REQ4- Design Rationale

Summary of the design of REQ3/REQ4

REQ3 introduced a small, engine-local status-effect framework and weapon actions; REQ4 used that framework to add coatings (Snow, Yewberry) that cause effects (Frostbite, Poison) when weapons hit. The implementation separates responsibilities:

- GameActor - stores and ticks StatusEffects (one list per actor).
- StatusEffect (interface) - polymorphic per-tick behavior: BleedEffect, BurnEffect, PoisonEffect, FrostBiteEffect.
- StatusRecipient & StatusRecipientRegistry - lightweight connector so non-GameActor code (FireGround) can add effects without instanceof.
- Coatable + CoatingType - small interface + enum to let items (weapons) store coatings.
- CoatAction, Snow, YewBerry - lets players apply coatings via standard allowableActions.
- Weapons (Axe, Bow) expose allowableActions that produce AttackAction objects; AttackAction consults weapon coating and applies the corresponding StatusEffect on hit.
- FireGround spawns fire and adds BurnEffect to actors stepping through it; underlying ground becomes Dirt.

This design uses the existing engine API (Actor.allowableActions, Action, Ground.tick, Actor.modifyAttribute, Actor.hurt) without changing engine classes.

Key Design Choices

DRY (Don't Repeat Yourself)

- All timed, per-tick damage or per-tick attribute reduction behavior is contained in StatusEffect implementations (Bleed/Burn/Poison/FrostBite). This avoids duplicating "tick & decrement" code across multiple classes as GameActor.tickStatusEffects() centralizes the tick loop.
- Coating behaviour is stored on weapons (Coatable) and applied in attack actions. The code to apply effects lives in StatusEffect classes hence no duplicated logic across Axe/Bow/Torch beyond small lookups.
- CoatAction is generic: same code works for Snow and YewBerry.

KISS (Keep It Simple, Stupid)

- Small, focused abstractions: StatusEffect has only 4 methods; Coatable has only 3. Complexity is localized.
- Defers special-case logic (e.g., tundra immunity) to effect implementations; attack actions only attach an effect instance.
- Uses existing allowableActions hook for providing coat options — avoids inventing new UI/command plumbing.

SOLID

- Single Responsibility: Each class has one reason to change:
    - Effects manage per-tick semantics.
    - GameActor manages lifecycle of effects on actors.
    - CoatAction handles applying coatings & consuming item.
    - Weapon items handle action generation and storing coating state.
- Open/Closed: New coatings or effects can be added by implementing StatusEffect and adding a CoatingType enum constant. Attack actions read weapon.getCoating() and can be extended to apply new effects without changing existing effect classes.
- Liskov Substitution: StatusEffect implementers conform to the same interface; they are interchangeable where a StatusEffect is needed.
- Interface Segregation: Small, focused interfaces (Coatable, StatusRecipient) keep the public API minimal.

- Dependency Inversion: High-level logic (attack actions) depends on StatusEffect types and Coatable interface rather than concrete implementations. StatusRecipientRegistry decouples ground logic from actor concrete types.

Connascence

- Name-based connascence: CoatingType enum is the canonical name set; attack actions use the enum values (tight coupling is intended and explicit). This is limited in scope and easy to update.
- Type-based connascence: GameActor exposes addStatusEffect because code needs to add effects; we used StatusRecipientRegistry to avoid instanceof and limit type coupling.
- Temporal connascence: The design depends on actor tick ordering (status effects being applied at the start of an actor's turn). This dependency is documented hence if engine turn-order semantics change, effect semantics would need review.

Pros / Cons

Pros

- Re-uses REQ3 framework: Coatings reuse the existing StatusEffect framework and GameActor lifecycle, minimizing new surface area and duplication.
- Simple developer model: Coating is stored on the weapon as a small state (Coatable + CoatingType) and applied by actions, keeping the runtime behaviour easy to follow.
- Extensible: Additional coatings (or even non-weapon coatable items) can be supported with minimal code: add a CoatingType and a StatusEffect, then apply in attack logic or via a factory.
- Testable: Coating application (CoatAction) and effect behaviour (poison, frostbite) are unit-testable in isolation from the engine.
- Respects constraints: No changes to the engine; interactions implemented using existing hooks (allowableActions, Ground.tick, Actor.modifyAttribute).

Cons

- Enum-based coating growth: As the number of coatings grows, the CoatingType enum and dispersed switch/if checks across actions could become unwieldy.
  Mitigation: refactor to a CoatingStrategy/CoatingEffectFactory or decorator pattern later.
- Weapon-state responsibility: Coating state lives on the item. If many items become coatable with varying lifecycle semantics, centralized policies may be needed.

- Implicit semantics: Some behaviour (e.g., frostbite interacting with actor WARMTH) depends on the presence of particular attributes or flags (tundra immunity). This requires careful documentation and consistent attribute usage in other requirements.

Alternatives evaluated

Alternative A - Coating as a first-class object / decorator on the weapon

- *Pros:* Encapsulates coating behaviour, allows per-coating state and behaviour, good for complex coatings.
- *Cons:* More boilerplate, wrapping/unwrapping items, increased runtime objects.
- *Decision:* Deferred as suitable if coatings become behavior-rich. For REQ4, enum + CoatingType is simpler and meets KISS.

Alternative B - Effects created and owned by coating items themselves (weapon triggers item then item manages effect lifecycle)

- *Pros:* Makes coatings active entities that can hold state (e.g., charges).
- *Cons:* Moves effect lifecycle out of actor context; ownership unclear; tougher stacking semantics.
- *Decision:* Rejected for clarity and ownership reasons.

Alternative C - Centralized coating registry/factory

- *Pros:* Central point to map CoatingType to StatusEffect (good for many coatings).
- *Cons:* Extra indirection; for the small number of REQ4 coatings, adds complexity.
- *Decision:* Consider for refactor if coatings grow beyond a few types.

Alternative D - No-coating / one-shot modifiers

- *Pros:* Simple: coating applies single immediate modifier on next hit.
- *Cons:* Cannot represent multi-turn coating effects (poison/frostbite semantics) cleanly or stacking behavior.
- *Decision:* Rejected as does not meet REQ4 functional requirements.

Why chosen approach wins

- Enum-state on weapon + CoatAction + StatusEffect gives the best trade-off: minimal new infrastructure, straightforward unit testing, clear ownership (effects live on GameActor) and straightforward extension path (refactor to strategies if needed).

Maintainability foresight and examples

- Adding a new coating (example: Acid):
    1. Add CoatingType.ACID.
    2. Implement AcidEffect implements StatusEffect.
    3. Update attack actions or add a small CoatingEffectFactory so actions apply AcidEffect on hit.
    - No engine changes required.
- Evolving to richer coatings: If coatings gain complexity (durations on the weapon, per-weapon charges), migrate to a CoatingStrategy or WeaponDecorator pattern. That refactor can be localized to the game.items package.
- Centralizing coating logic: If multiple actions handle coatings similarly, create a small helper CoatingApplier.applyCoating(weapon, target) to avoid repeating coating-check code (DRY).
- Stacking policy changes: If you later wish to cap stacking or change semantics (e.g., non-linear stacking), implement the policy in GameActor.addStatusEffect() as it already centralizes where effects are appended and can enforce caps/coalescing.
- Testing determinism: Where RNG (random number generator) is used (hit-chances), inject a Random instance for deterministic tests. Consider exposing a package-private constructor or setter to allow test injection.

Testing & verification strategies

Unit tests

- Coating application (CoatAction):
    - Verify coat item removed from inventory after apply.
    - Verify weapon.getCoating() returns expected value; re-coating replaces previous coating.
    - Verify torches cannot be coated (REQ4 rule) hence test refusal or not present in allowable actions.
- Status effects (PoisonEffect, FrostBiteEffect):
    - applyEffect() reduces health/warmth as expected.
    - decrementDuration() and isExpired() cycles work correctly.
    - Stacking is applying multiple effects and ensure additive behavior.
- Attack actions (Axe/Bow):
    - Test coated and uncoated branches, deterministic hit/miss using injected RNG.
    - Verify coating-triggered effects are added to target and that YewBerry is consumed upon coating (i.e., not present after coat action).
- Edge cases:
    - Attacking a tundra-spawned actor with snow coating and verify no frostbite applied (if actor tagged correctly).
    - Attempt coating when no coatable weapons present to check UI and allowableActions behaviour.

Integration tests

- Coat → Attack → Tick:
    - Coat Axe with YewBerry → Attack target twice → Simulate ticks and verify poison stacks: 4 HP per turn per application for 5 turns.
- Snow frostbite sequence:
    - Attack target with snow-coated weapon twice in successive turns; verify WARMTH reduction rules match description (including stacking and expiry).

- Torch exclusion:
    - Ensure CoatAction does not list torch as a coatable target; coating torches must be blocked.

Implementation notes

- Torches excluded: enforce in CoatAction / allowableActions that Torch cannot be coated (REQ4 explicit).
- YewBerry consumption: coat must consume the YewBerry from the actor inventory at time of coating.
- Coating replacement: re-coating replaces the current coating.
- Frostbite & WARMTH attribute: FrostBiteEffect should defensively check hasStatistic(WARMTH) before modifying; actors without the WARMTH stat are unaffected.
- Tundra immunity: ensure the flag/ability used to indicate tundra-spawned actors is named consistently and set in REQ1 actor creation; FrostBiteEffect must check this flag to respect immunity.