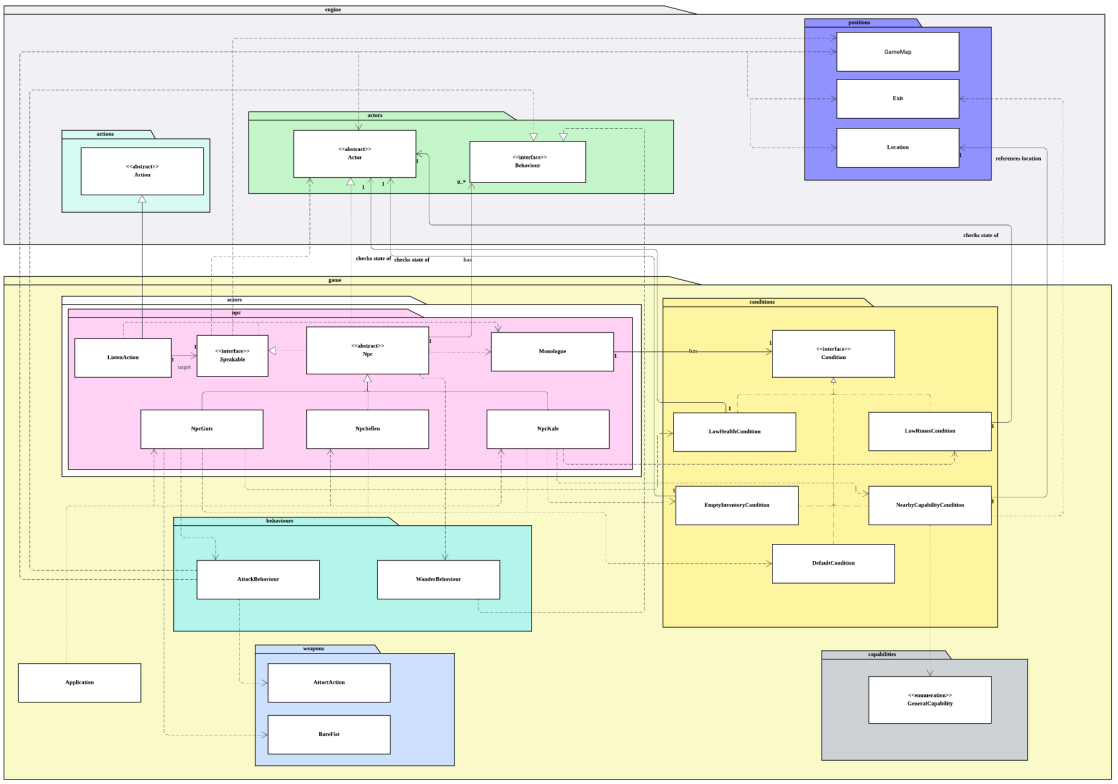
Design Rationale: The People of the Valley



1. Introduction

This document details the design for Requirement 3, with unique dialogues and behaviors. The objective is an interactive NPC system featuring a flexible, conditional monologue mechanism and distinct NPC actions (e.g., Guts's aggression), prioritizing extensibility, reusability, and object-oriented design principles (SOLID).

2. Design Goals

- Implement NPCs Sellen, Kale, and Guts with unique monologues, behaviors, and Player interaction via ListenAction.
- Support conditional monologues (e.g., Kale’s rune/inventory checks, Guts’s health-based dialogue) using Conditions.
- Enable Guts’s AttackBehaviour (targets >50 HP, BareFist) and shared WanderBehaviour for all NPCs.

3. Design Analysis & Decisions

1. NPC Core Structure and Basic Interaction

This feature addresses the fundamental representation of NPCs (Sellen, Kale, Guts) and how the Player initiates dialogue.

Option A: Abstract Npc Class with ListenAction

- **Description:** An abstract Npc class (extending Actor) serves as the base for Sellen, Kale, and Guts, managing shared behaviors (e.g., WanderBehaviour) and monologues via getMonologues(). Player interaction uses ListenAction.
- **Maintainability:** Good. Common logic is centralized; new NPCs are added as subclasses.

Pros	**Cons**
Enables polymorphic treatment (LSP) and code reuse (DRY) for behaviors and interaction.	Base Npc may need modification for future interaction mechanics.
Encapsulates NPC-specific data and separates interaction logic (SRP).	

Option B: Individual NPC Classes Extending Actor, Dialogue Logic in Player

- **Description:** Each NPC extends Actor. No common Npc base. Player class checks instanceof and calls NPC-specific dialogue methods or generates actions.
- **Maintainability:** Poor. Player class modified for new NPCs (violates OCP). Common behaviours duplicated.

Pros	**Cons**

Simpler for few NPCs with distinct behaviors.	Violates DRY (duplicated behaviors) and OCP (Player changes needed).
	Tight coupling and LSP violation via instanceof.

Decision & Justification

- **Chosen Design: Option A.**
- **Rationale:** Option A is chosen for its strong alignment with fundamental OOP principles, leading to a more organized, maintainable, and extensible system. The abstract Npc class provides a robust foundation for common functionalities (like offering ListenAction and default wandering) and enables polymorphic interactions. This promotes code reuse (DRY) and proper encapsulation of NPC-specific details, adhering to LSP and SRP. For example, adding a new NPC 'Blacksmith Bjorn' would involve creating NpcBjorn extends Npc and implementing getMonologues(); ListenAction would work with him without change (OCP for ListenAction). This is significantly more scalable and maintainable than Option B, which would lead to a brittle, tightly coupled design.

2. Monologue System and Conditional Dialogue Delivery

This feature details how NPCs manage and deliver dialogues, particularly conditional ones.

Option A: Monologue Class with Condition Interface and Speakable

- **Description:** NPCs implement the Speakable interface, overriding getMonologues() to return a list of Monologue objects, each pairing dialogue with a Condition (like LowHealthCondition,

NearbyCapabilityCondition). ListenAction calls getMonologues() on the Speakable target, filters by Condition checking, and randomly selects a monologue.

- **Maintainability:** Excellent. New dialogues/conditions are added without modifying core classes.

Pros	**Cons**
Separates condition logic (SRP, DIP).	May lead to many small Condition classes if not parameterized.
Enables extensibility (OCP) where new conditions/monologues added without modification	
Reusable Condition classes promote DRY	

Option B: Hardcoded Conditional Logic in NPC Methods

- **Description:** Dialogue logic (like if (farmer.getRunes() < 500)) is embedded in each NPC's getting monologue methods (getMonologue()) using if-else.
- **Maintainability:** Poor. Complex methods violate SRP; shared conditions require multiple edits.

Pros	**Cons**
Simpler for few NPCs with basic conditions.	Violates OCP (new conditions modify methods) and DRY (no reuse).
	Low cohesion and hard to test.

Decision & Justification

- **Chosen Design: Option A.**
- **Rationale:** Option A is chosen for its modularity, extensibility, and adherence to OOP principles (SRP, OCP, DIP) by utilizing the Monologue class with the Condition interface and Speakable for Npcs. This design cleanly separates the concern of checking a condition from the dialogue itself and from the NPC's core responsibilities. For example, if a new requirement arises for Kale to have a monologue when the Player carries a specific "RareItem," a PlayerHasItemCondition could be created and added to Kale's getMonologues() method without any changes to ListenAction or other NPCs, demonstrating OCP. This structured approach is significantly more robust and maintainable than hardcoding logic (Option B).

3. Guts's Aggressive Behavior

This feature covers NpcGuts's combat: attacking actors with >50 HP using BareFist.

Option A: Dedicated AttackBehaviour Class

- **Description:** An AttackBehaviour implements Behaviour. An instance, configured with an HP threshold of 50, is added to NpcGuts's behaviours list with a high priority. AttackBehaviour will scans surroundings, checks target HP (> 50), and returns an AttackAction using NpcGuts's intrinsic weapon (BareFist).
- **Maintainability:** Good. Attack logic is encapsulated and localized.

Pros	**Cons**
Encapsulates combat logic (SRP) within the engine’s behavior	Specific to ">x HP" rule; may need generalization for other NPCs.

system.	

Option B: Attack Logic Directly in NpcGuts.playTurn()

- **Description:** All logic for scanning for targets, checking their HP, and creating/returning an AttackAction would be embedded directly in NpcGuts.playTurn() method.
- **Maintainability:** Poor. playTurn() becomes complex, mixing concerns.

Pros	**Cons**
Simpler for a single attack rule.	Violates SRP and cohesion; logic isn't reusable.
	Harder to test due to complexity.

Decision & Justification

- **Chosen Design: Option A.**
- **Rationale:** Option A aligns with the behavior-driven architecture, promoting SRP and modularity (a new AttackBehaviour variant for an NPC attacking cursed entities can use a CursedCondition without modifying existing behaviors (OCP)). This keeps the NpcGuts class cleaner and its playTurn() method simpler, focusing on orchestrating behaviours based on priority. If new attack triggers arise, this design ensures minimal impact by adding Conditions, enhancing maintainability over Option B.

4. Conclusion

The design for REQ3, "The People of the Valley," successfully integrates interactive NPCs by leveraging an abstract Npc base class, a flexible

Monologue system employing the Condition interface (Strategy Pattern) for dynamic dialogue, and distinct Behaviour classes (WanderBehaviour, AttackBehaviour). This structure fulfills all specified functionalities for Sellen, Kale, and Guts, including their unique conditional dialogues and Guts's targeted aggression, adhering to SOLID principles (SRP, OCP, DIP) for a maintainable, extensible, and robust system. This paves the way for straightforward future enhancements, such as adding new NPCs with complex conditional dialogues or varied behaviors, with minimal impact on existing, well-encapsulated components.