REQ1: Design Rationale (Survival Mechanics: Hydration, Warmth, Sleep & Drink)

A. Representing player survival attributes

Design A1: Primitive fields on Player. Store int hydration, warmth on Player; decrement in playTurn.

Pros	Cons
Very simple to implement and read.	Violates DRY as soon as we add more attributes (duplicated increment/decrement/check code).
No extra classes.	Spreads attribute rules into Player.playTurn(), increasing class responsibilities (low cohesion).
	Harder to reuse the same logic on non-player actors later (closed to extension).

Design A2: Encapsulated attribute object(s). Introduce BaseActorAttribute with get(), increase(int), decrease(int). Player composes two instances: hydration, warmth.

Pros	Cons
High cohesion: all attribute logic lives in a single, reusable abstraction.	Adds a small layer of indirection.
Open for new attributes without changing Player logic.	
Works for any Actor later (LSP-friendly: using the same abstraction everywhere).	

B. Applying hydration changes (drinking)

Design B1: Action mutates attributes directly. DrinkAction.execute() downcasts to Player and updates fields.

Pros	Cons
Minimal classes.	Tight coupling to Player (downcasts).
Easy to follow inside the action.	Hinders reuse of the action for non-player actors
	Sprinkles attribute logic across many actions (low cohesion).

Design B2: Utility plus capability boundary. Provide HydrationUtils.increaseHydration(Actor, amount) and a marker/port HydrationCapability with a method increaseHydration(amount). Actions call the utility; the utility delegates via the capability (or via Player's attribute object).

Pros	Cons
Actions depend on an abstraction (capability/util) instead of concrete Player internals	Requires a small coordination layer (util + capability check).
Keeps attribute rules centralized	
Reusable for any Actor that advertises hydration capability	

C. Modeling sleep state & turn skipping

Design C1: Boolean flags only. SleepAction toggles isSleeping on SleepAction/Player; Player.playTurn() branches on flags to skip turns and suppress decay.

Pros	Cons
Quick to code.	State logic sprawls across SleepAction and Player (tight coupling).
	Easy to forget to pause all relevant decays; hard to extend, makes it fragile.

Design C2: Ability-gated behavior with a single owner of sleep timing. Introduce Abilities enum with SLEEPING. SleepAction owns timing (turnsRemaining, totalSleepDuration) and exposes getNextAction() to continue sleeping. While sleeping, we enable SLEEPING on the actor; Player.playTurn() checks abilities and pauses attribute decay if SLEEPING is present. Sleep requires a Bedroll on the ground.

Pros	Cons
Separation of concerns: SleepAction owns sleep lifecycle; Player just consults abilities (SRP, high cohesion).	Requires the abilities mechanism
Abilities are a clean, extensible gate for cross-cutting rules (OCP).	
Reusable: other actors could sleep using the same mechanism.	

D. Where to track bottle uses

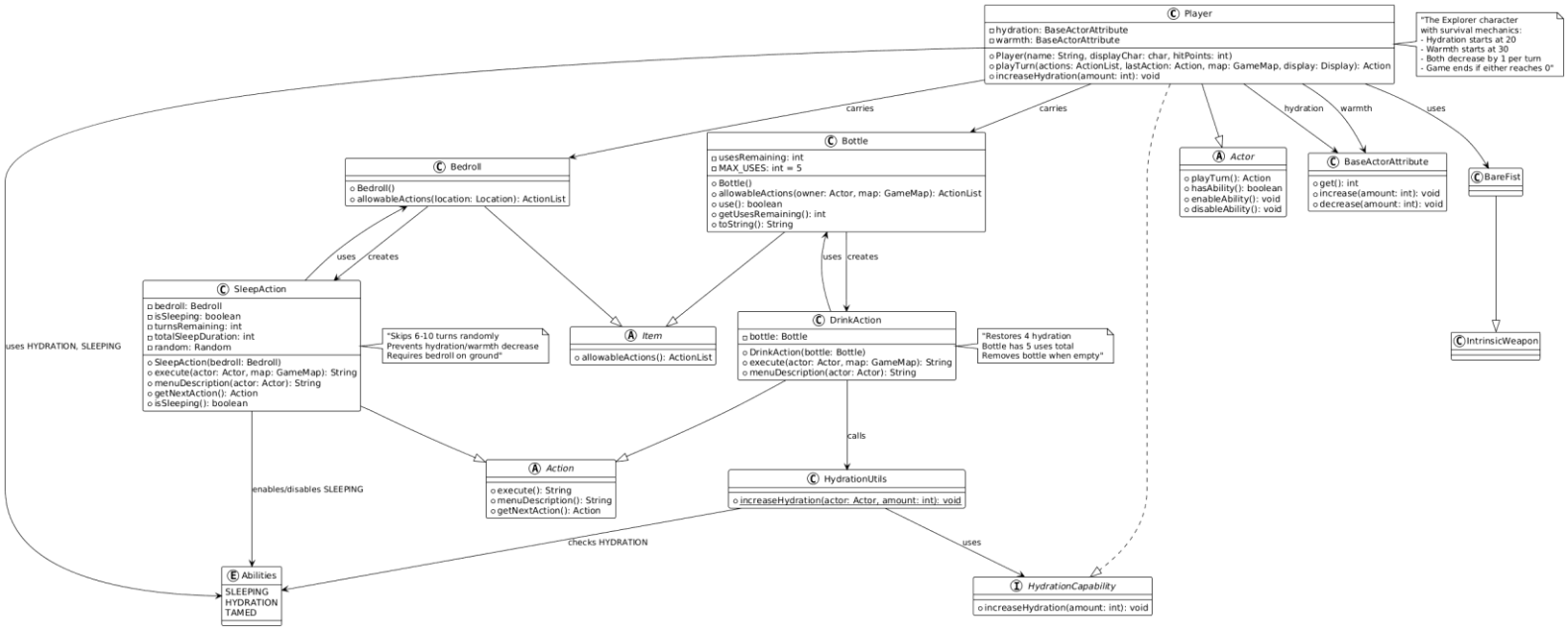
Design D1: DrinkAction tracks uses. The action stores and decrements remaining-use state.

Pros	Cons
Keeps Bottle “dumb”.	Violates object ownership: the item should know its state

	Duplicates logic if multiple actions can consume the same item.

Design D2: Bottle owns use count & life cycle. Bottle stores usesRemaining (max 5). DrinkAction asks the Bottle to use(), and removes the bottle from inventory when empty; toString() reflects remaining uses.

Pros	Cons
High cohesion: consumption rules live with the item	Slightly more code in the item.
Easy to reuse Bottle in different contexts/actions	
Clear invariants (cannot go below zero, etc.) centralized.	



The diagram above shows the final design of requirement 1, which utilizes each Design 2 of parts A, B, C and stated above to implement survival attributes (Hydration, Warmth), drinking from a Bottle, sleeping with a Bedroll, and the bottle-lifecycle management.

The Player composes two BaseActorAttribute objects (hydration, warmth), while DrinkAction and SleepAction extend the abstract Action class in the engine. Bottle and Bedroll are items that integrate via allowableActions, and HydrationCapability (interface) together with HydrationUtils

provides a clean boundary for adjusting hydration. Since attributes share common rules (increase/decrease, clamping, death at zero) and consumable items share lifecycle behavior (uses remaining, removal on empty), it is logical to abstract these identities to avoid repeated code (DRY) and keep logic cohesive in one place (Single Responsibility Principle).

Using an ability flag (Abilities.SLEEPING) to represent the sleep state, and a capability boundary (HydrationCapability) to represent "can be hydrated," separates orthogonal responsibilities into small, well-named units. This results in better maintainability (Single Responsibility Principle) and safer extensions: sleep timing is owned entirely by SleepAction, while hydration changes are mediated through a capability rather than direct field access. By keeping traits and abilities distinct (e.g., an actor may have hydration without ever sleeping), we avoid bloated interfaces and keep each contract focused on its purpose (Interface Segregation Principle).

HydrationUtils depends on the HydrationCapability interface rather than on concrete Player internals. This removes multiple direct dependencies from actions to specific classes and prevents scattering attribute logic across the codebase. When introducing more consumables (e.g., Canteen, HotSoup) or enabling NPCs to also hydrate, we do not need to change existing actions, the new classes simply expose the same capability (Open-Closed Principle). Likewise, actions and high-level flow depend on the abstraction (HydrationCapability) while low-level details (how hydration is stored/enforced) live behind it (Dependency Inversion Principle).

REQ2: Design Rationale (Animals & Combat)

A. Modeling attack behaviour

Design A1: Per-species attack actions (e.g., WolfAttackAction, BearAttackAction)
Each predator ships its own action computing hit chance/damage.

Pros	Cons
Each class contains its own constants; "close to where it's used."	DRY violation: duplicated hit/miss resolution, damage application, death handling.
Straightforward to follow in isolation.	Increases coupling between species and combat rules.
	Harder to add a new species.

Design A2: Shared AttackAction + species-specific IntrinsicWeapon. All combat is handled by one AttackAction (already in the engine/demo). Species provide their intrinsic weapon via dedicated classes (WolfBite, BearClaw, both extend IntrinsicWeapon) that encode damage, verb and chance.

Pros	Cons
High cohesion: combat flow is centralized in one action.	Requires small indirection to map each species to its weapon
Open for extension: add a species by adding an IntrinsicWeapon	
Clear ownership: weapon encapsulates numbers; action encapsulates process.	
Enables non-actors (e.g., the Player with a weapon item) to use the same action	

B. Where to expose "attackability"

Design B1: Attacker-only initiation. Predators decide to attack when adjacent. Victims provide nothing.

Pros	Cons
Minimal API surface.	Prevents the Player attacking passives unless attacker knows every victim type.
	Tighter coupling between attackers and potential targets.

Design B2: Engine affordance: targets advertise allowable actions. Each Actor overrides allowableActions(Actor other, String dir, GameMap) to publish actions others may perform. Our Deer returns a new AttackAction(this, dir) so the Player can attack it. Predators still create AttackAction during their own turns when they find an adjacent target.

Pros	Cons
Low coupling: attackers don't need type checks	Slightly more boilerplate per target class.
Consistent with engine design; reusable for any attacker (Player, Wolf, Bear).	

C. Species AI (movement + aggression)

Design C1: Big “if-else” monolith per species. One playTurn crams detection, targeting, movement, and fallback logic.

Pros	Cons
Compact file count.	Low cohesion; harder to test.
	Hard to tune priorities (attack vs wander) without side-effects.

Design C2: Lightweight, explicit priorities inside playTurn

Wolf/Bear: if any adjacent hostile target, return AttackAction; otherwise, choose one exit at random and try MoveActorAction; else DoNothingAction.

Deer: always wander randomly if destination is free; else do nothing.
This matches the uploaded Deer implementation and keeps aggression localized to predators.

Pros	Cons
Simple, readable, cohesive rules	Not as configurable as a full behaviour tree;
Easy to slot in Behaviours later if required	

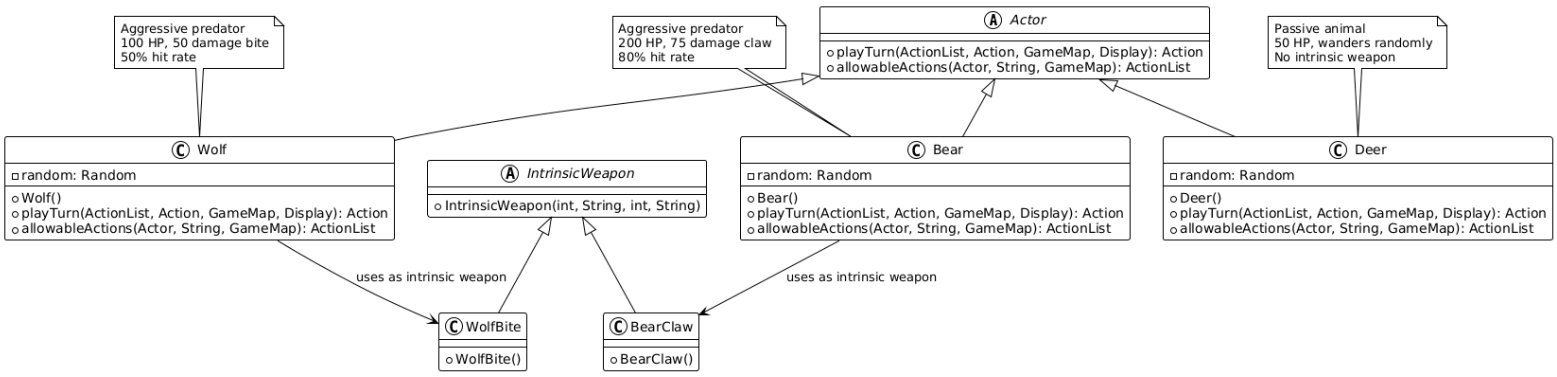
D. Representing weapons

Design D1: Put damage/chance as fields on Actor. AttackAction pulls numbers from the attacker’s fields.

Pros	Cons
Fewer classes.	Conflates state (HP, name) with capability (weapon stats), lowering cohesion.
	Hard to swap weapons or support multi-weapons.

Design D2: Subclass IntrinsicWeapon per species (WolfBite, BearClaw). Each weapon encodes damage, verb, and hit chance; Actor.getIntrinsicWeapon() returns an instance. Deer returns null/default.

Pros	Cons
SRP: weapon parameters live with the weapon.	A couple of small classes to hold constants.
Plays nicely with AttackAction which already expects a Weapon.	
Natural extension point (e.g. FrostBite, IceClaw)	



The diagram above shows the final design of requirement 2, which utilizes each Design 2 of parts A, B, C and D stated above to implement predator animals (Wolf and Bear) with intrinsic weapons, a passive Deer, and a unified combat flow using AttackAction.

The new concrete intrinsic-weapon classes WolfBite and BearClaw extend the abstract IntrinsicWeapon class, while the animal classes extend the abstract Actor class. Since we want predators to attack via the engine's generic AttackAction and the Deer to act as an interactable target that exposes actions to others, it is logical to abstract weapon identity into dedicated classes and expose affordances through allowableActions to avoid repetitions in code (DRY).

Using the engine's affordance pattern (allowableActions) to define what can be done to a target, and species-specific playTurn overrides to define when a predator chooses to attack, separates different responsibilities into different places, resulting in better maintainability of the system (Single Responsibility Principle). This approach also allows more extensions to fauna, by adding new intrinsic-weapon classes or new actors without modifying the combat action, since the current abstractions are already separated by their own purposes (Interface Segregation Principle).

AttackAction depends only on the Weapon/Actor abstractions and not on concrete species. This implementation allows us to remove multiple dependencies from combat flow to specific animal classes. When introducing more new animals or weapons, we do not need to modify existing actions to use the new types, as concrete species simply provide an intrinsic weapon and/or expose the relevant affordances (Open–Closed Principle).

REQ3: Design Rationale (Trees, Consumables, Periodic Drops)

A. Modeling plant types

Design A1: Each tree is a standalone Ground with its own timer & drop logic. WildAppleTree, HazelnutTree, YewBerryTree each override tick and implement counters and spawning.

Pros	Cons
Straightforward; minimal extra classes.	DRY violation: each class repeats timer/counter, adjacency scan, placement checks.
Easy to read each tree in isolation.	Harder to add a fourth tree: copy–paste risk; mixed responsibilities (appearance + production).

Design A2: Abstract Tree with a template hook. Create a base class that extends Ground and centralizes: dropInterval, turnCounter, adjacency selection, and spawnItem(Location) (abstract). Concrete trees only implement the item to spawn and the interval.

Pros	Cons
High cohesion: shared scheduling & placement live in one place.	Adds one base class layer
Open for extension: new tree = small subclass setting interval + item factory	
DRY: no repeated timers or scanning code.	

B. Consuming items

Design B1: Per-item action classes (e.g., EatAppleAction, EatHazelnutAction). Each consumable ships a custom action with its own effect.

Pros	Cons
Effects are “near the item”.	DRY violation: duplicated menu text, inventory removal, death/HP updates.
	Tight coupling between items and actions; harder to add new consumables.

Design B2: Shared ConsumeAction + abstract ConsumableItem. One ConsumeAction handles selection, messages, removal, and success/failure flow. Items subclass ConsumableItem and implement applyEffect(Actor) (e.g., Apple: heal +3 and call HydrationUtils.increaseHydration(actor, -2); Hazelnut: increaseMaxHp(+1); YewBerry: set HP to 0 / trigger death).

Pros	Cons

SRP/DRY: all consumption flow in one action; items declare only their effects.	Requires a small abstract base.
Works with inventory or ground usage uniformly; consistent UX.	
Easy to extend with new food types without changing the action (OCP).	

C. Scheduling & placement of drops

Design C1: Global "flora manager" scans the map each turn. A manager iterates over all plants, updates timers, and spawns items.

Pros	Cons
One place to toggle production for events.	Adds global coupling; every plant must register/unregister; breaks locality of behaviour.
	Harder to reuse outside the main map; violates information hiding.

Design C2: Per-tree tick with standardized placement. Each Tree (via the base class) increments a counter on tick; when counter % dropInterval == 0, it selects a random free adjacent tile (Location.getExits()), and places the item if available; otherwise, it silently skips that turn.

Pros	Cons
Behaviour lives with the plant (high cohesion).	Slight duplication of random selection across different Ground types if others adopt it (mitigated by base class).
No global registry or cross-component coupling.	
Naturally supports "surroundings" placement and graceful failure when tiles are full.	

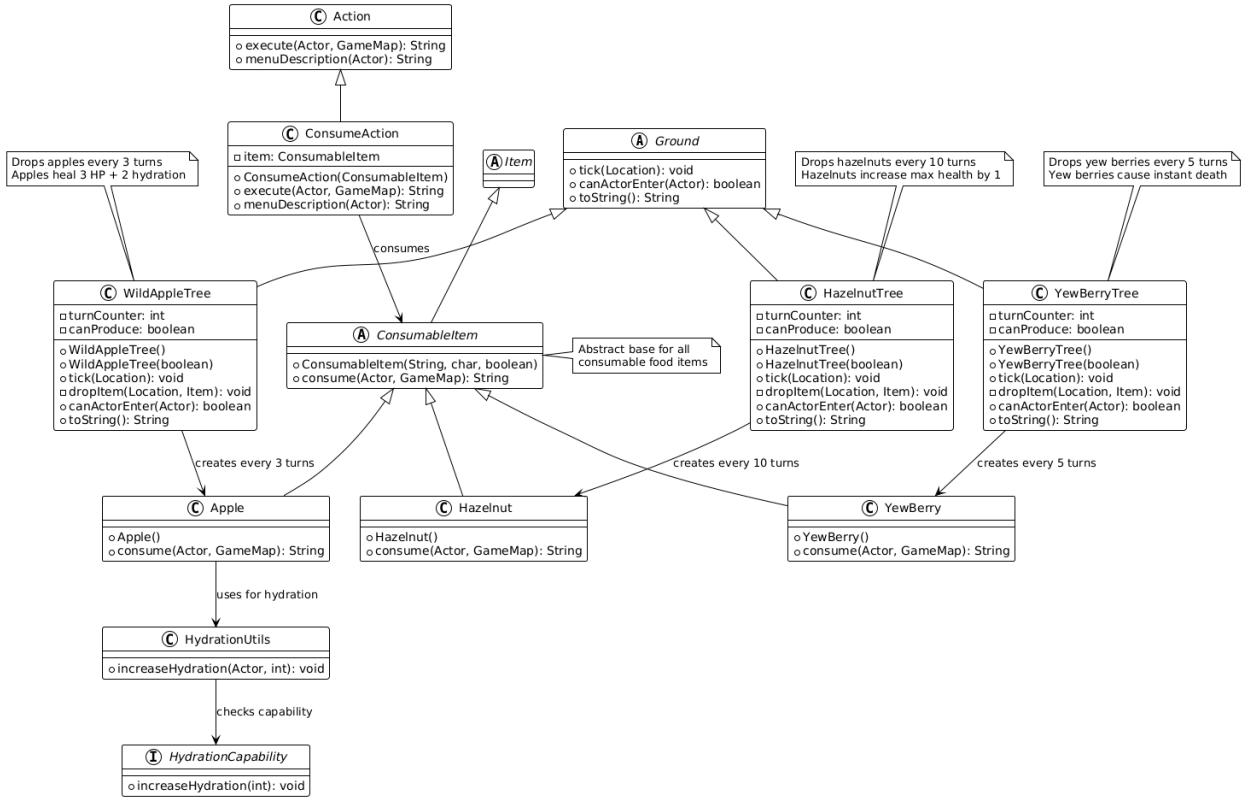
D. Handling plants that produce nothing

Design D1: Base class always produces; non-producers override with "no-op" item
Force every plant into the production pipeline and simulate "no production" by returning null or a dummy item.

Pros	Cons
Simplifies the base flow.	Leaks fake states (null/dummy) into the pipeline
	Forces non-producers to inherit irrelevant behaviour (low cohesion).

Design D2: Capability interface gates production (e.g., ProducesConsumables). Only plants that implement ProducesConsumables (or set dropInterval > 0) participate in spawning. Others remain plain Ground.

Pros	Cons
Interface Segregation: non-producers don't inherit unused behaviour.	One more small interface.
Clear capability boundary: systems can query "can this plant produce?" without type checks	
OCP: add new producer/non-producer types without disturbing existing code.	



The diagram above shows the final design of requirement 3, which utilizes each Design 2 of parts A, B, C and D stated above to implement flora that periodically spawn consumables around them and a unified consumption pathway.

The new abstract Tree (or AbstractFlora) extends the engine's Ground, while Apple, Hazelnut, and YewBerry extend the abstract ConsumableItem. Since we want plants to behave like terrain (actors can stand on them) and fruits to behave like items with consistent "consume" behaviour, it is logical to abstract these identities to avoid repeated code (DRY).

Using a capability interface (ProducesConsumables) to mark producer plants and a shared ConsumeAction to apply effects separates responsibilities cleanly (Single Responsibility Principle). This also enables easy extension, new plants can opt-in to production by implementing the capability, and new foods only implement an applyEffect hook while the action flow remains unchanged (Interface Segregation and Open–Closed Principle).

ConsumeAction depends only on the ConsumableItem abstraction; HydrationUtils is invoked from item effects (e.g., Apple reducing thirst), not hard-wired into the action. This removes multiple dependencies between actions and specific food types. When we introduce more consumables or plants, including non-producers, we do not need to modify existing actions or tree classes: producers follow the interval/placement template, and items encapsulate their effects (Open–Closed Principle).

REQ4: Design Rationale (Taming the Wildlife: Interfaces, Behaviours, Unified Actions)

A. Modeling taming & ownership

Design A1: Per-species booleans/state. Each animal class (Wolf, Bear, Deer) declares fields like isTamed, owner, and inlines accept-bait logic; playTurn branches on these flags.

Pros	Cons
Minimal extra types; quick to write.	DRY violation: ownership and taming rules duplicated across animals.
	Low cohesion: movement, combat, taming all mixed in playTurn.
	Hard to extend to new animals or new taming items (closed to extension).

Design A2: Tameable capability + TameableAnimal base. Introduce a Tameable interface (query/commands like isTamed(), getOwner(), tame(Actor owner, Item bait)), and an abstract TameableAnimal that extends Actor and owns taming state (owner reference, success rules, post-tame hooks). A TameAction handles the interaction: if the bait is valid for the target, it consumes the item, sets the owner, enables a TAMED ability/capability, and triggers hook methods to register follow/assist/collect behaviours.

Pros	Cons
SRP: taming lifecycle is localized; species classes stay focused on species stats.	Adds a small abstraction layer.

OCP/LSP: any future animal can become tameable by extending TameableAnimal or implementing Tameable without touching existing actions.	
TameAction depends on the Tameable abstraction, not concrete species.	

B. Following the owner

Design B1: Inline following logic inside species playTurn. Each animal's playTurn calculates a step toward the owner and otherwise wanders.

Pros	Cons
Simple to read in one file.	The same follow code in multiple animals.
	Hard to evolve (e.g., keep distance, wait when adjacent) without editing every class.
	Couples movement policy to species.

Design B2: Follower interface + reusable follow routine. Define a Follower interface (e.g., getOwner(), nextStepTowardOwner(GameMap)), and place shared follow logic in a helper used by TameableAnimal. In playTurn, a tamed animal first attempts to move toward any adjacent square that reduces Manhattan distance to the owner; if already adjacent, it holds position (so it can hand off items or fight); otherwise it falls back to species wandering.

Pros	Cons
SRP/DRY: one place for follow policy; species remain small.	Slight indirection to call the helper.
Clear priority ordering: give items → collect nearby → follow → wander.	

C. Fighting alongside the player

Design C1: Modify AttackAction to "know" allies. Teach AttackAction about ownership/teams and auto-target hostiles near a tamed animal.

Pros	Cons
Central place to inject ally logic.	Tight coupling: rewires engine combat for a single feature
	Violates OCP for the engine; risk of regressions elsewhere.

Design C2: CombatAssistant behaviour that yields standard AttackActions. Keep AttackAction unchanged. Provide a CombatAssistant helper/behaviour used by tamed animals: on their turn, if a hostile is adjacent to the owner or to the pet, return a standard AttackAction(target, dir). Hostility is evaluated via existing team/capability checks (no instanceof). If no hostile is adjacent, defer to follow/wander.

Pros	Cons
Low coupling to engine code; uses the same combat path as everyone else.	Another small component to wire into the turn policy.
SRP: assistant decides when to fight; AttackAction decides how.	
OCP/LSP: works for any tamed species with a weapon.	

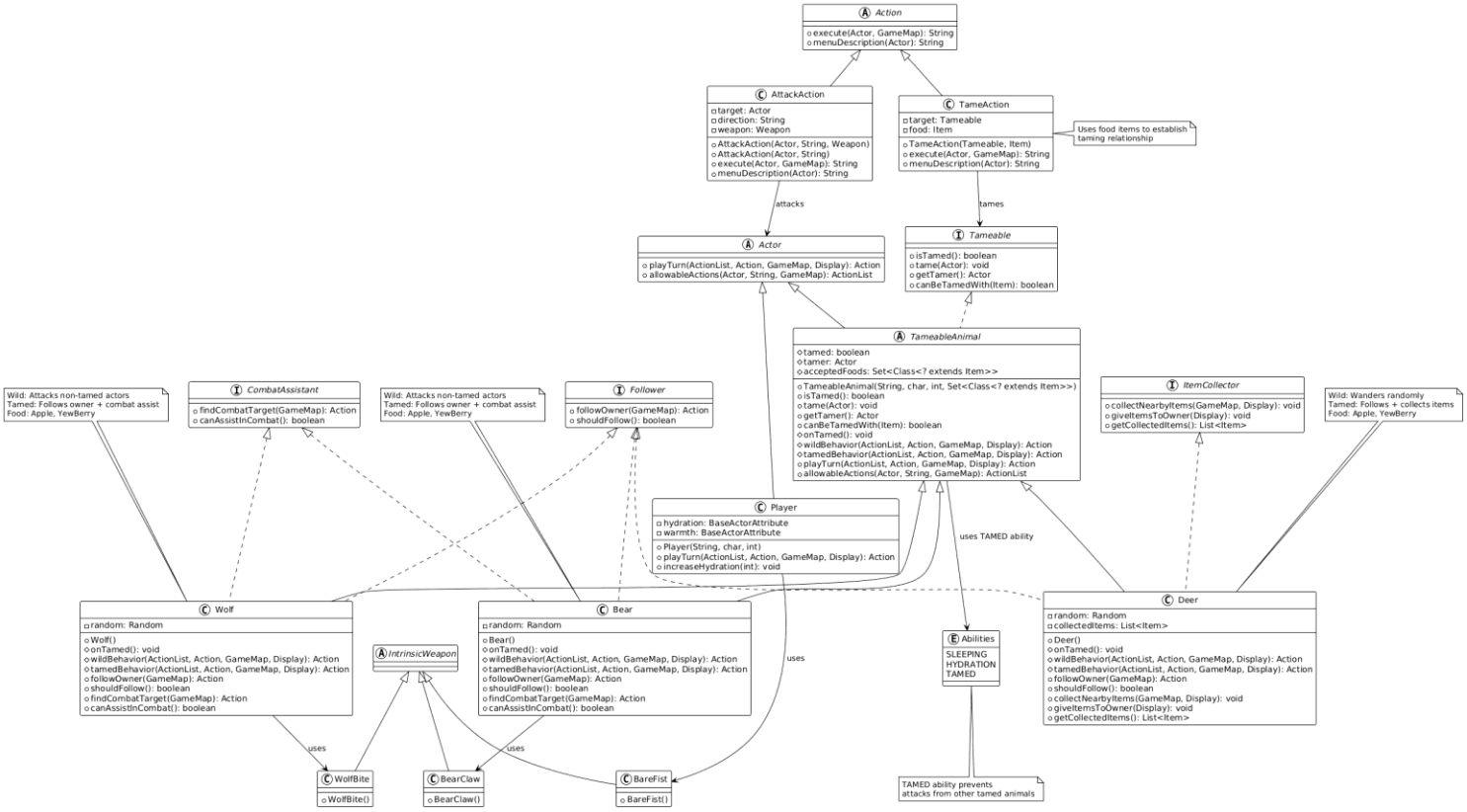
D. Collecting and handing over items

Design D1: Hard-code in Deer only. Implement ad-hoc scanning/pickup/transfer in Deer when tamed.

Pros	Cons
Quick to satisfy the example.	DRY violation; can't reuse for other tameables.
	Couples item flow to one species; breaks OCP for future extensions.

Design D2: ItemCollector interface + shared pickup/hand-off. Provide an ItemCollector interface with collectNearbyItems(GameMap, Display), giveItemsToOwner(Display), and getCollectedItems(). A default implementation in TameableAnimal (or a mixin helper) scans current and adjacent tiles for Items (typically fruits), picks them up, and when adjacent to owner, transfers them to the owner's inventory. Species opt-in (e.g., Deer implements ItemCollector; predators can too if desired).

Pros	Cons
SRP/DRY: one generic collection pipeline; species merely opt in.	Requires a tiny buffer to hold collected items.
OCP/ISP: "collection" is an independent capability from "following" or "combat."	
Natural priority policy: give → collect → follow → wander.	



The diagram above shows the final design of requirement 4, which utilizes each Design 2 of parts A, B, C and D stated above to implement taming, following, ally combat, and item collection/hand-off.

The Tameable interface and the abstract TameableAnimal base class encapsulate taming state and ownership, while TameAction provides a generic affordance that any tameable animal can expose via allowableActions. Since we want animals to remain regular Actors (so they still use standard movement and AttackAction) but gain new capabilities when tamed, it is logical to abstract these identities to avoid repeated code (DRY) and to keep responsibilities cohesive (Single Responsibility Principle).

Using separate interfaces to model distinct abilitie, Follower for movement toward the owner, ItemCollector for pickup/handoff, and the CombatAssistant behaviour to decide when to attack hostiles, separates concerns and keeps contracts small and focused (Interface Segregation Principle). This approach also supports easy extensions (e.g., a future Boar that only follows and fights, or a Fox that collects but never fights) by composing the relevant capabilities (Open–Closed Principle).

Crucially, ally combat is implemented without modifying AttackAction: pets simply produce the same AttackAction the engine already understands. High-level taming logic depends on abstractions (Tameable, Follower, ItemCollector) rather than concrete species, while low-level details (wolf bite vs bear claw, specific baits) live behind species or weapon classes (Dependency

Inversion Principle). When introducing more tameable animals or more post-tame behaviours, we do not need to change the core actions; we extend by adding new capability implementations or helper behaviours, keeping coupling low and the design robust.