REQ3: The People of The Valley

Implementation A: Introduce Talkable interface

We define a Talkable interface for any NPC that can be listened to. This abstracts dialogue capability into its own contract, so that ListenAction and other systems depend only on Talkable (an interface) rather than concrete character classes. In practice, only NPC classes with dialogue implement Talkable; non-speaking objects remain unaffected. This follows the Interface Segregation Principle: no class is forced to implement unused methods. It also supports Dependency Inversion: high-level actions (like ListenAction) depend on the Talkable abstraction, not on specific NPCs. In summary, the Talkable interface cleanly separates conversational capability from other NPC behavior, at the cost of one extra abstraction layer.

Pros	Cons
Decoupling: Any NPC can implement Talkable and be listened to without changing ListenAction (supports DIP and OCP). *Interface Segregation:* Only dialogue-capable NPCs implement the interface, so other actors aren't forced to include unused talk methods. *Polymorphism:* ListenAction can work with any Talkable object, making future NPCs easy to add (open for extension).	*Additional abstraction:* Introduces an extra interface layer. If only one or two NPCs have dialogue, this may seem like overkill. *Interface evolution:* Changing the interface (e.g. adding methods) would require updating all implementers. *Minor complexity:* Developers must remember to implement Talkable on new NPC classes that need dialogue.

Implementation B: Generic ListenAction class

We implemented a single ListenAction class that operates on any Talkable NPC. This consolidates the listening behavior: the action invokes getTalkingLines() on the target Talkable, without needing separate classes for each NPC. This design follows the DRY principle by centralizing logic that would otherwise be duplicated. It also adheres to the Open/Closed Principle: adding new talkative NPCs requires no change to ListenAction (it's already closed for modification but open for extension). All special behavior per NPC is handled in the NPC's override of getTalkingLines(), so ListenAction remains generic. In short, one reusable action class covers all dialogue interactions via the Talkable interface.

Pros	Cons
Reusability/DRY: One class handles listening for all NPCs, avoiding duplicate action classes (follows DRY). *Extensibility (OCP):* New Talkable NPCs automatically work with the same ListenAction, so we don't modify existing code to add them. *Simplicity:* ListenAction logic is centralized, making it easy to maintain and test.	*Complexity of edge cases:* A single class may need conditionals if some NPCs require special handling (risking a bloated conditional in one place). *Single responsibility risk:* If too much varied logic is added to this one action, it could violate SRP. *Less granular control:* Unique dialogue rules for one NPC might be harder to encode in a generic class versus a specialized action

Implementation C: NPC classes with overridden getTalkingLines

Each NPC (Sellen, Merchant Kale, Guts) is implemented as a subclass of PeacefulCharacter or HostileCharacter and also implements Talkable. The method getTalkingLines() is overridden in each class to return that NPC's dialogue (either a fixed string or conditional content based on game state). This uses classic OOP polymorphism: each NPC class customizes only its dialogue output. The base classes (PeacefulCharacter/HostileCharacter) handle common behavior, so NPC-specific code is limited to the overridden method. Because all NPC classes share the same interface or base type, they can be substituted interchangeably (satisfying Liskov Substitution). In effect, adding a new talking NPC is as simple as creating a new subclass with its own getTalkingLines(); the rest of the system (actions, utils, etc.) does not need modification.

Pros	Cons
Polymorphism: Each NPC defines its dialogue internally by overriding, so they behave like their base class but with custom lines (LSP). *Reuse of base behavior:* Common movement or combat logic lives in PeacefulCharacter/HostileCharacter, minimizing duplication. *Clear extension:* New NPCs can be added via subclasses without changing existing code (OCP). *Contextual dialogues:* Methods can incorporate game state checks to enable dynamic lines.	*Mixing concerns:* NPC classes hold dialogue text, which could bloat the class and mix presentation with behavior (risking SRP). *Code changes for new lines:* Altering or adding dialogues requires editing the class, which can be less flexible than data-driven approaches. *Static vs conditional:* Hard-coded (static) lines are simple but inflexible; adding conditional lines introduces complexity and coupling to game state.

Implementation D: SurroundingUtils for nearby checks

We introduced a SurroundingUtils helper class to encapsulate logic that checks tiles around an actor for cursed entities. Instead of repeating the same tile-scanning code in multiple places, NPCs call static methods in SurroundingUtils. This achieves DRY by removing duplicated environment-check code. Because SurroundingUtils has a narrow focus (only surrounding-tile logic), it respects SRP: it only changes if the scanning logic itself changes. All actor classes simply reuse this utility, so the implementation of nearby checks is consistently maintained in one place. The trade-off is that this approach is essentially procedural (static methods), so it doesn't use polymorphism; however, as a pure helper it has no state and won't need to vary at runtime, so it's acceptable in this context.

Pros	Cons
DRY: Centralizes repetitive tile-check code into one class, avoiding duplication across NPC classes. *Single Responsibility:* SurroundingUtils only handles environment scanning, making it easy to update if rules change. *Convenience:* Simplifies caller code (NPCs only invoke a utility call instead of manual loops).	*Static utility downsides:* Hard to substitute or mock in tests (violates DIP, since callers depend on static methods). *Limited extensibility:* If scanning logic needed state or dependency injection in the future, the static design might require refactoring. *Risk of bloat:* If too many unrelated helper methods accumulate, it could become a catch-all violating cohesion (so it must stay focused).

Implementation E: GutsAttackBehaviour strategy

For Guts, we created a GutsAttackBehaviour class that encapsulates his special aggression rule (attack only actors with HP > 50). This class implements a generic AttackBehavior interface (Strategy pattern). Guts's character is composed with this behavior: when deciding to attack, Guts delegates to GutsAttackBehaviour rather than hard-coding the rule. This modularizes the logic and follows SRP: Guts's class no longer has embedded attack-condition logic (that resides in the behavior class). It also makes the design open for new behaviors: we could add different behavior classes for other characters without modifying existing ones (OCP). Overall, this isolates the aggression criterion into one place and makes it easy to extend or change.

Pros	Cons
Modularity (Strategy): Attack logic is isolated in its own class, so the actor and the algorithm vary independently. *Extensible:* New enemy behaviors can be added by creating new Behavior subclasses without touching Guts's code (OCP). *SRP:* Guts's class delegates its one special rule out, so each class has a clear single responsibility. *Reusability:* The behavior can be reused or adapted for other characters with similar rules.	*Extra abstraction:* Introduces another class/interface, which is a slight overhead. *Complexity:* If only one behavior exists, some may see it as needless indirection (though it pays off if rules diversify).

Summary: OO Principles and Extensibility

This design cleanly follows object-oriented best practices. Each class or module has a focused responsibility (Single Responsibility) and there is minimal code duplication (DRY). We depend on interfaces (Talkable, AttackBehavior) so high-level code relies on abstractions, not concrete classes (Dependency Inversion). The system is open to extension but closed to modification (Open/Closed): for example, adding another NPC with dialogue simply means implementing Talkable and overriding its lines, without altering existing logic. All subclasses preserve the behavior expected by their base types (Liskov Substitution) — a PeacefulCharacter or HostileCharacter can stand in for a generic character without breaking code. By segregating interfaces (only dialogue-capable NPCs implement Talkable), we avoid forcing irrelevant methods on other classes.

In practice, this means future extensions are straightforward: adding a new talking NPC involves creating a new class (or using an existing base class) with its own getTalkingLines(), and it will automatically work with the ListenAction. New enemy behaviors (e.g. different attack conditions) can be added by making new AttackBehavior classes. Dialogue rules can be enriched by extending or adding methods in the NPC classes or behaviors, again without changing core systems. This modular, interface-driven design aligns with SOLID principles and the project’s goals of maintainability and flexibility. The resulting codebase should be easier to maintain, test, and extend as the game grows.