REQ2: Design Rationale

(Animal Spawning II: Revenge of the Spawners)

A. Spawner Structure and Placement

Design A1: Centralized Manual Placement Manager

A single SpawnManager handles placement and rules for all spawners across maps.
Pros:

- Centralized control
- Consistent placement logic
- Easier to verify placement constraints

Cons:

- Violates SRP (single class responsible for multiple biomes)
- Difficult to extend with new spawner types
- Tight coupling between maps and spawn logic
- Hard to customize per-biome behavior

Design A2: Biome-Specific Spawner Ground Classes

Each biome spawner is a concrete Ground subclass (Tundra, Cave, Meadow, Swamp) encapsulating spawn rules and timing.
Pros:

- Adheres to SRP (each spawner manages its own logic)
- Supports OCP (add new spawners without modifying existing ones)
- Clear separation of concerns
- Natural integration with terrain system
- Map-specific customization straightforward

Cons:

- More classes to maintain
- Risk of code duplication without careful design
- Requires consistent interface/abstract base

Selection: Design A2

Design A2 is selected because it follows SRP and OCP, aligns with the engine's terrain model, and supports map-specific constraints (e.g., Forest swamps spawn crocodiles + deer; Plains swamps spawn only crocodiles). It enables extensibility and keeps responsibilities focused.

B. Spawning Timing and Probability

Design B1: Fixed-Interval Spawning

All spawners use a fixed turn interval (e.g., every 5 turns).
Pros:

- Simple and predictable
- Easy to test and balance
- No dependency on map state

Cons:

- Cannot model conditional spawning (e.g., Swamp requires actor nearby)
- Less flexible for varying spawn conditions
- Cannot handle probability-based mechanics (50% chance)

Design B2: Conditional and Probability-Driven Spawning

Spawners check conditions and probabilities for each tick. Swamps spawn with 50% chance only when an actor is nearby. Other spawners use their own timing/probability rules.
Pros:

- Supports conditional spawning (swamp proximity check)
- Flexible probability mechanics (50% chance)
- Realistic and dynamic
- Per-biome customization

Cons:

- More complex logic per spawner
- Requires careful testing of probabilities
- Potential performance overhead (checks each tick)

Selection: Design B2

Design B2 is selected because REQ2 requires conditional spawning (swamp proximity) and probability-driven mechanics (50% chance). It is more flexible and realistic, allows per-biome customization, and supports future requirements.

C. Animal Selection Logic

Design C1: Single Species Per Spawner

Each spawner spawns only one animal type.

Pros:

- Simple logic
- No randomization needed
- Predictable behavior

Cons:

- Cannot support multiple species per spawner
- Inflexible for requirements (Meadows in Forest spawn deer + crocodile)
- Requires many spawner instances for variety
- Violates DRY if spawners share logic

Design C2: Randomized Selection from Allowed Species Set

Each spawner maintains an allowed species list (per map). On spawn, one is selected randomly with equal probability (unless specified otherwise).

Pros:

- Supports multiple species per spawner
- Flexible for map-specific profiles
- Matches requirements (Meadows spawn deer + crocodile in Forest)
- Centralized species-to-biome mapping

Cons:

- Requires seedable RNG for testing
- More complex selection logic
- Need to track allowed species per map/biome

Selection: Design C2

Design C2 is selected because REQ2 requires multiple species per spawner (e.g., Meadows in Forest spawn deer + crocodile; Tundra in Plains spawn crocodile). It supports map-specific profiles, reduces duplication, and enables extensibility.

D. Extensibility and Biome Differentiation

Design D1: Centralized Spawn Controller

A single SpawnController manages all spawner types with conditional logic (if-else/switch) to handle differences.

Pros:

- All spawn logic in one place
- Easy to see all rules
- Single point of modification

Cons:

- Violates OCP (modify controller for new biomes)
- Violates SRP (controller handles all spawner types)
- Large conditional blocks
- High connascence of algorithm
- Hard to test in isolation

Design D2: Abstract Spawner with Biome-Specific Subclasses

An abstract Spawner (or Ground subclass) defines template methods for timing, probability, and allowed species. Concrete classes (Tundra, Cave, Meadow, Swamp) implement specific behavior.

Pros:

- Adheres to OCP (add subclasses without modifying base)
- Adheres to SRP (each spawner has focused responsibility)
- Isolated changes per biome
- Template Method pattern supports reuse
- Easy to test per spawner

Cons:

- More class hierarchy
- Requires consistent abstract interface
- Risk of duplicate code without careful design

Selection: Design D2

Design D2 is selected because it follows OCP and SRP, isolates change by biome, uses a clear template for timing/probability/species, and enables extension without modifying existing code.

E. Connascence Analysis

Identified Forms of Connascence

Connascence of Name (CoN):

- Post-spawn effect registry keys must match animal class names
- Spawn profile keys must match map names and terrain class names

Connascence of Meaning (CoM):

- Constants like SWAMP_POISON_DURATION = 10 must be understood consistently
- Warmth threshold (0 = unconscious) must match across classes

Connascence of Algorithm (CoA):

- RNG seeding for deterministic tests must be consistent
- Probability calculations (50% chance) must use the same formula

Connascence of Position (CoP):

- Registration order in static initializers (not critical but exists)
- Spawn profile structure (map → terrain → species list)

Connascence of Timing (CoT):

- Post-spawn effects must run after actor is added to map
- Warmth decay must occur each turn

Connascence of Type (CoTy):

- Registry keys must use Class<? extends Actor> consistently
- Status effect interfaces must match capabilities

Connascence Reduction Strategies

Type-Safe Keys:

- Use Class<? extends Actor> instead of strings to reduce CoN
- Compile-time type checking reduces errors

Centralized Constants (Tuning):

- Single source for all numeric constants reduces CoM
- Constants referenced directly, not duplicated

Explicit Interfaces:

- PostSpawnEffect interface standardizes effect contracts (reduces CoTy)
- AnimalFactory interface standardizes creation (reduces CoTy)

Registry Pattern:

- Centralized lookup reduces CoP (no scattered mappings)
- Explicit registration makes dependencies visible

Template Method Pattern:

- Abstract spawner defines structure, subclasses fill details (reduces CoA)
- Consistent lifecycle hooks reduce CoT

Future Connascence Reduction Opportunities

1. Dependency Injection: Replace static registries with injected dependencies
2. Configuration Objects: Extract spawn profiles to configuration classes
3. Effect Composition: Chain effects through a pipeline interface
4. Builder Pattern: Use builders for complex spawner setup

F. Future Extensibility

Adding New Biomes

1. Create a new Ground subclass (e.g., Desert)
2. Implement abstract spawner methods (timing, probability, allowed species)
3. Add spawn profile entry in Earth with map-specific allowed species
4. Register any biome-specific effects if needed

No modifications to existing spawners.

Adding New Animals

1. Create the animal class (e.g., Lion extends TameableAnimal)
2. Register factory: AnimalRegistry.register(Lion.class, Lion::new)
3. Register post-spawn effect: PostSpawnEffectRegistry.register(Lion.class, new LionRoarEffect())
4. Update spawn profiles in Earth (add Lion.class to relevant terrain lists)

No modifications to existing spawning code.

Adding New Post-Spawn Effects

1. Implement PostSpawnEffect (e.g., LionRoarEffect)
2. Register in PostSpawnEffectRegistry
3. Optionally create composite effects for multiple behaviors

No changes to spawners or existing effects.

Stable Extension Seams

1. AnimalRegistry: Central lookup for animal creation
2. PostSpawnEffectRegistry: Central lookup for post-spawn behaviors
3. Abstract Ground spawner pattern: Template for new biome spawners
4. StatusEffect framework: Extensible status system (reused for poison)
5. SpawnHelper: Utility methods for spawn coordination

OCP Compliance

The design adheres to OCP: new biomes, animals, and effects are added via registration and subclassing without modifying existing code. This maintains system stability while enabling extension.

G. Post-Spawn Effects (REQ2 Extensions)

Requirements from REQ2

Deer Spawn: Drop one apple in a random exit of the spawner.
Bear Spawn: Scatter yew berries in surrounding exits with 50% chance per exit.
Wolf Spawn: Grow a mature YewBerryTree in one random exit. This tree drops a berry when any actor is adjacent (reactive, not every 5 turns).
Crocodile Spawn: Poison all actors in the spawner's surroundings for 3 turns, dealing 10 damage per turn. These effects apply regardless of spawner type (Swamp, Tundra, Meadow, Cave).

Design G1: Hardcoded Effects in Each Spawner

Each spawner (Tundra, Cave, Meadow, Swamp) contains conditional logic to apply effects based on spawned animal type.
Pros:

- Effects co-located with spawn logic
- No additional registry needed
- Direct control over effect application

Cons:

- Violates SRP (spawners manage both spawning and effects)
- Violates DRY (effect logic duplicated across spawners)
- Violates OCP (modify spawners to change effects)
- High connascence of algorithm (must match effect logic in all spawners)
- Difficult to test effects in isolation
- Hard to add new effects without modifying existing spawners

Design G2: Central Post-Spawn Effect Registry with Strategy Pattern

A PostSpawnEffectRegistry maps animal types to effect strategies. Each effect implements the PostSpawnEffect interface. After spawning, the registry is consulted and the effect is applied.
Pros:

- Adheres to SRP (effects separated from spawners)
- Adheres to DRY (effect logic in one place)
- Adheres to OCP (register new effects without modifying spawners)
- Low connascence (single registry, clear interface)
- Easy to test effects in isolation
- Supports effect composition/chaining
- Spawners remain spawner-agnostic

Cons:

- Additional abstraction layer
- Requires consistent registration during initialization
- Slight indirection (registry lookup)

Selection: Design G2

Design G2 is selected because it follows SRP, DRY, and OCP, reduces connascence, supports universal application across spawners, and enables easy extension. Effects are testable and composable, and spawners remain focused on spawning.

H. Requirements Traceability

New Animal: Crocodile

Requirement	Design Decision	Implementation	Testing
Crocodile (<) with 300 HP	Tuning.CROCODILE_HP = 300	Crocodile constructor	crocodile_creation_has_correct_stats_cases()
Bite attack: 80 damage, 75% hit	CrocodileBite weapon, Tuning constants	Crocodile.setIntrinsicWeapon()	crocodile_bite_attack_stats_cases()
Warmth starts at 55, removed at 0	BaseAttributes.WARMTH, Earth.handleAnimalWarmthDecrease()	Crocodile constructor + warmth decay	crocodile_warmth_decay_and_removal_cases()

Spawner Updates

Requirement	Design Decision	Implementation	Testing
Meadows in Forest spawn crocodiles	SPAWN_PROFILES in Earth	forestProfile.put(Meadow.class, Arrays.asList(Deer.class, Crocodile.class))	Integration test
Tundra in Plains spawn crocodiles	SPAWN_PROFILES in Earth	plainsProfile.put(Tundra.class, Arrays.asList(Wolf.class, Crocodile.class))	Integration test

New Spawner: Swamp

Requirement	Design Decision	Implementation	Testing
Swamp (~) spawner type	Swamp extends Ground	Swamp class with tick()	swamp_spawning_mechanics_cases()
50% chance if actor nearby	Condition + probability check in tick()	Swamp.tick() proximity check + Tuning.SWAMP_SPAWN_CHANCE	swamp_spawning_mechanics_cases()
Swamp-spawned animals get poison (10 turns, 5 dmg/turn)	Poison applied immediately after spawn	Swamp.tick() applies PoisonEffect	swamp_poison_application_cases()
≥1 Forest swamp (crocodile + deer)	ensureRequiredSpawners()	Earth constructor + spawn profile	Integration test
≥1 Plains swamp (crocodile only)	ensureRequiredSpawners()	Earth constructor + spawn profile	Integration test

Post-Spawn Effects

Requirement	Design Decision	Implementation	Testing
Deer → drop 1 apple in random exit	DeerAppleDropEffect	PostSpawnEffectRegistry + DeerAppleDropEffect.apply()	deer_apple_drop_effect_cases()
Bear → 50% chance yew berry per exit	BearYewberryScatterEffect	PostSpawnEffectRegistry + BearYewberryScatterEffect.apply()	bear_yewberry_scatter_effect_cases()
Wolf → grow reactive yewberry tree	WolfTreeGrowthEffect + YewBerryTree.enableProximityDrop()	PostSpawnEffectRegistry + tree with proximity observer	wolf_tree_growth_effect_cases(), wolf_spawned_tree_proximity_drop_cases()
Crocodile → poison surroundings (3 turns, 10 dmg/turn)	CrocodilePoisonPulseEffect	PostSpawnEffectRegistry + CrocodilePoisonPulseEffect.apply()	crocodile_poison_pulse_effect_cases()
Effects apply regardless of spawner	Universal application in SpawnHelper	PostSpawnEffectRegistry.getFor() called after spawn	Integration tests

I. Risks, Testing & Balancing

Identified Risks

- Random Clustering: Animals may cluster near swamps due to proximity-based spawning
    - Mitigation: Limit spawn frequency and ensure spawn location is unoccupied
- Spawn Storms: Multiple swamps near player could trigger many spawns
    - Mitigation: 50% chance per tick limits spawn rate; location must be unoccupied
- Resource Starvation: If deer spawn in Plains but requirements don't specify, verify spawn profiles
    - Mitigation: Spawn profiles explicitly define allowed species per map/biome
- Poison Double-Counting: Swamp poison + crocodile pulse could stack
    - Mitigation: Status effect system handles stacking; durations and DPT are separate
- Performance: Proximity checks every tick for all swamps
    - Mitigation: Early exit if no actor nearby; efficient exit iteration
- Deterministic Testing: Random probabilities make testing difficult
    - Mitigation: Seeded RNG in tests (Random(42)) for reproducible results

Testing Strategy

Unit Tests:

- Crocodile stats, warmth decay, bite attack
- Swamp spawning mechanics (proximity, probability, no-spawn conditions)
- Post-spawn effects (apple drop, berry scatter, tree growth, poison pulse)
- Probabilistic tests with seeded RNG

Integration Tests:

- End-to-end spawning from different spawner types
- Effect application verification
- Spawn profile validation across maps

Boundary Cases:

- Warmth at 1 (next tick removal)
- Spawner with no exits (no crash)
- Multiple actors nearby (all receive poison)
- Empty allowed species list

Balancing Verification:

- Constants from Tuning verified in tests
- Probability distributions validated with seeded RNG
- Damage/duration values match requirements

J. UML & Code Alignment

UML Structure

The design maps to:Abstract Layer:

- Ground (engine base class)
- PostSpawnEffect (interface)
- AnimalFactory (functional interface)

Concrete Spawners:

- Tundra extends Ground
- Cave extends Ground
- Meadow extends Ground
- Swamp extends Ground

Registries:

- AnimalRegistry (static factory registry)
- PostSpawnEffectRegistry (static effect registry)

Post-Spawn Effects:

- DeerAppleDropEffect implements PostSpawnEffect
- BearYewberryScatterEffect implements PostSpawnEffect
- WolfTreeGrowthEffect implements PostSpawnEffect
- CrocodilePoisonPulseEffect implements PostSpawnEffect

Animals:

- Crocodile extends TameableAnimal
- Bear, Wolf, Deer (existing, extended with post-spawn effects)

Utilities:

- Tuning (constants)
- SpawnHelper (shared spawn coordination)

Code Alignment Confirmation

Implementation matches the rationale:

1. Biome-specific spawners: Swamp, Tundra, Cave, Meadow as Ground subclasses
2. Conditional/probability spawning: Swamp.tick() checks proximity + 50% chance
3. Randomized species selection: SpawnHelper selects from allowed species list
4. Registry-based effects: PostSpawnEffectRegistry maps animals to effects
5. Centralized constants: Tuning provides all numeric values
6. Universal effect application: SpawnHelper.attemptSpawn() calls registry after spawn

The code structure enables adding new biomes, animals, and effects without modifying existing code, satisfying OCP. Responsibilities are separated (SRP), constants centralized (DRY), and dependencies inverted (DIP) through registry abstractions.

Conclusion

This design follows SOLID and DRY principles through:

- Single Responsibility: Each spawner and effect has one clear purpose
- Open/Closed: Extension via subclasses and registration, no modification of existing code
- Dependency Inversion: Spawners depend on registry abstractions, not concrete implementations

Connascence is minimized through type-safe registries, centralized constants, and explicit interfaces. The structure supports adding new biomes, animals, and effects without modifying existing code, ensuring maintainability and stability while enabling future extension.