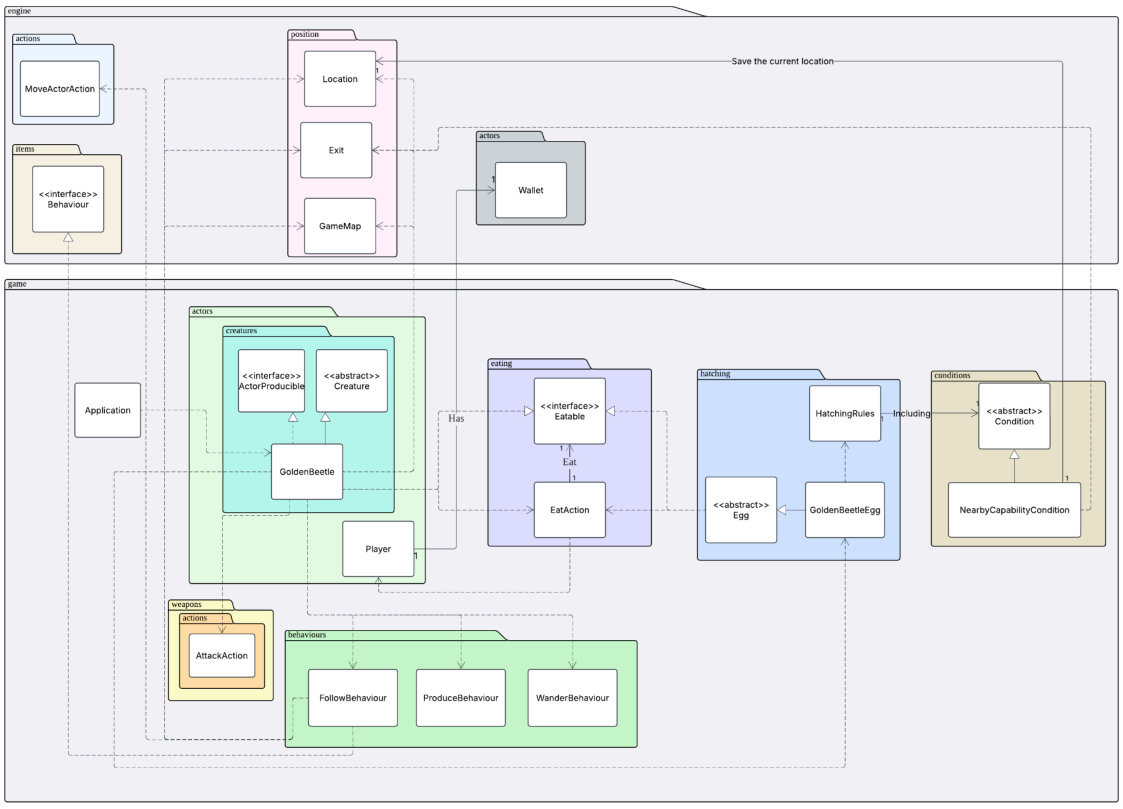
Revised Design Rationale: Golden Beetle and Interactions



1. Introduction

This document outlines the design rationale for implementing the Golden Beetle and its interactions in the game world. It builds on REQ1’s systems, addressing design considerations for extensibility and reusability, aiming for a robust, modular system.

2. Design Goals

- Implement GoldenBeetle Creature with 25 HP, rot immunity, 5-turn GoldenBeetleEgg production, Player consumption (healing 15 HP, granting 1000 runes), wandering/following(egg production prioritized), and Player attack support.
- Implement GoldenBeetleEgg as an Eatable Egg, hatching near cursed entities (Blight), with Player consumption restoring 20 stamina.
- Integrate systems (Eatable, ActorProducible, Player's Wallet) to promote extensibility for varied hatching conditions, eatable entities, and reusable capability checks.

3. Design Analysis & Decisions

1. Golden Beetle Egg Production

How the GoldenBeetle (an Actor) manages its timed production of GoldenBeetleEgg (an Item). GoldenBeetle implements the ActorProducible interface.

Option A: Logic in GoldenBeetle's ActorProducible Methods

- **Description:** GoldenBeetle implements ActorProducible. Its interface methods will check a 5-turn timer. Its canProduceOffspring() method creates a GoldenBeetleEgg and places it on the map. Using ProduceBehaviour, it calls canProduceOffspring() hen true and returns ProduceAction.

- **Maintainability:** Good. Changes to production rules are localized within GoldenBeetle.

Pros	Cons
ActorProducible, ProduceBehaviour, and ProduceAction improve SRP and DRY through reusability.	ActorProducible producing an Item as "offspring" requires clear documentation. Timer logic in GoldenBeetle limits reusability.

Option B: Production Logic in GoldenBeetle.playTurn()

- **Description:** All logic (5-turn timer, GoldenBeetleEgg creation, adding to map) would reside directly in GoldenBeetle.playTurn(). The ActorProducible interface might be trivial or unused for initiation.

- **Maintainability:** Poor. playTurn() becomes complex and brittle.

Pros	Cons
Simple for a single producing creature.	playTurn() handles movement and production which violates SRP.
	Low cohesion where production logic mixed with other concerns.

Decision & Justification

- **Chosen Design:** Option A.
- **Rationale:** Option A decouples production using ProduceBehaviour and ProduceAction, adhering to SRP. GoldenBeetle's ActorProducible implementation supports extensibility (a SilverMoth producing a SilverEgg every 3 turns can implement ActorProducible without changes (OCP)).

2. Consumption Mechanics for GoldenBeetle and GoldenBeetleEgg

The Player can consume an adjacent GoldenBeetle directly from the map (healing 15 HP, granting 1000 runes) or a GoldenBeetleEgg (extending Egg) from inventory (restoring 20 stamina). Both GoldenBeetle and GoldenBeetleEgg implements Eatable.

Option A: Generic EatAction using Eatable Interface

- **Description:** GoldenBeetle and GoldenBeetleEgg implement Eatable, defining effects (GoldenBeetle heals 15 HP, adds 1000 runes, GoldenBeetleEgg restores 20 stamina). Generic EatAction handles consumption for any Eatable entity.
- **Maintainability:** High. Effects are confined to each entity's Eatable

methods, keeping EatAction generic.

Pros	Cons
EatAction handles all Eatable entities, ensuring DRY.	Eatable methods must support diverse effects, potentially complicating debugging for complex logic.
New entities can implement Eatable without modifying EatAction, aligning with OCP and ISP.	

Option B: Specific Action Classes per Eatable Type

- **Description:** Specific actions (ConsumeGoldenBeetleAction, EatGoldenEggAction) handle consumption for GoldenBeetle and GoldenBeetleEgg, directly defining effects without Eatable.

- **Maintainability**: Poor. Multiple new action classes have to be created for new Eatable entities.

Pros	Cons
Specific actions simplify initial logic for each type.	Violates DRY and OCP; new eatable entities require new action classes.

Decision & Justification

- **Chosen Design:** Option A.
- **Rationale:** Option A uses EatAction with the Eatable interface, promoting DRY and OCP by enabling polymorphic handling of any Eatable entity, supporting the design consideration of eatable

entities beyond items (a RotCrab granting 20 HP can implement Eatable without modifying EatAction). Varied effects (like healing, stamina restoration) are encapsulated, ensuring extensibility.

3. **Golden Beetle Following Player**

GoldenBeetle can wander and follow the Player using FollowBehaviour. GoldenBeetle adds FollowBehaviour to its behaviours.

Option A: Following Logic in GoldenBeetle.playTurn()

- **Description:** GoldenBeetle.playTurn() will prioritize ProduceBehaviour initially, then scan map locations for the Player (FOLLOWABLE), then directly call MoveActorAction moving to an adjacent location.

- **Maintainability:** Poor. Hardcoded logic is rigid and non-reusable.

Pros	**Cons**
Simpler for a single creature with fixed logic.	Mixes concerns in playTurn() (poor SRP). Non-reusable logic (poor DRY

Option B: GoldenBeetle FollowBehaviour Logic

- **Description:** GoldenBeetle adds FollowBehaviour to its behaviours. Creature.playTurn() iterates behaviours by priority, with ProduceBehaviour taking precedence. FollowBehaviour identifies a followable target (Player with GeneralCapability.FOLLOWABLE).

- **Maintainability:** Good. Following logic is isolated.

Pros	**Cons**
FollowBehaviour encapsulates logic, ensuring high cohesion.	Requires map access and target logic, adding slight complexity.

Reusable (DRY), FollowBehaviour can be used by other actors.	

Decision & Justification

- **Chosen Design:** Option B.
- **Rationale:** Option B is a strong design due to using a dedicated FollowBehaviour, as implemented and used by GoldenBeetle (A LoyalWolf can reuse FollowBehaviour for a new target without changes). It adheres to SRP and is reusable (DRY). This significantly improves maintainability and flexibility over embedding logic directly.

4. GoldenBeetleEgg Hatching Logic

GoldenBeetleEgg hatches into a GoldenBeetle only when near "cursed entities" (Blight). GoldenBeetleEgg will extend Egg.

Option A: Hardcoded Logic in GoldenBeetleEgg.tick()

- **Description:** GoldenBeetleEgg overrides Egg's tick() hatching logic, directly implementing the "cursed environment", without any common rules and conditions.
- **Maintainability:** Poor for evolving hatching rules, as changes require modifying GoldenBeetleEgg.

Pros	Cons
Simple for a single, fixed condition with direct implementation.	Check logic cannot be reused (poor DRY). Hardcoding violates OCP, new conditions require modifying GoldenBeetleEgg.

Option B: HatchingRules with Condition in tick()

- **Description:** GoldenBeetleEgg extends Egg and uses a list of HatchingRules, each with a Condition. A NearbyCapabilityCondition checks for CURSED entities (Blight) at or adjacent to the egg's location.

- **Maintainability:** High for managing diverse hatching rules.

Pros	Cons
Supports OCP with new conditions	HatchingRules adds abstraction.
NearbyCapabilityCondition is reusable (DRY).	

Decision & Justification

- **Chosen Design:** Option B.
- **Rationale**: Option B provides a robust, extensible solution by embedding logic in GoldenBeetleEgg.tick() while using HatchingRules and the Condition interface for flexibility. GoldenBeetleEgg uses HatchingRules with NearbyCapabilityCondition. NearbyCapabilityCondition's reusability promotes DRY, fulfilling the design consideration for reusable capability checks.

4. Conclusion

This design integrates the GoldenBeetle using interfaces and behaviors: GoldenBeetle implements ActorProducible for 5-turn egg production, Eatable for direct and inventory-based GoldenBeetleEgg consumption, and FollowBehaviour for following, while GoldenBeetleEgg extends Egg with

HatchingRules and NearbyCapabilityCondition for hatching near cursed entities. It prioritizes SRP, OCP, DIP, and DRY, supporting varied hatching conditions, eatable entities, and reusable capability checks (Blessed).