# REQ1: Design Rationale (Flora II: Attack of the Plants)

## A. Representing plant growth stages

### *Design A1: Distinct Ground subclasses for each stage*

AppleSprout, AppleSapling, WildAppleTree (and similarly YewBerrySapling, YewBerryTree) are separate Ground subclasses. Each subclass stores minimal local state (turn counters, canProduce flag) and implements tick(Location) to advance its own lifecycle and perform production (drop fruit).

| Pros | Cons |
|---|---|
| High cohesion: lifecycle behaviour lives with the object that has that lifecycle. | Slight class proliferation (one class per stage). |
| Simple mapping to engine: Ground.tick(Location) fits naturally with engine tick loop. | Behaviour common to stages (turn counting, drop logic) may be duplicated if not factored to a helper. |
| Easy to reason about and unit-test single-stage behaviour. | |

### *Design A2: Single parametric Ground with internal stage enum/state*

One WildApple ground class holds an enum/state {SPROUT, SAPLING, TREE} and switches behaviour based on that state. Tick logic advances state and applies production rules based on current state and map context.

| Pros | Cons |
|---|---|
| Fewer classes; shared logic centralised. | Slightly larger, more complex class with branching (if (state==X)). |
| Easier to change stage transitions in one place. | Harder to mock/extend per-stage behaviour without additional conditionals. |

**Decision (chosen approach)**

**We chose A1 (Distinct Ground subclasses) with a small shared helper utility for common ops (drop-in-adjacent-tile). This matches the engine model (grounds are different types), keeps each class simple and readable, and fits the assignment requirement to keep logic inside game packages and use Ground.tick. Where behaviour was repeated we factored helper methods (e.g., dropItem(Location, Item)).**

## B. Where to implement fruit production & dropping

### *Design B1: Ground.tick handles production & dropping*

Each tree/sapling's tick(Location) decides when to produce and uses Location/Exit to place items in adjacent tiles.

| Pros | Cons |
|---|---|
| Straightforward and local: production logic together with lifecycle rules. | Must handle edge cases (adjacent tiles full / occupied) carefully to avoid exceptions. |
| No engine changes; uses Location API. | |

### *Design B2: Central PlantManager that schedules production*

A single manager outside ground instances receives events/ticks and manages when each plant should produce.

| Pros | Cons |
|---|---|
| Centralized scheduling, easier to change production policies across plant types. | Extra global state and coupling; harder to map to engine Ground.tick lifecycle; more invasive and unnecessary for this scope. |

**Decision**

**We used B1: production in Ground.tick. To avoid pitfalls, dropItem(...) tries adjacent tiles in predictable order and does nothing if no free location is available (no exceptions). This is simpler and aligns well with the engine's ground tick lifecycle.**

## C. Map-specific growth & production rules

### *Design C1: Encode map semantics in the ground classes (if/else by map)*

Grounds query location.map().toString() or a map flag at runtime to decide behaviour (e.g., Plains sprout skips sapling).

| Pros | Cons |
|---|---|
| Minimal additional structures. | Name-based coupling to map names; brittle if maps renamed. |

### *Design C2: Factory / Policy-per-map*

Use PlantFactory.createAppleSproutForMap(isPlains) or a PlantPolicyRegistry that creates the correct ground object or injects a policy object (growth durations, production frequency) when the ground is constructed. Tests and map-specific semantics are controlled at creation time.

| Pros | Cons |
|---|---|
| Clean separation: creation-time policy determines behaviour, no runtime string-matching. | Slight additional indirection in the factory/policy code. |
| Easy to add a new map type by registering a policy. | |

**Decision**

**We chose C2: a PlantFactory that encapsulates map-specific semantics and returns the appropriate Ground subclass (or subclass configured for that map). This keeps runtime decisions simple and makes tests deterministic i.e. the factory is the single place to adjust growth timings for maps.**

## D. Probabilistic growth (Yew Berry) & testability

### *Design D1: Use Random internally inside ground class*

Yew sapling calls random.nextBoolean() inside tick() to decide growth (50% every 3 turns).
Pros / Cons

| Pros | Cons |
|---|---|
| Simple to implement. | Hard to test deterministically unless you expose or seed the RNG. |

*Design D2: Extract RNG to a testable helper (SpawnHelper)*

Introduce Snow.SpawnHelper (or a small static helper) that holds a Random instance and offers setRandom(Random r) for tests. Ground classes consult SpawnHelper for probabilistic decisions.
Pros / Cons

| Pros | Cons |
|---|---|
| Deterministic tests are trivial: seed the RNG before ticks. | Introduces a small global point of indirection (acceptable for testing). |
| Keeps randomness injectable for tests without changing engine. | |

**Decision**

**We adopted D2: a SpawnHelper that encapsulates the RNG and exposes setRandom for test seeding. This keeps production code simple while making unit tests deterministic and repeatable.**

## How the chosen design addresses software principles

### *DRY*

- Common drop-and-find-adjacent logic is factored into a helper used by both Apple and Yew trees, avoiding duplication.
- PlantFactory centralizes map-specific creation parameters rather than scattering map checks throughout grounds.

### *KISS*

- Each Ground subclass has one responsibility: evolve and produce. No centralized complex scheduler is introduced.

### *SOLID*

- Single Responsibility: Sprout/Sapling/Tree each manage one stage. PlantFactory handles creation. SpawnHelper handles randomness.
- Open/Closed: To add a new plant, create new ground classes (or new factory policy) without modifying existing ones.
- Liskov: Ground subclasses behave as Ground.
- Interface Segregation: Classes expose only needed methods (tick overrides).
- Dependency Inversion: Deterministic tests are enabled by inverting RNG dependency into SpawnHelper (higher-level logic depends on abstraction for randomness).

### *Co*nnascence

- Name-based connascence exists when creation depends on map name (factory input). This is localized in PlantFactory and therefore manageable.
- Temporal connascence (dependence on tick ordering) is documented i.e. plants expect the engine to call tick every global turn.

## Pros / Cons of the chosen approach

### Pros

- Clear, testable lifecycle behaviour per-stage; easy unit testing of each ground class.
- Factory centralizes map rules hence adding new map semantics is straightforward.
- No changes to the engine package required.
- Deterministic tests supported via SpawnHelper.setRandom.

### Cons and mitigations

- Class proliferation: mitigated by keeping each class small and by extracting shared helpers.
- Global RNG helper: a small global is introduced for testability and mitigated by resetting RNG in test teardown and documenting setRandom.

## Alternatives evaluated

- Central PlantManager (rejected): overkill and would create global scheduling complexity.
- Stateful single-class with enum (kept as potential refactor): would reduce classes but increase branching; can be considered if plant count explodes.
- Coarse-grained configuration file (future): central config (YAML/JSON) for per-map rules and attractive for many maps, but out-of-scope for this assignment.

## Why the chosen approach wins

- It aligns naturally with the engine's Ground.tick lifecycle, is minimal-intrusive (no engine edits), yields high local cohesion, and is easy to unit test. PlantFactory and SpawnHelper keep map semantics and randomness controllable and decoupled from tick behaviour.

## Maintainability foresight & examples

- Add new plant species: create new Ground subclasses for stages, add creation policy in PlantFactory, and add unit tests. No engine changes.
- Change growth policy: update PlantFactory to return a different configuration or swap a policy object.
- Support new map: register a new policy in PlantFactory (or extend it) so sprouts/saplings are created with the correct timings/production frequency.

- If production becomes complex (e.g., seeds that reproduce), migrate dropItem logic to a small ReproductionStrategy injected at creation time. This keeps the Ground class focused on lifecycle and defers reproduction rules to a strategy object.

## Testing & verification strategies

### Unit tests

- WildPlantsLifecycleTest includes 3 cases per expectation: normal, boundary, and an edge/negative case. Examples:
    - Forest sprout -> after 2 ticks still sprout (boundary), after 3 ticks -> sapling, after 3+5 ticks -> tree.
    - Plains sprout -> after 3 ticks -> tree (skip sapling); production frequency tested (apples appear within expected ticks).
    - Yew sapling -> seedable RNG to force growth or non-growth; deterministic assertion for both seeds.
    - Edge: block adjacent tiles; ensure no apple is placed on the tree tile and no exception thrown.

### Integration / manual tests

- Playthrough checks: placing sprout on different maps and visually verifying display char transitions (, t, T etc.) and that fruit characters (a, x, etc.) appear on adjacent tiles per spec.

### Deterministic RNG

- SpawnHelper.setRandom(Random) allows seeding in tests so probabilistic behaviour is reproducible.