REQ3: Design Rationale (Weapons & Status Effects)

A. Status-effect ownership & ticking

Design A1: Actor-local StatusEffect list (chosen)

Each GameActor stores a List<StatusEffect> and exposes addStatusEffect(...) and tickStatusEffects(GameMap). GameActor.tickStatusEffects() iterates the list each turn, applies every effect and decrements durations.

Pros	Cons
Local ownership: effects belong to the actor they affect → clear lifecycle.	Slight duplication of per-actor tick invocation (each actor must call tickStatusEffects in their playTurn).
Stacking semantics are natural (multiple effect instances = additive behaviour).	In pathological cases with many effects, per-actor iteration could be heavier than centralized optimisations.
Easy to unit-test StatusEffect implementations in isolation and GameActor.tickStatusEffects().	
Minimal central coordination; scales with active effects per actor.	

Design A2: Central StatusEffectManager (rejected)

A global manager holds and ticks all effect instances each global tick.

Pros	Cons
Single place to optimise scanning, scheduling and coalescing.	Ownership ambiguity (who is responsible for removing effects on actor death).
Easier to run global queries (who has active poison?).	Potentially more coupling to engine tick loop; harder to reason about actor-local stacks.
	Harder to unit-test actor-local stacking semantics.

Decision: A1 (Actor-local) selected for clarity, testability and minimal coupling.

B. Boundary for delivering effects to actors (who can receive effects)

Design B1: instanceof GameActor checks at call sites (simple)

Caller checks if (target instanceof GameActor) and casts to add effects.

Pros	Cons
Very straightforward to implement.	High type coupling; brittle during refactor.
Minimal extra infrastructure.	instanceof flagged as a code smell.
	Violates open/closed if new recipient types appear.

Design B2: Capability flag + StatusRecipientRegistry (chosen)

Actors that accept status effects enable a capability (StatusAbilities.CAN_RECIEVE_STATUS) and register a StatusRecipient with StatusRecipientRegistry. Code checks the capability and uses the registry to add effects.

Pros	Cons
Avoids brittle casts; decouples add-effect call sites from concrete class hierarchy.	Introduces a small indirection (capability + registry lookup).
Works across engine boundary without modifying engine code.	Adds a global registry (but mitigated by WeakHashMap and simple usage).
Registry uses WeakHashMap to avoid memory leaks.	

Decision: B2 chosen to comply with the “no instanceof” guidance and to keep engine untouched.

C. How to represent weapon attacks

Design C1: Per-weapon Action classes (AxeAttackAction, TorchAttackAction, BowAttackAction) (chosen)

Item.allowableActions produces specific Action objects for each valid target; those Actions encode hit chance, immediate damage and effect-creation rules.

Pros	Cons
Single Responsibility: item discovers targets, action executes attack logic.	More classes to write, one per weapon type (but each is small).
Easy to put weapon-specific numbers and special behaviours (torch spawns fire) near the attack implementation.	
Easy to unit-test hit/effect branches.	

Design C2: Shared general AttackAction + parameterized weapon descriptors (rejected)

One AttackAction consumes a WeaponDescriptor object for damage/hit/effect rules.

Pros	Cons
Fewer Action classes; centralised attack flow.	Descriptor pattern requires mapping descriptors → effects (less explicit).
	Harder to localise special behaviours like torch spawning fire (would need callbacks).
	Slightly more indirection and complexity for special-case behaviours.

Decision: C1 chosen for clarity and to keep special-case logic localised.

D. Environmental fire representation

Design D1: Fire as transient Ground (FireGround) with underlying Dirt (chosen)

Attacking with Torch spawns FireGround tiles around the attacker. FireGround.tick() either adds BurnEffect to recipient (via registry) or hurts directly, and reverts to Dirt when expired.

Pros	Cons
Natural map-level representation (fire behaves like terrain).	Requires mapping changes on spawn; need to ensure map state consistency.
Ground tick() neatly models per-tile lifetime and interaction with actors.	FireGround must look up recipients via registry (extra indirection).
Easy to visually represent and to reason about actor traversal.	

Design D2: Attach temporary effect objects to Location but not change Ground (rejected)

Place ephemeral objects overlayed on Location, separate from Ground.

Pros	Cons
	More bookkeeping (overlay lifetime distinct from ground lifetime).
	Harder to visualise and to revert map state to original ground.

Decision: D1 chosen for simplicity and clear UX mapping.

E. Effect representation and stacking

Design E1: Effects as objects (BleedEffect, BurnEffect, PoisonEffect) stored as instances (chosen)

Each effect is a small object: (remainingTurns, perTickValue) implementing StatusEffect. Adding multiple instances naturally stacks additively.

Pros	Cons
Very simple stacking model: N instances = N×effect per tick.	Lots of small objects if stacking grows high (profiling may be needed in extreme cases).
Encapsulates expiry and per-tick logic.	
Easy to add new effect types.	

Design E2: Aggregate effect entries with counters (rejected initially)

Store a single effect per type with stackCount and aggregated duration/values.

Pros	Cons
Potential memory/iteration savings with high stacks.	More complex aggregation logic; harder to model multiple different durations per instance.

Decision: E1 chosen (object-per-instance). If performance becomes an issue, consider coalescing identical effects.

F. Where to expose attackability (who publishes AttackAction)

Design F1: Attacker decides attacks only (predator-centric)

Predators compute AttackAction during their playTurn; targets do not expose allowable attack actions.

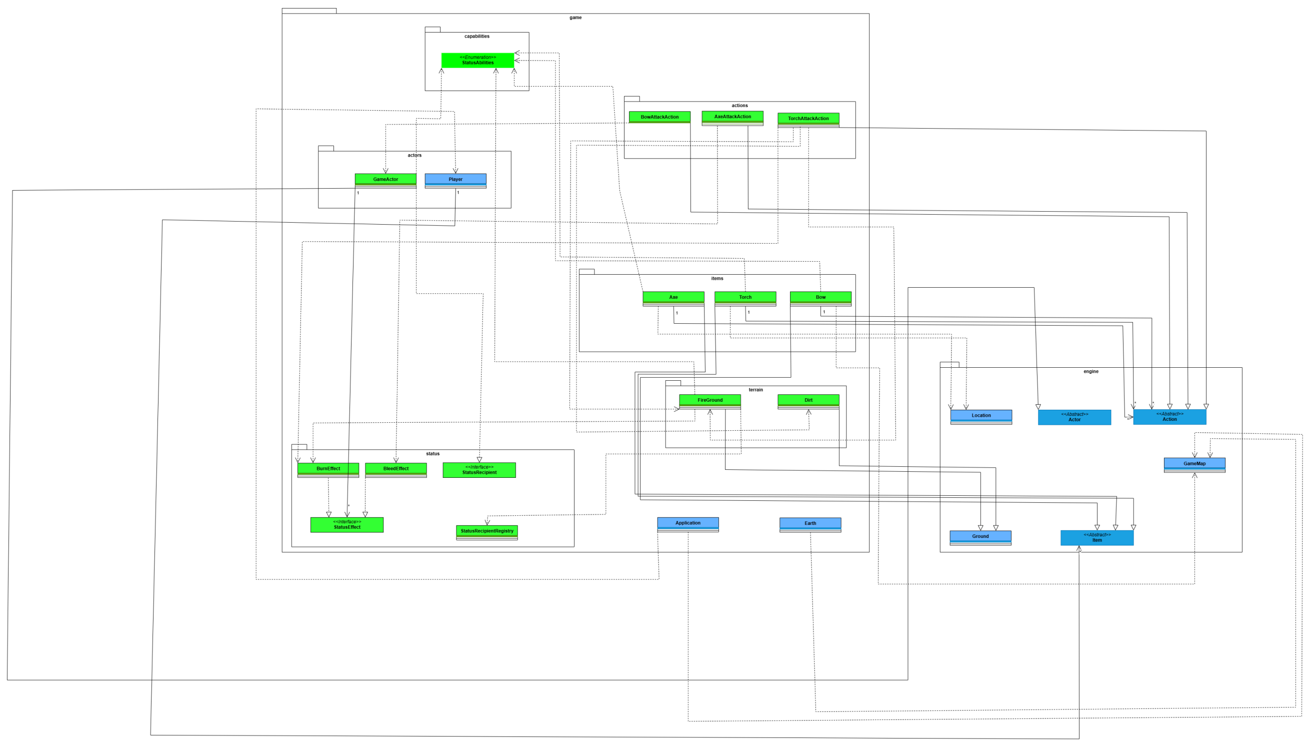
Pros	Cons
Simple for predators; minimal target API.	Prevents Player from easily attacking otherwise passive actors unless attacker knows every victim type.

Design F2: Targets advertise allowableActions(Actor other, String dir, GameMap) (chosen)

Targets publish actions others can perform on them (e.g., Deer.allowableActions() returns AttackAction(this, dir) so explorers can attack deer consistently).

Pros	Cons
Low coupling: attackers don't need to know target internals.	Slightly more boilerplate per target.
Consistent with engine's existing design affordance.	
Reusable: any attacker (player, wolf, bear) can use the same target-provided action.	

Decision: F2 chosen to follow engine conventions and reduce coupling.



The diagram above shows the final structure for REQ3: weapon items (Axe, Torch, Bow), per-target attack actions, a lightweight status-effect framework, and an environment tile that models temporary fire. At the top level the game-specific GameActor class extends the engine Actor and realizes StatusRecipient; GameActor owns a list of StatusEffect instances and exposes addStatusEffect(...) and tickStatusEffects(GameMap) so effects are applied and expired each turn. StatusEffect is an interface implemented by concrete classes BleedEffect and BurnEffect (both shown as realizations). StatusRecipientRegistry is a tiny lookup utility (weak map) that maps engine Actor → StatusRecipient so non-actor code (for example, FireGround) can add effects without instanceof casts hence this is shown as a dependency from FireGround to StatusRecipientRegistry.

Weapon items are modelled as game Item subclasses: Axe, Torch, and Bow. Each weapon creates its corresponding attack Action instances via allowableActions(...): Axe → AxeAttackAction, Torch → TorchAttackAction, Bow → BowAttackAction. Those Action classes extend the engine Action class (realizations) and depend on GameActor and StatusEffect implementations: on a hit they call target.hurt(...) for immediate damage and/or target.addStatusEffect(...) to attach BleedEffect / BurnEffect. The TorchAttackAction additionally spawns temporary FireGround tiles (dependency), and FireGround.tick(Location) either adds a BurnEffect to StatusRecipients or calls hurt(...) directly for actors that cannot receive status effects; when its lifetime ends it reverts the tile back to Dirt. The diagram shows these as usage/dependency arrows rather than inheritance or composition, which keeps the design aligned with the engine API and avoids changing engine classes.

Actor and item relationships are deliberately low-coupled: GameActor manages status effects (association), weapon items generate actions (dependency) rather than forcing an action into engine internals, and StatusEffect implementations are tiny value-like objects whose stacking semantics are expressed by simply adding multiple instances to the GameActor’s list. This architecture enforces the Single Responsibility Principle (actions do attack execution; effects encapsulate per-tick logic; ground handles environment state), is open for extension (add new StatusEffect classes or new weapon Actions without modifying GameActor), and minimizes type connascence by using the StatusRecipientRegistry and StatusAbilities.CAN_RECIEVE_STATUS capability rather than instanceof checks.