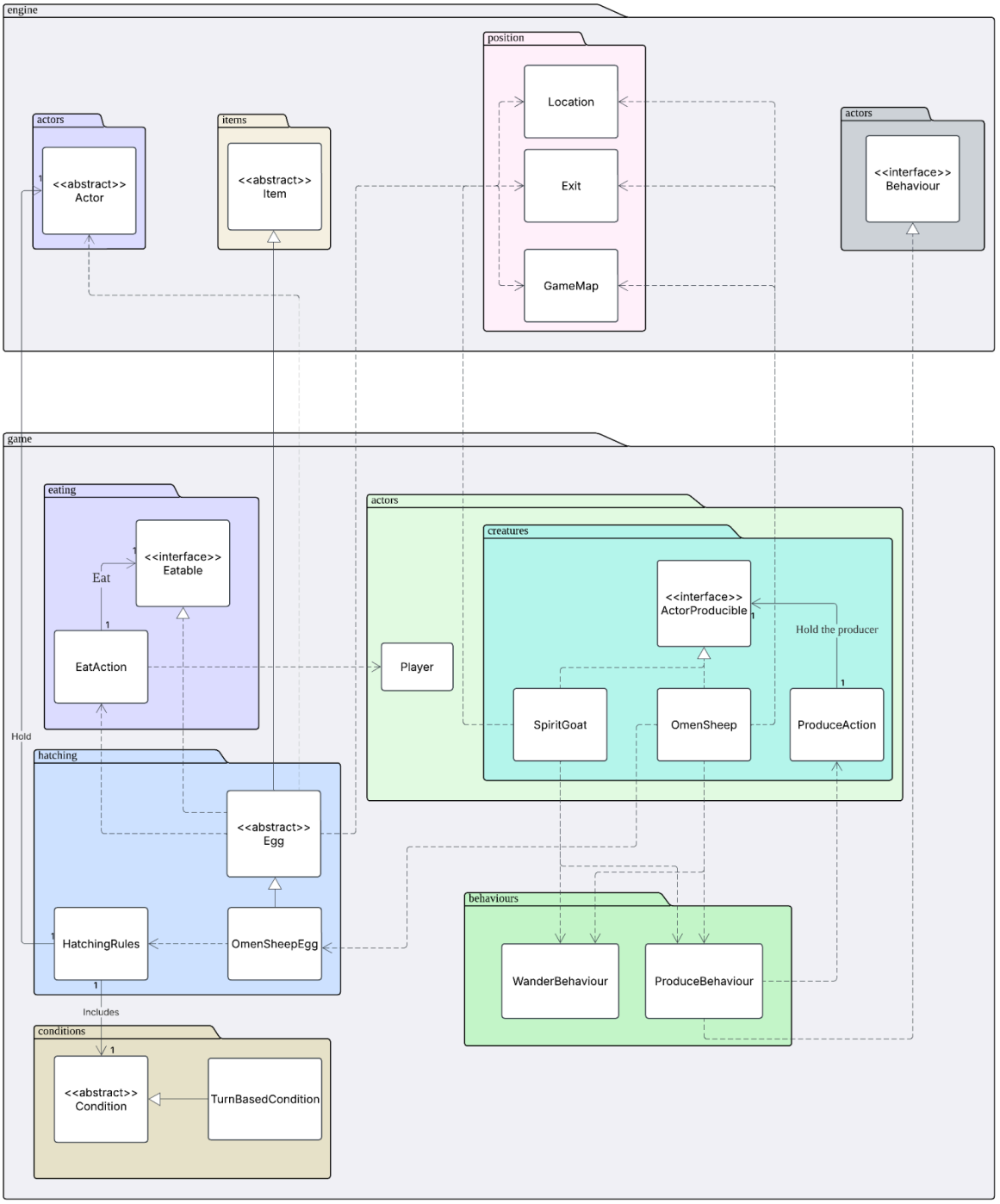
Design Rationale: Creature Offspring and Interactions



1. Introduction

This document outlines the implementation of creature offspring production and interactions, focusing on Spirit Goats, Omen Sheep, Eggs, and the Farmer. It emphasizes future extensibility and reusability, aiming for a robust, flexible system.

2. Design Goals

- Implement ActorProducible for Spirit Goat reproduction near "blessed" entities and Omen Sheep periodic OmenSheepEgg production.
- Implement OmenSheepEgg mechanics: timed hatching (on ground), pickup, and consumption (Eatable, EatAction).
- Promote Extensibility & Reusability for new producers, grace sources, eatables, and hatching conditions.

3. Design Analysis & Decisions

1. Creature Production Logic Implementation

This explains how creatures like OmenSheep (timed item production) and SpiritGoat (conditional actor production) manage their production logic.

Option A: Full production logic in playTurn()

- **Description:** All production logic (timers, condition checks, creation, map addition) resides directly in each creature's playTurn(). No new interfaces/classes.

- **Maintainability:** Difficult. Changes require modifying playTurn().

Pros	Cons
Simple for basic rules, with logic included in creature's state.	Violates SRP by mixing production, movement & combat logic in playTurn().
	Violates OCP by adding new creature with different rules.

Option B: Trigger production logic via Behaviours/Actions with ActorProducible interface

- **Description:** Creatures implement the ActorProducible interface with methods like canProduceOffspring() and produceOffspring() containing production logic. A produce behaviour checks if production is possible, and a produce action executes it. OmenSheep counts turns to produce an egg, while SpiritGoat checks for "blessed" ground to spawn a goat.

- **Maintainability:** More maintainable than Option A and clean. Changes for production rules are localized in ActorProducible methods.

Pros	**Cons**
Adheres to SRP, ProduceBehavior handles triggering, while creatures manage production rules. ProduceBehaviour also promotes DRY.	Introduces more classes

ActorProducible's capability of defining production logic across creatures aligns with ISP.	

Decision & Justification

- **Chosen Design:** Option B.
- **Rationale:** Option B is chosen over Option A for its improved adherence to SOLID and DRY principles. By using Behaviour and Action components for triggering and ActorProducible for production logic, it achieves better separation of concerns, decoupling, and extensibility. For example, if a new creature like a SilverMoth produces a SilverEgg every 2 turns, it can implement ActorProducible without modifying the core production mechanism, demonstrating OCP. New producible creatures with unique logic can implement the interface without modifying the core production mechanism, supporting future growth.

2. Egg Hatching Logic

This feature explains how the egg (OmenSheepEgg as an Item) manages its hatching process (3 turns, on ground condition) to produce a creature (OmenSheep as an Actor).

Option A: Logic Embedded in Egg.tick() with HatchingRules & Condition

- **Description:** The abstract Egg class manages hatching in tick(Location), called when the egg is on the ground. Egg uses HatchingRules (pairing a Condition with a creature supplier). The Condition interface defines hatching conditions (OmenSheepEgg uses TurnBasedCondition for 3 turns). If a condition is met, HatchingRules triggers hatching.

- **Maintainability:** Changes to hatching conditions are localized to new Condition implementations, enhancing maintainability.

Pros	**Cons**
Leverages Item.tick() for integration.	Moderate SRP violation (Egg manages multiple roles: item, eatable, hatching).
Condition interface enables flexible hatching rules (OCP, DRY)	

Option B: Logic Hardcoded in Egg subclasses tick()

- **Description:** Hatching logic is duplicated in each egg subclass (e.g., OmenSheepEgg) without abstractions like HatchingRules or Condition. Each subclass independently implements its hatching logic in its tick() method.

- **Maintainability:** Poor. Any change to hatching logic requires modifying each subclass, leading to duplicated code and increased risk of errors.

Pros	Cons
Simpler for a single egg type by avoiding abstractions	Violates DRY (hatching logic duplicated). Violates OCP (new conditions require subclass modification).

Decision & Justification

- **Chosen Design:** Option A.
- **Rationale:** Option A provides a robust, extensible solution by embedding logic in Egg.tick() while using HatchingRules and the Condition interface for flexibility (a BirdEgg hatching near a river can use a new Condition without modifying Egg). This adheres to OCP (varied conditions without modifying Egg) and promotes DRY (reusable conditions). It meets the OmenSheepEgg requirement (3 turns on ground, resets in inventory) and supports future maintainability for new egg types and conditions with minimal impact.

3. Consumption Design

This feature explains how the action of eating an Eatable entity (OmenSheepEgg) is implemented, allowing the Player to consume the entity (egg) and apply the effects (increase 10 points max health). The design must support varied consumption effects and the consideration that other entities (not just items) might be eatable, ensuring flexibility and extensibility.

Option A: Specific Action Classes per Eatable Type

- **Description:** Separate action classes are created for each eatable type (like EatSheepEggAction, EatGoldenBeetleEggAction), each directly handling the specific effects of its type.

- **Maintainability:** Poor. Adding new eatable types requires creating new action classes, leading to code duplication and increased maintenance effort.

Pros	**Cons**
Simpler internal logic for uniquely complex effects.	Violates DRY (duplicated logic). Violates OCP (new types require new action classes).

Option B: EatAction using Eatable Interface

- **Description:** A single EatAction works with any object implementing Eatable, which defines consumption execution

methods. (Player eats OmenSheepEgg, EatAction increases Player's max health and removes the egg). This design leverages polymorphism, allowing EatAction to work with any Eatable entity.

- **Maintainability:** Very high. Adding new Eatable entities or modifying their effects only requires implementing or updating the Eatable interface, leaving EatAction unchanged.

Pros	**Cons**
EatAction handles all Eatable entities (DRY)	Eatable interface needs to be general enough for varied effects.
New Eatable entities align with OCP.	

Decision & Justification

- **Chosen Design:** Option B
- **Rationale:** Option B is chosen because it provides a clean, OO solution by using a generic EatAction with the Eatable interface, strongly aligning with fundamental design principles like OCP and DRY (A new eatable RotCrab granting 20 HP can implement Eatable without modifying EatAction). It leverages polymorphism for flexible interaction with any Eatable item, offers clear contracts, and provides sufficient flexibility for defining unique consumption effects.

4. Conclusion

The design for Requirement 1 implements creature offspring and interactions, prioritizing robustness and principles like SRP, OCP, and DRY. Production uses ProduceBehaviour and ActorProducible, consumption leverages EatAction with Eatable, and hatching employs HatchingRules with Condition for OCP-compliant condition management. This ensures a cohesive, extensible system.