Design Rationale REQ 1

Option 1. Using an abstract class for WeaponArt class

Pros	Cons
Shared implementation since in the future there may be more weapon arts with similar methods such as mana consumption and other features having a default implementation will reduce code duplication.	Less flexibility since abstract classes impose a more strict structure since all weapon arts will need to inherit the same base functionality, even if certain arts don't require some of the provided methods.
Partial implementation since unlike interfaces we can provide implementation to certain methods while others can be abstract and further customised in the subclasses.	Less open for extension since some weapon arts will have very different behaviours, it may require modifying the abstract class .

Option 2. Using an interface for WeaponArt class

Pros	Cons
Interfaces allow for more flexibility and extensibility since in the future if more weapon arts are added it can be done without changing existing code, an interface makes it easier to introduce new arts while keeping the existing structure the same.	Lack of code reusability, since some of the weapon arts may have similar methods and functions,an interface will not allow for default implementations since all methods are abstract.
	Repetition may also occur since some weapon arts may have the same methods in the future.

Summary:

For this REQ the design that was chosen follows option 1 where the WeaponArt class is an abstract class that is extended by two concrete classes (Quickstep and LifeSteal). Using the abstract class it has several benefits such as shared implementation, code reuse and a clear structure. It follows the single responsibility principle (SrP) by separating the common behaviour from specific weapon arts that extend from it, which allows each subclass to focus on their unique functions. It also supports the Liskov Substitution Principle (LSP), since any subclass of WeaponArt can be used wherever the abstract class is expected. The abstract class also promotes the DRY principle since common behaviours, constructors and attributes do not need to be included in the subclasses since they are able to reuse the code from the abstract class hence reducing duplicated code.