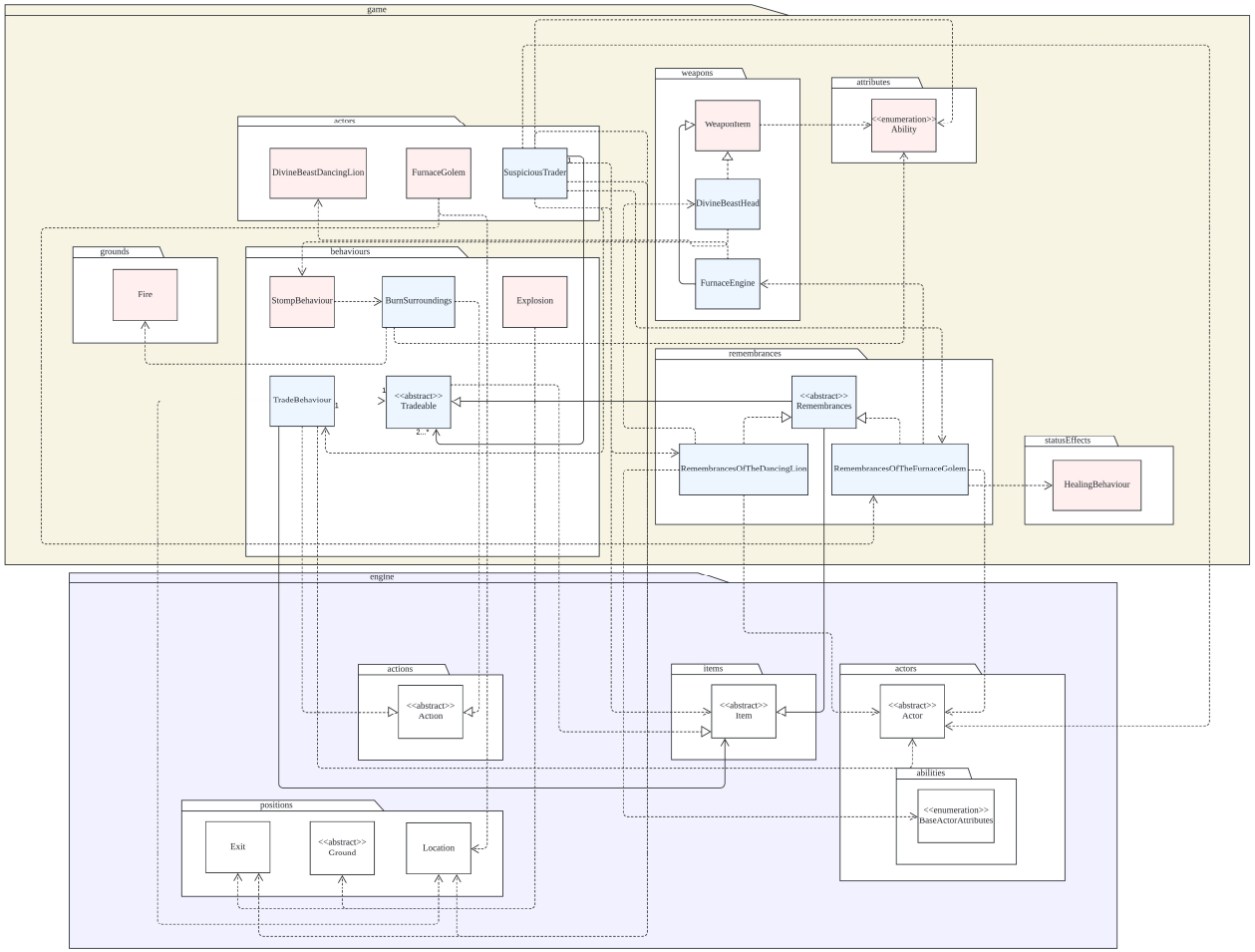
Req 2 – Remembrance of a Legend

UML diagram:



<table>
<tr><th>Class modified / Created</th><th>Roles and Responsibilities</th><th>Rationale</th></tr>
<tr><td>Remembrances
(extends Tradeable)</td><td>Abstract class showing Remembrances interface

Relationships:
1. Extends “Tradeable” to gain “Tradeable” methods like “applyTradeEffect” and “getTradeItem”</td><td rowspan="4">Alternate Solution:
Create a “Remembrances” concrete class to implement logic details of remembrance of the dancing lion and remembrance of the furnace golem into one single class.

Finalized Solution:
Instead of combining three remembrance classes into one solid class, create “Remembrances” abstract that extends “Tradeable”. Create “RemembranceOfTheDancingLion” and “RemembranceOfTheFurnaceGolem” that inherits “Remembrance” then implement their respective logic details.

Reason for decisions:
1. Single Responsibility Principle: By separating two different concrete class “RemembranceOfTheDancingLion” and “RemembranceOfTheFurnaceGolem” that inherits “Remembrance”, it can ensure that each class has its own responsibility. Moreover, it is more</td></tr>
<tr><td>RemembrancesOfTheDancingLion
(extends Remembrances)</td><td>Class showing Remembrance of the Dancing Lion

Relationships:
1. Extends “Remembrance” abstract class to show “Remembrance” properties</td></tr>
<tr><td>RemembrancesOfTheFurnaceGolem
(extends Remembrances)</td><td>Class showing Remembrance of the Furnace Golem

Relationships:
1. Extends “Remembrance” abstract class to show “Remembrance” properties</td></tr>
<tr><td>Tradeable
(extends Item)</td><td>Abstact class showing tradeable items</td></tr>
</table>

	Relationships: 1. Extends “Item” abstract class to show “Item” properties	scalable, and it will not bring complexity to the code when large amount of remembrances are added. 2. **Open-closed Principle:** When more remembrances are introduced, the new remembrance can be added and extended without modifying the class itself. For example, “Remembrance” abstract class provides a base implementation for remembrance-related functionality. By creating subclasses “RemembranceOfTheDancingLion” and “RemembranceOfTheFurnaceGolem”, it can add them without modifying the existing “Remembrance” class. 3. **Liskov substitution Principle:** “RemembranceOfTheDancingLion” and “RemembranceOfTheFurnaceGolem” classes extends the “Remembrance” and provides specific implementations for different remembrance types. They inherit the
TradeBehaviour (extends Action)	Class showing trading actions Relationships: 1. Extends Action abstract class to have an “execute()” method and “menuDescription()” method to ensure that the consume action is executed and shows in the menu	
SuspiciousTrader (extends Actor)	Class showing the SuspiciousTrader Relationships: 1. Extends Actor abstract class to have “playTurn” and “allowableActions” to make “TradeBehaviour” can be implemented to this actor 2. Depends on the “IMMUNE” enum from “Ability” to ensure that the “Player” cannot attack the trader	
FurnaceEngine (extends WeaponItem)	Class showing the Furnace Engine weapon	

	Relationships: 1. Extends WeaponItem to have weapon item properties	base functionality from "Remembrance" and can be used interchangedbly with "Remembrance" without affecting the program. 4. Dependency Inversion Principle High-level modules like the SuspiciousTrader class depend on the Remembrances abstract class rather than specific Remembrances implementations. This helps in reducing dependency on each and every specific remembrance so that the SuspiciousTrader will not be affected by changes in specific remembrances (low-level modules) in the future. 5. **Don't Repeat Yourself:** The execute method in "Explosion" method has already been implemented. "FurnaceGolem" and "FurnaceEngine" can just call the execute method of "Explosion", avoid implementing it again which adheres the DRY principle.
DivineBeastHead (extends WeaponItem)	Class showing the Furnace Engine weapon Relationships: 1. Extends WeaponItem to have weapon item properties	
BurnSurroundings (extends Actions)	Class showing burning grounds Relationships: 1. Extends Action abstract class to have an "execute()" method and "menuDescription()" method to ensure that the consume action is executed and shows in the menu	

		Limitations and Tradeoffs: 1. The use of multiple interfaces and abstract classes can increase the complexity of the codebase, making it harder to understand and maintain 2. Debugging issues in a highly abstracted and modular codebase can be more complex, it requires a deep understanding of the interactions between different components. 3. The use of multiple layers of abstraction (interfaces, abstract classes) can introduce overhead when understanding and navigating the codebase. **Connascence:** 1. Connascence of Name: The design minimizes connascence of name by ensuring that class names and method names are meaningful and consistent. For example, the method applyTradeEffect is consistently

		named across different Remembrances. 2. Connascence of Type: The design uses strong typing to ensure that objects are used correctly, reducing the risk of type-related errors. For instance, the applyTradeEffect method expects an Actor type, ensuring that only valid actors can use this method. 3. Connascence of Meaning: The design ensures that the meaning of data is consistent across the system, reducing the risk of semantic errors. For example, the RemembranceOfTheFurnaceGolem consistently represents the item dropped by the Furnace Golem. 4. Connascence of Position: The design avoids connascence of position by using named parameters and avoiding positional dependencies. For example, the constructor of WeaponItem uses named parameters to avoid confusion about the order of arguments. **Code Smells:**

		1. Long Methods: The design avoids long methods by breaking down functionality into smaller, more manageable methods. For example, the attack method in WeaponItem is concise and focused on a single responsibility. 2. Large Classes: The design avoids large classes by adhering to the Single Responsibility Principle and ensuring that each class has a single responsibility. For instance, RemembranceOfTheFurnaceGolem only handles the specific effects of trading that remembrance. 3. Duplicated Code: The design avoids duplicated code by abstracting common functionality into base classes and using inheritance and polymorphism. For example, the applyTradeEffect method is defined in the base class and overridden in subclasses. 4. Feature Envy: The design avoids feature envy by ensuring that methods operate on the data within their own class, rather than accessing

		data from other classes. For example, the attack method in WeaponItem operates on the weapon's own attributes.