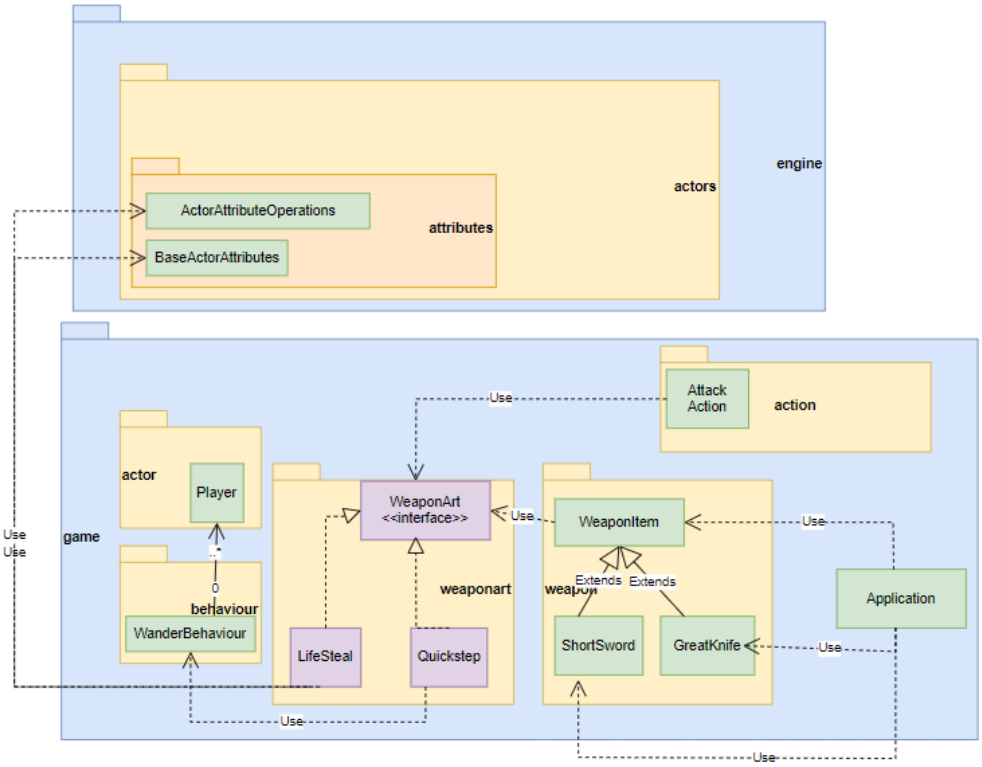
Design Rationale

Requirement 1: Weapon Arts

Design Goal

The main design goal is to extend the weapon system to support Weapon Arts, which are special abilities unique to certain weapons, without modifying the engine. The implementation must ensure that Weapon Arts can be dynamically assigned to weapons and automatically triggered during an attack if the player has enough mana. The design should also accommodate future expansion, allowing for additional Weapon Arts and new weapon types.

UML(Requirement 1):



Classes Added/ Modified	Roles and Responsibilities	Reasons
1. **GreatKnife (Modified)**	**Role**: A specific weapon class that can be assigned different Weapon Arts during instantiation. This demonstrates that the same weapon can have various arts based on the instance. **Responsibilities**: The class initializes the weapon with the assigned Weapon Art and delegates to it during attacks if applicable.	**Alternate Solution** An alternate solution could involve directly embedding the logic for each Weapon Art into the weapon or player class, rather than using the WeaponArt interface. While this approach might reduce the number of classes, it would tightly couple Weapon Arts with the player and weapon logic, violating SRP and making future extensions difficult. **Finalized Solution** The finalized solution uses the **Strategy Pattern** to decouple Weapon Arts from the weapon class, making the system more modular and flexible. Each Weapon Art is treated as a separate strategy that can be applied to a weapon. This approach allows new arts to be added without altering the weapon or player classes and provides a clear and maintainable design. **Reason for Decisions** 1. **Single Responsibility Principle (SRP)**: Each class has a single, well-defined responsibility. For example, WeaponArt defines the interface for arts, and each concrete art (e.g., LifeSteal and QuickStep) is responsible for implementing the logic of that art. This ensures that changes to one aspect of the system (like adding new Weapon Arts) don't impact unrelated areas. 2. **Open-Closed Principle (OCP)**: The design is open for extension but closed for modification. The WeaponArt interface allows new arts to be added in the future without changing existing weapon or player logic. The system can easily
2. **WeaponArt (New Interface)**	**Role**: Defines the contract for all Weapon Arts. Each Weapon Art must implement canActivate() and activate(). **Responsibilities**: Ensure that each Weapon Art can define its activation conditions and effects (e.g., consuming mana).	
3. **WeaponItem (Modified)**	**Role**: A generic weapon class that supports the dynamic assignment of Weapon Arts. **Responsibilities**: Manage weapon attributes (damage, hit rate, etc.) and invoke the Weapon Art if it exists.	
4. **LifeSteal (New Class)**	**Role**: Implements the Lifesteal Weapon Art. When activated, it consumes mana and restores health to the player.	

	Responsibilities: Deduct mana points from the actor and restore health points based on the amount specified.	accommodate new Weapon Arts by simply creating new classes that implement the WeaponArt interface. 3. **Liskov Substitution Principle (LSP)**: Subtypes (Weapon Arts) can be substituted for their base type (WeaponArt) without affecting the correctness of the program. For instance, switching between LifeSteal and QuickStep doesn't require any changes in how the player or weapon interacts with the art. 4. **Interface Segregation Principle (ISP)**: The WeaponArt interface contains only the methods relevant to all weapon arts (i.e., canActivate() and activate()). Each weapon art implementation provides its own logic for these methods, keeping the interface simple and focused. 5. **Dependency Inversion Principle (DIP)**: The AttackAction class depends on the WeaponArt abstraction rather than specific weapon arts. This makes the system more flexible, as AttackAction doesn't need to know which specific art is being used—it simply triggers the art via the interface. **Limitations and Trade-offs** 1. **Complexity**: The introduction of new classes (e.g., WeaponArt, LifeSteal, QuickStep) adds some complexity to the codebase, though this is necessary to ensure flexibility and scalability. 2. **Performance Overhead**: Each weapon attack now checks for the
5. **QuickStep (New Class)**	**Role**: Implements the Quickstep Weapon Art. This allows the actor to move to a random adjacent position after attacking. **Responsibilities**: Execute the Quickstep by invoking a random movement behavior after a successful attack.	
6. **Player (Modified)**	**Role**: The player can now use weapons with Weapon Arts. **Responsibilities**: The player invokes the Weapon Art during attacks if they have enough mana.	
7. **AttackAction (Modified)**	**Role**: The attack action is extended to support Weapon Arts. **Responsibilities**: Automatically activate the Weapon Art during an attack if the conditions (e.g., enough mana) are met.	

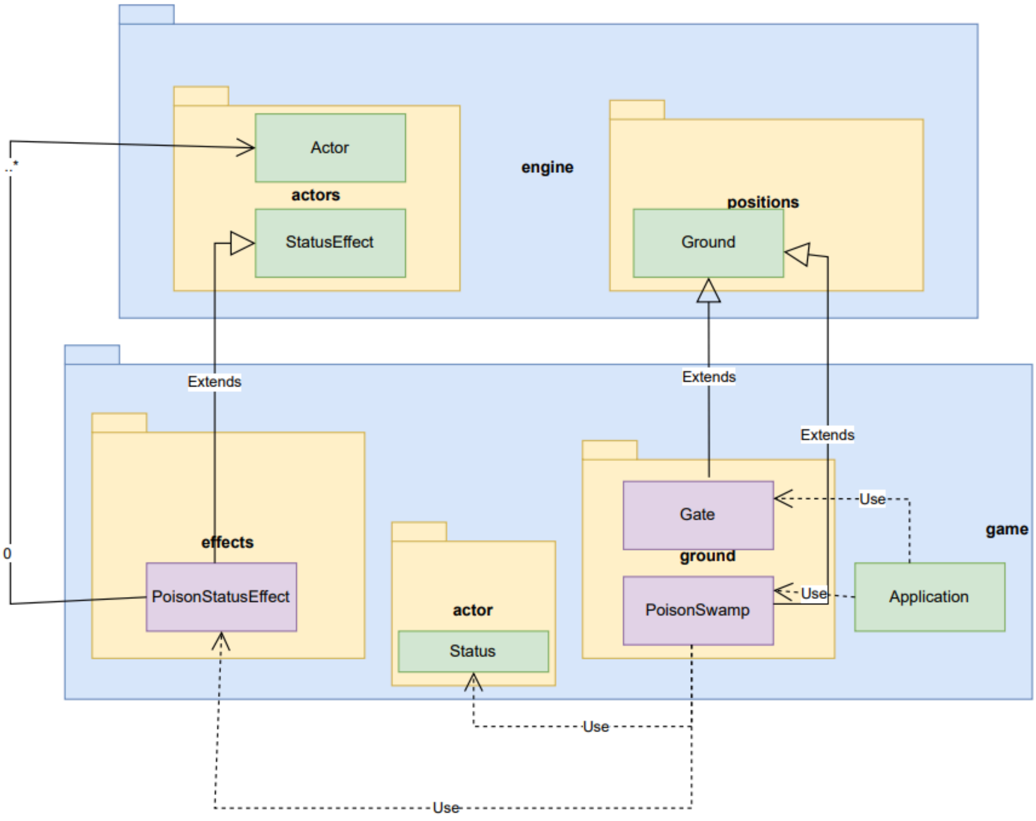
8. **ShortSword (Modified)**	**Role**: A specific weapon class that can be assigned a particular Weapon Arts named QuickStep during instantiation. This demonstrates that a weapon can have various arts or singular art based on the instance. **Responsibilities**: The class initialises the weapon with the assigned Weapon Art and delegates to it during attacks if applicable.	presence of a Weapon Art and evaluates its conditions (e.g., mana availability). This introduces a small performance overhead, but the impact should be minimal in most scenarios. 3. **Memory Usage**: Each weapon art must be instantiated and stored for each weapon, increasing memory usage. However, given the game's requirements, this trade-off is acceptable for the flexibility gained. 4. **Coupling**: Although the design follows the SOLID principles, there is still some coupling between the player, weapon, and weapon arts due to the shared state (e.g., mana). This coupling could be reduced with further abstractions, though that would add unnecessary complexity for this requirement.

REQ2: Belurat, Tower Settlement

Design Goal

The primary goal is to implement two new areas in the game world, Belurat Tower Settlement and Belurat Sewer, with the addition of mechanics such as poison swamps and dynamic gate traversal. This aims to increase the complexity and depth of the game world while adhering to SOLID principles to maintain scalability, maintainability, and flexibility for future extensions.

UML:



Classes Added/ Modified	Roles and Responsibilities	Reasons
Application (modified)	Role: Initializes the game world, adds the new maps, and sets up gates and poison swamps for seamless map traversal. Responsibilities: ● Creating two new maps (Belurat, Tower Settlement and Belurat Sewer). ● Adding poison swamps and handling their interactions with actors. ● Setting up gates that allow map traversal between Gravesite Plain, Belurat Tower, and Belurat Sewer.	**Alternate Solution**: An alternative solution for the map transition could have been handling the gate transitions through hard-coded logic directly in the Application class or Player actions. Similarly, poison effects could have been applied directly by the PoisonSwamp class, managing damage in its tick method without the need for a separate status effect class. **Finalized Solution**: ● The **Gate** class was created to handle map transitions dynamically by allowing flexible movement actions between maps. It allows different actions to be added for each gate, enabling the Tarnished to move between **Gravesite Plain**, **Belurat Tower**, and **Belurat Sewers**. ● **PoisonSwamp** was implemented as a new type of ground that applies the **PoisonStatusEffect** to any actor entering it. This status effect manages damage-over-time, applying poison effects that stack as the actor steps on additional swamps. ● Two new maps were created: **Belurat Tower** and **Belurat Sewer**. The maps are integrated into the game world and linked through the **Gate** class, ensuring seamless traversal for the player. **Reasons (SOLID Principles)**: ● **Single Responsibility Principle (SRP)**: Each class focuses on a specific task. For example, the PoisonStatusEffect is solely responsible for handling poison-related logic, and the Gate class is responsible for managing transitions between maps. ● **Open/Closed Principle (OCP)**: The **Gate** and **PoisonSwamp** classes can be easily extended for new
Gate(created)	Role: Represents gates that players can use to transition between maps. Responsibilities: ● Allowing the Tarnished to move between the **Gravesite Plain**, **Belurat Tower**, and **Belurat Sewer**. ● Dynamically adds transition actions based on the player's location.	

PoisonSwamp (created)	Role: Represents the poison swamp ground that applies poison status effects to actors stepping on it. Responsibilities: ● Applying poison status when the Tarnished or other actors step on the swamp. ● Ensuring the poison effect stacks if multiple swamps are traversed.	types of ground or gates without modifying the existing code. If additional status effects or map transitions are required, the system can handle them without impacting the core logic. ● **Liskov Substitution Principle (LSP)**: The **Gate** class can accommodate different map transitions without breaking the system. Any future gates can be added seamlessly by implementing the same actions without modifying the existing gate logic. ● **Dependency Inversion Principle (DIP)**: The **PoisonSwamp** class doesn't directly manage the poison effect. Instead, it relies on the PoisonStatusEffect class, making the poison application and duration handling independent and easier to modify in the future. **Limitations and Trade-offs**: ● **Map Complexity**: While this design is efficient for basic map transitions, more complex future maps may require more intricate gate logic, including conditional access or environmental effects. ● **Extension Complexity**: The current poison system is simple but adding additional effects like freezing, burning, or slowing down actors may require further development of the **StatusEffect** framework, adding to the system's complexity.
PoisonStatus Effect(created)	Role: Manages the poison status effect, including damage over time and poison stack duration. Responsibilities: ● Tracking the poison's damage and duration on affected actors. ● Applying damage based on the number of swamps traversed. ● Ensuring the poison status is removed once the effect's duration ends.	

Status (modified)	Role: Manages various status effects, including the new **POISONED** status. Responsibilities: ● Tracking whether an actor is poisoned and managing the poison’s behavior. ● Providing a unified way to manage various statuses in the game.	