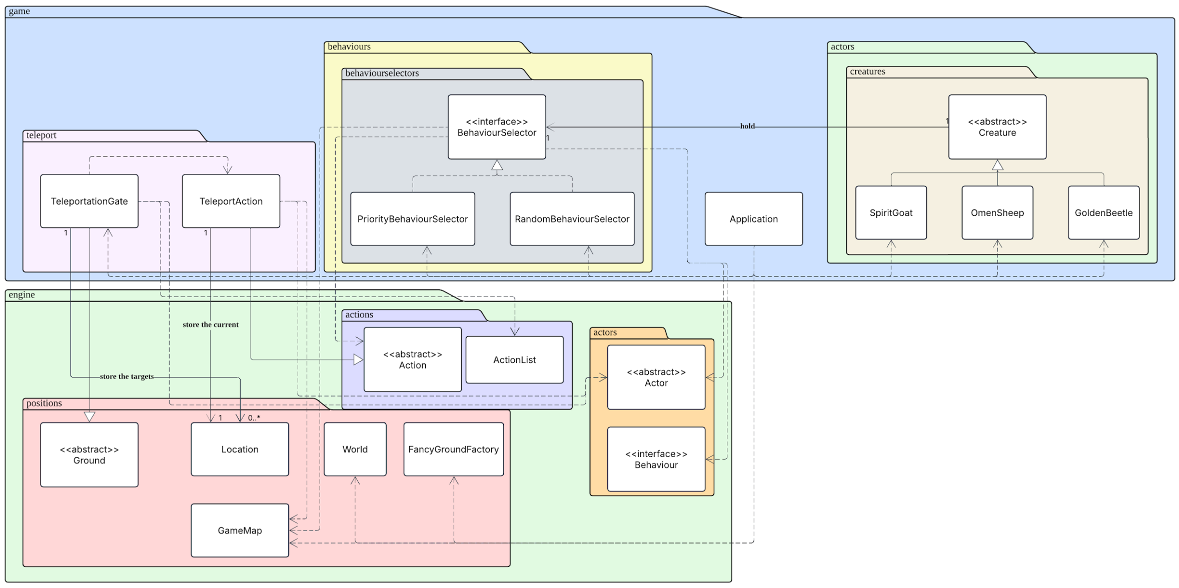
Design Rationale: Travelling to Limveld and Creature Behaviour Strategy



1. Introduction

This document describes the design and implementation of teleporting to Limveld map and the creatures' behaviour selection strategies, focusing on supporting both priority-based and random behaviour selection. It also covers the teleportation mechanic between Limveld and the Valley of Inheritree. The design prioritizes extensibility, reusability, and adherence to SOLID principles.

2. Design Goals

- Implement Limveld as a new map connected to the Valley of Inheritree via teleportation circles.
- Support multiple behaviour selection strategies (priority-based and random).
- Ensure creatures cannot move between maps except via teleportation controlled by the player.
- Facilitate future extension of new behaviour selection strategies and multi-map integration.

3. Design Analysis & Decisions

3.1 Teleportation & Map Connectivity

Option A: Logic in Player.playTurn()

- **Description:** In this model, the Player class would be "smart." Inside its playTurn() method, it would explicitly check the ground it is standing on. If ground can teleport, the Player class itself would then get the list of destinations from the gate and create the TeleportActions to add to its menu.
- **Maintainability:** Low. The Player class becomes bloated with special-case logic for different types of interactive ground, making it hard to maintain

Pros	Cons
Keeps all Player-specific interaction logic centralized in one place.	Violates SRP by mixing teleportation movement and map navigation logic into player
	Hard to maintain or expand teleportation rules globally

Option B: Centralized Teleportation Management in Map/World Layer

- **Description:** Teleportation circles are map objects with configured destinations. Player triggers teleportation to switch maps. Creatures cannot move between maps except via player-initiated teleportation.
- **Maintainability:** High. Centralized management simplifies future modifications and testing.

Pros	Cons
Clear separation between creature and map logic	Requires design of map connection infrastructure
Easy to manage teleportation globally and extend with multiple destinations or new maps	

Decision & Justification:

- **Chosen Option:** B
- **Rationale:** Option B is a good design that leverages the engine's polymorphic allowableActions system. It creates a decoupled and highly extensible architecture where objects are responsible for their own functionality. This follows the "Tell, Don't Ask" principle—the Player doesn't ask the ground what it is; the TeleportationGate *tells* the Player what actions are available. This is a more scalable and maintainable solution that perfectly aligns with SOLID principles and promotes low connascence.

3.2 Behaviour Selection Logic

Option A: Behaviour Selection in Creature.playTurn()

- **Description:** Each creature's playTurn() method contains explicit logic to iterate over behaviours sequentially and select the first valid behaviour. Adding random selection requires embedding extra conditional logic inside this method.
- **Maintainability:** Low. Changes to behaviour selection require editing all creature classes individually.

Pros	**Cons**
Simple initial implementation	Violates Single Responsibility Principle (SRP) by mixing selection and execution.
Easy to trace behaviour logic since it's centralized in the creature class	Difficult to extend or modify selection logic without changing creature code (violates Open-Closed Principle, OCP).
	Duplicate code if random selection logic is added to multiple creatures.

Option B: Strategy Pattern with BehaviourSelector Interface

- **Description:** Introduce a BehaviourSelector interface with multiple implementations such as PriorityBehaviourSelector and RandomBehaviourSelector. Creatures delegate behaviour selection to the assigned selector instance.
- **Maintainability:** High. Behaviour selection logic isolated and easily testable. New strategies can be introduced without touching creature code.

Pros	Cons
Clean separation of concerns respecting SRP	Additional classes increase complexity and require upfront design
Easy to add new selection strategies without modifying creatures (OCP compliance)	
Behaviour selectors reusable across different creatures, promoting DRY	

Decision & Justification:

- **Chosen Option:** B
- **Rationale:** The strategy pattern offers a scalable, clean, and maintainable way to handle behaviour selection. It supports the requirement that some creatures use priority selection while others select randomly, fixed at instantiation. It respects key SOLID principles and facilitates future expansions. This design uses **connascence of name**: actors depend only on the interface name BehaviourSelector, not the actual implementation. This low-level connascence reduces tight coupling. We intentionally avoid **connascence of meaning** or **algorithm**, which would make the actor reliant on selector internals, harming modularity.

3.3 Creature Behaviour Setup

Option A: Behaviour Lists Inside Each Creature

- **Description:**
  Each creature subclass explicitly defines its own list of behaviours internally. The behaviours are hardcoded in methods like initializeBehaviours(). This makes the creature fully responsible for managing its behaviour set and how it executes them.
- **Maintainability:**
  Moderate. Any change to behaviours or their priority requires modifying each affected creature class directly. Behaviour code is duplicated across creatures that share similar behaviour logic.

Pros	Cons
Simple and intuitive to implement and understand	Violates the DRY principle because similar behaviour lists are repeated across multiple creature classes
Behaviour logic is immediately visible inside the creature, easing debugging and inspection	Reduced flexibility if future extensions require dynamic behaviour changes or alternative selection strategies
No additional abstraction layers, reducing complexity in smaller projects	

Option B: External Static Behaviour Configuration

- **Description:**
  All behaviour configurations for creatures are defined in a centralized utility class (BehaviourConfig), mapping creature types to behaviour lists and their order. Creatures fetch their behaviour set at runtime via this static mapping.
- **Maintainability:**
  Poor. Changes to a creature's behaviour require modifying the global configuration. Centralization makes reasoning and debugging harder as project scales.

Pros	Cons
Promotes separation of concerns and reusability of behaviour sets	More complex to implement, requiring additional classes and interfaces
Supports runtime flexibility for behaviour switching if desired	Behaviour logic is less transparent inside creature classes, possibly increasing debugging difficulty for beginners
	Overhead might be unnecessary for small projects with few creature types

Decision & Justification:

- **Chosen Design:** Option A
- **Rationale:**
  We chose to define behaviours directly in each creature class for clarity and modularity. It aligns with encapsulation and allows different creatures to have customized logic easily. Although Option B sacrifices modularity for centralization, making logic harder to trace and increasing coupling, it was rejected.

4. Conclusion

The design of Limveld teleportation and the creature behaviour system adopts modular architecture using the Strategy pattern for behaviour selection and centralized map teleportation management. This design promotes separation of concerns, extensibility, and adherence to SOLID principles, ensuring maintainable and scalable code for future expansions.