REQ4 – Design Rationale: Planting, Fruit and Watering System

In REQ4, I built a planting and watering system that includes things like cursed ground recovery, hydration-based plant growth, refillable watering tools, and harvestable plants. I wanted the design to be clean, modular, and easy to extend later on, so I used interfaces and inheritance where it made sense.

The system covers:

- Recovering corrupted soil (like Blight) using either a shovel or a cure action
- Growing plants like Inheritree and Bloodrose, which depend on being watered regularly
- Using manual (WateringCan) and automatic (Sprinkler) tools that can be refilled
- A failure state (WitheredSoil) when plants die, which blocks further planting
- Harvestable items (e.g. InheritreeFruit) that can be sold or eaten

Design Alternatives

Design	Description	Pros	Cons
Design 1: Hardcoded logic in subclasses	Each plant subclass handles its own growth and watering logic independently.	Simple to implement for one-off plants.	High code duplication, poor scalability.
Design 2: Interface-based growth logic	Use of Waterable and Harvestable to unify plant behavior.	Promotes consistency and polymorphism; enables clean tool interaction.	Requires consistent tagging and adherence to interface contracts.

Design 3: Watering tool hierarchy	Tool classes inherit from WateringDevice, separated by use-case.	Encapsulates refill/use logic well; reusable refill actions.	Needs extra logic to handle auto-watering timing/range.
Design 4: Interface-based soil recovery	Soil like Blight can be restored via Shovelable and Curable.	Decouples ground logic from specific actors or tools.	Slightly more complex due to status checks.
Design 5: Irreversible failure state	Dead plants convert to WitheredSoil, cannot be revived.	Clear and predictable consequence for failed hydration.	Limits flexibility for future “revival” features.

Among these, I mainly used Designs 2–5 because they made the code easier to scale and reuse, especially when adding new tools, actions, or plants.

Justification

My goal for REQ4 was to make a small but complete farming system that works around hydration. I focused on the plant lifecycle (grow → harvest → die), how tools work (refillable/manual/automatic), and how players interact with both tools and the environment. I also added a basic shop system through an NPC named Kale.

The plants I added — InheritreePlant and BloodrosePlant — both extend from a base Plant class. They implement Waterable (for hydration tracking) and Harvestable (for maturity and

yields). If they aren't watered in time, they turn into WitheredSoil, which is basically dead land and can't be used again — this makes players think a bit more about timing.

Watering tools are organized into a class hierarchy. WateringDevice is the base class, and it branches into ManualWateringDevice and AutomaticWateringDevice. WateringCan is manual and used directly by the player, while Sprinkler automatically waters nearby plants every few turns. Both tools can be refilled using RefillWateringCanAction or RefillSprinklerAction, usually through Kale the shopkeeper.

For terrain, Blight is a type of cursed soil that can be converted back to regular soil. It implements Shovelable and Curable, depending on how the player wants to fix it. I tried to avoid instanceof and used hasCapability() checks instead, since they're cleaner and more flexible.

In a few places, such as retrieving specific item types from an actor's inventory (e.g., WateringDevice.getDevicesFrom(actor) or PlantFruit.getFruitsFrom(actor)), I used safe downcasting patterns with .filter() and .map(Class::cast) in stream pipelines. This approach is controlled and type-checked using instanceof-like filters (.filter(WateringDevice.class::isInstance)), which ensures the casts are valid and avoids runtime errors.

While downcasting is often discouraged in OOP when misused, in this case it avoids cluttering the system with additional boilerplate or hard-coded logic (e.g., manual instanceof checks in multiple locations). Instead, it encapsulates the type selection into static helper methods, promotes separation of concerns, and improves code readability. Since the actions operate on interface contracts like Sellable, Waterable, or Refillable, they remain decoupled from the exact subclass implementations, thus preserving maintainability and extensibility.

Some items like InheritreeFruit implement the Sellable interface, which means they can be sold at the shop or used for effects like restoring stamina. Shop logic uses actions like SellAction or BuyAction, which are simple but reusable.

SOLID Principle

I tried to follow the SOLID principles as best as I could:

- **SRP (Single Responsibility)**: Each class has a clear job. For example, Plant handles growth, WateringCan deals with water usage, and RefillAction manages refill logic.

- **OCP (Open/Closed)**: I can add new types of tools or plants by extending the existing classes or implementing interfaces like Waterable or Harvestable, without changing old code.

- **LSP (Liskov Substitution)**: Any item that implements Sellable or Waterable can be used in SellAction or WaterAction directly. No special handling is needed for different types.

- **ISP (Interface Segregation)**: Interfaces are kept small and focused. Tools that can be refilled implement Refillable, but other items don't have to.

- **DIP (Dependency Inversion)**: Actions don't depend on specific tool classes, just on the interfaces they implement. For example, WaterAction works with anything that implements ManualWatering.

Code Smell Avoidance

To keep the codebase clean and maintainable, I tried to avoid common code smells. For example, I moved shared logic like resetWaterCountdown() into the Plant superclass to reduce duplication. Each class has a clear focus — Kale handles shop logic, plant classes manage their own lifecycle, and watering tools handle usage and refilling — which helps avoid "God Classes." I also kept changes localized, so if I want to tweak how BloodrosePlant works, I only need to modify that class without touching the rest of the system.

Connascence Reduction

To reduce tight coupling between classes, I used interfaces instead of relying on specific concrete classes. For example, watering tools just need to implement ManualWatering, so it doesn't matter whether it's a WateringCan or some other tool — the actions don't need to know the exact class. Similarly, the watering logic works through a simple water() method, and the internal countdown logic is handled inside the plant itself. This makes the system more flexible and easier to expand.

Design-by-Contract Principles

Each key part of the system follows clear rules. Waterable ensures that a plant can be watered and its hydration resets. Harvestable guarantees that the plant will provide a yield when mature. Sellable defines how items are priced and sold. The tick() method in Plant handles hydration decay and triggers withering when needed. WitheredSoil can't be replanted or watered, keeping

that rule consistent. Interfaces like Curable and Shovelable define the conditions needed to convert or dig up soil. These rules make behavior predictable and help the system stay reliable and easy to test.