REQ 1:

1. Implementing Destination Wiring for TeleportCircle

Design A:

Uses a no-argument constructor for **TeleportCircle** and internally maintains a *List<Location>* named destinations. In the **Application** setup, each **TeleportCircle** is instantiated first and then wired to one or more destinations via repeated calls to *addDestination(Location location)*. The *allowableActions()* method iterates through this list and dynamically generates **TeleportAction** objects pointing to each target **Location**. This wiring approach separates teleport logic from scene setup.

Pros	Cons
By using addDestination() after the instantiation of the teleport circle, it adheres to the **Open Closed Principle (OCP)** in which new teleport targets can be introduced without modifying the **TeleportCircle** class, making it highly adaptable to future maps.	The process of instantiating the teleport circle and then calling *addDestination* increases the risk of incomplete configuration e.g., If a developer forgets to call *addDestination()*, the circle will appear in the game but without the teleport functionality, which is the most important feature of the teleport circle itself.
Maintainability is enhanced by enabling teleportation logic to develop independently, by supporting multiple destinations without necessitating code modifications to the core class.	
Unit testing and setup for dynamic maps become easier because the teleport circles can be instantiated independently and configured later. By separating teleport logic from game initialisation, this upholds the **Single Responsibility Principle (SRP)**.	

Design B:

Each **TeleportCircle** is instantiated with a single destination directly in the constructor, e.g., *TeleportCircle(GameMap destMap, int x, int y)*.

Pros	Cons
By integrating the teleport logic directly into the instantiation, it minimises the number of method calls in the **Application**, which simplifies the setup of the circle in situations with only one destination.	This method completely does not allow for multiple destinations without rewriting the class by hardcoding a single destination into the constructor. This design would require a redesign of the class and all of its constructor calls if future gameplay calls for conditional teleportation or teleport circles with multiple options, which would go against the **Open Closed Principle (OCP)** and increase maintenance work.
	Tightly coupling construction with logic makes the class harder to reuse or reconfigure, complicating testing and future modifications.

Comparison:

Design A was chosen due to its compatibility with modularity and extensibility. This model scales more smoothly with complex teleporting behaviour because the **TeleportCircle** class uses *allowableActions()* to support multiple destinations and actions. Design B may be easier to develop initially, but it makes it more difficult to expand and maintain in the future. By following the **Open Closed Principle (OCP)** and **Single Responsibility Principle (SRP)**, Design A guarantees that existing systems will be minimally disrupted in the event that future requirements call for multi-location teleportation or conditional destination logic.

2. Implementing Behaviour Selection Strategy for Creatures

Design A:

A **BehaviourSelector** instance is accepted as a constructor parameter by creatures like **GoldenBeetle**, **OmenSheep**, and **SpiritGoat**. The selection logic can be fully abstracted out thanks to this interface. During *playTurn()*, implementations such as **PriorityBehaviourSelector** and **RandomBehaviourSelector** control the selection of actions from the behaviours map.

Pros	Cons
Encapsulates selection logic in a separate class, which promotes flexibility and reuse. Since new behaviour selectors can be added without changing already-existing creature classes, this supports the **Open Closed Principle (OCP)**.	In order to understand decision-making, developers must trace from the creature class through the selector, which adds a layer of complexity and may reduce clarity.
Both parent and offspring creatures can inherit **BehaviorSelector** instances, minimising duplication and maintaining consistent behaviour inheritance.	

Design B:

Creatures hardcode their behaviour selection within *playTurn()* using if-else statements.

Pros	Cons
Behaviour flow is instantly visible and able to be understood easily due to the centralised logic of the creature class.	Reduces modularity of the code by tightly coupling creature functionality with decision logic. Editing each creature separately will be necessary to add or modify behaviours.
	Becomes increasingly difficult to implement changes consistently across all creatures as their numbers rise. This violates the **Don’t Repeat Yourself (DRY)** principle.

Comparison:
Design A was chosen to ensure maintainability and extensibility as the number of behaviours and creatures grows. Although Design B offers simplicity, its tightly coupled logic and duplication risks make it unsuitable for large-scale systems. By utilising the **Open Closed Principle (OCP)** and **DRY principle**, Design A guarantees that current logic is unaffected if new creature types are created.

3. Implementing Egg Hatching For Whichever Type Of Behaviour Selector The Parent Is

Design A:

Uses the **CreatureType** enum stored in each egg to determine which creature to hatch. The **BaseHatchCondition** interprets this enum and creates the corresponding creature using a centralised *createCreature()* method. The switch-case logic links creature types with their instantiation logic.

Pros	Cons
Because the creature creation process is centralised and simple to follow, debugging is made easier and all instantiation logic is handled in one location.	Ties creature type expansion directly to the *createCreature()* method which allows for tighter coupling.
This method keeps the logic simple and manageable while avoiding unneeded abstractions.	

Design B:

Each egg is given a **CreatureFactory**, which is in charge of creating instances of the appropriate creature behaviour selector type. These factories may be used for condition setup or construction.

Pros	Cons
Supports the **Open Closed Principle (OCP)** since new creature types can be introduced without changing existing code by defining new factories.	Complexity is increased by the extra abstraction layer which is unneeded, especially when only few creature types exist.
	Creation logic is distributed across several factory classes, making it more difficult to trace bugs that may exist.

Comparison:

Design A was selected due to its simplicity and suitability for the number of creature types. On the other hand, Design B increases the difficulty of debugging by distributing creation logic across several factories and adding more layers of abstraction. If future requirements require new types of creatures or different ways of creating them, the centralised *createCreature()* method in Design A can still be extended with minimal impact, preserving the existing interface and logic flow.

REQ 2:

1. Implementing specific BoCAttackBehaviour and BoCAttackAction classes rather than reusing existing AttackAction and AttackBehaviour.

Design A:
The first design approach considered was to reuse the existing generic AttackAction and AttackBehaviour classes from the game package to handle the Bed of Chaos's attacks. This would involve treating the Bed of Chaos as a typical actor with no specialized logic for its unique combat style.

Pros	Cons
Consistent attack behaviour across all actors in the game.	Lacks flexibility for the Bed of Chaos's unique combat logic, which is variable damage scaling with growth.
Reuses existing code, reducing duplication.	Would require workarounds or conditionals inside the generic attack classes, introducing potential code smells.
	Harder to extend the Bed of Chaos's combat features in future requirements.

Design B:

The second approach, which was implemented, was to create specialized classes: BoCAttackBehaviour and BoCAttackAction. These classes encapsulate the Bed of Chaos's unique attack behaviour, including variable damage based on its growing parts.

Pros	Cons
Fully encapsulates Bed of Chaos's combat logic, adhering to the Single Responsibility Principle (SRP).	Introduces more classes, which could marginally increase complexity.
Follows the Open Closed Principle (OCP) by allowing future extension (for example unique attack animations, status effects) without	Potential maintenance burden if future changes need to coordinate between generic and specific attack systems.

modifying the generic action and behaviour classes for attacking.	

Comparison:
Design B was chosen because the Bed of Chaos is a boss with combat mechanics that deviate significantly from normal actors. By creating dedicated attack classes, the implementation adheres to SOLID principles (SRP and OCP), and minimizes connascence by decoupling boss-specific combat logic from generic attack logic. This avoids tight coupling that would arise if I tried to force boss-specific features into generic attack classes. The minor increase in class count is justified for the sake of cleaner design and better extensibility.

2. Implementing effects package classes: TreeEffect interface, and BoCHealingEffect to manage healing effects by Leaf.

Design A:
Initially, I considered directly placing the healing logic (adding HP to the boss) in the Leaf class itself whenever it grows. This approach would avoid creating extra classes or interfaces.

Pros	Cons
Simplifies the code, with fewer classes and reduced boilerplate.	Increases connascence between growth logic and healing logic, making future changes like adding other effects riskier.
Direct healing logic in the leaf’s apply healing method, which means fewer levels of indirection.	Harder to maintain if more types of effects (like damaging effects or buffs) are added.

Design B:
The second approach, and the one implemented, is to create a dedicated effect interface (TreeEffect) and a specific BoCHealingEffect class for healing. This separates the effect logic from the growth logic of the leaves.

Pros	Cons
Adheres to the Single Responsibility Principle. Leaf's job is to trigger effects, not manage them.	Requires an extra interface and class even for a single healing effect, which can feel extra for smaller scaled projects.
The design follows the Open Closed Principle, that new effect types like poison or defence boosts can be added by simply implementing the TreeEffect interface, without touching existing growth logic.	Increases the number of classes which may be harder to maintain.
Improved modularity and readability, effects are encapsulated and clearly named.	

Additionally, the healing effect class was renamed from HealingEffect to BoCHealingEffect to avoid name clashes with other healing classes in the project or game engine. This renaming follows the principle of minimizing connascence of name, ensuring that class names are specific and unambiguous in the larger project context.

Comparison:
Design B was chosen to ensure a flexible and clean design that can accommodate future effects without modifying core leaf or growth logic. While it adds a few extra classes, it avoids code smells of merging distinct responsibilities and promotes a scalable, maintainable system that respects SOLID and minimizes potential for unintended changes, reducing connascence of change.

3. Avoiding downcasting by moving growable looping logic outside of GrowAction and ensuring recursive growth in BedOfChaos and Branch.

Design A:
Initially, I considered handling the entire growth process inside the GrowAction class, including casting the growable parts(including Bed of Chaos) to specific types to handle special and recursive growth logic for branches.

Pros	Cons
Keeps growth logic inside one place (GrowAction), which may seem simpler in the early stages.	Violates the Liskov Substitution Principle, that is when downcasting is used, it’s a sign that objects aren’t substitutable as intended.
	Downcasting increases connascence of type, where changes to concrete types propagate through multiple classes, making maintenance harder.
	Introduces type-checking logic in GrowAction, violating the Single Responsibility Principle. It now does more than trigger a growth action.

Design B:
The refined approach that I implemented here is to keep GrowAction minimal, simply triggering the polymorphic growThisTurn() method on Growable objects. The actual recursive growth logic (like branches growing more parts and recursively calling sub-growth) was implemented in the BedOfChaos and Branch classes themselves.

Pros	Cons
Fully adheres to the Open Closed Principle, that new growable types (new tree structures) can be added without modifying GrowAction.	The growth logic is spread across BedOfChaos and Branch rather than being centralized, which can make it slightly harder to trace the complete growth logic in a single class.
Completely avoids downcasting and instanceof checks, removing connascence of type and keeping polymorphic behaviour clean.	Adds a small amount of repeated logic in BedOfChaos and Branch classes (recursive traversal).
Clearly separates responsibilities. GrowAction only triggers growth, while actual growth logic resides with the growable entities.	

Comparison:
Design B was chosen because it respects polymorphism and avoids code smells associated with type-checking and downcasting. It fully follows the SOLID principles, especially Liskov Substitution and Single Responsibility, while keeping each class focused on its own responsibility. GrowAction triggers growth, while BedOfChaos and Branch own the logic of their growth. By eliminating connascence of type, the design ensures future maintainability and makes the growth system flexible for future extensions, for example adding new types of growable tree parts.

REQ 3:

1. Create seed classes for each of the plans: CarrotSeed, MushroomSeed and LettuceSeed..

   Design 1
   Create the seed classes directly.

Pros	Cons
This design is simple and straightforward as all the necessary functions are implemented directly within the class itself.	This design violates the DRY principle as common attributes and methods are repeated across each class.
This design reduced the dependency between the abstract class and the concrete classes.	

   Design 2
   Create the classes by inheriting the abstract Seed class.

Pros	Cons
This design promotes code reusability by centralizing common functionality in the abstract class, so there's no need to have duplicated code in each seed subclass. Thus, it adhered to the DRY principle.	The concrete seed classes are tightly coupled with the abstract Seed class, which can make it hard to modify without affecting the subclasses.
Each concrete seed is only responsible for one responsibility for a specific plant, so this design is adhered to the Single Responsibility Principle.	
The new types of seeds are added without modifying the existing classes, so this design is adhered to the Open Closed Principle.	

   Design 2 is eventually chosen as it provides a more scalable and maintainable solution compared to Design 1, allowing for easier extension and better code reusability.

2. For each plant, define the corresponding sprout, mature plant, and plant item classes as follows:
    - Carrot:
        - sprout - CarrotSprout
        - mature plant - MatureCarrot
        - plant item - Carrot
    - Lettuce:
        - sprout - LettuceSprout
        - mature plant - MatureLettuce
        - plant item - Lettuce
    - Mushroom:
        - sprout - MushroomSprout
        - mature plant - MatureMushroom
        - plant item - Mushroom

    Design 1

    Create the sprout, mature plant and plant item classes directly for each of the plants.

Pros	Cons
This design promotes simplicity as each class is independent and directly defines the functions related to the specific sprout, mature plant and plant item.	This design violates the DRY principle as common attributes and methods are repeated across each specific class.
This design reduced the dependency between the concrete sprout, mature plant and plant items classes to their corresponding abstract classes.	

Design 2

Create abstract Sprout, MaturePlant and PlantItem classes and let the corresponding concrete classes extend from them.

Pros	Cons
This design promotes code reusability as all the common functionality is centralized in abstract base classes, which reduces duplication. Thus there will be no repeated code, which adhered to the DRY principle.	Introducing the abstract Sprout, MaturePlant and PlantItem classes adds another layer of abstraction, which can make the system more complex and harder to understand.
This design also adhered to the Open-Closed Principle as if more types of plants are added in the future, they can be easily added by creating subclasses for sprout, mature plant and plant item without modifying the existing code.	

Design 2 is chosen because it promotes code reusability, reduces duplication, and supports easier extension without modifying existing code.

3. Implementation for watering system.

    - Create a WateringCan class that extends the Item class.

<u>Design 1</u>
Create the WateringCan class which adds the new ability CAN_WATER to the player, and we can now check the ability of the player to determine if the player can water plants.

Pros	Cons
The player's ability is extended without modifying the existing codes, which adhered to the Open-Closed Principle.	It might be too simplistic if in the future, different plants require different types of watering.
This design encourages loose coupling and reduce connascence, because the system can check the capability of the player instance instead of checking the specific class type (instanceof).	

<u>Design 2</u>
Create the WateringCan class and use the built-in instanceof to check if the player is having the WateringCan to determine if the player can water plants.

Pros	Cons
This design promotes simplicity as it is easy to implement.	This design violates Open-Closed Principle as it requires modification on the condition check when adding new watering tools in the future.
	The logic is directly tied to the specific class, which makes it hard to reuse.

Design 1 is eventually chosen as Design 1 violates the SOLID principles and does not promote code reusability.

- Create a Waterable interface that is implemented by the abstract Sprout class.

Pros	Cons
This design adheres to the Interface Segregation Principle as Waterable interface defines a clear contract for the objects that can be watered.	It potentially violates the Liskov Substitution Principle if, in the future, a sprout is introduced that should not be waterable.
This design follows the Open-Closed Principle as if a new Waterable entity exists in the future, it can simply implement the Waterable interface without modifying the existing code.	

- Create a WaterAction that extends the abstract Action class

Pros	Cons
The WaterAction class is only responsible for executing the harvest action, which means that it only has a single responsibility which adhered to the Single Responsibility Principle.	The WaterAction class has a tight coupling (high connascence) to the Waterable interface.
The WaterAction class can be reused whenever there is an object that can be harvested. This improves the reusability of the code, so this design is adhered to the DRY principle.	

REQ 4:

1. Implement an affection level and allow human NPCs to receive items.
   Design 1:
   Let the human actor implements Receivable interface to indicate that the actor can receive an item, add an affectionLevel field for all human npc.

Pros	Cons
It is relatively simple to implement and easy to understand. An actor can receive items by simply implementing the `Receivable` interface. Additionally, the affection level can be easily increased or decreased	Need to repeatedly add an affection level field for every human actor which violates DRY principle.
	Need to use instanceOf() to check if the actor is a Receivable which violates Open closed principle.

Design 2:
Create a Human abstract class that extends Actor, and let all human npc extend it. Create NPCUniqueAttribute and NPCUniqueAttributeEnum to handle new human attributes(in this case affection level).
NPCUniqueAttribute implements ActorAttribute.

Pros	Cons
By inheriting from the Human class, all human subclasses can reuse shared code, reducing code repetition and adhering to the DRY (Don't Repeat Yourself) principle.	Increases the number of classes in the code, which will decrease readability.
By creating the `NPCUniqueAttribute` and `NPCUniqueAttributeEnum` classes, unique attributes can be added to human NPCs through their actor attributes. Additional attributes for future NPCs can also be included in the enum, increasing extensibility.	
Check for the presence of the affection level attribute in actors to determine whether they can receive items. This approach avoids the need to use `instanceOf` to check an actor's	

capability to receive items. In the future, any actor that should be able to receive items only needs to include the affection level attribute.	

The final design chosen is Design 2, as it has lesser code duplication and higher extensibility compared to Design 1. Additionally, Design 2 is more flexible when it comes to adding new features or making modifications in the future, as its structure supports scalability and reuse.

2. Implement giveable items.

   Design 1:
   Create a new GivableItem abstract class, GivableItem extends Item. Let all givable item extends GivableItem to indicate the item is givable.

Pros	Cons
Reduce code duplication, every shared givable code will be implemented in the givable parent class.	Overly abstract hierarchies, such as `Flower` extending `GivableItem` which in turn extends `Item`, can lead to unnecessary complexity and tight coupling between classes.
	This design can lead to conflicts if an item needs to have multiple functionalities in the future. For example, if an item is both a weapon and givable, the implementation would fail because Java does not support multiple inheritance, the item cannot extend both `WeaponItem` and `GivableItem` at the same time.

   Design 2:
   Create Givable interface, let givable item implements Givable interface to indicate it is Givable.

Pros	Cons
By introducing the `Givable` interface, the system becomes more extensible. Any new item that can be given simply needs to implement this interface,	Creating a Givable interface will increase the number of classes in the system, which may decrease readability and maintainability.

without requiring changes to existing code. This adheres to the Open Closed Principle.	
The Givable interface defines a clear and concise contract for item that are givable and it does not force any unrelated method upon the implementing class which adhered to the Interface Segregation Principle.	
Creating a specific interface improves scalability and extensibility. If new givable items exists, they can introduce their own give effect.	

The final design chosen is Design 2 because Design 1 is overly abstract and may lead to conflicts. In contrast, Design 2 offers greater extensibility.

3. Implement action of giving giveable items to human npc.

Design 1:
Cramming all 'give item' logic into each givable item.

Pros	Cons
It is simple to implement and easy to understand, as the give logic only needs to be added inside each givable item without requiring the creation of additional classes.	This violates the Single Responsibility Principle, as cramming all logic into each givable item gives the class too many responsibilities and may lead to code that is harder to maintain, test, and extend in the future.
	High code repetition, because every giveable item needs to have the same code.

Design 2:
Create a GiveAction class that extends from the abstract Action class.

Pros	Cons
The GiveAction class is only responsible for executing the give action, which means that it only has a single responsibility which adhered to the Single Responsibility Principle.	The GiveAction class has a tight coupling (high connascence) to the Givable interface.

The GiveAction class can be reused whenever there is an object that can be given. This improves the reusability of the code, so this design is adhered to the DRY principle.	

The final design chosen is Design 2 because it offers higher maintainability and avoids code repetition. Unlike Design 1, which crams all "give item" logic into each givable item leading to duplicated code and violating the Single Responsibility Principle. Design 2 separates this logic into a `GiveAction` class. This approach adheres to both the Single Responsibility Principle and the DRY.

Appendix:

- Refactor XXXPuchaseAction class (XXX is a weapon), into only one PurchaseAction class
- Added a Purchasable interface instead of XXXPurchasable class.
- Modified the logic of each seller class based on the modification above