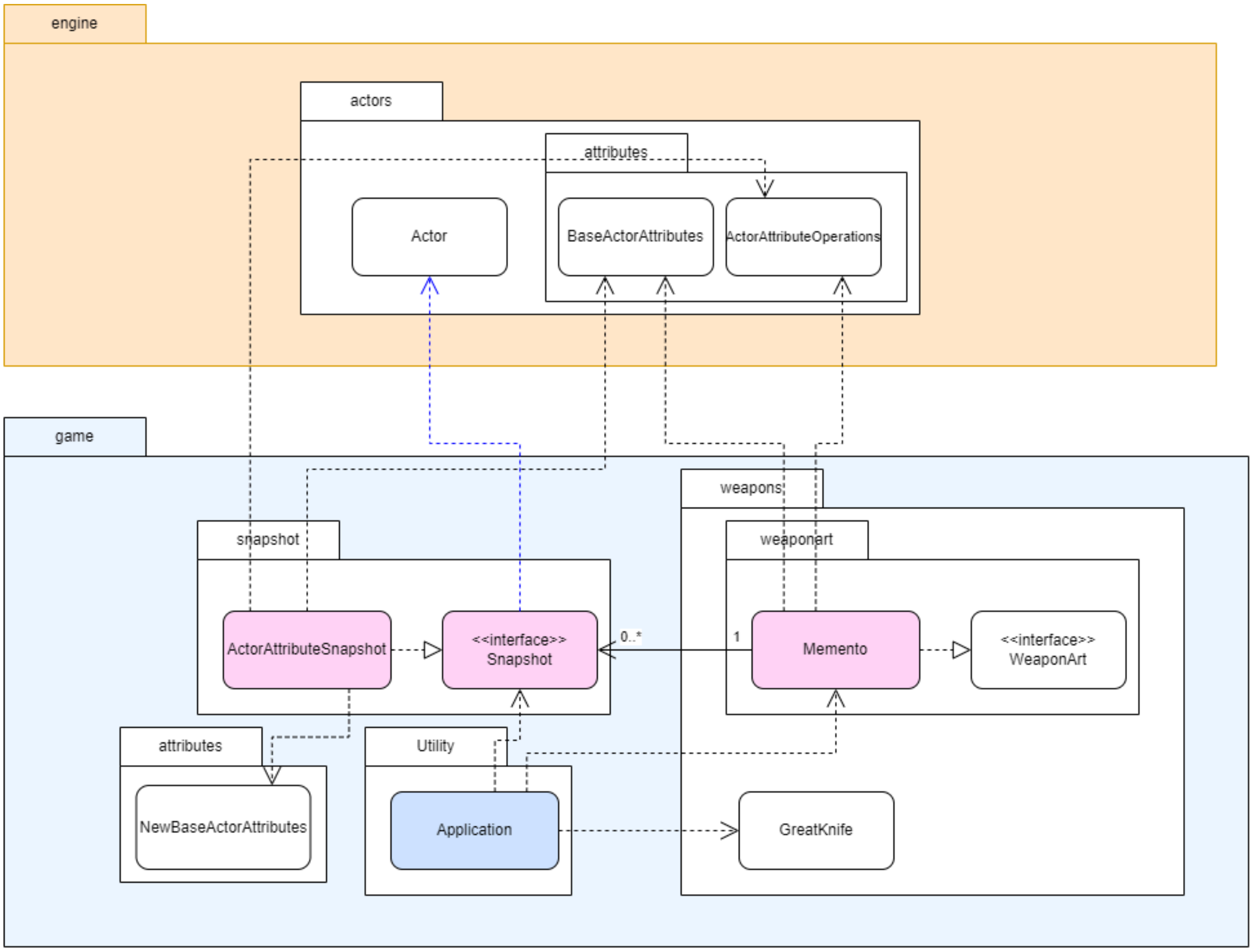
REQ 3: Memento



Pink = Class Created
Blue = Class Modified
REQ 3 UML Class Diagram

<table>
<tr><th><b>Classes Modified / Created</b></th><th><b>Roles and Responsibilities</b></th><th><b>Rationale</b></th></tr>
<tr><td>Snapshot Interface</td><td><b>Role &amp; Responsibility:</b>
The Snapshot interface defines the methods for saving and restoring an actor’s state. It standardizes a snapshot concept, allowing any class that implements it to save and restore states for various aspects related to actors.

<b>Relationship:</b>
- Acts as an interface, implemented by classes
- Defines the basic contract for saving (save(Actor actor)) and restoring (restore(Actor actor)) states for Actor classes
- Used by the Memento class to handle saving and restoring actor attributes during attacks.</td><td rowspan="2"><b>Alternative Design:</b> Create one WeaponArt implementation called PlayerSnapshot that handles all of the tasks related to snapshotting the Player Class’s attributes. It implements the Snapshot saving, restoring and and activation functionalities all in one class to save the Player’s Health, Mana and Strength specifically.

<table>
<tr><td>Pros</td><td>Cons</td></tr>
<tr><td>The design is much more simple and efficient for this specific scenario</td><td>This design is not easily extensible to other actors that may have different attributes. Violates the Open/Closed Principle</td></tr>
<tr><td>Less overhead and abstraction</td><td>Violates Single Responsibility Principle (SRP) by combining all of the functionalities into one class. This is a major code smell called God Class.</td></tr>
<tr><td></td><td>The class is tightly coupled to the Player class and its specific</td></tr>
</table>
</td></tr>
<tr><td>ActorAttributeSnapshot (Implements Snapshot)</td><td><b>Role &amp; Responsibility:</b>
The ActorAttributeSnapshot implements the Snapshot interface. it manages saving and restoring all of the Actor's attributes by creating copies of those attributes and storing them in a stack. It holds multiple snapshots of the attributes and restores the most recent one.

<b>Relationship:</b>
- Implements Snapshot
- Depends on Actor to retrieve and modify actor attributes.
- Interacts with BaseActorAttributes and NewBaseActorAttributes to handle actor attributes.</td></tr>
</table>

<table>
<tr>
<td></td>
<td>- Contains a stack (snapshotStack) that stores snapshots of actor attributes as a Map<Enum<?>, Integer&gt;.</td>
<td rowspan="2">
<table>
<tr><td></td><td>attribute, reducing the design’s adaptability and increases maintenance costs when changes occur</td></tr>
</table>

Finalised Solution:
The finalized solution uses a combination of Snapshot implementations for state-saving (ActorAttributeSnapshot for attributes) and Memento to invoke this functionality selectively.

Reasons for Decision:

Single Responsibility Principle (SRP): Each class has a clear purpose. The Snapshot defines the interface for future snapshots, while ActorAttributeSnapshot specifically handles all of the attributes an actor could be holding. The Memento handles all of hte WeaponArt related functions such as managing health and mana costs, and the activation condition for Snapshots.

Open/Closed Principle (OCP): The system can be extended with additional implementations of Snapshot if new saving/restoring needs arise, without modifying existing classes. For example, the LocationSnapshot can easily be implemented from Snapshot to store the locations of the actor’s in the GameMap.

Liskov Substitution Principle (LSP): The Snapshot interface allows any Snapshot-implementing class to
</td>
</tr>
<tr>
<td>Memento
(Implements WeaponArt)</td>
<td>**Role &amp; Responsibility:**
The Memento class implements the WeaponArt interface, allowing an actor to either "remember" (save) or "restore" a previous state during combat. It manages snapshots of an actor’s attributes and provides a chance-based mechanic for saving or restoring an actor's state while attacking. It also decreases the Actor’s Health for every successful activation.

Relationship:
- Implements WeaponArt

- Depends on the Snapshot classes to handle the actor’s attribute state.

- Interacts with Actor and GameMap to get and modify its states.

- Contains a stack (snapshotStack) that stores snapshots of actor attributes as a Map<Enum<?>, Integer&gt;.</td>
</tr>
</table>

		replace ActorAttributeSnapshot if needed, ensuring compatibility across all Snapshot implementations. **Interface Segregation Princple (ISP):** Snapshot interface is lightweight, offering only the methods directly related to its purpose. This makes it easy for other snapshot-related implementations to focus solely on save and restore operations without unnecessary dependencies. **Dependency Inversion Principle (DIP)**: Memento interacts with Snapshot through the abstract interface rather than a concrete implementation. This would make it easier to update and change the Snapshot interface or implementations without affecting Memento. **DRY Principle:** The finalized solution contains very little of duplicated code thanks to modularization of the methods. The general nature of the ActorAttributeSnapshot makes it so that there is no need to duplicate codes for different types of Actors as it will work for all of them. This design ensures that the snapshotting mechanism is handled in a single place, making it easier to maintain, update, or extend in the future if new attributes or actors are introduced, thus keeping the codebase clean and consistent. **Connascence:** **Connascence of Name:** This design effectively reduces the connascence

		of naming. The Memento Class interacts with all of Snapshot implementations through the Snapshot interface. This reduces the Connascence of names as it reduces rigid reliance on specific classes, increasing flexibility and lowering coupling. **Connascence of Name/Meaning:** Multiple classes such as Actor, ActorAttributeSnapshot and Memento all must agree on the name and meaning of values of BaseActorAttributes and NewBaseActorAttributes. This is a justified connascence as the attributes of these Enum classes are integral to how the Actor class functions. **Connascence of Algorithm:** The restore method of ActorAttributeSnapshot has a reliance of the Stack-based Last-in-First-Out order for restoring snapshots. This is unavoidable as it is an expected behavior for the game design to restore the latest snapshot taken. **Code Smells:** **God class:** By strictly adhering to the Single Responsiblity Principle, this design ensures that no class is too large and not try to handle too many different functionalities. **Long methods:** Long methods such as use() in the Memento class are effectively refactored by modularizing the functionalities into smaller methods. **Feature Envy:** There is an unavoidable feature envy in

		ActorAttributeSnapshot as it heavily relies on the methods of the Actor class. This is a balanced trade-off as it still promotes encapsulation of actual snapshotting functionalities and also because the Actor class is unmodifiable in the context of the Assignment. **Pros and Cons of the Finalized Design:** ● **Adheres to all of the SOLID principles:** This ensures that the design is flexible and maintainable to future modifications. ● **Encapsulation and Separation of Concerns:** Maintains encapsulation by keeping snapshotting logic separate from the Actor class, reducing direct dependencies. ● **Low Overall Connascence:** Lower coupling between classes through the use of interfaces enhances flexibility and adaptability. ● **Modular and Extensible:** Code modularization minimizes duplicated code, ensuring ease of updates and enhancements. **Cons:** ● **Performance Overhead:** The nature of the snapshot storage in a stack can be problematic in an edge case where snapshots are continuously stored and not restored. This could slow down the overall system.