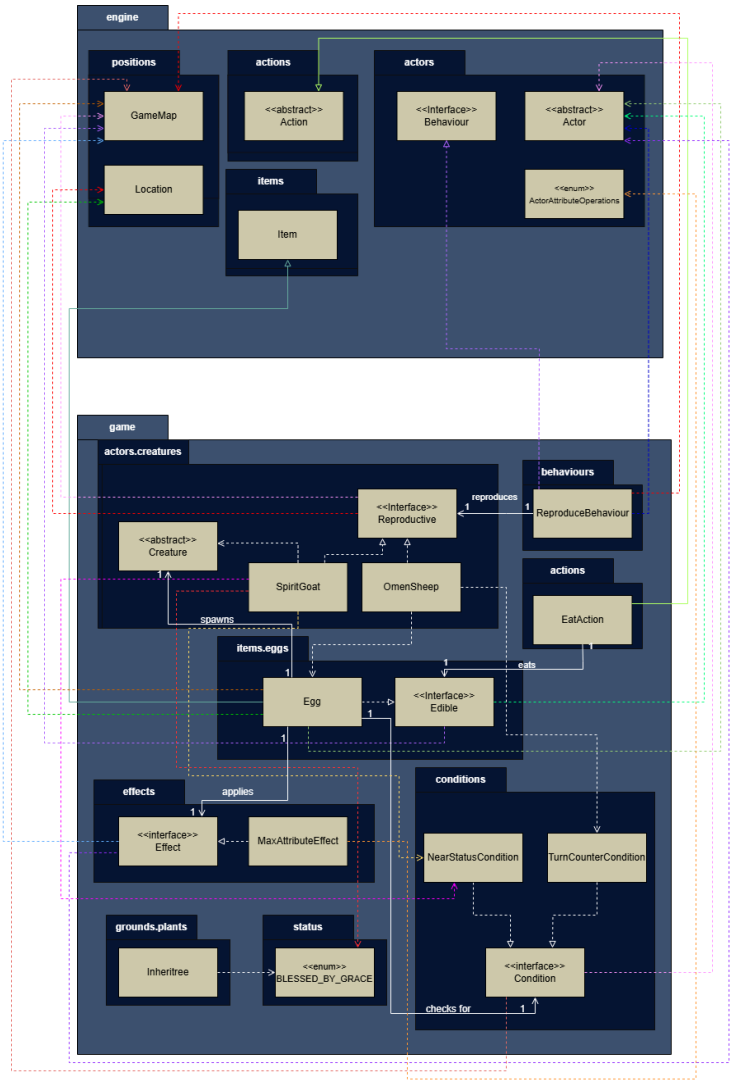
Design Rationale 2 - ELDEN THING: The Valley of the Inheritree

FIT2099 Assignment 2

Adji Ilhamhafiz Sarie Hakim	**33212422**
Anfal Muhammad Ahsan	**34309225**
Arielle Dela Cruz	**34086315**
Mohanad Al-Mansoob	**34303715**

Requirement 1



Design Goals

- Add new offspring types and hatching conditions without having to change existing code
- Separation of concerns regarding condition, effect and behaviour in reproduction logic
- Make use of existing reusable classes to avoid reinventing and unnecessary code addition.
- No instanceof or downcasting

Feature Summary

This feature introduces reproduction behaviour to SpiritGoat and OmenSheep, with SpiritGoat spawning a new SpiritGoat if adjacent to entities blessed by grace, and OmenSheep laying an edible egg every few turns, which hatches based on different conditions into different creatures. If the Farmer picks up the Egg, it is unable to hatch while in the Farmer's inventory and can be eaten, increasing max health by 10.

Design Decisions

We chose to implement a concrete Egg class with attributes for hatching condition, eat effect and hatchling.

Pros	**Cons**
One class handles all egg types; no need to implement multiple subclasses or have repeated code	Constructor has multiple parameters
Clear separation of concerns between ‘when’ (condition) and ‘what’ (effect)	Hard to track the code flow since it splits between different classes
Eating behaviour lives in reusable Effect implementations	It might be difficult to track in the code flow
Adheres strictly to SRP (each class has their own responsibility) & Open-Closed (add new conditions without changing Egg, reusable for other turn-based features	Partial reproduction logic still remains in creatures

Justification

Implementing one concrete Egg class allows us to avoid duplicate subclasses by using a single, highly configurable Egg. If we were to add new hatching conditions or eating effects, we would only require new Condition or Effect classes, never making changes to the actual Egg class. This design scales cleanly as more egg-types or conditions are introduced.

Design Alternatives and Comparison

Alternative Design	**Pros**	**Cons**
Chosen design: Concrete Egg with Condition, Effect and Creature attributes	- Easily maintainable as there is only a single class - Highly configurable - Follows SRP & OCP - Reusable across features	- Constructor needs multiple parameters
Abstract Egg class & subclasses	- Shared logic in Egg abstract class - Subclasses implement their own hatching logic within their classes	- One subclass per egg type, introducing many new classes - Inflexible inheritance makes mixing behaviours hard

Hard-coding logic within Egg.tick() [instead of using existing Condition and Effect]	- Easy to implement, all code in one place	- Mixes condition checking, hatching and eating logic together - Not reusable for other features

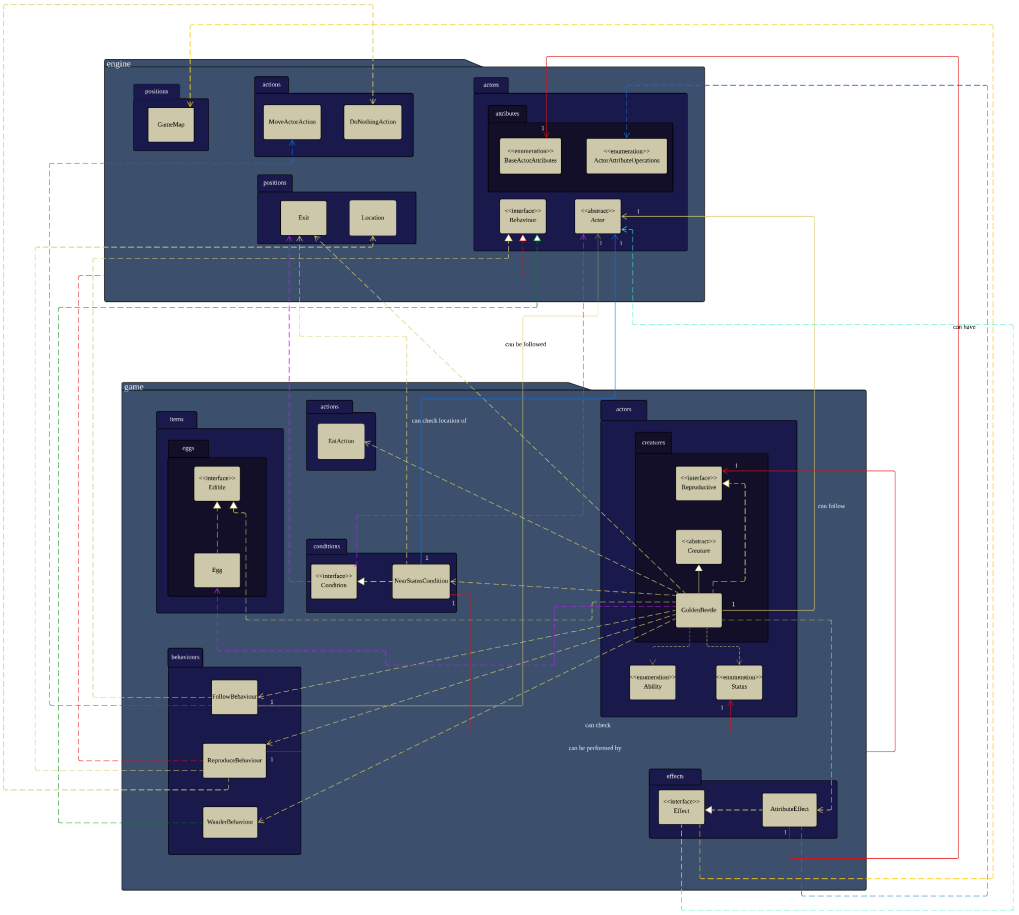
Summary

With our concrete Egg implementation, we can utilise reusable Conditions and Effects. This allows us to use and create reusable classes for other features, and follows SRP as Egg tracks turns and handles hatching logic, Condition handles hatching conditions, and Effect only applies effect. We adhere to OCP if we would like to add a new hatching condition or effect by supplying a different Condition or Effect without having to change the Egg itself. Egg depends on the abstractions Condition and Effect and not on concrete types, thus adhering to DIP. We implement dependency injection by passing in abstractions rather than hard-wiring logic, decoupling Egg from concrete implementations. We were able to avoid rigid inheritance without the use of abstract and deep subclasses. We achieve low connascence as collaborators communicate with well-defined interfaces (Condition, Effect, behaviours) and minimise hidden coupling. We avoided code smells like God classes, as each class has its narrow focus.

In regards to future extensions, we are able to add new conditions, eat effects and egg types without having to change the Egg class, and using existing reusable classes for efficiency. For instance, if eating an egg causes an increase in stamina, we can inject a new Effect.

In summary, this design strikes a balance between flexibility, clarity and adherence to the SOLID principles, all the while leveraging existing reusable classes. Our design is clean, maintainable and scalable, reusing existing classes to maximise consistency.

Requirement 2



Design Goals

- Ensure the GoldenBeetle class only focuses on the responsibilities specific to the beetle, namely, executing its defined behaviours and responding to interactions (like being eaten).
- Ensure all behaviours of the GoldenBeetle are executed in order of priority, with reproduction being evaluated before other behaviours like following or wandering.

Feature Summary

This feature introduces a new creature, the Golden Beetle, which extends the Creature class. The Golden Beetle can produce a Golden Egg, which can later hatch into another creature. Both the beetle and its egg are edible by actors. The Golden Beetle performs three behaviours: Reproduce, Follow, and Wander, which are prioritized and executed based on their importance. This modular design allows for flexible addition of behaviours and supports future scalability across different creatures.

Design Decision

Design 1: Use ReproduceBehaviour and a Reproductive Interface

In this design, a separate ReproduceBehaviour class encapsulates the logic for reproducing, which is executed based on its priority. The GoldenBeetle uses a Reproductive interface and simply declares the ability to reproduce. The game engine calls behaviours in order of priority. The Followable enum is also used to simplify detection of actors to follow.

Pros	Cons
Each class has a single responsibility: GoldenBeetle registers its behaviours, while ReproduceBehaviour handles reproduction logic.	Slightly more boilerplate as it requires defining and registering an extra class (ReproduceBehaviour) even for creatures with simple reproduction logic.
Simple and clean use of the Followable enum makes it easy to identify which actors should be followed, avoiding hard-coded type checks.	Logic spread across multiple classes might make debugging or tracing specific actions slightly more complex for beginners.
Promotes code reuse: uses Creature parent methods like addBehaviour() and getBehaviour() for consistent behaviour handling across creatures.	
Modular and extensible: new creatures can easily reuse ReproduceBehaviour without rewriting reproduction logic.	

Design 2: Integrate Reproduction Logic Directly in GoldenBeetle

In this design, the GoldenBeetle class implements Reproductive and directly handles the reproduction logic inside its playTurn() method, rather than delegating to a behaviour.

Pros	Cons
Simpler to implement for a single creature: all logic is in one place without the need to	Reproduction is not executed based on priority: it becomes part of the general

create extra behaviour classes.	playTurn() execution flow, potentially after lower-priority actions.
Reproduction is guaranteed to happen every turn without relying on external priority-based execution.	Violates the Single Responsibility Principle (SRP): GoldenBeetle now contains reproduction logic as well as behaviour management, reducing cohesion.
	Less reusable and harder to extend: other creatures can't reuse the reproduction logic without duplicating code.

Justification

The GoldenBeetle class extends Creature, aligning with inheritance and polymorphism principles. Since all creatures share core functionalities like movement and behaviour selection, inheriting from Creature prevents redundant logic and promotes consistency.

By offloading reproduction logic to a separate ReproduceBehaviour class, we follow the Single Responsibility Principle (SRP) — GoldenBeetle manages its identity and declares its behaviours, while the how of reproduction is handled externally. This separation enhances modularity, testability, and reusability.

The Followable enum abstracts which actors are follow targets, avoiding hard-coded checks or instanceof logic. This reduces code smells like type-checking or feature envy and ensures code conformity with clean object-oriented principles.

The use of NearStatusCondition and AttributeAffect ensures condition and effect logic is encapsulated, supporting the Open/Closed Principle (OCP) — new conditions or effects can be added without modifying core entity classes.

The design avoids God classes, long methods, and duplicated logic, which are common code smells in tightly coupled systems. By isolating responsibilities across well-named, purpose-driven classes, the structure stays easy to navigate and extend. We also maintain the Don't Repeat Yourself (DRY) principle by reusing generic condition, effect, and behaviour systems rather than hardcoding logic in each actor.

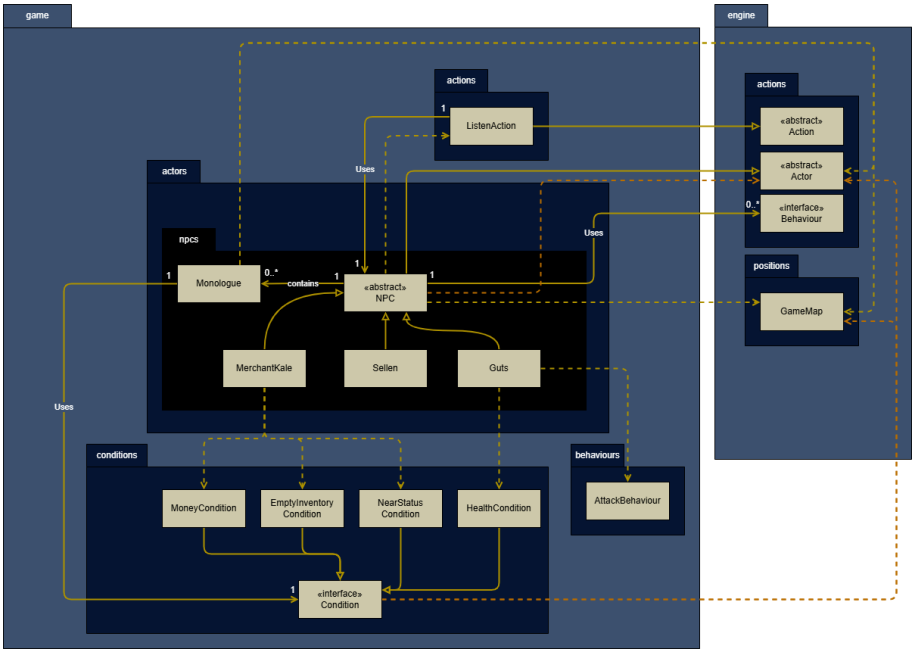
Summary

The GoldenBeetle class extends Creature and implements Edible and Reproductive. It supports being eaten and reproducing by laying a GoldenEgg. The beetle's turn logic is

driven by prioritized behaviours like reproducing, following, or wandering defined using separate behaviour classes. The egg laid uses NearStatusCondition to hatch when appropriate, and AttributeAffect to restore stamina when consumed.

Dedicated classes for behaviours, effects, and conditions, along with enums like Followable, promote modular, readable, and extensible code. This design avoids code smells like feature envy, duplication, or bloated classes, and conforms to clean architecture guidelines. It remains flexible for future additions, like new egg types, reproduction conditions, or edible effects.

Requirement 3



Design Goals

- Ensure common dialogue functionality can be shared across all NPCs
- Allow the addition of new NPCs, monologues, and dialogue conditions with minimal changes.
- Keep a clear separation between different categories of actors (NPCs vs creatures)

Feature Summary

This feature introduces NPCs (Sellen, Kale, and Guts) with contextual and random dialogue interactions. The “Farmer” can choose to listen to nearby NPCs and receive dialogue based on predefined monologues, some of which are conditionally shown based on game state.

Dialogue System Design Decision: Abstract Class vs Interface

We considered two main approaches to implementing NPC dialogue capabilities:

Criteria	Talkable Interface	NPC Abstract Class
Code Reuse	Requires repeating dialogue logic in each implementing class	Common functionality is implemented once
Interface Segregation Principle	Adheres to ISP	Does not adhere to ISP
Ease of implementation	Requires more effort to maintain	Quick and DRY implementation with minimal duplication

Justification:

We chose the abstract class NPC approach as we expect all NPCs to have a dialogue system. Using an abstract class made it easier to centralise monologuePool, dialogue filtering, and allowableActions() logic. This would save a lot of code repetition of identical behaviour across many classes. However, we acknowledge that silent NPCs might be introduced in the future, so we might use a Talkable interface for better flexibility, and this would not be a hassle, as of currently, there are only three NPCs introduced.

Possible Designs:

1. Using an abstract class for NPC, which extends to the children (e.g. Sellen) to implement the monologue system to all NPCs, where the condition checking would be placed within an abstract method that can be modified specifically for each NPC.

Pro	Con
Centralised logic: Shared code for all NPCs is easily reused in the base class.	Limited flexibility: Java supports single inheritance, so using an abstract class prevents NPCs from extending another class if needed.
Encapsulation: NPC-specific behaviour is tightly scoped to NPC subclasses.	Tightly coupled logic: Makes it hard to reuse or share monologue-related logic with non-NPC actors (e.g., player or allies).
Easy inheritance: Simple to create new NPCs with custom monologue behaviours.	Rigid design: Embedding condition logic in an abstract method means that adding new types of condition logic might require modifying every subclass.

2. Using an interface to implement the monologue system, where an NPC (or Actor in this design) would implement an interface that allows the NPC to speak while also allowing the condition checking to be placed within the interface method.

Pro	Con
High flexibility: Any class (NPCs, allies, players) can easily be made to support monologues.	Code duplication: Common functionality (e.g. monologue pool) must be manually re-implemented or pulled from utility classes.
Better separation of concerns: Encourages modular design; speaking behaviour is decoupled from core actor logic.	No state: Interfaces can't hold state before Java 8; even with default methods, managing internal monologue lists can be clumsy.
Promotes composition over inheritance.	Inconsistent structure: Without a common base class, implementers may diverge in how they structure or manage their monologues.

3. Using both abstract and interface, where the abstract would contain the common method for the NPC,s while the interface would be used to check for conditions for the monologue.

Pro	Con
Extensibility: New conditions can be created independently of NPC subclasses.	Slightly more complex: Requires understanding of both inheritance and composition.
Separation of concerns: The NPC doesn't need to know how conditions work—just when they apply.	Requires coordination: Developers must understand both the abstract class and interface contract to correctly implement new behaviours.
Reusability: Condition implementations can be reused across different types of NPCs or even non-NPCs.	

In our implementation for requirement 3, we have implemented the third design as it is the best of both worlds since by having abstraction we can minimize repeatability to the NPCs since we consolidated the common methods and attributes inside the NPCs such as monologue and allowableActions() method, while also using the interface to implement the condition which would allow the reusability of the condition (e.g. 1 condition or an if-else statement can be used by another class without re-writing the whole statement).

This design follows the SOLID principle as each class has its own responsibility; the NPC abstract class contains the common NPC behaviour, such as holding the monologues and determining the actions each turn. Moreover, the Condition interface with its implementations

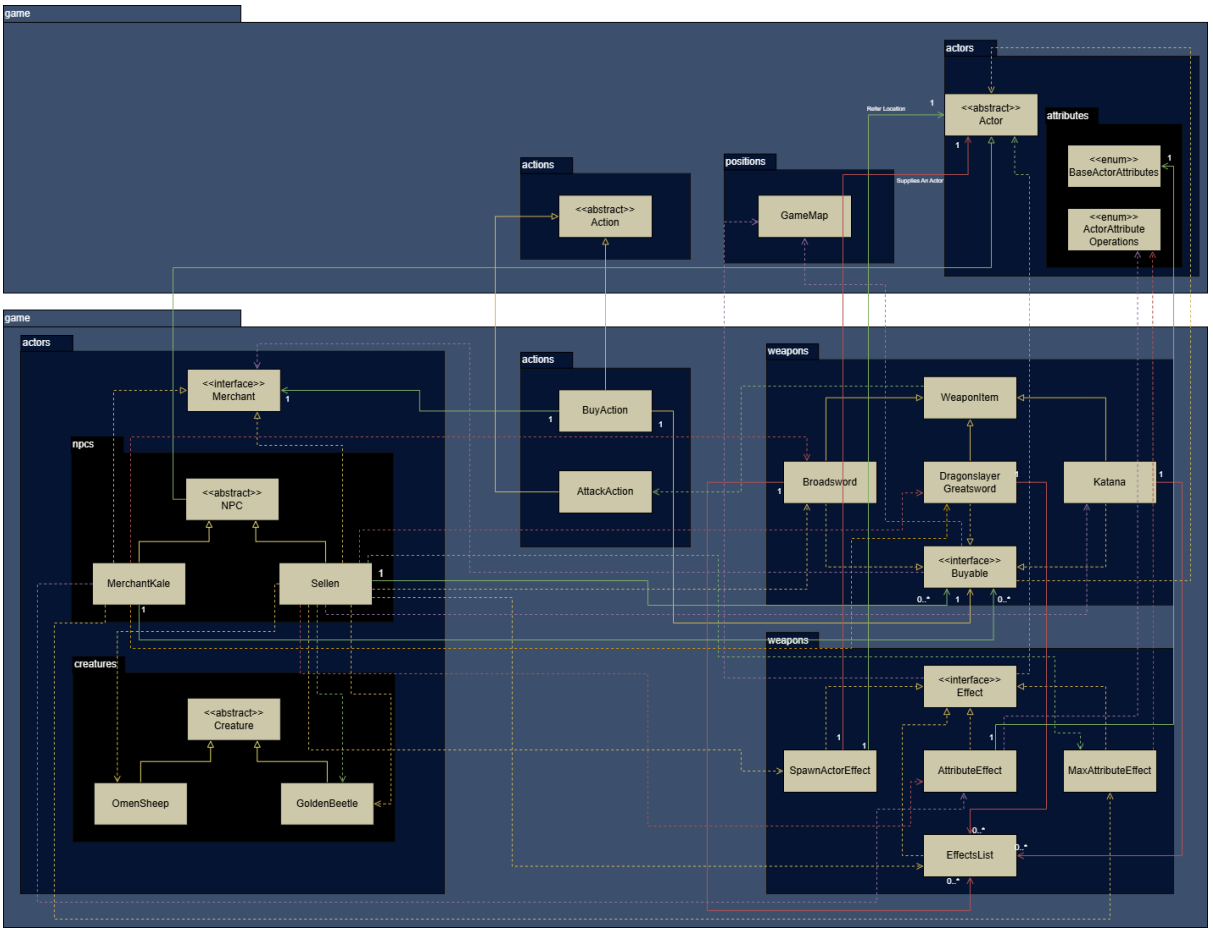
(e.g. NearStatusCondition, MoneyCondition) handles their own conditions to be used for the monologue display.

Moreover, with the way it is implemented, we can create new conditions without modifying any prior classes, such as a condition that checks how many turns have been taken by the NPC or anything for that matter. The interface that allows the Open/Closed Principle also helps with the Dependency Inversion Principle since the interface is abstract, which allows the Monologue class to be loosely coupled with the Condition classes instead of tying it to a specific condition.

In addition to being loosely coupled, since the Liskov Substitution Principle requires the subtypes to be substitutable for their base types, the NPC subclasses can be used whenever a class expects an NPC class without anything unexpected happening.

Since we only have 1 method inside the condition interface, we allowed the Condition class to be minimal and focused, which helps the NPC not implement any unrelated interface to support monologues.

Requirement 4



Design Goals

- Easily support new weapons and merchants with varying purchase effects.
- Centralizing buying logic and effect logic so they are reused across different weapons/merchants.
- Avoid hard-coding item-specific or merchant-specific logic inside unrelated classes.
- Enable future changes (e.g., new effects, weapons) with minimal code updates.

Feature Summary

The Wayward Sellers of Wares introduces a feature where the player (Farmer) can directly purchase weapons from NPC merchants like Sellen and Kale. Each weapon has different purchase effects depending on the seller, such as stat boosts, healing, spawning new entities. The buying process must happen through the standard action menu (no submenus), and each buy action consumes a game tick like any other actions. The system is designed to allow different merchants to sell the same or different items with custom post-purchase effects.

Strategy Pattern & Effect System

To support dynamic and modular behaviour when purchasing items from different merchants, I implemented an Effect System that utilizes the Strategy Pattern. This pattern enables the encapsulation of varying algorithms (in this case, post-purchase effects) and allows them to be interchanged independently of the core buying logic.

I introduced an Effect interface along with several concrete implementations:

- AttributeEffect: modifies a specific attribute (e.g., stamina or health).
- MaxAttributeEffect: increases the maximum value of an attribute
- SpawnActorEffect: spawns an actor near a target.

How the Strategy Pattern is Used

This pattern is used in the effect system to encapsulate post-purchase behaviours as interchangeable Effect objects. Each effect, such as increasing stamina or spawning a creature, is implemented as a separate class. Buyable items hold a list of these effects (EffectList) and apply them when purchased. This decouples the item logic from specific outcomes and allows flexible, dynamic combinations of effects without modifying the core logic which supports scalability and clean design.

Justification

This design allows me to easily define custom behaviours for purchase without tightly coupling the logic to the merchant or weapon classes. It also adheres to the Open/Closed principle where new effects can be added without modifying existing ones. Additionally, these effects can be used globally across all of other classes on the game which could make implementing features with them easier,

Use of Supplier<Actor> in SpawnActorEffect

To support the dynamic spawning of actors, I used a Supplier<Actor> in the SpawnActorEffect class. This functional interface allows us to create a new instance when the effect is applied. Using the same instance can lead to illegal state errors as the game cannot handle having the same instance on the same GameMap.

Why Supplier<Actor> is a Good Design Choice:

- Encapsulates actor creation logic: the effect class does not need to know which specific actor to spawn or how to construct it.
- Prevents shared state: ensure that each invocation produces a new instance, maintaining correct behaviour in the game.
- Promotes flexibility: different actors can be injected into the effect without changing the effect logic itself
- Supports testability: during testing, mock suppliers can be provided

Buyable Interface

To formalize the concept of items that can be bought, we created a Buyable interface. This interface requires implementing:

- Int getCost()
- Void onPurchase(Actor, Merchant, GameMap).

Each weapon such as BroadSword, Katana, and DragonslayerGreatsword implements this interface and defines its purchase cost and logic.

Justification

This allows any class (weapon or item) to be marked as purchasable while keeping the buying logic localized within the item itself. It simplifies the merchant's code and avoids violating Single Responsibility Principle.

Dependency Injection

When creating Buyable weapons, EffectsLists are injected through their constructor. This way, a single weapon like BroadSword can be reused by multiple merchants, each supplying different effects depending on their design.

```
EffectsList broadSwordEffects = new EffectsList();
broadSwordEffects.addEffect(new MaxAttributeEffect(BaseActorAttributes.HEALTH, amount: 20));
sellItems.add(new BroadSword( cost: 100, broadSwordEffects));
```

Justification

This design follows the Inversion of Control principles. It also avoids tight coupling between effect logic and specific actor classes.

Design Alternatives & Comparison

Decision	Alternative	Pros	Cons
Effect interface + EffectsList	Hardcoded logic in item/merchant	Easy for simple cases	Violates SRP, not extensible
Merchant interface	Make all NPCs able to sell	Fewer classes	Breaks Interface Segregation
Buyable interface	Extend the base item class directly	Fewer types	Doesn't support non-item purchases
Dependency Injection	Configure logic inside the weapon	Central control	Poor modularity and reuse
Use of Supplier	Pass a raw instance of actors	Simpler code	Less flexible, tightly coupled

Pre Requirements

To support the planting mechanic within the game, we introduced a general-purpose Seed class that models any plantable item.

Design Structure

The Seed class is a subclass of Item and contains:

- A reference to the Plant that will be grown,
- A stamina cost is required to plant it,
- A plant() method that handles the actual planting logic,
- An override of allowableActions() to dynamically expose the PlantAction only when the ground is marked as PLANTABLE

Refactor Reasoning:

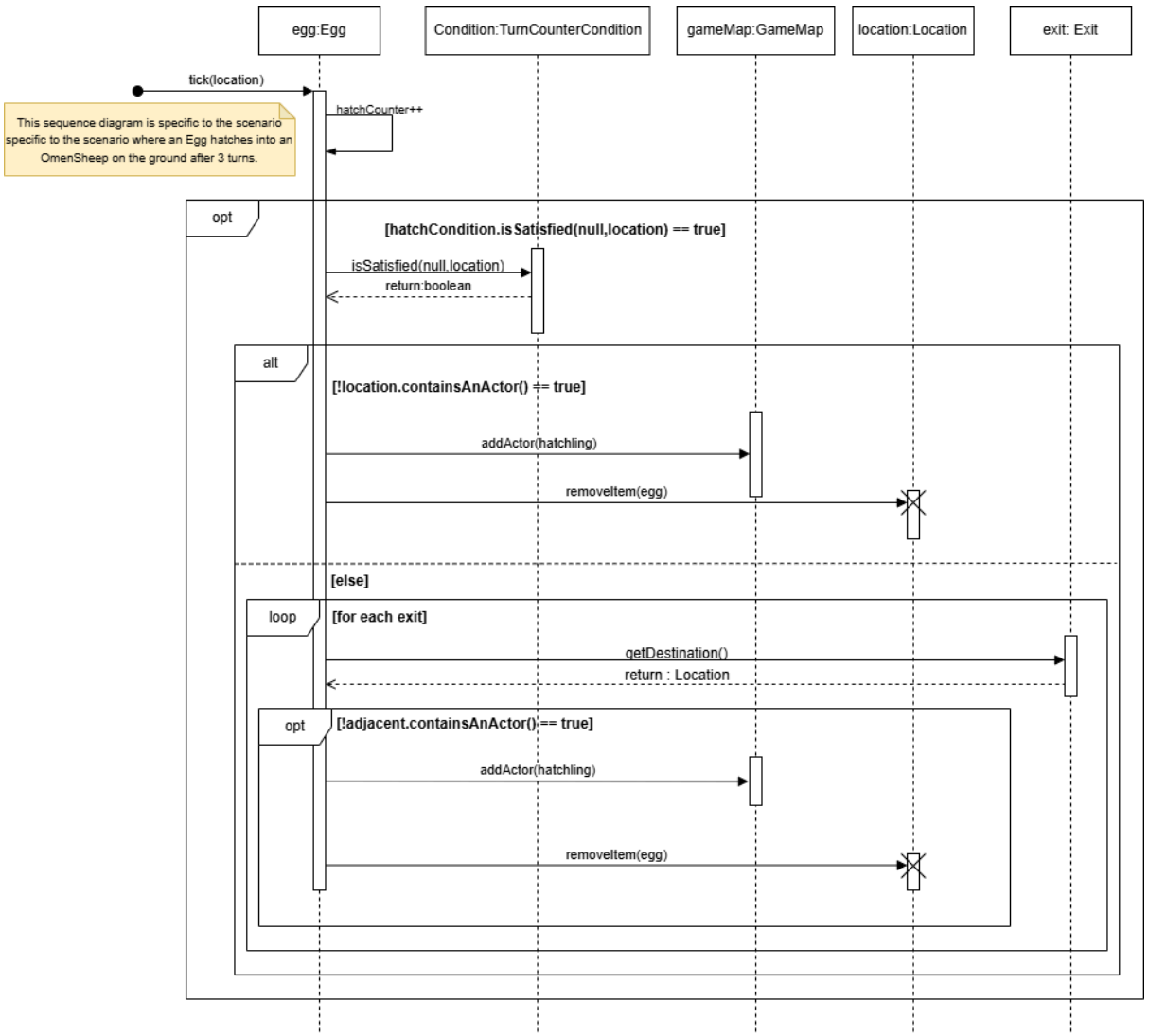
This design uses a single concrete Seed class that can be configured through constructor parameters instead of subclassing for each seed type. This avoids over-abstraction, which was an issue in our earlier submission, where we created separate concrete subclasses like InheritreeSeed and BloodroseSeed

Refactor Pros:

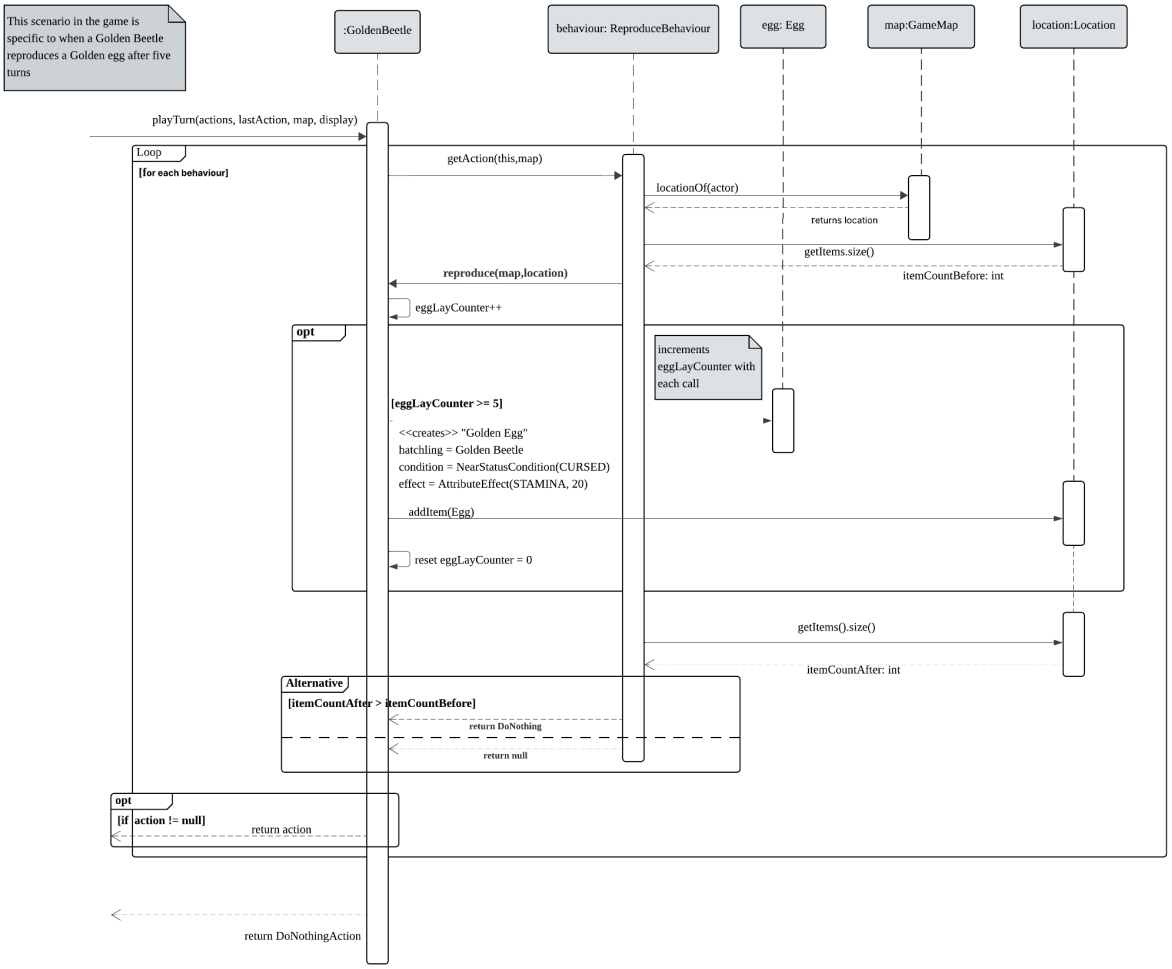
Principle	Application
Single Responsibility Principle	Seed encapsulates all planting-related behaviour in one place, rather than distributing it across multiple unrelated classes
Open/Closed Principle	New plant types can be added by passing in different Plant instances without modifying the existing Seed logic
Command Pattern	The PlantAction encapsulates the intent to plant and delegates to the seed's plant() method for execution

Addendum

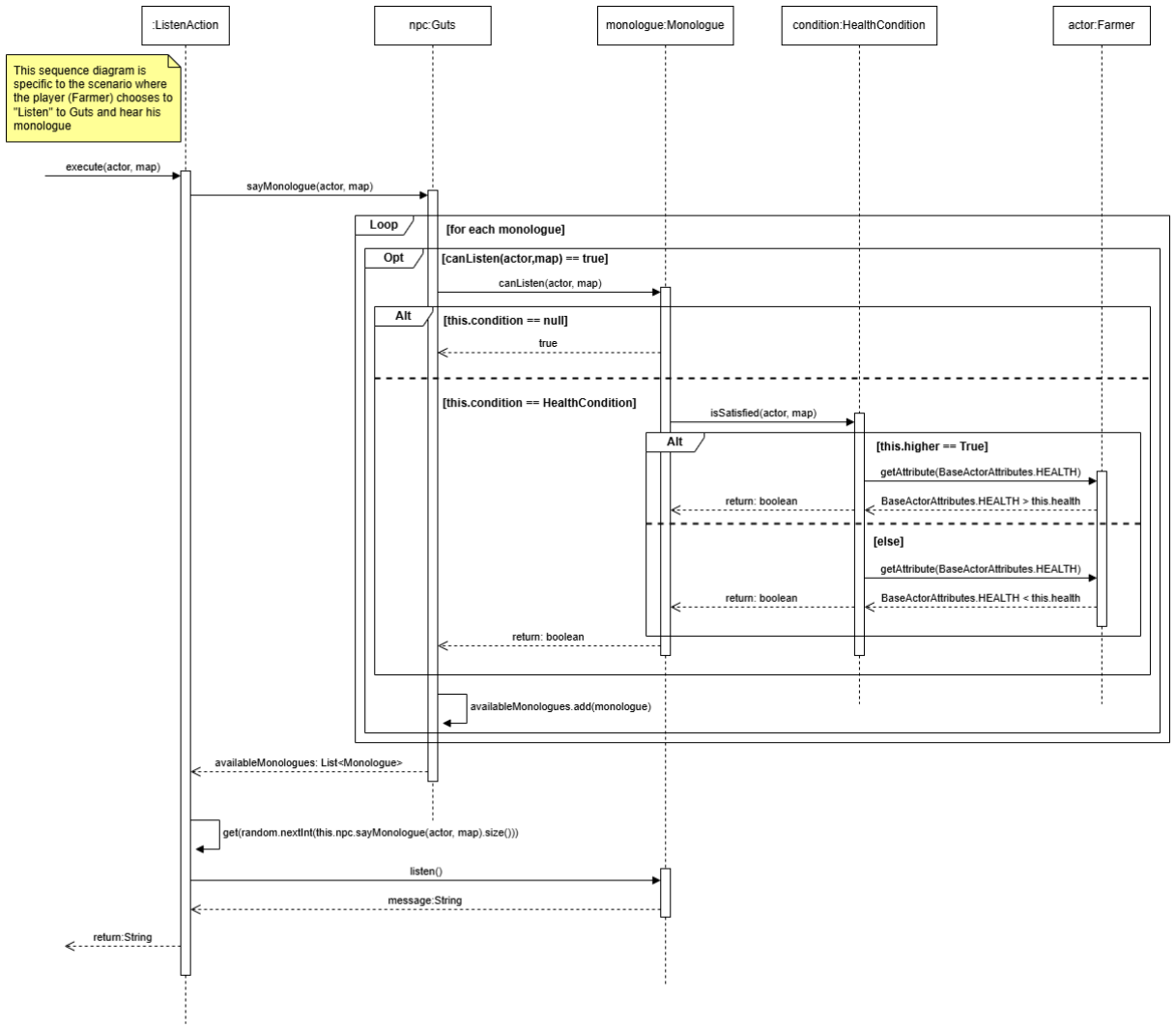
Requirement 1: Sequence diagram



Requirement 2: Sequence diagram



Requirement 3: Sequence diagram



Requirement 4: Sequence diagram