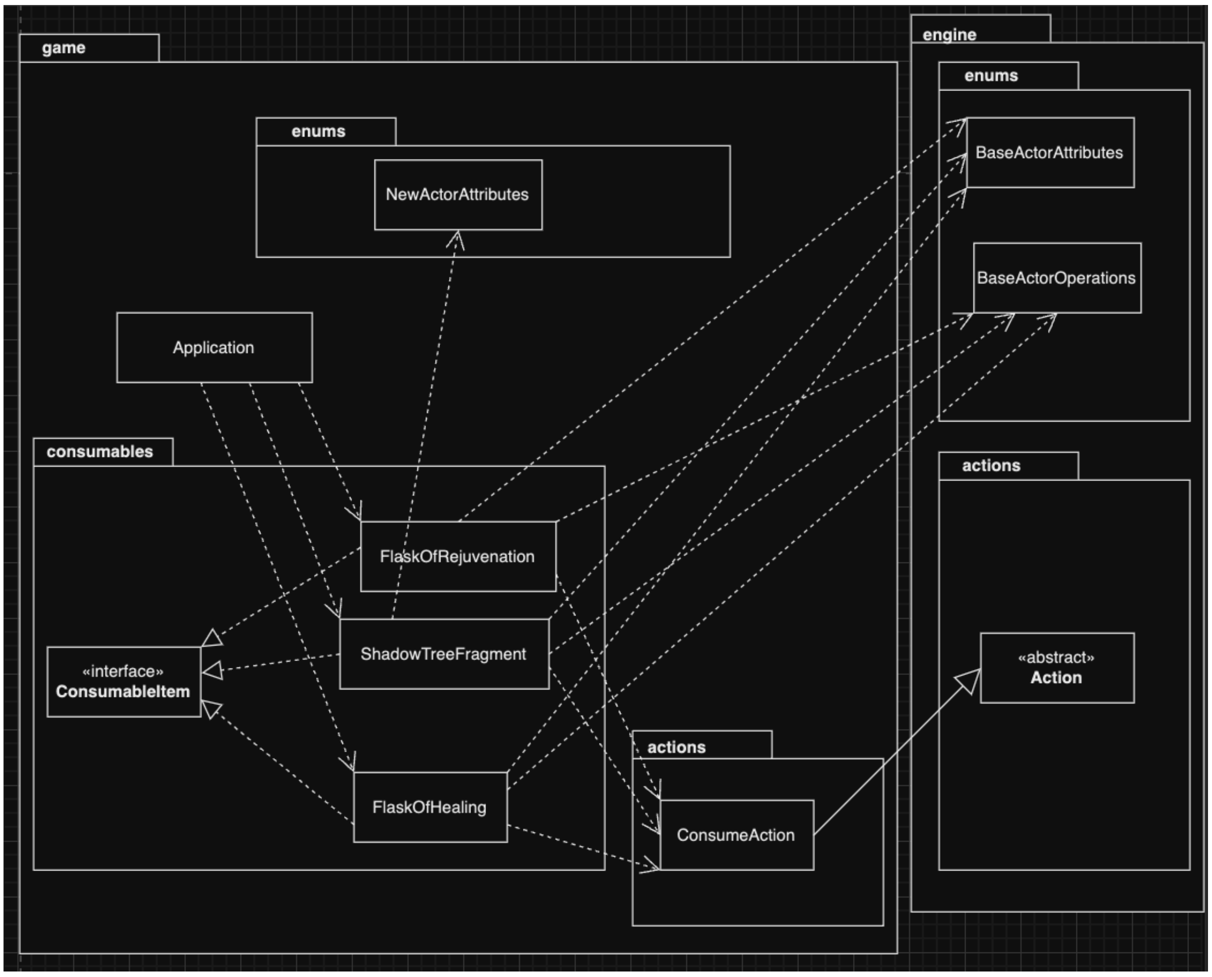
REQUIREMENT 2



Roles and Responsibilities of the Classes:

1. ConsumableItem Interface
   - Responsibility: Acts as a contract for any item or object that can be consumed. This ensures that every consumable will provide the consume method.
   - Justification: This design follows the *Interface Segregation Principle* (ISP) by allowing different consumables to define their own specific behaviour when consumed, without the need for unnecessary functionality. It makes the system extendable for future "consumable" objects.
2. FlaskOfHealing and FlaskOfRejuvenation
   - Responsibility: Encapsulate the behaviour of their respective flasks (healing or rejuvenating). Both classes keep track of their charges and provide feedback when charges are depleted.

    - Justification: These classes follow the *Single Responsibility Principle* (SRP) by focusing only on how their respective flask behaves when consumed. The logic for maintaining the charge count and providing feedback on depletion is contained within each class, ensuring cohesive and maintainable code.
3. ShadowTreeFragment
    - Responsibility: Provides a one-time consumable item that boosts multiple attributes of the Tarnished. Once consumed, it disappears from the inventory.
    - Justification: Similar to the flasks, this class adheres to SRP by focusing solely on its unique effect when consumed. It ensures the fragment is removed from the inventory upon use, which also adheres to real-world expectations (i.e., once consumed, it's gone).
4. ConsumeAction
    - Responsibility: Implements the action of consuming an item. It is responsible for invoking the correct consume method from the ConsumableItem interface.
    - Justification: Following the *Open/Closed Principle* (OCP), this class can work with any future consumables that implement the ConsumableItem interface, without the need to modify the ConsumeAction itself.

Interactions Between Classes:

- Player and ConsumableItem:
    - The Player can interact with any item that implements the ConsumableItem interface, allowing for seamless use of flasks, fragments, or any future consumable objects.
    - Upon choosing to consume a ConsumableItem, the ConsumeAction invokes the consume method. Depending on the item, it can heal, increase stats, or do nothing (if the item is out of charges).
- FlaskOfHealing & FlaskOfRejuvenation Logic:
    - The logic inside each flask class manages the charge count. Once a player uses up the charges, the flask does not vanish but instead informs the player that the flask is empty. This ensures that the flasks continue to exist in the game world while their consumable effect is limited.
- ShadowTreeFragment:
    - Unlike the flasks, the fragment is consumed only once and permanently removed from the player's inventory upon consumption. This difference is encapsulated in the consume method of the ShadowTreeFragment class, which modifies the player's attributes and removes itself from the inventory.

Coupling and Cohesion:

- Low Coupling: The ConsumableItem interface provides a flexible contract for any class implementing it. The Player and ConsumeAction interact with the ConsumableItem polymorphically, meaning that new consumable items can be introduced without changing these classes. This reduces dependencies between modules and improves maintainability.
- High Cohesion: Each class has a single responsibility, and all methods and attributes within a class are directly related to its purpose. For instance, FlaskOfHealing focuses entirely on managing healing and charges, while ConsumeAction is only concerned with carrying out the consumption process. This separation of concerns ensures that each class is easy to understand and modify.

Adherence to SOLID Principles:

1. Single Responsibility Principle (SRP): Each class has a single, well-defined responsibility. FlaskOfHealing only deals with healing, FlaskOfRejuvenation with mana restoration, and ShadowTreeFragment with increasing max stats. The ConsumableItem interface abstracts the consumption behaviour, leaving each specific class to implement its own logic.
2. Open/Closed Principle (OCP): New types of consumable items can be added without modifying existing code. For instance, if a new item (e.g., a poison flask) is introduced, it would simply need to implement the ConsumableItem interface, and the ConsumeAction class would still work with it. On of the key things here is that rather than putting everything into a ConsumableItem class I instead split it into a ConsumeAction class and a ConsumableItem interface. As such being able to consume something does not necessarily have to be from an item. If I wanted to add a feature where you could consume water from a puddle that could now be done.
3. Liskov Substitution Principle (LSP): Any consumable item (e.g., FlaskOfHealing, FlaskOfRejuvenation, or ShadowTreeFragment) can be used interchangeably where a ConsumableItem is expected. This promotes flexibility and allows future expansion.
4. Interface Segregation Principle (ISP): The ConsumableItem interface provides only the methods relevant to consumable items, ensuring that classes implementing this interface are not burdened with unnecessary methods.
5. Dependency Inversion Principle (DIP): High-level modules (such as the Player) depend on abstractions (ConsumableItem), not on specific implementations (e.g., FlaskOfHealing or ShadowTreeFragment). This decoupling allows for easier expansion and testing.

Scalability and Future Extensions:

- Future Consumables: The design is highly extendable. If more consumables are added (e.g., poison, stamina-restoring potions, or buffs), they simply need to implement the ConsumableItem interface and define their specific behaviour in the consume method.

- Effects Beyond Healing: Since the ConsumableItem interface doesn’t dictate the effects of consumption, it allows for flexible effects in the future (e.g., poisoning, slow healing over time, or boosting strength temporarily).
- Non-Item Consumables: Since the ConsumableItem interface can be implemented by any class, the system allows for non-item consumables (e.g., environment-based consumption like drinking from a fountain). This opens up the design for future enhancements without violating current design principles.