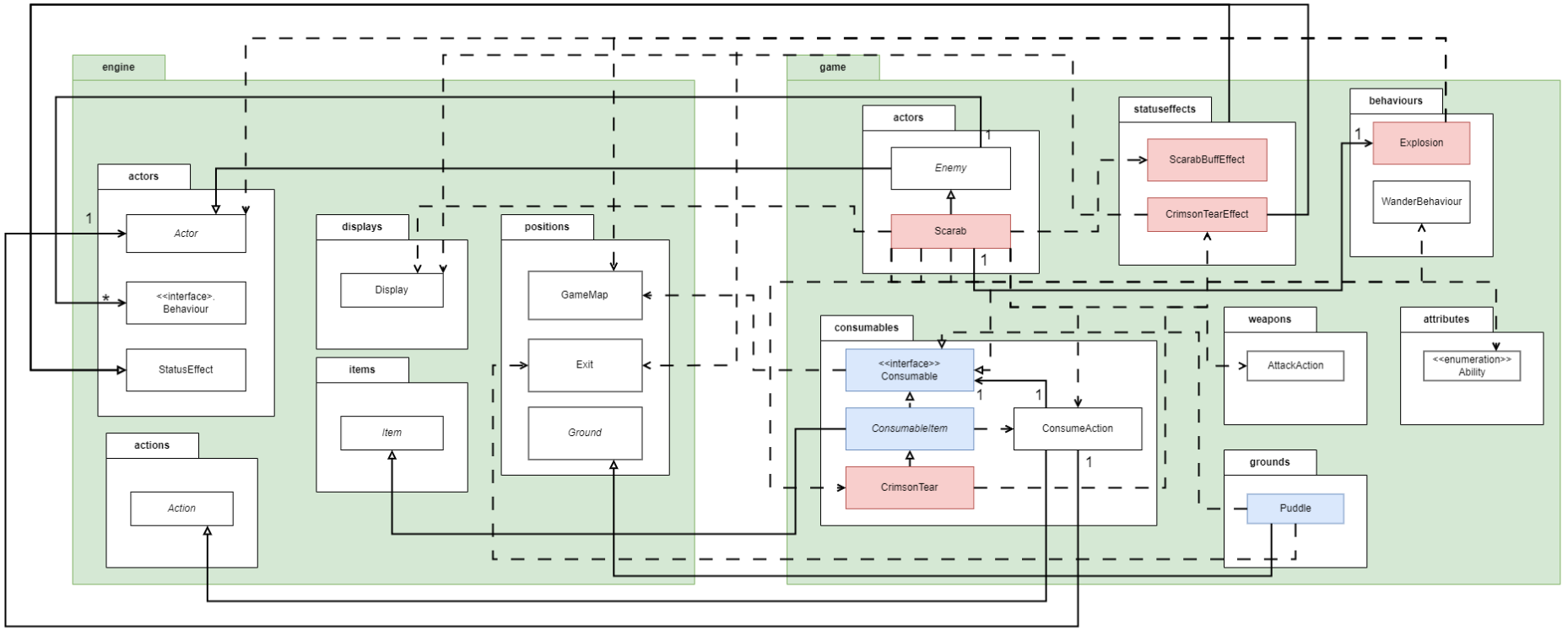
Design Rationale

REQ4: Be wary of dung (beetle)



Modified Classes = Light Blue
Classes Created = Light Pink

Classes Created / Modified	Roles and Responsibilities	Rationale
Created an Explosion class (extends Action)	Class representing explosion action. Relationship: Scarab and FurnaceGolem has association with Explosion to enable both the actors to create an explosion.	Reasons for decision: 1. Don't Repeat Yourself Principle - Since both Scarab and FurnaceGolem can create an explosion, instead of manually coding out the explosion mechanism in both the classes, by creating a separate Explosion class and calling the Explosion.execute() method in both the actor's classes can prevent code duplication. 2. Promotes reusability as if more enemies can create explosions in the future, we can simply reuse the explosion mechanism while passing their respective explosion damage as parameters.
Created a Consumable interface	Class representing consumables in the game. Relationship: Implemented by ConsumableItem, Scarab and Puddle to allow them to be consumed.	Reasons for decision: 1. Creation of Consumable interface allows us to define a common method which is consume() for all game entities that can be consumed such as Actor, Ground and Item. 2. Allows every game entity that can be consumed to have their own specific behaviour and implementation of how they can be consumed by overriding the consume() method. 3. Open-closed Principle – This interface also allows for future extensions by new consumables without modifying any existing code. 4. Since all Consumable Items can be consumed, therefore ConsumableItem abstract class can implement Consumable and leave consume() as an abstract method so that any consumable items that extend this class can
Modified ConsumableItem class to implement Consumable (extends Item and implements Consumable)	Class representing items that are consumable in the game. Relationship: Extended by CrimsonTear to allow CrimsonTear to be consumed while having Item attributes.	

		implement their own respective behaviour of being consumed().
Created a Scarab class (extends Enemy and implements Consumable)	Class representing Scarabs Relationship: 1. Scarab extends Enemy to achieve Enemy functionalities. 2. Scarab implements Consumable to achieve consume functionality. 3. Scarab depends on Ability as it has the capability Ability.POISON_RESISTANT. 4. Scarab has association with Explosion as it can cause an explosion when it dies. 5. Scarab has dependency with CrimsonTear as it spawn a Crimson Tear when it dies. 6. Scarab has dependency with ScarabBuffEffect to allow the actor to obtain the buff when Scarab is consumed. 7. Scarab depends on WanderBehaviour, AttackAction and ConsumeAction as it	Reasons for decision: 1. Promotes reusability as extending from Enemy allows the Scarab to inherit all standard behaviour of enemies such as playTurn(), reducing redundancy. By extending Enemy, Scarab achieves common Enemy functionalities while also being able to have its own specific behaviour such as exploding on death. 2. Liskov Substitution Principle (LSP): Since Scarab is an Enemy as well as a Consumable, it can be substituted in any methods as Enemy or Consumable without breaking Scarab's functionality. This also promotes code flexibility. 3. Dependency Inversion Principle (DIP): Scarab only interacts with high-level abstractions such as Enemy abstract class and Consumable interface rather than concrete implementations. This makes the code easier to maintain as if we were to make changes to all Enemy actors, we can simply change it in the Enemy class without modifying the Scarab class.

	behaves and act according to them.	
Created a CrimsonTear class (extends ConsumableItem)	Class representing Crimson Tear consumable item. Relationship: 1. Extends ConsumableItem to achieve consumable item functionalities. 2. Depended on by Scarab to allow Scarab to spawn a CrimsonTear on death.	Reasons for decision: 1. Single Responsibility Principle (SRP): This class focuses solely on representing a consumable item that temporarily heals the actor when is consumed. This makes the code easier to understand and debug. 2. It promotes easy expansion as if there were additional effects to be added to CrimsonTear, we can easily expand the functionality without modifying any other existing classes.
Created a ScarabBuffEffect class (extends StatusEffect)	Class representing a buff when Scarab is consumed. Relationship: 1. ScarabBuffEffect extends StatusEffect to achieve its functionality. 2. Depended on by Scarab to allow actor to gain the buff when Scarab is consumed.	Alternate Solution: Creating a single AllStatusEffects class that handles all status effects such as the ScarabBuffEffect and CrimsonTearEffect within that class. Disadvantages of alternative solution: 1. Violates Single Responsibility Principle (SRP): The combined class now has to handle two responsibilities which is to manage the temporary healing and the temporary increase of maximum hitpoints and mana instead of having one sole responsibility. This also increases the chances of having bugs as well as makes it harder to debug due to the class having multiple functionalities. 2. Violates Open-Closed Principle (OCP) – When new status effects are to be added, instead of extending from the StatusEffect abstract class, we now need to make changes in
Created a CrimsonTearEffect class (extends StatusEffect)	Class representing a buff when CrimsonTear is consumed. Relationship: 1. CrimsonTearEffect extends StatusEffect to achieve its functionality.	

	2. Depended on by CrimsonTear to allow actor to gain the buff when CrimsonTear is consumed.	AllStatusEffects class. This results in the AllStatusEffects class being not open for extension and not closed for modification. Finalised solution: Creating a ScarabBuffEffect class and CrimsonTearEffect class respectively that extends the StatusEffect abstract class to allow both the classes to have their own implementation but still adhering to the StatusEffect core functionality. Reasons for Decision: 1. Single Responsibility Principle (SRP): The ScarabBuffEffect class focuses solely on temporarily modifying the player's maximum hitpoints and mana whereas the CrimsonTearEffect class focuses solely on temporarily healing the player. By adhering to this principle, If there were changes to be made to ScarabBuffEffect, I can easily modify the ScarabBuffEffect class without affecting the CrimsonTearEffect class. 2. Open-Closed Principle (OCP): The usage of inheriting from the StatusEffect abstract class allows for new status effects to be created by simply extending from the StatusEffect abstract class without modifying any existing code. 3. Liskov Substitution Principle (LSP): Both ScarabBuffEffect and CrimsonTearEffect classes inherits the StatusEffect abstract class. This means that we can substitute StatusEffect with either of the two created classes without modifying the intended behaviour of the program.

		4. Dependency Inversion Principle – Both ScarabBuffEffect and CrimsonTearEffect classes extends the StatusEffect abstract class which is a high level module as it is serves as an abstraction. This reduces tight coupling as every class has its own implementation. 5. Don't Repeat Yourself Principle – Since both ScarabBuffEffect and CrimsonTearEffect classes extend the StatusEffect abstract class, we avoid duplicating the common functionalities for StatusEffects such as the toString() method as it returns the name of the status effect for all StatusEffect subclasses. Disadvantages of Finalised Solution: 1. More classes are created which can lead to memory overhead as well as complexity of the program especially when the number of classes increase by a significant amount.
Modified Puddle class to also implement Consumable (extends Ground and implements Consumable)	Class representing Puddle ground. Relationship: 1. Extends Ground abstract class to achieve ground functionalities. 2. Implements Consumable interface as it can be consumed.	Reasons for decision: 1. By implementing Consumable, it obtains the behaviour similar to all other consumables in the game as it can be consumed. 2. By implementing the consume() method in Puddle, it allows encapsulation of the specific logic of the consumption of Puddle making the code easier to maintain. 3. Promotes polymorphism as Puddle can be treated as a Consumable leading to greater flexibility of the program.

	3. Dependency with Scarab as it has a chance to spawn Scarab when consumed.	