, with certain Behaviours having priority over others, each NPC carries an list of BehaviourSorters, carrying a Behaviour and a BehaviourPriority Enumeration, while the NPC contains methods to add these and sort these by their priority. Doing this ensures that future behaviours can be added with an appropriate priority in the BehaviourPriority Enumeration, adhering to OCP. The list of BehaviourSorters can also accept any implementation of Behaviour, adhering to LSP. Different behaviours, such as WanderBehaviour, implement the Behaviour interface, where the getAction is used in the engine code to return the available actions for that NPC. Hence, each Behaviour can implement its own logic to get the appropriate actions for its behaviour, and hence each NPC can add the appropriate behaviours that represent its actions. Every single Behaviour implementation is required to use the getAction method for the game's logic to work, adhering to ISP – none of the Behaviours rely on unneeded methods. For example, the SpiritGoat and OmenSheep both move randomly, so the WanderBehaviour is added to these classes, giving them the ability for them to randomly select a MoveAction. Additionally, instead of the engine directly referencing each behaviour to know what actions to perform for the NPCs, it instead references the Behaviour interface and each hence the corresponding actions for each unique behaviour are performed, adhering to DIP.

This also allows for the implementation of an AttackBehaviour, which allows for NPCs to attack other actors (including the player) if they have the HOSTLE_TO_ENEMY status, and hence can control which creatures are hostile.

Disadvantages:

- Since the Behaviour interface cannot be modified (in the game engine), the logic for sorting the Behaviours had to fall in the NPC class, which violates SRP as the NPC class now has to hold behaviours, and know how to sort them.

- BehaviourPriority must be MODIFIED every time a new behaviour is added, violating OCP, however it is not possible without modifications to the game engine to sort behaviours and give them priorities without this Enumeration.
- Currently if a creature is given an AttackBehaviour it will attack any Actor that is given the HOSTILE_TO_ENEMY status, but if in future groups of creatures that are not hostile to each other are added, it requires lots of different statuses to be added, violating OCP and becoming very messy. The solution would be to add interfaces for different groups of creatures, that don't attack each other, however this is beyond the scope of this requirement as it is currently unknown what groups are required.

Implementation of Stamina

The farmer (Player) stamina is implemented through utilising the addAttribute method and the BaseActorAttribute.STAMINA both supplied in the game engine and displayed through the overriding of the toString method. This approach was done as it required no extra classes, just using the existing engine code, and an override of a method from its parent class Actor.

Implementation of Attack

The AttackAction method implements the attacking logic in the game. The Player can attack other creatures (NPCs) because in AllowableActions, it checks that the nearby actor has Statuses PLAYER and that the NPC has HOSTILE_TO_ENEMY, and if it does, offers an AttackAction on the creature. This was done as checks aren't needed in every other individual class to see if they can be attacked by the Player, its handled by the NPC class and dependent on their assigned status. It also means that AttackAction only has to worry about doing damage, adhering to SRP.

If an actor dies, a new DeathAction is created and executed, handling the death message by calling the unconscious method from the engine (removes the actor from the map and gives the message), and also handles the dropping of items. This also adheres to SRP as the death logic is handled separately to the attack logic.

Disadvantages and Justification

- Every time a new rule to dictate which creature can be attacked by the player/other actors, the allowableActions method must be modified, violating OCP. This again could be solved with extra interfaces for different groups of NPCs, however it is still unknown what groups would be present, so the current implementation fits the scope of the requirement.

Requirement 2

Implementation of Seeds and Plants

The Seed abstract class is created to represent seeds in the game, and extends the Item abstract class, adhering to LSP as the game engine can treat any extension of Seed as a normal item with performable actions on it. Each extension of Seed, in this case InheritreeSeed and BloodroseSeed only deal with what specific Plant is returned in the result of planting that seed, adhering to SRP. The same advantages apply for the InheritreePlant and BloodrosePlant – they extend the new abstract class Plant, which extends Ground. Each extension of Plant overrides the tick method from Ground to implement the requirement of their tick and immediate turn effects. This means that any new Plant can implement any tick effect, without modifying any of the high-level game engine logic, adhering to OCP.

The Seed class controls which Grounds can be planted on, by checking if it has GroundStatus.PLANTABLE enumeration – if future types of Grounds are created and can be planted on, they simply just need to add this status, using an existing method of addCapability provided in the game engine. This approach is chosen because it leverages the game engine's existing methods and avoids downcasting to check if there's a certain type of ground. Additionally, to implement the Inheritree's cursed ground removal, GroundStatus.CURSED is added, and used by Inheritree to check if nearby grounds are cursed – if they are, they can be turned into soil. This all adheres to OCP – no further modification of Inheritree is required to check for new types of cursed Ground. While for this requirement, CURSED and PLANTABLE pretty much represent the opposite of each other so it isn't explicitly required, in future there may be new types of CURSED ground that isn't PLANTABLE, so they are kept separate.

To allow for the Player to plant seeds, a PlantAction class is created that extends the Action class, making use of the execute abstract method – adhering to OCP as the existing engine doesn't need to be modified. PlantAction can also accept any type of Seed, adhering to LSP. PlantAction also is only concerned with the logic of setting the new Plant returned from the seed's plant method as the new ground at the location, adhering to SRP.

Disadvantages and Justification

- Each Plant must handle its immediate effect and tick effect in one method, which technically violates SRP as there's two different behaviours in one class. While an interface could be created to segregate these behaviours, only the tick method gets used in the engine class, meaning that logic would have to be created to prevent the tick method from executing, whilst creating specific logic for the immediate effect to occur instead (would probably be called in the PlantAction). The current approach fits in more seamlessly with the engine code, and does not create overengineering for a two fairly similar behaviours (simply just what a Plant does to the rest of the map/actors).
- While a Plantable interface would be the more LSP choice, as code can be written to purely accept Plantables when planting, it would also require the use of a visitor pattern, which when comparing to the provided addCapability from

the engine, is more complex and bordering on overengineering – simply leveraging the existing code from the engine to achieve the same result, despite slight SOLID violations, is the more efficient choice.

Additional Death Logic

Since in this requirement, creatures may die from the result of the Plants (not related to the actor), on every tick, there should be a check to see if the creature is still conscious. Hence, in the NonPlayerCharacter constructor, a new DeathBehaviour is added, which implements the Behaviour interface and returns a new DeathAction if the creature is no longer conscious. No additional modification is required for new creature classes, as the logic is already implemented in the abstract class, adhering to OCP and DRY. Additionally, DeathAction now has a second constructor for when there is not another actor involved with the Death, which allows for the BloodrosePlant to implement its potentially lethal damage logic. SRP is adhered to as DeathBehaviour is only concerned with checking if the actor is still conscious, and if it is, to return a new DeathAction to be executed.

For the Player, on the playTurn method, code is added to check if the player has fallen unconscious as a result of any non-actor's actions (hardcoded in instead of using Behaviours, as it would be pointless for the Player to implement the singular Behaviour of DeathBehaviour). The message display is also hardcoded into DeathAction, checking if the actor to die is the player using Status.PLAYER. Again, this leverages the engine code, therefore making the logic simpler.

Disadvantages and Justification

- Violation of SRP in playTurn (checks for unconsciousness) and DRY (logic repeated in DeathBehaviour). However, since Player is a special case of Actor, and is driven by Menu, if the Player used DeathBehaviour, the Player would have to 'choose' the DeathAction, due to how the engine works.

Requirement 3

Implementation of disappearing Actors

A new RotStatus, extending the abstract StatusEffect abstract class from the engine is created, implementing the tick method to count the amount of turns it takes for the Actor to disappear. Each actor who carries this status, may implement the amount of turns by carrying an instance of the RotStatus, and add it to its list of status effects using getStatusEffects so the engine can implement the tick effects. If the amount of ticks is reached, a new DeathAction is created and executed, causing the Actor to disappear. This approach adheres to SRP – RotStatus executes the logic for the amount of ticks required for the Actor to disappear, and the actual disappearing of the Actor is handled in DeathAction. The creation of RotStatus also ensures OCP is adhered to – all that was required to implement this logic was an extension of the StatusEffect abstract class. Moreover, this rot effect will seamlessly fit into the engine's code and have its tick method executed as it's a subclass of StatusEffect, adhering to LSP.

Disadvantages and Justification

- Tight coupling between SpiritGoat/OmenSheep, or any Curable, and RotStatus due to needing to access the resetRot method. This was done to avoid the use of downcasting or instanceOf to get the RotStatus from the list of status effects, removing runtime checks. Creating a whole new interface for the reset method would have required logic implemented in the cure method (working with a 'RotResettable') would have been overengineering for this minor of a part in the design.

Implementation of Curing

A Cureable interface, with one Cure method, is created as different types of classes in the game (Ground, Actor) may both be cured. This allows for the individual Ground or Actor, in this case, the Blight, OmenSheep and SpiritGoat to implement their own cure logic. This adheres to OCP – if there's any extra classes in the game that can be 'cured', they simply only need to add this interface. The cure method is only responsible for implementing the cure logic for that specific class, adhering to SRP. The CureAction class is created, extending the abstract Action class to provide the Player the option to cure in the menu, and simply just calls the cure method of the Curable – since it only accepts Curables in the constructor, there is no extra downcasting/instanceOf required. To allow the Talisman to provide the Player with the ability to cure, its pickUpAction and dropAction are overridden, to provide Ability.CURER, leveraging the addCapability method from the engine. Then in the allowableActions of each Curable, there's a check for this status on the other actor before adding the CureAction. Again, this approach avoids any downcasting or instanceOf and instead uses the game's engine to check whether a curing action can be done. CureAction also only depends on the abstraction Curable, instead of a concrete class such as OmenSheep, adhering to DIP. Curable also only has one essential method that will always be used by any Actor/Ground that can be cured – adhering to ISP.

Disadvantages and Justification

- Every curable must implement its own override of allowableActions, which slightly violates DRY, however since each CureAction consumes different amounts of stamina, this is a necessary evil – otherwise there would be one CureAction for all Curables with the same stamina consumed, which does not fulfil the requirement.
- If Talisman would need new abilities added to it, the code for it would again have to be modified, violating OCP.
- If new items, that also have the ability to cure are added, the same code that was written in the Talisman must be repeated, violating DRY. This could be solved by creating a CurableItem interface, however it is not yet known whether other items would be able to cure absolutely every Curable, or even cure at all, so it was not added for simplicity

Implementation of Stamina cost

Both the CureAction and PlantAction have an extra parameter in the constructor, that accepts the stamina cost as an integer and then uses the game's engine code to modify the stamina of the player. This effectively leverages the engine's existing methods and also allows for other classes that return any of these actions, to implement their own stamina costs.