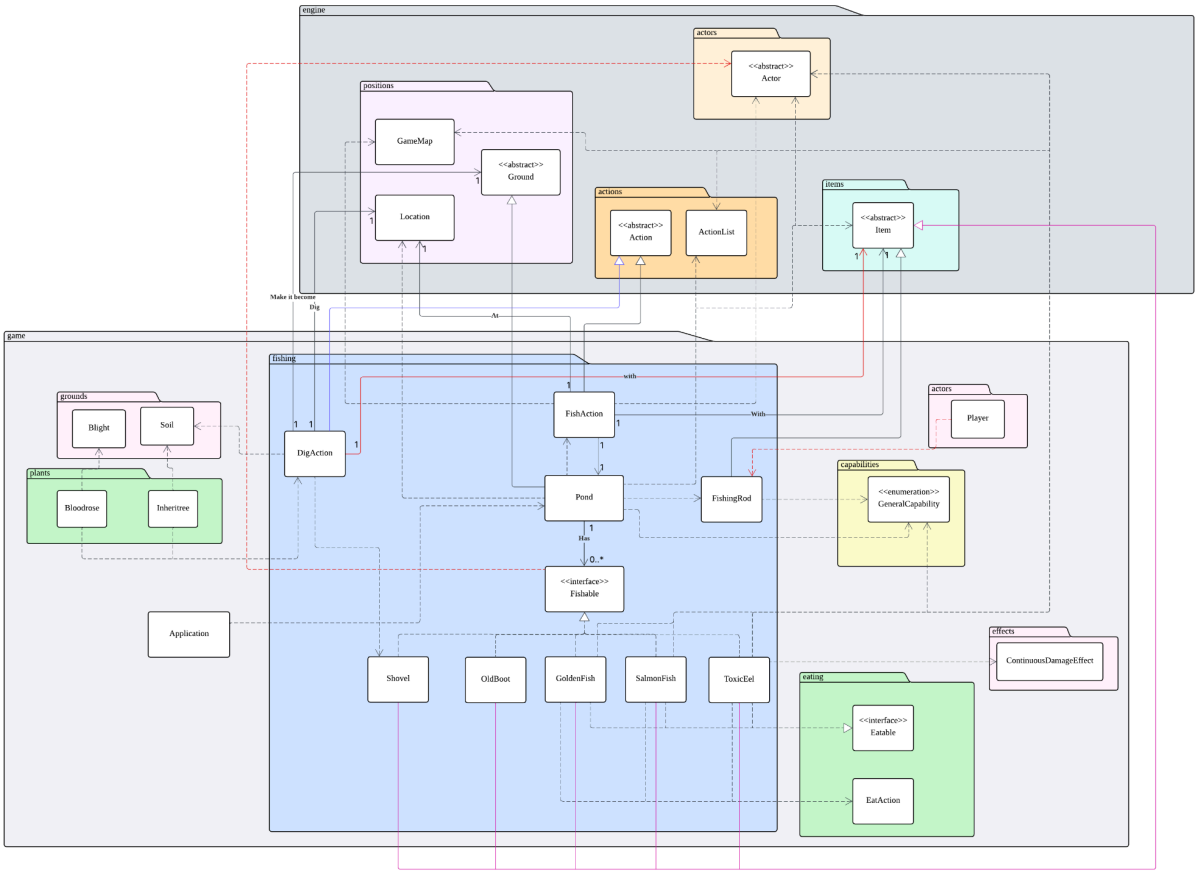
Design Rationale: Fishing System (REQ4)



1. Introduction

This document outlines the design for a fishing system. It allows players to fish in Pond areas with a FishingRod, catching one item per turn based on probabilities (SalmonFish: 30%, ToxicEel: 50%, Showel: 60%, OldBoot: 40%, GoldenFish: 5%). Caught items can be consumed via EatAction or used for digging (Shovel). The design prioritizes extensibility and SOLID principles for a robust system.

2. Design Goals

- Implement fishing at Ponds with a FishingRod, with probabilistic catches added to inventory.
- Allow consumption of fishable items via EatAction or use via DigAction.
- Ensure fishable items dropped at a Pond disappear (to be implemented).
- Define how Ponds are populated with items, ensuring encapsulation.

3. Design Analysis & Decisions

3.1 Fishing Mechanics and Catch Logic

Option A: Logic in FishAction with Probabilities

- **Description**: Create a FishAction implementing all fishing logic in its execute method, including probability checks and inventory updates. Catch probabilities are hardcoded in FishAction's logic, and a single item is selected using a first-success approach.
- **Maintainability**: Poor. Adding new fishable items requires modifying FishAction's execute method.

Pros	Cons
Simple for small implementation with three fishable items	FishAction handles probability checks and inventory updates violates SRP. Violates OCP where adding new fishable items require modification
	High connascence of algorithm: FishAction tightly couples catch logic to specific item types.

Option B: Fishable Interface with Polymorphic Catch Logic

- **Description**: Fishable interface defines catching probability and "who catch", implemented by the fishable items in pond. FishAction uses Random.nextDouble() to add the "within probability" items into a list and randomly catch one of them.
- **Maintainability**: High. New fishable items can implement Fishable and define their own probability, without modifying FishAction.

Pros	Cons

Adheres to SRP where FishAction handles catch logic, and Fishable items define their own probability. New items can be added without modifying FishAction. (OCP)	Complex and unintuitive probabilities where the indicated probabilities are meant just eligible for catching.
FishAction depends on Fishable interface, not item types, which shows low connascence.	

Decision & Justification

- **Chosen Design**: Option B.
- **Rationale**: Option B adheres to SOLID principles (SRP, OCP) and DRY, reducing connascence by decoupling FishAction from specific item types. The Fishable interface allows new items (SharkFish with 20% chance) to be added without modifying core logic, ensuring extensibility. The first-success approach simplifies the catch logic, catching one item per turn.

3.2 Fishing, Eating, and Digging Effects

Option A: Effect Logic in FishAction, EatAction, and DigAction

- **Description**: FishAction, EatAction, and DigAction directly implement effect logic for each item capability (fishing,eating,digging) by hardcoding effects in their execute methods.
- **Maintainability**: Poor. Adding new items requires modifying FishAction, EatAction, and DigAction to handle new effects, leading to code duplication.

Pros	Cons
Simple for initial implementation with three items, as effects are centralized.	Violates SRP and OCP where all actions handle effect alongside other responsibilities and new item types require modifying the actions

	Effect logic is tightly coupled to specific item types. (High connascence of algorithm)

Option B: Polymorphic Effects via Fishable and Eatable Interfaces

- **Description**: The created classes like SalmonFish and ToxicEel implement Fishable, defining “who catch” to add to inventory and fished effects. The legit fishes are to implement Eatable, defining “who eat” for consumption effects. Shovel will enables DigAction and set grounds.
- **Maintainability**: High. New items define their own effects without modifying the actions.

Pros	Cons
Supports OCP where new items define their effects without modifying actions and promotes DRY where the effect logic is defined once per item type.	If new item that need a new type to be used or existed in the pond will need to implement new item type interface.
Low connascence where the actions depend on interfaces, not specific item types.	

Decision & Justification

- **Chosen Design**: Option B.
- **Rationale**: Option B aligns with SOLID principles (SRP, OCP, DRY), reducing connascence by decoupling effect application from FishAction, EatAction, and DigAction. The Fishable and Eatable interfaces ensure new items (e.g., Dolphin) can define their own effects without modifying core logic, promoting extensibility and maintainability.

3.3 Pond's Fishable Population and Impassability

Option A: Fishable in FishAction and impassable in Pond

- **Description**: The list of fishable items is hardcoded directly within the FishAction class. Separately, the Pond class implements the canActorEnter(actor) method to return false, making it impassable.
- **Maintainability**: Poor. The Pond's state logic is split between two different classes.

Pros	Cons
Simple to implement for a game with only one type of fishing area	Violates SRP where FishAction is managing the items and the creating new fishing area with different items would need to modify the action (OCP)

Option B: Encapsulated State in Pond

- **Description**: The Pond class is made fully responsible for its own state. For population, it initializes its own list of fishable items in its constructor and provides a defensive copy to FishAction via a getter. For impassability, it implements the canActorEnter(actor) method to return false.
- **Maintainability**: High. All state logic for the Pond is located within the Pond class itself.

Pros	Cons
Adheres to SRP: The Pond correctly encapsulates population and supports OCP where new areas created with unique populations.	The use of defensive copying for the item list introduces a negligible performance overhead.

Decision & Justification

- **Chosen Design**: Option B.
- **Rationale**: This design is chosen for its strong adherence to SOLID principles, particularly SRP and OCP. By making the Pond class responsible for its own population and physical properties, the design is far more robust, maintainable, and extensible. This

allows for the easy addition of new, varied fishing locations in the future without altering existing action logic.

4. Conclusion

The design for Requirement 4 implements a fishing system with a focus on extensibility, SOLID principles and connascences. Fishing mechanics use FishAction with Fishable for polymorphic catch logic, fishing, eating, and digging effects are designed via Fishable and Eatable interfaces, and the Fishable item lifecycle (population) is managed by Pond. The system promotes SRP, OCP, and DRY, with low connascence, ensuring a robust, maintainable, and extensible solution for interactive fishing.