FIT2099 Assignment 3 Design Rationale

Requirement 1 - Teleportation and Behavioral Strategies System

Introduction & Design Goals

This requirement introduces teleportation mechanics and behavioural strategies to the three non-hostile creatures (Omen Sheep, Spirit Goat, Golden Beetle). Our design prioritizes:

- **Extensibility**: Add new teleporters without modifying core logic
- **Reusability**: Centralize common behavioral strategies (random/priority).
- **Decoupling**: Minimize dependencies between systems
- **SOLID Compliance**: Enforce single responsibilities and interface segregation

Key Design Decisions & SOLID Alignment

1. Teleportation System

   Mechanism
   - *TeleportAction*: Pure execution logic (SRP)
   - *TeleportationCircle:* Manages destinations and action generation (OCP)
   - Bi-directional setup in Application.java , both valley and limveld teleporters are available for teleportation between the two.

   SOLID Alignment
   - **SRP**: *TeleportAction* handles movement execution, *TeleportationCircle* manages destinations
   - **OCP**: New destinations added via *addDestination()* without modifying existing code
   - **DIP**: Both classes depend on engine abstractions (*Location*, *GameMap*)

   Pros:
   - Clean separation of concerns
   - Zero coupling with creature logic
   - Reusable for any map transitions

   Cons:
   - Teleporter availability tied to destination occupancy (no fallback logic)

2. Behaviour Strategy Pattern

   Mechanism
   - *PrioritySelectionStrategy*: Original priority-based selection
   - *RandomSelectionStrategy*: New random selection variant

SOLID Alignment
- **OCP**: Add new strategies without modifying NPCs
- **DIP**: NPCs depend on `BehaviourStrategy` abstraction
- **LSP**: Strategies interchangeable at runtime

Pros:
- Decouples default logic from NPC implementations
- Enables creature personality variants (e.g., unpredictable Golden Beetle)
- In Application.java - Different strategies per instance

Cons:
- Slight performance overhead from strategy indirection

Alternatives Considered

1. **Monolithic Ground Class**
   - **Proposal**: Combine teleporter and creature spawning in one class
   - **Rejected**: Violates SRP, creates god object

2. **Hardcoded Teleport Destinations**
   - **Proposal**: Directly embed locations in `TeleportAction`
   - **Rejected**: Breaks OCP, limits reuse

Conclusion

This design achieves:
+ Extensibility via capabilities and strategies
+ Reusability through centralized systems (teleporters)
+ SOLID Compliance via SRP, OCP, and ISP

The key strengths are polymorphic creature behaviors and decoupled systems allowing independent evolution. While minor compromises exist in attribute management, they're acceptable tradeoffs for the flexibility gained. Future requirements (new creatures, teleporter types, or behavioral strategies) can be implemented through existing extension points without core modifications.

Requirement 2 - Bed of Chaos

Introduction & Design Goals

In Limveld stands the Bed of Chaos (“T”), a stationary boss with 1 000 HP that never moves. When a player comes close, the boss attacks with a base 25 damage at 75 % accuracy. Otherwise, each turn it “grows” by adding Branches or Leaves in a 50/50 split.

- Extensibility: New part types (e.g. PoisonBud, ShieldSpore) can plug into the growth system without changing existing classes.
- SOLID Compliance: Each class has one responsibility; we depend on simple interfaces, stay open to new parts, and avoid instanceof or downcasting.
- Maintainability: Growth logic, attack logic, and healing logic are clearly separated to make future changes safe and easy. REQ2 Bed of Chaos

Key Design Decisions & SOLID Alignment

1. **PlantPart Interface**

Responsibility:

Declares two methods:

- getDamage() for attack power.
- grow(boss, display) for per-turn growth effects.

SOLID:

- SRP: PlantPart only defines part behavior.
- DIP: Bed of Chaos relies on the abstraction, not concrete Branch or Leaf.
- OCP: New parts implement this interface, no core edits needed.

2. **Branch Class**

Responsibility:

- Contributes 3 damage plus its children’s damage.

- On grow(), prints “Branch is growing…”, then randomly spawns another Branch or a Leaf.
- If it spawns a Leaf, immediately calls that leaf’s grow() so the boss heals in the same turn.

SOLID:

- SRP: Branch only handles its own damage and growth.
- LSP/ISP: Satisfies PlantPart without extra methods.

3. Leaf Class

Responsibility:

- Contributes 1 damage.
- On grow(), heals boss by 5 HP and prints “Bed of Chaos (HP/HP) is healed.”
- Never spawns new parts.

SOLID:

- SRP: Leaf handles only healing and damage.
- OCP: Alternative healing parts can follow the same pattern.

4. BedOfChaos Class

Responsibility:

- Manages HP, attack vs. growth decision, and part list.
- Growth turn sequence:

1. Print “... is growing...”

2. Snapshot existing parts
3. Spawn one new top-level Branch or Leaf (50/50), printing “it grows a X...”
4. Call grow() on every part from the snapshot (ensures top-level leaves wait until next turn to heal)
5. Return a silent No-Op action to avoid engine errors

- Attack turn: builds an IntrinsicWeapon each turn with damage = 25 + sum(parts) and 75 % accuracy.

SOLID:

- SRP: Bed of Chaos only orchestrates growth and attacks, not part details.
- DIP: Uses PlantPart and IntrinsicWeapon abstractions.
- OCP: Any new part type or attack modification can be added without changing growth logic.

5.BedOfChaosIntrinsicWeapon

Responsibility:
Encapsulates the boss’s own crushing attack (damage + hit-rate) as a named class.

SOLID:

- SRP: Weapon logic sits in its own class.
- OCP: Attack effects can be extended by subclassing this weapon.

Alternatives Considered

Using WeaponItem for the boss’s attack:

- Rejected: Implies pickup/dropping, complicates dynamic damage updates.

Immediate leaf healing on spawn:

- Rejected: Violates REQ2 (“leaf heals only when the boss grows,” not when it appears).

Down-casting to detect leaf vs. branch

- Rejected: Banned by SOLID guidelines; replaced with polymorphic grow() and snapshot ordering.

Limitations & Trade-Offs

- Random Growth: Makes testing exact part counts less predictable; use seedable RNG for deterministic tests.

- Display Overhead: Printing every growth step may clutter logs; can be toggled off in production.

- Turn-Based Healing Only: Leaves heal only once per growth turn; real-time or chained growth would require a strategy pattern.

Conclusion

My design cleanly separates the boss's, part behaviors, and weapon logic. By depending on the PlantPart interface, using snapshots to control heal timing, and following SOLID principles, we meet all REQ2 requirements which are stationary boss, conditional attack/growth, 50/50 branching/leaves, recursive damage accumulation, and healing, while remaining open to future enhancements without core changes.

Requirement 3 - LLM-Powered Monologue NPC

Introduction & Design Goals

Requirement 3 brings three LLM-powered NPCs into the world: the Ghost of Whispers, the Wandering Poet, and the Seer, each capable of speaking dynamically based on the player's past actions. Our primary design goals were:

- Extensibility: Allow new LLM-driven characters or dialogue styles to be added without touching existing code.
- Reusability: Centralize common logic (API calls, action history) so behaviours can share it.
- Separation of Concerns: Keep prompt construction, LLM interaction, and game integration in different modules.
- SOLID Compliance: Ensure each class has one clear responsibility, depends on abstractions, and is closed to modification but open to extension.
- Engine Compatibility: Integrate easily with FIT2099's Behaviour and Action abstractions, no instanceof checks or down-casting.

Key Design Decisions & SOLID Alignment

LLMDialogueBehaviour Interface

- Defines constructPrompts() and formatResponses() for any LLM-driven behavior.
- SRP: Behaviors focus strictly on dialogue content.
- OCP: New behaviors simply implement the interface.
- DIP: High-level modules depend on this abstraction, not concrete classes.

ActionHistoryProvider Abstraction

- Supplies player action history to behaviors, decoupling them from Player.
- ISP: Exposes only the methods and behaviors needed.
- DIP: Behaviors and managers rely on the ActionHistoryProvider interface.

LLMMonologueManager as API Gateway

- Centralizes all geminiCall() interactions and any caching or error handling.
- SRP: Manages only external API concerns.
- OCP: Switching LLM providers or adding logging requires changes here alone.
- DIP: Other classes call the manager, not the API directly.

Concrete Behaviour Classes

- PoetBehaviour, SeerBehaviour, and GhostBehaviour each implement LLMDialogueBehaviour and the engine's Behaviour.
- SRP: Each class handles one style of monologue (poetry, prophecy, riddles).
- LSP: Any behaviour can replace another without breaking the dialogue flow.
- OCP: Adding a new "HistorianBehaviour" or similar is just a new class.

LLMDialogueAction Integration

- Extends Action to inject LLM dialogue into the turn system by invoking a behaviour and the manager.
- SRP: Executes only the dialogue event.
- DIP: Depends on LLMDialogueBehaviour, not concrete behavior classes.
- OCP: Reusable for any future LLM-powered NPC.

Alternatives Considered

- Hard-Coded Dialogue Trees: Deterministic and low-latency, but quickly unmanageable as contexts and branches grow, violating OCP.
- Inline Prompt Logic in NPC Classes: Would mix game and dialogue concerns, break SRP, and complicate testing.

Conclusion

By cleanly separating prompt construction, LLM communication, and game integration into focused, SOLID-compliant modules, this design makes it straightforward and safe to add new LLM-powered characters or dialogue styles. Maintenance, testing, and future extension all become simpler.

Requirement 4 - Day/Night System

The implementation of Requirement 4 introduces a dynamic day/night system handled by a central TimeManager. This system cycles through four phases which are Morning, Afternoon, Evening, and Night, each with unique effects on the game environment and its actors. The design prioritizes modularity, extensibility, and adherence principles like SOLID.

A. Core Architecture for Time Management

The foundational decision for this feature was how to model the passage of time. Two primary approaches were considered.

Design 1: Strategy Pattern with a TimePhase Interface (Chosen Approach)

A TimePhase interface defines a common contract (applyEffects, getName) for all possible time phases. Concrete classes (MorningPhase, EveningPhase, etc.) implement this interface, encapsulating the specific logic for that period. The TimeManager acts as the context class, holding a list of these TimePhase strategies and cycling through them to apply their effects.

Pros	Cons
High Cohesion (SRP): Each phase class has one responsibility which is managing its logic of the time of the day. This simplifies understanding, testing, and maintenance.	**Increased Number of Classes:** This approach introduces several new classes and an interface, very slightly increasing the number of project files.
Extensible (OCP): New phases (e.g., "Blood Moon") can be added by creating new classes without modifying the existing code.	**Minor Performance Overhead:** TimeManager and TimeController add a layer of processing per tick, though the impact is negligible.
Interchangeable (LSP) & Loose Coupling (DIP): Any TimePhase object can be used by the TimeManager interchangeably. The TimeManager depends on the TimePhase abstraction, not concrete implementations, hence promoting flexibility.	

Design 2: Monolithic Enum-Based System

An alternative was a monolithic enum where a single TimeManager class would use a large switch statement to handle the logic for all phases.

Pros	Cons
Fewer Files: Centralizes all time-related logic into one or two main big files.	**Violates SRP & OCP:** The TimeManager becomes a central object. Adding new phases requires modifying this central class, risking bugs in existing code and causing inefficiency inevitably..
Initial Simplicity: May seem conceptually simpler for a very small number of phases.	**Poor Maintainability:** The central switch statement becomes bloated and difficult to manage as the system grows.
	Tight Coupling: Logic is centralized, making isolated testing of phase behaviour difficult.

Justification: Design 1 was chosen for its adherence to SOLID principles, which provides a robust and maintainable foundation. The benefits of decoupling, clear separation of concerns, and ease of future extension have a major advantage over the minor increase in class count.

B. Applying Phase Effects to Actors

The second major design decision was how TimePhase objects should modify actor behaviour.

Design 1: Behavioural Capability System (Chosen Approach)

Instead of coupling the time system to specific actor types (e.g., via enums like OMEN_SHEEP), the design was refactored to use generic, behavioural capabilities like BECOMES_HOSTILE and AFFECTED_BY_NIGHT. TimePhase classes to iterate through actors on the map and check for these behavioural flags. For instance, if an actor has BECOMES_HOSTILE, EveningPhase grants it the HOSTILE_TO_FARMER status. Actors are defined by their potential behaviours, not their species.

Pros	**Cons**
Extreme Decoupling: This design achieves a high degree of decoupling. The time system has zero knowledge of concrete actor types, only of abstract behaviours. This reduces connascence to almost nothing.	**Indirect Logic Flow:** The connection between a phase and an actor's behaviour is indirect, requiring a developer to look in both the phase and actor class to understand the full effect.
True Open/Closed Principle (OCP): The system is fully open to extension without modification. For example, to add a new Werewolf creature that becomes hostile at night, one simply creates the class and assigns it the BECOMES_HOSTILE capability. The EveningPhase class requires zero changes to support this new actor.	**Map Iteration:** This requires iterating through all actors on the map each phase, which could be a performance concern on extremely large maps.
High Cohesion & Role-Based Design: Actors are defined by the roles they play (BECOMES_HOSTILE), not by their species (OmenSheep). This leads to a cleaner, more cohesive design .	

Design 2: Direct Actor Method Calls

An alternative was for TimePhase classes to directly call methods on actor objects, requiring instanceof checks and downcasting to identify and command them on it.

Pros	**Cons**
Direct Logic Flow: The logic is explicit and easy to follow in a single direction.	**Tight Coupling & instanceof:** Creates a rigid dependency where phase classes must know about concrete actor classes. This use of instanceof is a known anti-pattern that violates OCP and DIP and makes code resistant to change.
	Breaks Encapsulation: Phase classes would become aware of the internal implementation details of actors, violating a core OOP principle.

Justification: Design 1 is vastly superior as it avoids the issues of tight coupling and instanceof. Using abstract, behavioural capabilities as a communication bridge, ensuring the components remain decoupled and extensible, which is critical for the future of the code base.

Future Maintainability and Conclusion

The chosen architecture ensures excellent future maintainability. The combination of the Strategy pattern and a behavioural capability system creates an exceptionally flexible and robust system.

- Adding a New Phase: A DuskPhase could be created and inserted into the TimeManager's cycle with minimal effort and no risk to existing code.
- Adding a New Affected Actor: A new RabidChicken that becomes hostile at night requires no changes to any TimePhase class; the chicken is simply instantiated with the BECOMES_HOSTILE capability.

Conclusion

This design effectively leverages fundamental object-oriented principles to deliver a feature that is not only functional but also clean and prepared for future expansion. It successfully avoids anti-patterns like monolithic classes and instanceof checks, resulting in a high-quality and efficent implementation.