FIT2099 Assignment 2 Design Rationale

Requirement 1

Introduction & Design Goals

Requirement 1 improves the valley by giving Spirit Goats and Omen Sheep new life-cycle mechanics: goats spawn offspring when near blessed entities like inheritree, and sheep lay eggs that hatch after a delay. Our design aims for:

- Extensibility: Allow new blessed entity types, egg varieties, and hatching rules without modifying core actor or item logic.
- Reusability: Centralize adjacency checks and egg behaviors to avoid redundant code..
- Single Responsibility: Keep each class focused-actors decide when to reproduce, utilities handle where, and eggs manage their own lifecycle and consumption.
- SOLID Alignment: Depend on abstractions (interfaces, enums), remain open to extension, closed to modification, and avoid down-casts.

Key Design Decisions & SOLID Alignment

1. Capability-Driven Spawning (Spirit Goat)

Mechanism

SpiritGoat delegates adjacency checks to EntityUtils.hasHereOrNearbyWithCapability, which scans for entities with the BLESSED_BY_GRACE capability. If found, EntityUtils.findFreeLocation selects a valid spawn point for offspring.

SOLID Alignment:

- SRP: SpiritGoat focuses on reproduction timing; EntityUtils handles adjacency logic.
- Open/Closed: Any future "blessed" entity, shrines, artifacts, NPCs, can simply add the BLESSED_BY_GRACE tag. No edits to SpiritGoat.
- Dependency Inversion: Code relies on a shared utility and the GameCapability enum, not concrete ground or actor classes.

2. Egg Item with Turn-Based Hatching (Omen Sheep)

Mechanism

OmenSheep uses an EggItem with TurnBasedHatch, which spawns a parameterized EggItem. That item is made with:

- A TurnBasedHatch strategy (3-turn countdown),
- A factory for OmenSheep::new,
- A consumer value (+10 max health).

Each turn, the egg's tick() calls its strategy's canHatch, and after 3 turns, the egg hatches, spawns a sheep, and removes itself.

SOLID Alignment

- SRP: OmenSheep tracks egg timing; EggItem handles lifecycle and consumption; strategies encapsulate each hatching rule.
- Open/Closed: New hatching rules (e.g., proximity-based) require only a new HatchStrategy implementation.
- Dependency Inversion: EggItem depends on the HatchStrategy interface, not on specific strategy classes.
- Interface Segregation: The Consumable interface exposes only consumption logic to actors; eggs don't need to know about farming code.

3. Flexible Consumption System

Mechanism

EggItem implements Consumable, enabling Farmer actors to use ConsumeAction. On consumption, EggItem.onConsume modifies health/stamina and removes the egg.

SOLID Alignment

- ISP: By keeping consumption in its own interface, the Farmer class remains unaware of individual egg internals.
- Open/Closed: New edible entities can simply implement Consumable and provide their own onConsume logic.

4. Avoiding Hard-Coded Checks

Mechanism

Rather than checking for specific classes (e.g., InheritreeGround), all proximity tests use EntityUtils.hasHereOrNearbyWithCapability(loc, cap). We unified "blessed" and future tags under a single GameCapability enum. This avoids coupling to specific classes like InheritreeGround.

Alternatives Considered

- Centralized Spawning Manager

    - A global service scans all actors for reproduction each turn.
    - Rejected: Centralizes unrelated logic, violates SRP

- Hard-Coded Class Checks
    - Actors explicitly test for instanceof InheritreeGround.
    - Rejected: Breaks OCP, any new blessed type would require code edits in each actor.
- Separate Egg Subclasses
    - One subclass per creature (OmenSheepEgg, GoldenBeetleEgg, etc.).
    - Rejected: Duplicates hatching and consumption logic; harder to maintain.

Limitations & Compromises

- Performance: Repeated adjacency scans add overhead but are acceptable for small to medium maps in a turn-based game.
- Turn-Based Focus: All strategies assume discrete turns; real-time or event-driven requirements would need new strategy implementations.
- Enum Growth: As capabilities proliferate, consider grouping or splitting enums to maintain clarity.

Conclusion

Our implementation for Requirement 1 cleanly separates concerns and adheres to all SOLID principles:

- SRP: Clear separation of spawning, adjacency, and lifecycle logic.
- OCP: Extensible via capabilities, strategies, and interfaces, no core modifications needed.
- DIP/LSP: Dependencies on abstractions (e.g., HatchStrategy, Consumable) ensure substitutability.
- ISP: Narrow interfaces like Consumable prevent leakage of unrelated logic.

This foundation supports future requirements, whether new reproductive triggers, cursed-egg mechanics, or more complex follow-behaviours, by simply creating new strategy or behaviour classes and tagging entities appropriately.

Requirement 2

Requirement Overview

Requirement 2 introduces the Golden Beetle, a small non-hostile NPC with the following behaviors:

- Egg-laying every five turns, producing a “Golden Egg” (EggItem display ‘0’).
- Conditional hatching of eggs only when adjacent to cursed ground (GameCapability.CURSED) via a SurroundingsHatchStrategy.
- Consumption by the Farmer when adjacent (without pickup), healing 15 HP and granting 1000 runes.
- Follow behavior: beetles scan only adjacent tiles for any actor tagged Status.FOLLOWABLE and pursue that target until it disappears, then wander.
- Egg consumption in inventory restores 20 stamina through the egg’s consumerValue parameter.

Design Goals

- Extensibility: Future consumables (e.g., special fruits) and new NPC behaviors (e.g., territorial creatures) should plug in without modifying Golden Beetle core logic.
- Reusability: Leverage the generic EggItem, Consumable, and Behaviour interfaces so we don’t rewrite egg or consumption mechanics.
- Separation of Concerns: Isolate egg-laying, hatching, following, and consumption into dedicated modules or strategies.
- Compatibility: Easily integrate with the existing engine’s tick-based update loop, capability system, and action framework.
- SOLID Compliance: Each class or interface must have a single responsibility, depend on abstractions, and remain open for extension but closed for modification.

Key Design Decisions & SOLID Alignment

1. Unified Egg Lifecycle via EggItem

Mechanism

We reuse the existing EggItem class by parameterizing it with:

- A HatchStrategy (SurroundingsHatchStrategy for cursed-ground adjacency),
- A factory (GoldenBeetle::new),
- A capability to modify
- A consumerValue of 20 for stamina restoration.

SOLID

- SRP & ISP: EggItem handles ticking, hatching delegation, and consumption; the Consumable interface keeps eat-action logic separate from Farmer code.

- OCP & DIP: New egg types require only new HatchStrategy implementations, with no change to EggItem itself. The item depends on the HatchStrategy abstraction.

2. Scheduled Egg Laying

Mechanism

GoldenBeetle tracks an eggCounter in playTurn and increments it each turn. On the fifth increment, it creates an EggItem, passing a CursedSurroundingsHatch strategy, GoldenBeetle::new factory, and consumerValue 20, adds it to its current tile, resets the counter, and returns a DoNothingAction to prioritize egg-laying.

SOLID

- SRP: GoldenBeetle schedules eggs; EggItem and hatch strategies manage lifecycle and hatching.
- OCP: Changing intervals or hatch rules requires only new strategy parameters, not core edits.
- DIP: Beetle depends on the abstractions EggItem and HatchStrategy rather than concrete implementations.
- LSP: Any HatchStrategy can replace the current one without altering beetle logic.

3. Contextual Hatching Strategy

Mechanism:

SurroundingsHatchStrategy inspects adjacent tiles using EntityUtils.hasNearbyWithCapability(loc, GameCapability.CURSED) rather than hard-coding cursed-ground checks. When canHatch returns true, it delegates to the egg's factory to spawn a new beetle.

SOLID

- SRP: Hatching rules live entirely within strategy classes.
- OCP: New strategies (e.g., timed or event-triggered) plug in easily.
- DIP: EggItem invokes only the CursedSurroundingHatch class from HatchStrategy interface.

4. Adjacency-Based Follow Behavior

Mechanism

Instead of a global scan, FollowBehaviour first checks adjacent tiles for any actor marked Status.FOLLOWABLE. When found, it sets it as the target and allows the existing beetle to pathfind. If the target disappears, the beetle clears its target and reverts to a WanderBehaviour.

SOLID

- SRP: The FollowBehaviour encapsulates when and how to follow.
- OCP: Any actor can become followable simply by acquiring the FOLLOWABLE tag, no beetle code changes.
- DIP: Both behaviors depend on the abstract Behaviour interface, not concrete movement code.

5. Flexible Consumption via Consumable

Mechanism

GoldenBeetle and its eggs implement Consumable, exposing a ConsumeAction when adjacent (beetle) or in inventory (egg). The beetle’s onConsume grants health and runes, while the egg’s logic grants stamina.

SOLID

- ISP: Players interact through the Consumable interface; core NPC and item classes stay focused on their own rules.
- OCP: Future edible NPCs or items need only implement Consumable to gain eat-action support without modifying the action framework.

Alternatives Considered

- Dedicated Egg Subclasses

    - Creating GoldenBeetleEgg with its own logic.
    - Rejected: Duplicated code and violated DRY; new egg types would demand new classes.

- Central NPC Manager

    - A global service to handle all NPC reproduction, hatching, and following.
    - Rejected: Introduced a class that violates SRP and complicates unit testing.

- Hard-Coded Follow Logic in playTurn()

    - Embedding movement and pathfinding directly in GoldenBeetle.
    - Rejected: Mixed concerns; avoided reusable FollowBehaviour.

Limitations & Trade-Offs

- Performance Overhead: Adjacent-tile scans for both egg hatching and follow-target selection occur each turn. While acceptable for typical map sizes, a very large world could amplify this cost.
- Capability Coupling: Dependence on GameCapability.CURSED or FOLLOWABLE enums can become brittle if names change. Central enum maintenance is needed to handle ripples.

Conclusion

The Golden Beetle system for Requirement 2 cleanly handles generic behaviors, strategies, and utilities to deliver all specified mechanics:

- Egg-laying at fixed intervals through a reusable behavior.
- Contextual hatching via the strategy pattern and unified adjacency utilities.
- Follow-target selection is local to each beetle, using the existing follow behavior for pathfinding.
- Player consumption through the Consumable interface, dividing item/NPC logic from player actions.

By adhering to SOLID principles, isolating responsibilities, depending on abstractions, and remaining open to extension, this design supports future expansions (new NPCs, dynamic hatching conditions, or consumables) with minimal friction and maximum reuse. The separation of concerns makes each component straightforward to test, maintain, and evolve as the assignment's requirements grow.

Requirement 3

Requirement Overview

Requirement 3 adds three story-driven NPCs, Sellen, Merchant Kale, and Guts, each of whom the Farmer can listen to when adjacent. Listening triggers a randomly selected "monologue" drawn from an NPC's pool, which may vary based on game state (player runes, inventory, health, or cursed terrain nearby). In addition, Guts will attack any actor with more than 50 HP, while Sellen and Kale merely wander.

Design Goals

- Extensibility: Support new NPCs with conditional or static monologues without rewriting core listening logic.
- Reusability: Leverage a single MonologueSource interface and a shared ListenAction so all NPCs uniformly expose dialogue options.
- Separation of Concerns: Keep NPC behavior (monologue composition, attack triggers, wandering) separate from action dispatch (i.e. listening mechanics).
- Compatibility: Integrate with the existing turn-based engine, capability system (Status.CAN_PROVIDE_MONOLOGUE, Status.FOLLOWABLE, etc.), and ActionList framework.
- SOLID Alignment: Ensure each class has a single responsibility, depends on abstractions, and remains open for extension while closed to modification.

Key Design Decisions & SOLID Alignment

1. MonologueSource Interface

- Mechanism: All three NPC classes implement game.interfaces.MonologueSource, which defines List<String> getMonologues(Actor listener, GameMap map).
- SRP: NPCs supply only monologue content; the interface doesn't define how those lines are chosen or displayed.
- OCP/DIP: New NPCs simply implement MonologueSource, or existing ones add new conditions, without altering listening mechanics. The listening system depends on the interface, not concrete NPC types.

2. Capability-Driven Dialogue Availability

- Mechanism: Each NPC's constructor tags itself with Status.CAN_PROVIDE_MONOLOGUE. In their allowableActions(...), they check adjacency and non-empty monologue lists to add a single ListenAction.
- SRP & ISP: The NPC's allowableActions method handles both combat and dialogue choices, but dialogue logic is limited to "should I offer a ListenAction?" The ListenAction class itself retrieves, picks, and returns a random line.

- OCP: Any actor with the CAN_PROVIDE_MONOLOGUE capability can be “listenable”, no need to modify ListenAction or the engine.

3. NPC-Specific Monologue Pools

- Sellen: Returns a fixed list of two lore-driven lines unconditionally.
- Kale: Builds a dynamic list by checking, in order,

    - Farmer’s runes < 500, adds a poverty line.
    - Farmer’s empty inventory, adds an inventory quip.
    - Proximity to cursed ground (GameCapability.CURSED), adds a caution line.
    - All matching lines are collected, and ListenAction picks one at random.

- Guts: If the Farmer’s health ≤ 50, returns only the mocking line; otherwise returns two battle cries.

SOLID:

- SRP: Each NPC’s getMonologues(...) method owns only its own content-selection logic.
- OCP: Adding new conditional lines (e.g., “if time of day is night”) only requires editing that NPC’s getMonologues, the listening framework remains untouched.
- LSP: Any MonologueSource implementation can be substituted and will function identically from the engine’s perspective.

4. Action Injection via allowableActions

- Mechanism: Within each NPC’s override of allowableActions(Actor otherActor, String dir, GameMap map), after combat checks they:

    - Verify adjacency by comparing exits.
    - Call getMonologues(otherActor, map) to see if any lines are available.
    - If non-empty, add a single ListenAction.

- SRP & DIP: NPCs simply decide when to offer dialogue; ListenAction handles how to select and present a line.
- OCP: New NPCs or new dialogue conditions are accommodated by editing only that NPC class’s getMonologues or adjacency check, not the engine or ListenAction.

Alternatives Considered

1. Hard-coded Dialogue in playTurn

- Embedding monologue logic in the NPCs’ playTurn methods.

- Rejected: Mixed action sequencing and dialogue generation, violating SRP and making maintenance harder.

2. Central Dialogue Manager

- A global service that scanned all actors for monologue sources each turn and presented a menu.
- Rejected: Centralizes unrelated logic, and complicates testing and extension for individual NPC behaviors.

3. Exclusive Monologue Conditions

- Enforcing strict priority (e.g., only one line per visit) via if–else chains.
- Trade-Off: The current design allows multiple qualifying lines to join the pool, enhancing variety, at the cost of slight divergence from a strict “first-match only” interpretation.

Conclusion

Our Requirement 3 implementation cleanly layers dialogue mechanics atop the existing engine:

- Monologue generation is governed by the MonologueSource interface, letting each NPC freely define its own pool.
- Dialogue availability uses a shared ListenAction and capability tag, keeping NPCs agnostic to the underlying action framework.
- SOLID compliance is maintained throughout: single-purpose methods, dependence on interfaces and enums, and the ability to extend dialogue features by adding new lines or NPCs, without altering the engine or the core listening code.

This design ensures that as the valley’s cast of characters grows (with novel conditional monologues, dynamic content, or even conversation trees), we need only implement or extend individual MonologueSource classes and adjust NPC constructors, leaving the listening infrastructure entirely stable and reusable.

Requirement 4

Introduction & Design Goals

Requirement 4 adds an economic layer: the Farmer can buy three weapons directly from adjacent merchants, each purchase costing one turn and triggering merchant-specific effects. Primary goals were:

- **Extensibility** – allow new merchants or weapons without touching existing logic.
- **Separation of Concerns** – keep selling, combat stats, and stat-modification distinct.
- **No RTTI** – meet the ban on `instanceof` while still knowing "who sold the item."
- **Single-menu Flow** – purchase options appear in the same `Menu` already rendered by the engine.
- **SOLID & DRY** – every class holds one reason to change; duplicated code is factored away.

Overall Concept

Merchants are normal actors tagged with a new capability enum `MerchantId` (`SELLEN`, `KALE`).
When a merchant's `allowableActions()` runs, it inspects its own capability and adds the appropriate **buy-actions** (`BuyBroadswordAction`, `BuyDragonslayerAction`, `BuyKatanaAction`) straight into the player's main action list.
Each buy-action deducts runes from the wallet, inserts the weapon into the Farmer's inventory, and applies global and seller- specific bonuses (HP heal, max-stat boosts, creature spawns).

Weapons themselves are simple subclasses of `WeaponItem` carrying only combat numbers and glyphs; they know nothing about costs or bonuses.

Key Design Decisions & SOLID Alignment

1. ### Capability-driven merchant identity

    - Merchants call `addCapability(MerchantId.SELLEN)` (or `KALE`).
    - Buy-actions receive this enum, avoiding any class comparisons.
    - *Liskov Substitution Principle* – any actor with a `MerchantId` behaves legally as a seller.

2. ### One action class per purchasable weapon

    - `BuyBroadswordAction` encapsulates price, wallet deduction and HP + max-stat logic; other weapons follow the same pattern.
    - *Single Responsibility Principle* – an action has exactly one reason to change: the rules of that purchase.
    - *Open/Closed Principle* – a new weapon requires a new action, leaving existing classes unchanged.

3. ### Capability-driven weapon identification

    - WeaponItem subclasses tag themselves with `ItemCapability.WEAPON`
    - `allowableActions()` and `AttackAction` scan the inventory for that capability rather than using `instanceof`.
    - Dependency Inversion & Open/Closed – combat logic depends on an abstract capability, not concrete classes.

4. ### Creature spawn encapsulation

    - The Dragonslayer action instantiates a `GoldenBeetle` near the Player when the seller is Sellen; Katana does similar for an `OmenSheep`.
    - Spawn code lives only inside the relevant action, preventing bloated merchant classes.

5. ### Wallet-first validation

    - Each buy-action checks funds before proceeding; failure returns a descriptive string and consumes no runes.

    - Aligns with *Interface Segregation Principle* – the wallet API stays tiny (`addBalance`, `deductBalance`).

Advantages

- **Menu cohesion** – the Farmer sees "Buy Broadsword (100 runes)" alongside ordinary actions; no extra submenu.

- **Low coupling** – merchants need not know weapon effects; weapons need not know merchants.

- **Ease of extension** – adding "Edgar the Smith" requires: tag him `MerchantId.SMITH`, create `BuyWarhammerAction`, and expose that action in `allowableActions()`, no core edits.

- **Testability** – each buy-action can be unit-tested by mocking a Player and verifying wallet and attribute deltas.

Limitations & Trade-offs

- **Many small files** – every new weapon entails a new action class. The clarity gain outweighs the minor overhead; code navigation remains simple.

- **Hard-coded prices inside actions** – changing a cost means editing one constant. Central pricing tables were considered unnecessary complexity at this scale.

- **Creature-spawn coupling** – actions instantiate concrete creature classes. If spawn logic becomes more varied a factory pattern can be introduced later.

Alternatives Considered

- **Generic `ShopAction` that opened a sub-menu** – rejected because the requirement explicitly forbids secondary menus.

- **Single `BuyWeaponAction` with a switch on weapon enum** – would violate SRP and grow unwieldy as weapons multiply.

- **Embedding cost/effect data inside the weapon class** – couples economic rules to combat equipment, hindering the reuse of the same weapon found as loot.

Conclusion

The design meets Requirement 4 while honouring SOLID principles, banning `instanceof`, and adhering to the one-menu constraint.
 Capability tags identify merchants; dedicated buy-actions encapsulate economic logic; helper methods in `Player` prevent scattering attribute code.
 Future growth, new currencies, discount rules, or exotic vendors, slots cleanly into the same pattern, ensuring the valley's fledgling economy remains maintainable and robust.