REQ3 - Design Rationale

Goals & constraints

REQ3 adds three weapons (Axe, Torch, Bow) with status effects (Bleed, Burn), stacking effects, and a small environmental mechanic (fire ground that burns actors). Constraints: do not modify edu.monash.fit2099.engine (engine package). Keep code in game.*. Keep solution testable, maintainable and readable.

Key design choices

1. Centralized status-effect framework (StatusEffect, GameActor, StatusRecipient, StatusRecipientRegistry) so weapons add effects rather than changing actor internals. Effects are value-like objects that are ticked by GameActor.tickStatusEffects(map). Why: isolates timed behaviour, enforces single responsibility, makes stacking straightforward.
2. Capability-based boundary (StatusAbilities.CAN_RECIEVE_STATUS) plus a registry (StatusRecipientRegistry) to avoid *instanceof* while keeping the engine unchanged. GameActor enables the capability and registers itself as a StatusRecipient. FireGround and weapon actions check the capability and look up the recipient to deliver effects. Why: removes brittle type-coupling (connascence of position) and avoids engine changes.
3. Per-weapon action classes (AxeAttackAction, TorchAttackAction, BowAttackAction) created by their respective Item.allowableActions(...) methods. Actions encapsulate attack rules (hit chance, immediate damage, status application, environment spawning). Why: single-responsibility as Item chooses valid targets; Action executes attack effect.
4. Environment tile FireGround with an underlying ground (Dirt) so fire is temporary and reverts automatically. FireGround.tick() handles actor-on-tile logic (adds burn or hurts immediately).
5. Stackable effects like BleedEffect and BurnEffect are small objects (duration + damage) that are added to an actor's effect list; GameActor.tickStatusEffects iterates the list and applies all effects, producing additive damage naturally.

How design addresses software principles

DRY

- The StatusEffect contract centralizes per-turn behaviour for timed damage; all weapons reuse it instead of duplicating per-turn logic.
- The per-item allowableActions pattern is reused (Axe/Torch/Bow), avoiding repeated scanning code elsewhere.

KISS

- Each class does one clear thing: Item exposes actions, Action executes attack, StatusEffect describes per-tick change, GameActor stores/ticks effects, FireGround handles tile logic. Randomness and simple percentages keep the rules easy to read and reason about.

SOLID

- Single Responsibility: Weapon items produce actions; actions apply immediate damage and add effects; GameActor only stores and ticks effects.
- Open/Closed: Add new weapons or effects by implementing new Action or StatusEffect classes; GameActor needs no change.
- Liskov Substitution: GameActor is an Actor and can be used wherever Actor is expected. StatusRecipient allows treating any recipient uniformly.
- Interface Segregation: StatusEffect is focused and minimal. StatusRecipient only declares addStatusEffect(...).
- Dependency Inversion: Higher-level modules (weapons/terrain) depend on abstractions (StatusRecipient) and capability checks rather than engine internals.

Connascence

- The implementation reduces connascence of type (avoids instanceof) by centralizing the contract (CAN_RECIEVE_STATUS) and registry mapping. There is still connascence of name (the CAN_RECIEVE_STATUS flag must be enabled in GameActor), but that is explicit and localized to one place. This is preferable and maintainable.

Pros / Cons

Pros

- Extensible: New effects (poison, freeze, slow, etc.) are added by implementing StatusEffect and creating the effect at the point of attack or world interaction so no changes to the effect engine required.
- Testable: Each StatusEffect is a small unit with deterministic applyEffect and lifecycle methods; GameActor.tickStatusEffects() can be unit-tested for stacking and expiry semantics.
- No engine changes: The design respects the constraint of not modifying edu.monash.fit2099.engine, avoiding grading or compatibility issues.
- Low coupling: Weapon Action classes do not need to reach into actor internals; they either call hurt(...) for immediate damage or addStatusEffect(...) via the StatusRecipient abstraction (through the registry when necessary).

Cons

- Global registry: StatusRecipientRegistry is a central lookup. This introduces a form of global state/indirection. Mitigations:
    - Use WeakHashMap so actor lifecycle does not leak memory.
    - Centralize registration/unregistration (in GameActor constructor and unconscious() cleanup).
- Small indirection / verbosity: The capability + registry lookup is a two-step approach rather than a single instanceof cast. This adds a little code at call sites but reduces brittle type coupling and enables easier future refactors.

Alternatives evaluated and their comparisons

Alternative A - Use instanceof and cast to GameActor at call sites

- *Pros:* Very simple to implement; minimal new classes.
- *Cons:* High type-coupling; fragile to refactor; violates teacher guidance to avoid instanceof as a code smell.
- *Decision:* Rejected.

Alternative B - Add status-effect support to the engine Actor API

- *Pros:* Natural, uniform, and direct; all actors would natively accept effects.
- *Cons:* Violates the assignment constraint (must not alter engine package) and would be invasive.
- *Decision:* Rejected.

Alternative C - Central StatusEffectManager / global manager

- *Pros:* Single point to tick all effects and centralized optimization opportunities.
- *Cons:* Ownership ambiguity (who owns effects?), singleton (restricting class to single instance while providing global access to it), and coupling to engine tick loop. Harder to reason about actor-local stacks and lifecycles.
- *Decision:* Rejected in favour of actor-local ownership.

Alternative D - Event Bus / Publish–Subscribe

- *Pros:* Highly decoupled; flexible extension points.
- *Cons:* Extra complexity (ordering, priorities, debugging), potential for unexpected side effects and non-determinism; overengineered for requirement scope.
- *Decision:* Rejected.

Alternative E - Decorator / WeaponDecorator for coatings

- *Pros:* Encapsulates coating behaviours; good if coatings grow in complexity.
- *Cons:* More complex, requires wrapping/unwrapping weapons, increases runtime objects.
- *Decision:* Deferred as the current enum-as-state implementation is KISS and easily refactorable to the decorator approach if coatings become behaviour-rich.

Why the chosen approach

- The chosen solution (actor-local StatusEffect + capability + registry + per-weapon Action classes + enum coatings) gives the best balance of non-invasiveness, low coupling, testability, and simplicity. It avoids instanceof and is straightforward to extend.

Maintainability foresight and examples

- Adding a new weapon: implement an Item exposing allowableActions(...) and a corresponding Action class that applies immediate damage and any StatusEffect instances. No core changes needed.
    - *Example:* add Spear and SpearAttackAction that applies ArmorBreakEffect (implement StatusEffect).
- Changing RNG (random number generator) for tests: actions currently use Random locally. For deterministic unit tests you can inject a seeded Random or a test RNG helper into actions.
- Coating evolution: current enum-based coatings are simple and practical; if coatings later become behaviour-rich (own state, multiple uses, conditional decay), you can migrate to a CoatingStrategy or Decorator implementation without touching the effect engine.
- Effect lifecycle and cleanup: using WeakHashMap for StatusRecipientRegistry reduces memory leak risk; still consider explicitly unregistering in GameActor.unconscious(...) for clarity.
- Performance scaling: tickStatusEffects() iterates only active effects. If stacking becomes pathological, options include:
    - coalescing identical effects into a single aggregated effect
    - limiting stack count per effect type
    - moving to a priority queue for very large numbers of effects

Testing & verification strategies

Unit tests

- Effect classes (BleedEffect, BurnEffect, PoisonEffect, FrostBiteEffect):
    - applyEffect applies the expected change to health/warmth.
    - decrementDuration reduces remaining turns; isExpired toggles when expected.
- GameActor.tickStatusEffects():
    - Multiple stacked effects apply additively across ticks.
    - Expired effects are removed and have no lingering effect.
- Action classes (AxeAttackAction, BowAttackAction, TorchAttackAction):
    - Test hit/miss branches deterministically with injected RNG.
    - Test that coatings result in correct effect creation and that coating items are consumed.
- CoatAction:
    - Verify setCoating() semantics and that the coating item is removed from the actor inventory.

Integration tests

- Coat + Attack:
    - Coat an Axe with YewBerry, attack a target twice; simulate ticks and verify poison stacks: 4 HP/turn per application for 5 turns.
    - Coat with Snow, attack, ensure WARMTH reduces by 1 per coating + 1 permanent coldness where applicable; stacking semantics match description.
- Torch & FireGround:
    - Torch attack spawns FireGround for 5 turns; actor stepping into the fire receives the stepping burn (5 dpt for 5 turns); ensure ground reverts to Dirt.