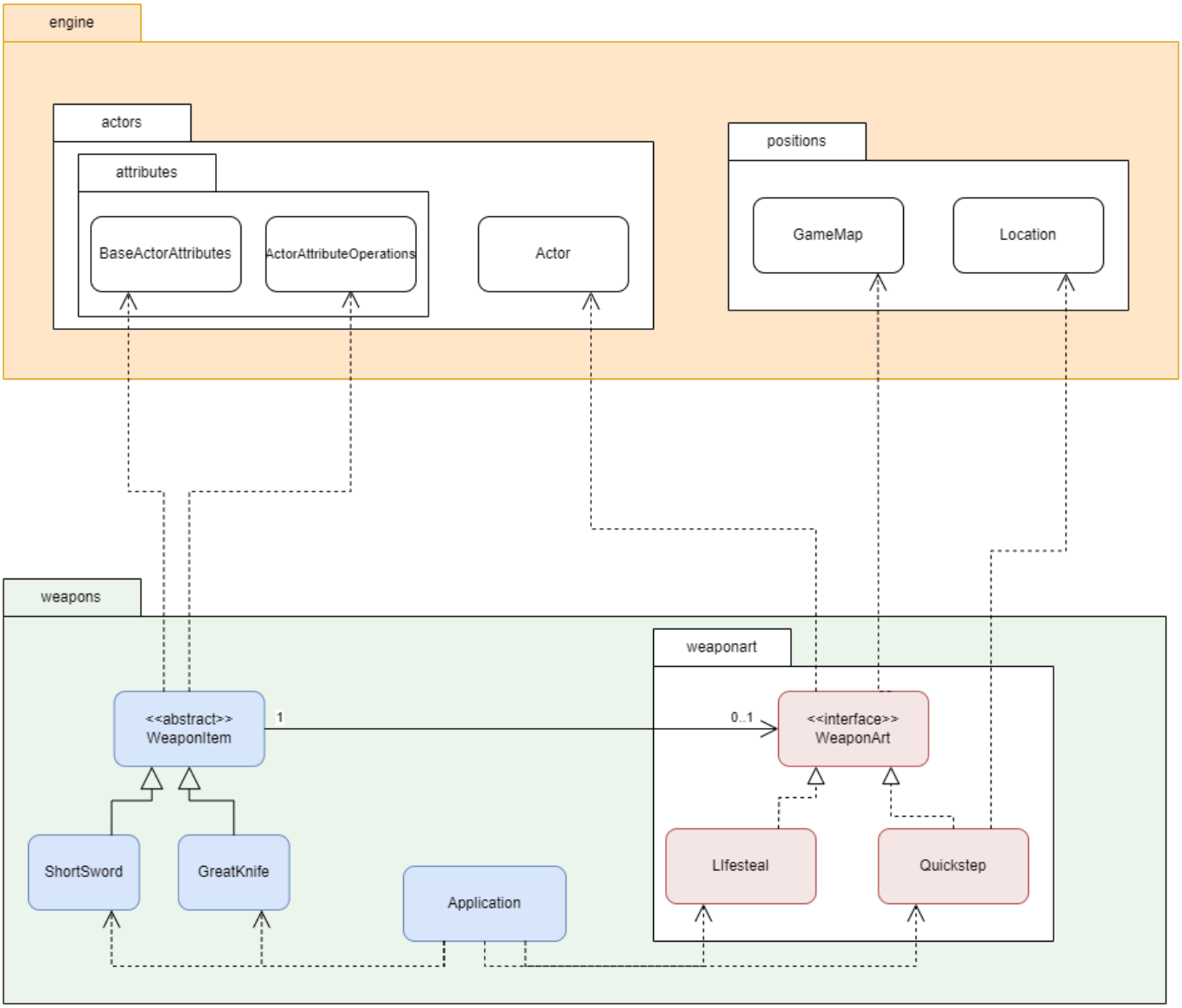
REQ 1: Weapon Arts



REQ 1 UML Class Diagram

<table>
<tr><th>Classes Modified / Created</th><th>Roles and Responsibilities</th><th>Rationale</th></tr>
<tr><td>Modified WeaponItem abstract class (extends Item, implements Weapon)</td><td>Role & Responsibility:

WeaponItem is a base class for types of items that can be used as weapons for attacking. It encapsulates the weapon behaviours from damage, hit rate and weapon arts.

Relationship:
- Extends Item
- Implements Weapon
- Has one WeaponArt for each instantiation of an implementation
- Depends with the Actor classes, GameMap and Location.
- attack() allows damage interaction between two actors, and activates a WeaponArt if it exists and mana conditions are met for the holding actor.</td><td rowspan="2">Alternate Solution based on Assignment 1 code:
A Concrete WeaponItem class with an encapsulated override of getPickUpAction that all weapon classes can inherit

<table><tr><td>Pros</td><td>Cons</td></tr><tr><td>Promotes code reuse and prevents repetition</td><td>Comes with the assumption that all weapon classes will need to use the same implementation of getPickUpAction.

This introduces tight coupling.</td></tr><tr><td></td><td>A concrete WeaponItem class can be instantiated and would introduce a loose contract which can be problematic with Liskov Substitution Principle</td></tr></table>
Finalised Solution:
The finalised solution uses the abstract WeaponItem class extending Item and implementing Weapon that encapsulates all the necessary</td></tr>
<tr><td>Modified ShortSword</td><td>Role & Responsibility:

ShortSword represents the ShortSword weapon with specific attributes such as damage, hit rate, and strength requirement.
It also determines if an actor can pick up the weapon based on their strength.

Relationship:
- Extends WeaponItem
- Overrides PickUpAction to implement a check of the Actor’s strength before allowing pick up.</td></tr>
</table>

Modified GreatKnife	**Role & Responsibility:** GreatKnife represents the Great Knife weapon with specific attributes such as damage, hit rate, and strength requirement. It also determines if an actor can pick up the weapon based on their strength.. **Relationship:** - Extends WeaponItem - Overrides PickUpAction to implement a check of the Actor's strength before allowing pick up.	common logic for interacting with a weapon child class. Inheriting from WeaponItem, both the GreatKnife and ShortSword can apply these common logic while also implementing the specific strength checking getPickUpAction method that is required for the two classes, but not necessarily for WeaponItem. **Reasons for Decision:** **Single Responsibility Principle (SRP)**: The WeaponItem abstract class is solely responsible for weapon-specific logic such as attacking, executing weapon art and returning AttackAction. The concrete implementations such as GreatKnife and ShortSword are only focused on implementing specific attributes and behaviours like the strength-checking getPickUpAction **Open/Closed Principle (OCP)**: The WeaponItem can be extended for any types of weapons without needing to be modified. New weapons can be created without alteration to core logics, promoting scalability. **Liskov Substitution Principle (LSP)**: Overriding getPickUpAction ensures that ShortSword and GreatKnife can be treated as any WeaponItem or Item object while still applying their own logic such as getPickUpAction. ShortSword and GreatKnife can also be interchangeably used for AttackAction that utilizes the Weapon interface. **Interface Segregation Principle (ISP):** The WeaponItem class implements the Weapon interface, which provides weapon-specific methods. This separates weapon-related behavior from other possible item functionalities, **Dependency Inversion Principle (DIP)**:

<table>
<tr><td></td><td></td><td>The system ensures that high level classes such as actors and other engine classes utilize abstractions such as the Weapon interface or the abstract WeaponItem class as opposed to specific implementations of weapons. This reduces coupling between the high level classes and lower level weapon classes.

DRY Principle: The centralization of common functionality such as damage, hit rate, and weapon arts avoids duplication of code across different weapon classes like ShortSword and GreatKnife, promoting reuse of common logic.

Limitations and Trade-offs:
If future concrete weapon classes utilizes the exact same override implementation of getPickUpAction, then the overall scalability of the system would be affected as the code has to be repeated multiple times. However, it is a reasonable expectation to assume that not all weapons would require a strength check to be picked up.</td></tr>
<tr><td>WeaponArt Interface</td><td>Role & Responsibility:

WeaponArt is an interface enforcing the contract for any weapon arts. It standardizes every weapon art by ensuring that they implement a use method and a manaCost method.

Relationship:
- The use method of the interfaces expects interaction with two different actors and the GameMap.</td><td rowspan="2">Alternative Solution 1:
Implement both Lifesteal and Quickstep logic into a weapon class with an attribute indicating which to activate.

<table><tr><td>Pros</td><td>Cons</td></tr><tr><td>Simplified system with reduced class count</td><td>Violates SRP as the weapon class is now responsible for behaviours beyond the scopes of weapon item</td></tr></table></td></tr>
<tr><td>Lifesteal</td><td>Role & Responsibility:

Implements the WeaponArt interface</td></tr>
</table>

<table>
<tr><td></td><td>to create a weapon art that heals the attacker by a set amount after attacking.

Relationship:
- The use method of the interfaces expects interaction with two different actors and the GameMap.</td><td>
<table>
<tr><td></td><td>logic which includes attacking and picking up.</td></tr>
<tr><td></td><td>Violates OCP as it will be harder to expand the system, requiring modification of the weapon class itself for future updates.</td></tr>
<tr><td></td><td>The code implementation would not be reusable unless repeated across all weapon classes.</td></tr>
</table>

Alternative Solution 2: Create WeaponArt interface without the manaCost() requirement. Implement the manaCost() and checking logic in Lifesteal only.

<table>
<tr><td>Pros</td><td>Cons</td></tr>
<tr><td>Simplifies the system by reducing the number of methods needed to be implemented</td><td>This breaks the expectation that most future WeaponArt implementations would require mana.</td></tr>
<tr><td>Reduced code duplication for WeaponArt</td><td>The system is not as extensible as any future</td></tr>
</table>
</td></tr>
<tr><td>QuickSttep</td><td>Role & Responsibility:

WeaponArt is an interface enforcing the contract for any weapon arts. It standardizes every weapon art by ensuring that they implement a use method and a manaCost method.

Relationship:
- The use method of the interfaces expects interaction with two different actors and the GameMap.</td><td></td></tr>
</table>

		<table><tr><td>implementations that would have a manaCost of 0.</td><td>weapon arts would need to define a duplicated logic for handling mana.</td></tr></table> **Finalised Solution:** The finalised solution utilizes the WeaponArt interface to enforce a contract and structure for any weapon arts. Lifesteal and Quickstep classes implement the interface while implementing their own unique behaviours like healing or moving the attacker on activation. This solution allows any weapon to associate with one weapon art. **Reasons for Decision:** **Single Responsibility Principle (SRP)**: The WeaponArt interface is solely responsible for defining weapon art behavior, and each concrete weapon art class like Lifesteal and Quickstep has its own distinct, single responsibility. **Open/Closed Principle (OCP)**: The system allows for any weapon art with unique implementations to be created without the need to modify any preceding clases. This makes it scalable. **Liskov Substitution Principle (LSP)**: Lifesteal, Quickstep and any future weapon arts can be used interchangeably in any system that expects a WeaponArt interface. **Interface Segregation Principle (ISP):** The WeaponArt interface is cleanly separated from other interfaces or responsibilities, ensuring that it only requires methods specific to weapon arts.

		Dependency Inversion Principle (DIP): Higher-level classes like actors depend on the abstraction of the WeaponArt interface, rather than concrete classes like Lifesteal or Quickstep. This promotes decoupling. **DRY Principle:** There is no code duplication as common logic for weapon arts are enforced from the WeaponArt interface, leaving any implementations of the interface to define their own specific logic. **Limitations and Trade-offs:** One limitation of this design is that some weapon arts, like Quickstep, have a mana cost of 0. While keeping the manaCost() method ensures consistency, having to define this for every weapon art might seem redundant. However, this consistency is beneficial for future-proofing the system with the expectation that future weapon arts will most likely come with a manaCost of more than 0.