Design Rationale:

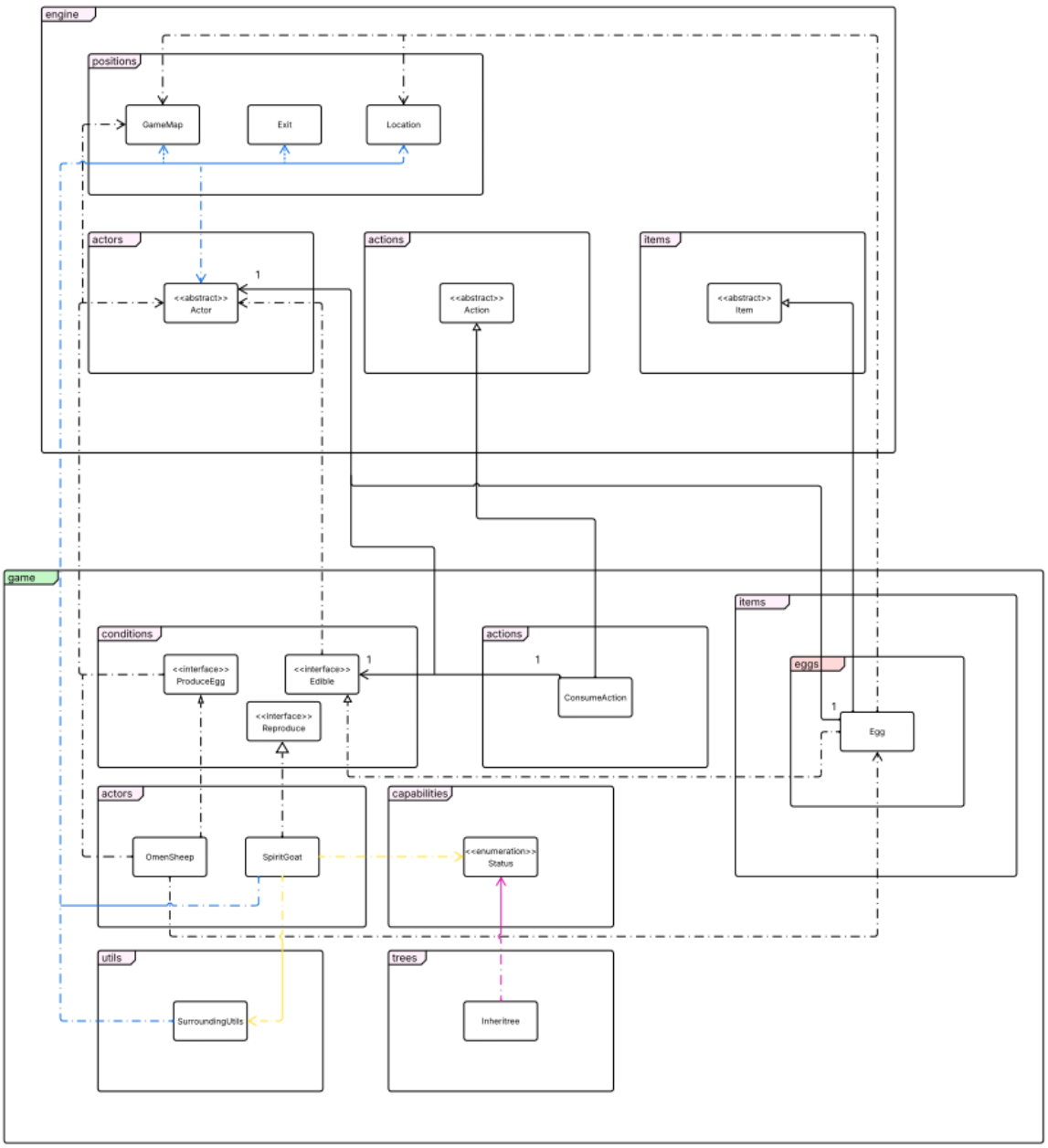
REQ 1:

Design 1: Hardcoded logic for specific entities where spirit goat explicitly checks for Inheritree instances and Egg logic is tied to Omen Sheep

Pros	Cons
Simple to implement initially as it strays from complex concepts like abstraction	Tight coupling as Spirit Goat depends on Inheritree which when modified, requires modification of Spirit Goat
Reproduction logic is unambiguous	Code duplication as adding new entities with grace would mean modifying Spirit Goat
Straightforward implementation as inventory checks are self-contained in Egg	No abstraction for consumption

Design 2: Abstraction driven architecture with Spirit Goat checking surroundings for a capability type and Egg has unique hatching behaviours

Pros	Cons
Can work with multiple entities that have grace without the direct modification of Spirit Goat	Slightly more complex due to more abstraction layers
Edible interface allows for future edible items	
Eggs can be used for other creatures	



The diagram above shows the implementation of requirement 1 using design 2, in which abstract classes and interfaces are used to encapsulate shared logic for reproduction, consumption and interactions with surroundings. Interfaces such as Reproduce and ProduceEgg provide a guide for entities that spawn offspring or eggs. This design allows for both scalability and abstraction, which adheres to the Interface Segregation Principle by ensuring that classes implement relevant methods. The Edible interface supports the Single Responsibility Principle as Edible handles consumption logic and SurroundingUtils handles a generalised surrounding check. The Egg class utilises the Supplier<Actor> to decouple hatching mechanics for specific creature types which

allows for more diverse creatures to spawn in the future through dependency injetion. This approach allows for the minimisation of code duplication and ensures that the system adheres to the Open/Closed principle as new grace-blessed interactions or hatching conditions can be added without modification of existing code. This was further shown with the usage of SurroundingUtils in Spirit Goat which dynamically checks grace-blessed entities, enabling the Spirit Goat reproduction to adapt to any other enumeration. Overall, this design was chosen as it ensures scalability, and aligns with the best practices for a maintainable and extensible system