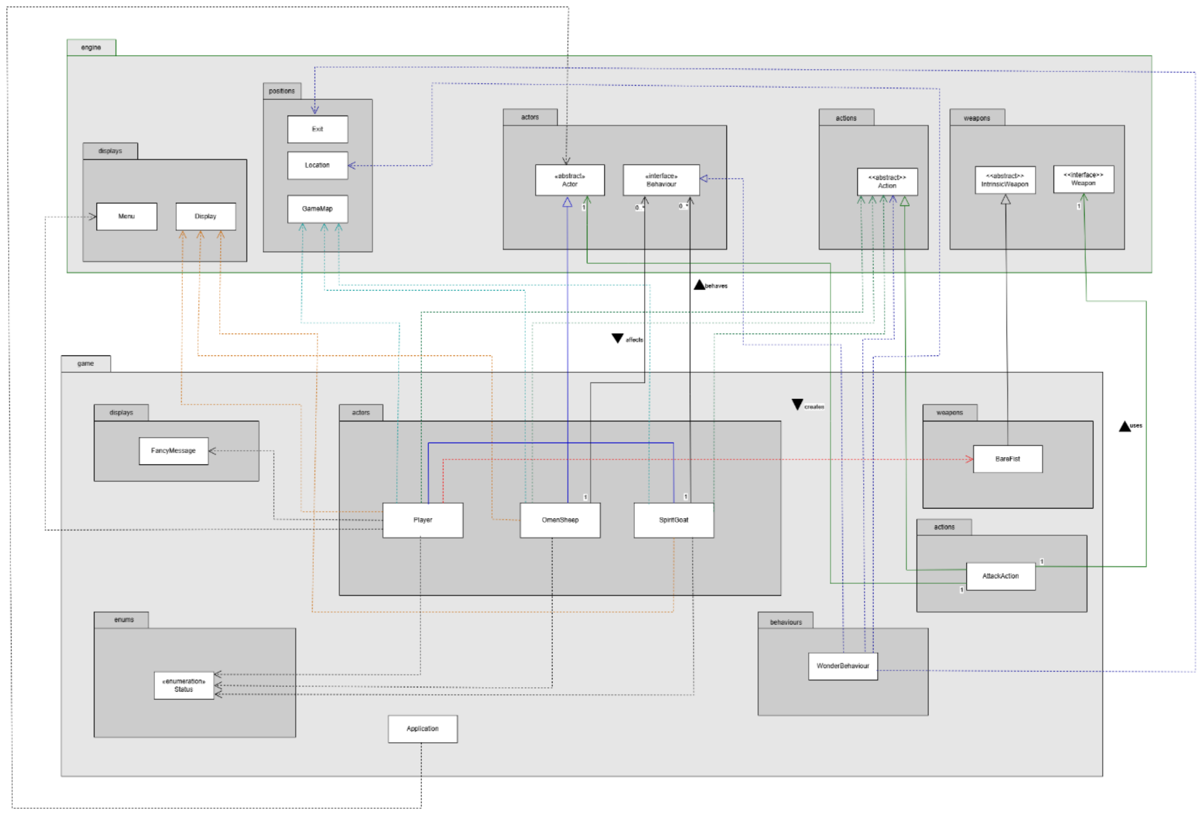
FIT 2099 Assignment 1 Report

Name: Tee Zhi Hong

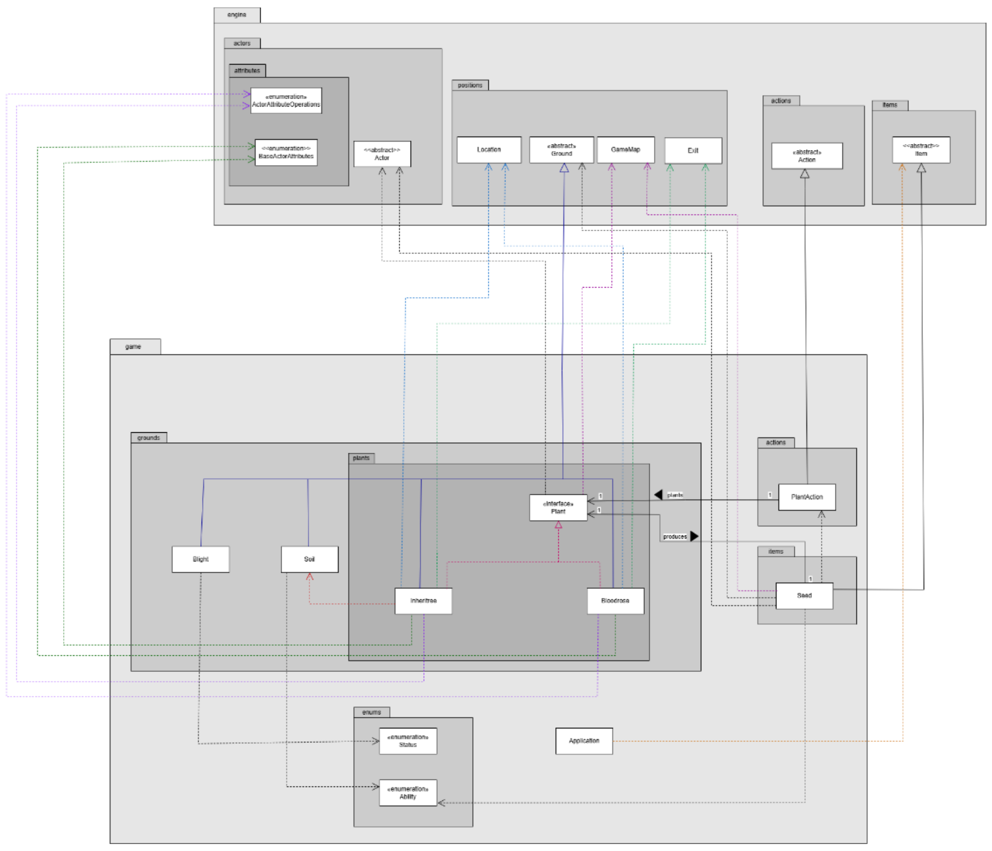
Student ID: 34570403

Design Diagrams

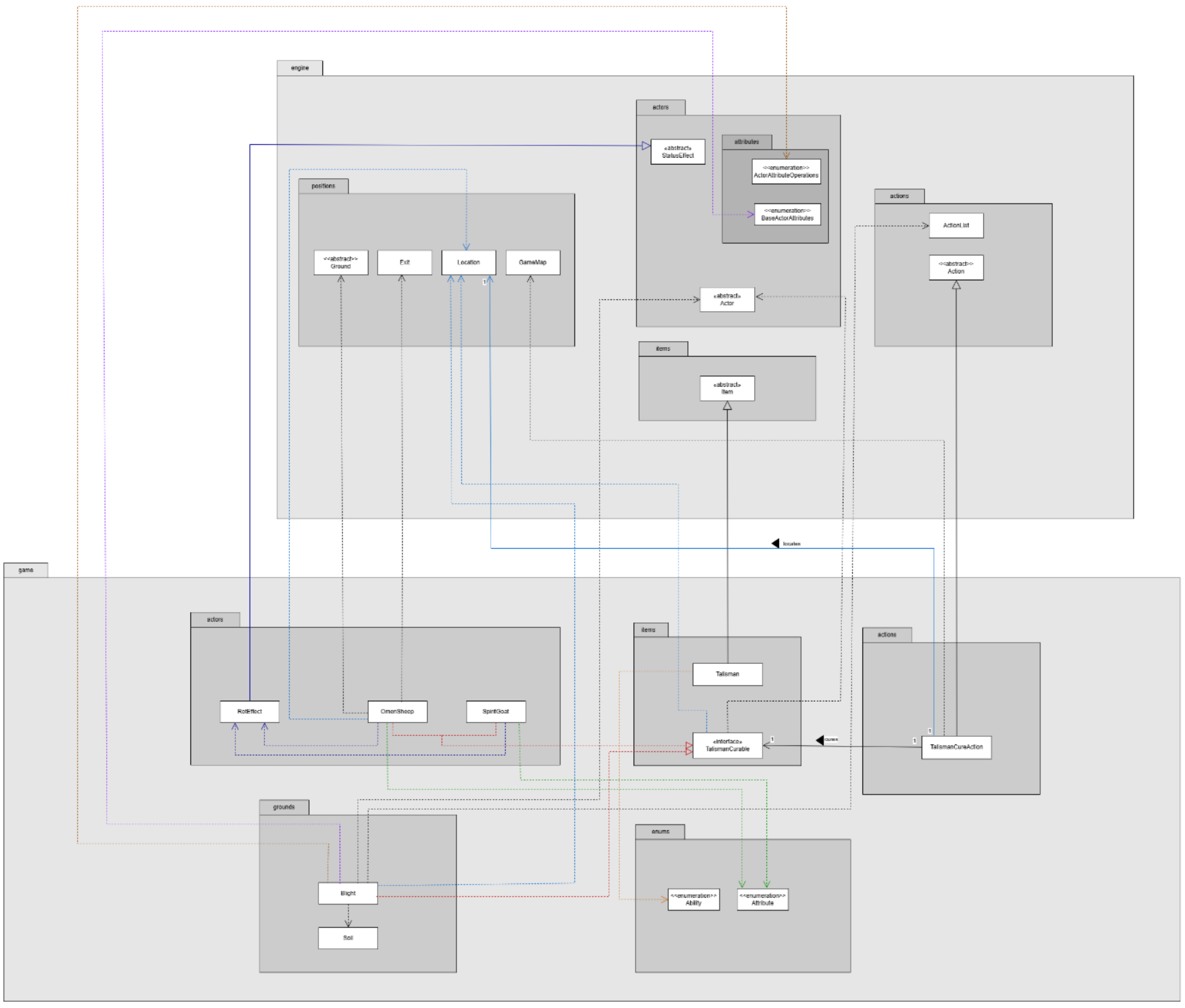
<u>REQ 1 UML Class Diagram:</u>



REQ 2 UML Class Diagram:



REQ 3 UML Class Diagram:



Design Rationale

REQ 1

A: Implement a player called “Farmer” for Elden Thing.

Design 1: Make Player class an abstract class and create a Farmer class that inherits the Player abstract class.

Pros	Cons
Easy to extend without modifying the Player class, especially when there are multiple types of players.	Multiple layers of abstraction cause tight coupling between them, which increases code complexity and increases the chances of the parent class method logic being lost in its deeply nested child classes.
	It can cause redundancy if all types of Player classes have the same characteristics.

Design 2: Use the Player concrete class to create a "Farmer" player for Elden Thing.

Pros	Cons
Easy to maintain or extend if all types of players have the same characteristics, or if it has only one type of character.	Hard to maintain or extend if many types of players have different characteristics.
No need to create concrete classes for each type of Player, which reduces code redundancy.	

B: Implement non-playable characters (NPCs) for Elden Thing.

Design 1: Create an abstract class called NonPlayableActor which extends the Actor abstract class to represent all non-playable actors.

Pros	Cons
Avoid code redundancy if they have similar characteristics across all non-playable actors.	Multiple layers of abstraction would result in tight coupling between them that increases code complexity, which in turn harder to maintain or debug
Scalable, as it can extend to new non-playable actors without any code modifications inside the abstract class.	Increase the chances of the parent class method logic being lost in its deeply nested child classes
	Overcomplicates the game design if a few non-playable actors are inside the game.

Design 2: Non-playable actors are implemented as concrete classes and extend to the Actor abstract class from the “engine” package.

Pros	Cons
It has only one layer of abstraction, which is the Actor abstract class, so the classes can be maintained and debugged easily.	May produce code redundancy if all non-playable actors have the same characteristics as each other but are different in name.
Provide separation of concerns by giving each non-playable actor its dedicated concrete class if they vary.	
Utilises the resources from the game engine package.	

C: Implement a behaviour that allows spirit goats and omen sheep to wander around the valley.

Design 1: Create a concrete class called "WanderBehaviour" that allows non-playable actors to wander around the valley without any interfaces implemented.

Pros	Cons
Easy implementation and simple logic.	If a non-playable actor has many kinds of behaviours, more methods and attributes are needed to accommodate each behaviour, which causes code redundancy.
	Does not fully utilise the game engine. The "Behaviour" interface is provided to represent all Behaviour classes.
	Causes low code readability, which hardens the code maintenance for a non-playable Actor class if the class has many behaviours.

Design 2: Create a concrete class called "WanderBehaviour" that allows non-playable actors to wander around the valley, and the concrete class implements the "Behaviour" interface from the game engine.

Pros	Cons
The "Behaviour" interface can represent all behaviour classes, which reduces the number of attributes for behaviour classes inside each non-playable Actor class.	It might cause code redundancy if all behaviour classes have the same characteristics, including the same method implementation across all behaviour classes.
Able to store different kinds of behaviour classes under a single list attribute.	
Easy to extend the behaviour of a non-playable Actor, as no additional attributes or methods are needed to extend it.	

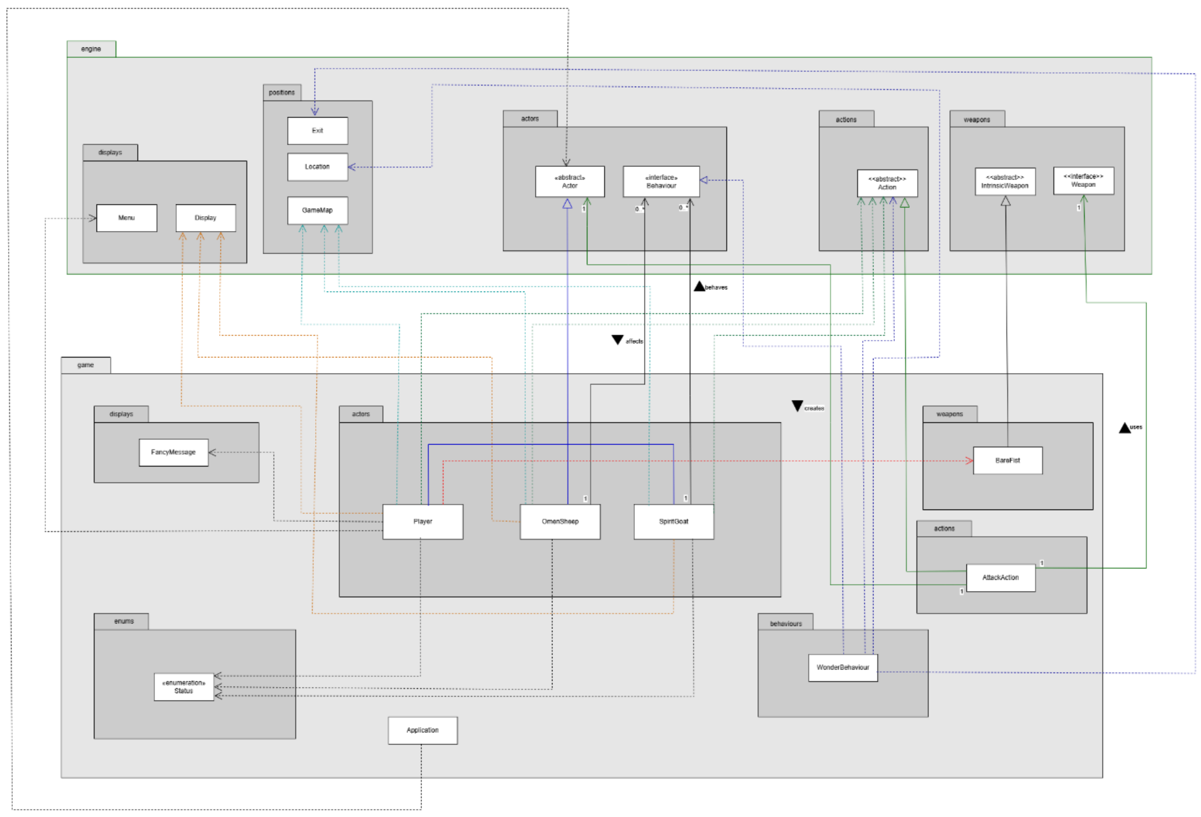
D: Implement a feature which the "Farmer" can use their bare fist to attack any creatures in the valley

Design 1: Create a concrete class that represents the attack action of an Actor to another Actor ("AttackAction") without extending to any super classes.

Pros	Cons
Easy to debug and maintain when there are a few actions in the game.	It can cause code redundancy if many actions possess similar characteristics (e.g. same implementation of methods).
	Need to have many attributes and methods if the Actor has many actions, which also causes code redundancy.
	Not fully utilising the game engine, the abstract class Action in the engine already gives the base methods and attributes for an action in the game.

Design 2: Create a concrete class that represents the attack action of an Actor ("AttackAction") and extend it to the Action abstract class from the game engine.

Pros	Cons
Allow scalability as the abstract class can be extended easily to a new action class without modifying it.	May cause code redundancy if all actions behave the same way.
Prevents downcasting, which checks whether the action matches the specific action.	
Easy to debug and maintain, even for an Actor who has many actions.	



The diagram above shows the final design of Requirement 1, which utilises Design 2 in parts A, B, C, and D to implement the Player class that can spawn a "Farmer", SpritGoat, and OmenSheep classes that wander around the valley. The farmer can also attack the creatures using their intrinsic weapon, a bare fist.

Since the Player, OmenSheep, and SpiritGoat have the same method implementation and attributes as the Actor abstract class, it is useful and logical to extend these classes to the Actor abstract class, as it can reduce code repetitions among these classes (Don't Repeat Yourself). Also, these classes can be represented as an Actor class object (upcasting), which makes it easier for external classes that have Actor attributes, especially the AttackAction class, as the new actor class can extend the Actor abstract class without the need to modify the external class code (Open-Closed Principle).

The WonderBehaviour class implements the Behaviour interface from the game engine. The benefit of it is that all kinds of behaviours for non-playable characters can be represented as a single class object, so it can have a single attribute that covers all derived classes of Behaviour instead of an attribute for each behaviour used inside the external class (Dependency Inversion Principle and Reduce Dependencies Principle). (983 Words in Total)

REQ 2

A: Implementing Inheritree, Bloodrose and other plants in Elden Thing.

Design 1: Implement the concrete class representations of plants without any abstraction.

Pros	Cons
Simple to implement as the design does not require the implementation of any abstraction layer (abstract class or interface).	Does not have a representation that unifies all the plant classes.
Easy to debug and maintain if there are a few plants in the game.	The external class that uses many kinds of plants needs to have a dependency for each plant, which makes the external class more complicated, that it hard to maintain and debug.
	Need to modify the external class that uses all plants if there is a new plant in the game.
	Not fully utilise the game engine as the abstract Ground class has attributes and methods for a Ground.

Design 2: Implement an interface called "Plant". The concrete class representations of plants will extend to the abstract Ground class and implement the interface.

Pros	Cons
Allows the external class that depends on plants to have a lesser dependency, as it can depend on the Plant interface to obtain all concrete class representations of plants.	It might be redundant if all plants have similar attributes and characteristics.
No modifications in the external class that depends on plant classes when a new plant is added to the game.	
The interface will represent all plants without adding another layer of abstraction through a new abstract class, which can make code maintenance harder.	

B: Implement seeds in Elden Thing.

Design 1: Implement an abstract Seed class that extends the Item abstract class, and the concrete class representation of seeds will extend the abstract Seed class.

Pros	Cons
Shows a clear relationship between seeds in the design.	Multiple layers of abstraction will cause tight coupling between the parent class ("Seed" abstract class) and child classes, in which changes in the parent class will affect its child classes.
Allow the external class that depends on seeds to have a lesser dependency, as it can depend on the Seed abstract class.	Hard to debug as the concrete class representation of seeds has 2 layers of abstraction, so the developer needs to go through the parent classes to see and debug unexpected behaviour in the concrete class.

Design 2: Implement a concrete Seed class to represent all plant seeds and extend the Item abstract class. Each Seed object will have its corresponding Plant attribute that represents a specific plant.

Pros	Cons
Reduce redundancy if each plant seed has the same attributes and behaviours.	Becomes a "God" class if each plant seed has different attributes and behaviours, causing the code inside the Seed class to be difficult to extend and maintain.
Simple to implement the concrete class as it only implements the seed functionality instead of implementing a concrete class for each plant seed.	
Shows a clear relationship between the Seed class in the design.	
Easy to maintain and debug if a few plant seeds are in the game.	

C: Check whether the location has soil so the “Farmer” can plant a seed.

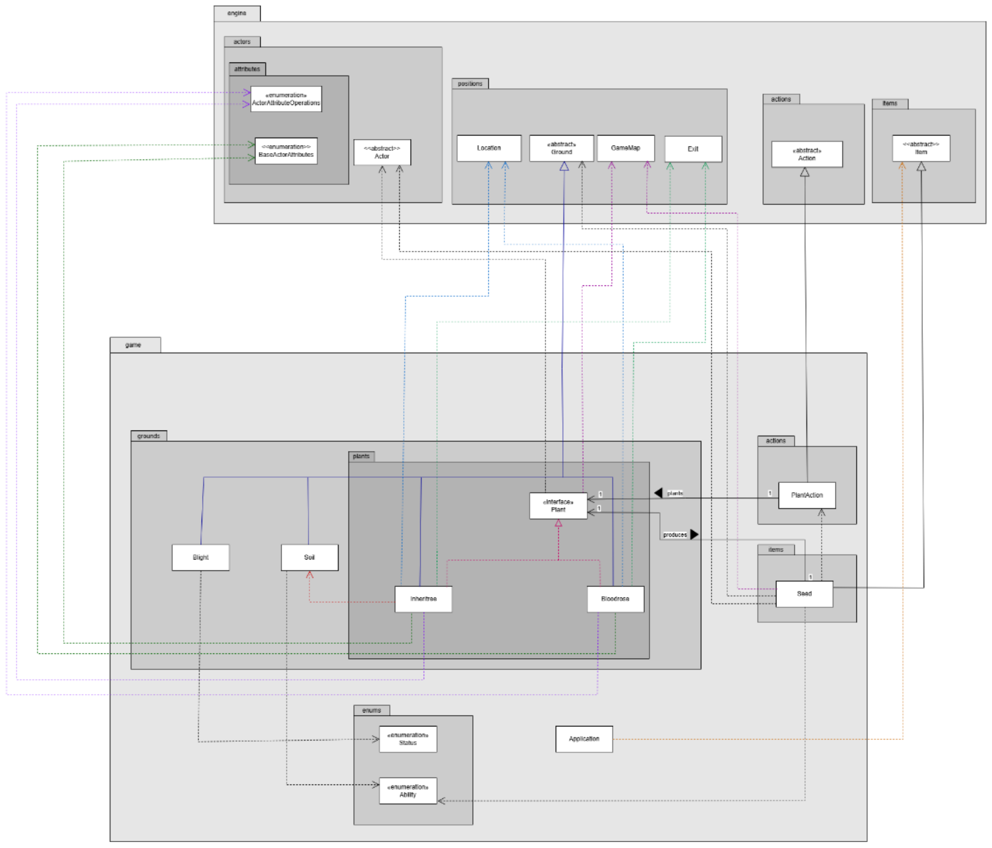
An Ability enumeration constant called “Ability.PLANT_SUITABLE” represents all Ground classes that can be planted. The enumeration constant is added inside the class constructor of the Soil class, such that a seed can be planted on the soil. The benefit of creating an enumeration constant for this is to check the Ground status and prevent the use of downcasting and type checking, such as the use of instanceof, which in turn depends on the Ground subclass type from the Ground abstract class. Therefore, the Liskov Substitution Principle has been violated because it indicates that the abstract Ground class is not substitutable with Ground subclasses.

D: Implement the planting effects of plants such as Inheritree and Bloodrose

The effects of planting are implemented inside the “plant” method for plant classes that implement the Plant interface, such as Inheritree and Bloodrose, so it will only affect the Actor once, which is planting. However, the “plant” method would have many responsibilities if there were many and complex implementations for the effect, reducing code readability and maintainability.

E: Implement the blooming effects of a plant such as Inheritree and Bloodrose

The effects of blooming are implemented inside the “tick” method inside the Inheritree and Bloodrose classes, as the purpose of this method is to spread the effects of the Ground class for each game tick. In this case, these plant classes affect the surrounding actors in each game tick. This implementation utilises the game engine, where the method from the Ground abstract class (“tick”) is reused.



The diagram above presents the final design of Requirement 2, which utilises Design 2 of parts A and B along with the design of parts C, D, and E mentioned above to implement the Inheritree and Bloodrose classes that fulfil the Plant interface. These classes influence the actor who plants them and the surrounding actors when the plant blooms during each game turn.

The Inheritree and Bloodrose classes extend the Ground abstract class, while the Seed class extends the Item abstract class. These subclasses share similar attributes and behaviours with the superclasses provided in the game engine, enabling them to leverage the same attributes and methods as their respective abstract superclasses to minimise code repetition (Don't Repeat Yourself).

The Inheritree and Bloodrose classes implement the Plant interface, creating an abstraction layer that allows these concrete plant classes to be represented as a single class. This arrangement enables external classes, such as the Seed and PlantAction classes, to depend on the plant classes without forming direct dependency relationships with these concrete plant representations (Dependency Inversion Principle). Furthermore, it facilitates future code extensibility, particularly with new plant classes added to the game. The addition of a new plant class that implements the Plant interface will not require modifications to the external classes that depend on the Plant interface (Open-Closed Principle). (980 Words in Total)

REQ 3

A: Implement a countdown timer for all creatures that can rot, which are “SpiritGoat” and “OmenSheep”.

Design 1: Implement an attribute called “countdownTimer” in rotting creature classes and decrement the countdown timer by 1, and check whether the countdown timer has run out of time inside the “playTurn” method in the SpiritGoat and OmenSheep classes.

Pros	Cons
Easy to implement with simple logic.	The “playTurn” method would have combined multiple responsibilities instead of one.
	The” playTurn” method would be hard to maintain and debug.

Design 2: Create an enumeration class called “Attribute” and create an enumeration constant called “COUNTDOWN_TIMER” inside the class. The enumeration constant is used as an enumeration attribute with values inside the “OmenSheep” and “SpiritGoat” classes. Implement a class called RotEffect, which extends the StatusEffect abstract class from the game engine and decreases the countdown timer by 1. The RotEffect class object is added to the “statusEffects” list for the “OmenSheep” and “SpiritGoat” classes.

Pros	Cons
Ensure that every method inside the “SpiritGoat” and “OmenSheep” has only one responsibility.	It will overcomplicate the design if a few creatures can rot.
Modularises the code, separates the responsibility of the rot effect for rotting creatures in each game tick to another class.	The countdown timer will start at total time - 1 and end at 0 instead of 1, which is odd and inaccurate for a timer. The reason is that the game tick is called before calling each Actor inside the
Utilises the game engine code as the Rot class has similar attributes and behaviour to the StatusEffect abstract class. Also, the “addStatusEffect” method inside the abstract Actor class will allow StatusEffect objects to be added to the list and affect the actor from the status effect in each game tick.	

B: Implement an indicator to check whether the rotting creatures have a countdown timer of zero and turn them unconscious if it is

The indicator logic is implemented inside the “playTurn” method for all rottable creatures. The downside of this is that the “playTurn” method will have multiple responsibilities instead of one, which is to return an action that a user or behaviour chooses, which could violate the Single Responsibility Principle. However, there is no “tick” method inside the Actor abstract class, and only the GameMap class has the “unconscious” method it so the “playTurn” method is suitable to be used in implementing the indicator as the method has a GameMap parameter.

C: Implement the curing effects of the talisman onto the things that are curable by it.

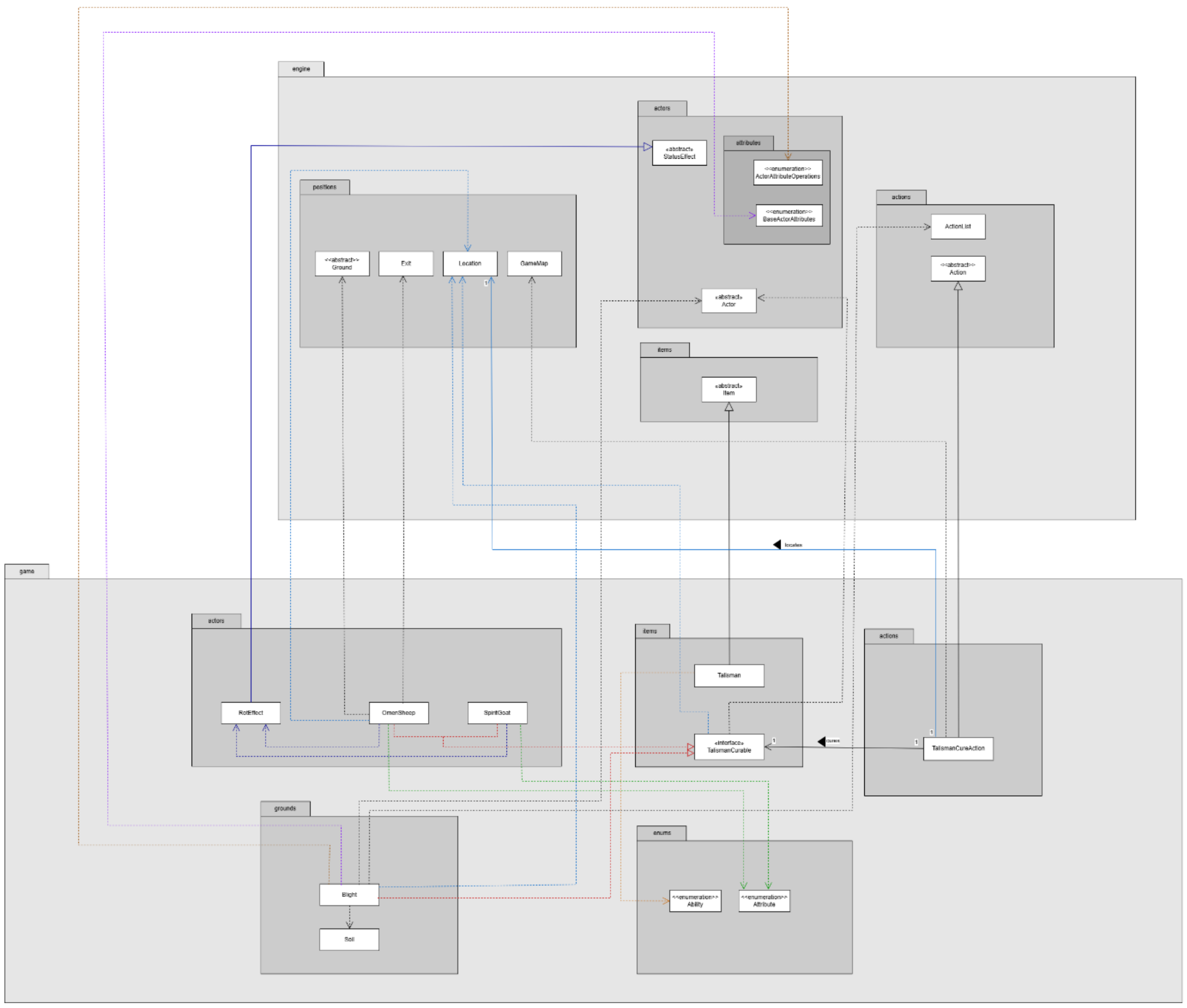
Design 1: Implement the “CureAction” class, which extends the “Action” abstract class from the game engine and has two attributes for curing relics and curable things. Also, create the "Curable” interface and implement the interface for concrete classes that represent curable things.

Pros	Cons
Easy to implement the logic.	Not all entities are curable by the same curing item, so concrete classes that represent them will have empty implementations of the interface methods.
Easy to maintain and debug when there is a small number of curing items and curable things.	Has a lot of dependencies inside the “Curable” interface, as each curing item depends on the interface. This hardens the code maintenance and debugging as the number of curing items increases.

Design 2: Implement the “TalismanCureAction” class, which extends the “Action” abstract class from the game engine and has an attribute for things curable by a talisman. Also, create the "TalismanCurable” interface and implement the interface for concrete classes that represent entities curable by a talisman.

Pros	Cons
Segregates the interface for each curing item, ensuring that no empty implementation interface methods are inside the concrete class if the class implements the interface specific to the curing item.	May cause code redundancy if all curing relics have the same behaviour.
The interface represents all concrete classes that are curable by the talisman, so the TalismanCureAction will have a single TalismanCurable attribute.	
No need to modify the external class when there is a new curing item by a talisman inside the game.	

To apply a TalismanCureAction for entities that are curable by Talisman, the Talisman item has an enumeration constant called “Ability.TALISMAN_CURE”, and the Actor “hasCapability” method can be used to check whether the actor has a Talisman by checking the presence of an enumeration constant of each item inside the “allowableActions” method for each class that implements the “TalismanCurable” interface. The benefit of this is to prevent the use of downcasting and “instanceOf” for type checking, which is dependent on the concrete subclasses that violate the Liskov Substitution Principle and produce code smells.



The diagram below shows the final design for Requirement 3, which utilises Design 2 of Parts A and C and design of Part B stated above to implement the effects of planting "Bloodrose" and "Inheritree" on the Actor's stamina, countdown timer for rotting creatures, and curing actions for things that are curable by a talisman.

The RotEffect class extends the abstract StatusEffect class as the subclass has similar attributes and behaviours to the superclass. The identical attributes and methods present in the abstract superclass are not implemented in the RotEffect subclass to avoid repetitions in code (Don't Repeat Yourself).

The "TalismanCurable" interface can represent all entities that can be cured by a talisman so that external classes can have a single TalismanCurable attribute instead of an attribute for each concrete class that represents an entity that is curable by a talisman (Dependency Inversion Principle). In addition, no code modification is needed for external classes when a new class implements the interface, resulting in fewer dependencies and maintainable code (Reduce

Dependencies Principle and Open-Closed Principle). Furthermore, the interface prevents any empty method implementation as the interface is only for things that are curable by a specific cure relic called a talisman instead of all curing items, so all methods inside the interface are not empty from derived classes that implement the interface (Interface Segregation Principle). (988 Words in Total)