REQ4- Design Rationale

A. Representing coating state on weapons

Design A1: Enum-state on item (CoatingType) (chosen)

Store a CoatingType enum field inside weapon items that implement Coatable (get/set/clear). CoatAction sets the enum on the chosen weapon; attack actions read weapon.getCoating() to decide whether to add an effect on hit.

Pros	Cons
Very simple to implement and understand (KISS).	As number of coatings grows, action code will contain more if/switch checks (scattered logic).
Low boilerplate; easy to inspect and test.	Limited if coatings need per-coating state (charges, decay) and would require refactor.
Re-coating is simple: setCoating() overwrites prior value.	

Design A2: Coating as object / strategy (CoatingStrategy / decorator)

Wrap weapon with a Coating object (or provide a CoatingStrategy) that encapsulates behaviour/state for a coating.

Pros	Cons
Encapsulates coating behaviour fully (good for complex coatings, stateful coatings).	Extra complexity and runtime objects; more boilerplate to implement.
Avoids switch logic in many places; supports per-coating methods.	Requires wrapping/unwrapping logic for weapons; more invasive change.

Decision: A1 chosen now (enum on item). If coatings increase in complexity/number, refactor to A2 (Strategy/Decorator).

B. How coating effects are applied on hit

Design B1: Attack action inspects weapon.getCoating() and creates StatusEffect instances (chosen)

Attack Actions (AxeAttackAction, BowAttackAction) check weapon.getCoating() at execution time. On a successful hit they add the corresponding StatusEffect (e.g., PoisonEffect(5,4) for YEWBERRY, FrostBiteEffect(3,1) for SNOW) to the target via StatusRecipient/GameActor.

Pros	Cons
Localized and direct: attack executes immediate damage and applies effect in one place.	Coating-application logic is repeated in each attack action (minor duplication).
Reuses the REQ3 status-effect framework (no extra infra).	Requires defensive checks in effect application (WARMTH attribute, tundra immunity).
Easy to reason about stacking (each application is a new StatusEffect instance).	

Design B2: Central CoatingEffectFactory / mapping

Attack actions delegate to a central factory that maps CoatingType → StatusEffect instance(s) to apply.

Pros	Cons
Centralized mapping reduces duplication; easy to add new coatings.	Adds an indirection layer (small complexity overhead).
Cleaner single point to modify coating semantics.	Slightly more code to maintain for few coatings.

Decision: B1 chosen initially (direct check). If coatings grow, switch to B2.

C. Where the coat UI/action comes from (how player coats weapon)

Design C1: Coat item exposes allowableActions(owner, map) and creates CoatAction entries per coatable weapon (chosen)

The coat item (Snow, YewBerry) inspects the owner's inventory for Coatable items and returns CoatAction options. CoatAction.execute() removes the coat item from inventory and calls weapon.setCoating(...).

Pros	Cons
Reuses engine hook allowableActions; consistent UX (select coat item then weapon).	Coat item must enumerate inventory (cheap but some code duplication if many coatable types exist).
Coat consumption (YewBerry) is handled at the time of action.	
Simple testability: callable from the coat item context.	

Design C2: Weapon exposes "apply coating" action and prompts for coat item

Weapon provides an action that, when chosen, looks up applicable coat items from inventory and consumes one to apply the coating.

Pros	Cons
Action appears where the weapon resides (natural to UX).	Weapon must scan inventory and own selection UI handling, this increases coupling between weapon and coat items.
Keeps coat logic centralized with the weapon.	Slightly more complicated action flow.

Decision: C1 chosen for simplicity and minimal changes to weapon code.

Additional enforcement (REQ rule): Torches must not be coatable so ensure torches only appear if item implements Coatable (or coat item explicitly filters out torches).

D. Special semantics: stacking, replacement, consumption, frostbite & tundra immunity

Design D1: Stack-by-instance & expiration via GameActor tick (chosen)

Each coating-application produces a new StatusEffect instance appended to target's GameActor effect list. GameActor.tickStatusEffects() iterates and applies every instance each turn, so stacking is additive and expirations are independent.

Pros	Cons
Simple, predictable stacking semantics (N instances → N×per-turn effect).	Potential growth of many identical instances in pathological cases (mitigation: add caps/coalescing later).
Expiry is per-instance so naturally models the "oldest effect drops off" behaviour.	
No central stack manager needed.	

Design D2: Coalescing/capped stacks in GameActor.addStatusEffect()

addStatusEffect() coalesces new effects into aggregated state up to a cap.

Pros	Cons
Prevents unbounded stacking; can implement caps or aggregated durations.	More complex logic in GameActor; harder to reason about individual instance lifetimes.
More memory/time efficient in extreme stack cases.	

Decision: D1 chosen; consider D2 as future optimization.

D (cont.): YewBerry consumption & re-coating semantics

Design: CoatAction.execute() immediately removes the coat item from actor inventory (so YewBerry disappears). Re-coating is implemented by weapon.setCoating(...) replacing the prior enum state.

Pros	Cons
Matches requirement (YewBerry consumed once used).	If players expect multi-use coatings on a weapon, enum approach must be extended.
Replacement semantics simple and predictable.	

D (cont.): Frostbite & WARMTH / Tundra immunity

Design: FrostBiteEffect.applyEffect() will:

- Check hasStatistic(WARMTH) before reducing warmth (defensive).
- Check for SPAWNED_FROM_TUNDRA flag/ability and skip applying frostbite if set (tundra-spawned immunity).

Pros	Cons
Safe in absence of WARMTH stat; enforces requirement that tundra spawns are immune.	Requires consistent naming/usage of the tundra flag and WARMTH stat across codebase; document these invariants.
Keeps responsibility in the effect implementation.	