REQ 1:

1. Handling entities that should be treated as 'blessed' by grace.

   Design 1:
   Manually check the ground type using the built-in instanceof operator. If the ground is an instance of Inheritree, then it is considered blessed.

Pros	Cons
It is simple to implement and easy to understand as this approach does not require adding any extra status or structures, it just checks the ground type directly.	This design violates the Open Closed Principle. If a new ground type needs to be treated as 'blessed' in the future, the existing code must be modified to handle it.
	The classes that perform the checking become tightly coupled to the specific type.

   Design 2:
   Add status BLESSED as there might have more entities that are blessed by grace, for now, only Inheritree has this capability.

Pros	Cons
When there are other new grounds that are blessed by grace, we can simply add capability without modifying the existing class which adhered to the Open Closed Principle.	If different types of blessings needed to be handled in a unique way, a single status BLESSED might be too simplistic.
It encourages loose coupling and reduce connascence, because the system can check the capability of the entity instance instead of checking the specific class type (instanceof).	

   Design 2 is chosen because it adhered to the OCP which can add additional features without modifying the existing class and it also has higher maintainability than design

2. Implementing the logic and behavior for creatures that produce offspring by giving birth directly rather than laying eggs.

   A. Creating ProduceOffspringBehaviour and ProduceEggBehaviour classes.

Create concrete ProduceOffspringBehaviour and ProduceEggBehaviour classes that implement Behaviour interface.

Pros	Cons
The ProduceOffspringBehaviour class is only responsible for one responsibility which is to get the produce offspring action, while the ProduceEggBehaviour only responsible to get the produce egg action. This design adheres to Single Responsibility Principle.	We might need to explicitly assign behaviours when a certain condition is met to each individual creature which may be not efficient when creating many creatures.
If there are new creature that can produce offspring, we can reuse the ProduceOffspringBehaviour and ProduceEggBehaviour to avoid code repetition which adhered to Don't Repeat Yourself Principle	

B. Creating ProduceOffspringAction and ProduceEggAction classes.

Created concrete ProduceOffspringAction and ProduceEggAction classes that extends from the abstract Action class.

Pros	Cons
The ProduceOffspringAction class is only responsible for one responsibility which is executing the produce offspring action of a creature, while the ProduceEggAction is only responsible for executing the produce egg action of a creature, which adhered to the Single Responsibility Principle.	Different creatures might have different ways of producing offspring. For example, SpiritGoat and OmenSheep use different methods, so we need to create separate classes to handle each of them.
The ProduceOffspringAction and ProduceEggAction classes can be reused whenever a creature can produce offspring without creating multiple methods for each specific case. This improves the code reusability and is adhered to DRY principle.	

C. Checking the specific capabilities for game entities.

Created a concrete LocationUtils class.

Pros	Cons
The LocationUtils class can be reused whenever we need to check for specific capabilities in nearby game entities at the specific location without creating multiple methods for each case. This adhered to the DRY principle.	It might violate SRP because the utility class may take on too many unrelated functions.

3. Implementing the logic and behaviour for creatures that produce offspring by laying eggs.

A. Creating the OmenSheepEgg class.

Design 1: Created a concrete OmenSheepEgg class without defining a shared abstract Egg class.

Pros	Cons
The implementation is easy to understand and does not involve additional layers of abstraction.	As more egg types might be introduced in the future, similar logic will need to be duplicated across each egg class, which violates the DRY principle.
	Any change to shared egg functionality would require modifications across multiple classes, which makes this design less maintainable.

Design 2: Created concrete class OmenSheepEgg class to represent the egg that was laid by OmenSheep. It extends from the newly created abstract Egg class.

Pros	Cons
It promotes code reusability because there are many similarities	Introducing the abstract Egg class adds another layer of abstraction,

between the egg subclasses in which the same thing can be placed inside the abstract Egg class. This design is adhered to the DRY principle.	which can make the system more complex and harder to understand.
If more egg types are added, they can simply extend from abstract Egg class, making the system easier to extend and maintain when there exists a new feature without modifying the existing code, in which the design adhered to the Open Closed Principle.	

Design 2 was used because the design is easier to extend and maintain for future extension and make the system cleaner and easier to manage.

4. Implementing the hatching mechanism for eggs if the eggs are left on the ground.

   Created a HatchCondition interface, abstract BaseHatchCondition class that implements HatchCondition interface and TurnBasedHatchCondition class that extends from abstract BaseHatchConsition class to handle eggs that hatch by turns.

   - Created HatchCondition interface.

Pros	Cons
The HatchCondition interface focuses only on the hatching behavior and allows other classes to implement it without being forced to include irrelevant methods. This design is adhered to the Interface Segregation Principle.	The introduction of interfaces increases the complexity, which makes the code harder to understand
If there are any new hatch conditions in the future, they can implement this interface without changing the core logic. This adhered to the Open Closed Principle	

- Created abstract BaseHatchCondition class that implements HatchCondition interface.

Pros	Cons
The abstract BaseHatchCondition class contains shared logic for all hatch conditions, so this design is encouraging code reusability and reducing duplication. This adhered to the DRY principle	Since all hatch condition classes extend the abstract BaseHatchCondition class, this can lead to tight coupling relations.

- Created concrete TurnBasedHatchCondition class that extends from abstract BaseHatchConsition class to handle eggs that hatch by turns.

Pros	Cons
The TurnBasedHatchCondition class is only responsible for managing the turn based hatching condition, which is adhered to Single Responsibility Principle.	If the TurnBasedHatchCondition class is not frequently used in the future, implementing the whole class for this condition might be unnecessary.

5. Implementing the logic for consuming produced eggs if the eggs are collected by the player.

    - Created a Consumable interface that was implemented by the abstract Egg class.

Pros	Cons
The Consumable interface defines a clear and concise contract for objects that can be consumed and it does not force any irrelevant methods upon the implementing class which adhered to the Interface Segregation Principle.	It potentially violates the Liskov Substitution Principle if, in the future, an egg type is introduced that should not be consumable.
If a new consumable entity exists in the future, it can simply implement	

the Consumable interface without modifying existing code which adhered to the Open Closed Principle.	

- Created a ConsumeAction class that extends from the abstract Action class.

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

REQ 2:

1. Implementation of GoldenBeetle class to represent golden beetle
    A. Implementation of Golden Beetle being an actor in the game

Design 1

Create concrete class that represent golden beetle without extending any super classes

Pros	Cons
Simple to implement and understand, as all features are contained within a single class.	High code repetition, as many features are repeatedly implemented across similar creature classes
	Low maintainability, as changing a piece of shared logic(such as the consume logic) requires manually updating each consumable class individually.

Design 2

Create GoldenBeetle concrete class to represent Golden Beetle. GoldenBeetle class extends the abstract Creature class and implements Consumable interface.

Pros	Cons
Extending the `Creature` class enhances code reusability, as `GoldenBeetle` shares common features defined in the `Creature` class. This approach follows the DRY (Don't Repeat Yourself) principle.	When a concrete class extends an abstract class, it creates a tight coupling between them. If the abstract class changes, it might force changes in all its subclasses, making the code is potentially harder to maintain.
GoldenBeetle can be used in place of the abstract Creature class since they have the same attributes and methods of abstract Creature class. This design adhered to the Liskov Substitution Principle.	
By implementing the `Consumable` interface, it ensures that `GoldenBeetle` provides a concrete implementation of the consume() action, thereby increasing consistency and reliability in how consumable objects behave.	

The final design chosen is Design 2 because by extending the `Creature` class, it reduces repeated code and improves maintainability. Common features are placed inside the Creature class, so updates only need to be made in one place. This also makes it easier to add new creature types and keeps the code more organized.

B. Implementation of Golden Beetle following actors that are followable.

- Mark actors as followable

Design 1

Add Followable interface and let actor that are followable implement it

Pros	Cons
Easy way to mark actors as followable. To add new followable actors in the future, they only need to implement the `Followable` interface.	Need to use instanceOf() in GoldenBeetle class to check for surrounding Followable objects. Violates Open Closed Principle.

Design 2

Added ability FOLLOWABLE to represent actors that can be followed.

Pros	Cons
It is a simple way to mark actors as followable. Any followable actors added in the future will only need to include the followable ability, without modifying existing code, which adheres to the Open Closed Principle.	If there are multiple followable conditions that need to be handled differently in the future, a single FOLLOWABLE ability may be too simplistic.
It promotes loose coupling and reduces connascence by allowing the system to verify an actor's ability rather than relying on specific class types (e.g. using instanceOf). This helps decouple the game logic from individual actor classes.	

The final design chosen is Design 2, as it adheres to the Open-Closed Principle, allowing new features to be added without modifying existing code. It also has higher maintainability and readability compared to Design 1.

- Created a concrete FollowBehaviour class that implements the Behaviour interface.

Pros	Cons
Implementing Behaviour interface provides a clear contract of methods that the FollowBehaviour should implement.	Creating a FollowBehaviour class will increase the number of classes in the system, which may decrease readability and maintainability.
The `FollowBehaviour` class is focused solely on handling the follow behaviour of a creature, adhering to the Single Responsibility Principle.	
Can be reused when there are new creatures that can follow a followable actor which adhered to the DRY principle	

2. Implementation of Golden Egg

    - Implementation of Golden Egg being an item and consumable

    Design 1
    Create concrete classes that represent golden egg without extending any super classes

Pros	Cons
Easy implementation, enabling egg to implement only the functionality they require.	Much of the code is repeated because all eggs have many shared attributes and methods.

Design 2
Created concrete class GoldenBeetleEgg class to represent the egg that was laid by GoldenBeetle. It extends the abstract Egg class.

Pros	Cons
By extending the Egg class, the design encourages code reusability, as the egg subclasses share many common features. These shared features do not need to be repeated inside the GoldenBeetleEgg class, which adheres to the DRY (Don't Repeat Yourself) principle.	Creating a GoldenBeetleEgg class will increase the number of classes in the system, which may decrease readability and maintainability.
By extending the Egg class which	

implements Consumable, Golden Beetle Egg automatically becomes consumable and provides a clear contract of methods that the GoldenBeetleEgg should implement.	
The `GoldenBeetleEgg` class is solely responsible for managing the golden beetle's egg, separating its logic from other egg types. This design follows the Single Responsibility Principle.	

The final design chosen is design 2, because by extending Egg class, design 2 will not have much repeated code and have higher extensibility.

- Implementation of egg hatching when a nearby entity has a specific status (in this case CURSED) and also handles cases where the same egg can hatch into different creatures depending on the condition.

Design 1

The `tick()` method in GoldenBeetleEgg checks for nearby entities each turn, and triggers hatching if an entity with a specific status is detected.

Pros	Cons
Easy to implement, only need to add a conditional check for nearby entities in the egg's tick() method.	Violated OCP, for every hatch condition, needs to modify existing code to check for nearby entities.
	Violated DRY principle, since the conditional check for entity with specific status might be reused in the future,

Design 2

Added ProximityHatchCondition which extends BaseHatchCondition to handle egg hatch when there is an entity with certain status nearby.

Pros	Cons
The ProximityHatchCondition class is only responsible for managing the hatching condition when an entity with a certain status is nearby, which is adhered to Single Responsibility Principle.	Creating a separate class to handle the proximity hatch condition may lead to an excessive number of classes, which can make the system harder to maintain and decrease its readability.
By extending the abstract `BaseHatchCondition` class, this design promotes reusability by allowing common features to be defined once in `BaseHatchCondition`, avoiding the need to redefine them again.	
Handled scenarios where the same egg can hatch into different creatures under different conditions, by specifying the target creature for each condition (for example, new condition BLESSED in the future). This design adheres to the Open Closed Principle, avoiding adding a conditional statement in GoldenBeetleEgg class to determine what creature it will hatch into based on the condition.	

The final design chosen is design 2 because design 2 enables hatch conditions to be reusable and easier to extend if there are additional features.

REQ 3:

1. Implementing Conditional Monologue Behaviour for NPCs

Design A:

Created a **MonologueCondition** interface and multiple concrete condition classes like **LowRunesMonologueCondition**, **EmptyInventoryMonologueCondition**, **LowHpMonologueCondition**, and **ProximityMonologueCondition**. Each monologue line has a specific condition assigned to it, and only those classes that inherit the MonologueSpeaker abstract class whose conditions are met are able to be listened by the player

Pros	Cons
Very extensible as if a new type of monologue condition is to be added, a new condition class can be created. There is no need to change any existing code in any People This adheres to the **Open Closed Principle**.	Creating a separate class to handle the proximity hatch condition may lead to an excessive number of classes, which can make the system harder to maintain and decrease its readability.
Promotes code reuse across MonologueSpeakers that share condition logic. For example, different MonologueSpeakers can have their own monologues for when the player has low runes (low rune condition), following **DRY principle**.	It might violate the Dependency Inversion Principle as all MonologueSpeaker depend directly on concrete monologue condition classes. This design was chosen deliberately to balance **simplicity** and **readability** in a contained game world.
Supports loose coupling by abstracting the monologue logic into separate **MonologueCondition** classes. The classes that inherit from **MonologueSpeaker** are independent of the condition logic, so changes to a specific monologue condition do not impact these classes themselves. This separation ensures that each monologue speaker remains unaffected by modifications or additions in conditions.	

Design B:

Embed monologue logic directly within each individual class using if statements in a method such as **getMonologue()** that checks player state or surroundings.

Pros	Cons
Easier to implement and understand for small systems as all the monologue-related logic will be in one method for each class that has monologues	Duplicates logic across monologue speakers (e.g. repeated checks for low HP for each monologue speaker), which violates **DRY principle**.
	Any change to condition logic requires modifying methods in the monologue speakers themselves, which violates **Open Closed Principle**.

Comparison:

Design A was chosen because it is significantly more maintainable and scalable. As the number of monologue speakers and condition types increases, Design A minimizes code duplication and isolates condition logic, making the system easier to extend. For example, adding a new monologue speaker that reacts to player hunger would only require a class which may be called **LowHungerMonologueCondition**.

2. Monologues via Interfaces and Abstract Classes

Design A:

Created a **CanSpeakMonologue** interface and an abstract **MonologueSpeaker** class which handles shared functionality such as storing and evaluating monologues. NPCs like **Sellen**, **MerchantKale** and **Guts** extend from this class and implement a common method for speaking.

Pros	Cons
Promotes polymorphism as NPCs who can say monologues can be referenced via **CanSpeakMonologue**, which allows them to be listened to via	When all speaking NPCs extend from **MonologueSpeaker**, they become coupled to its internal logic, whereby if **MonologueSpeaker** has a bug, every subclass inherits that problem.

ListenAction, thus following the Liskov Substitution Principle. So all NPCs that can speak inherit from **MonologueSpeaker**, which implements the **CanSpeakMonologue** interface, thus allowing the **ListenAction** class to treat any speaking NPC as a **CanSpeakMonologue** object. This allows **ListenAction** to interact with different NPCs interchangeably.	
New NPCs that are able to be listened to and say monologues can reuse logic by inheriting **MonologueSpeake**r class which implements the **CanSpeakMonologue** interface and add their own monologue conditions.	

Design B:

Store all monologue lines as a list of strings and use switch-case to pick the correct one

Pros	Cons
Easily readable and understandable as understanding exactly which monologues can be said and when, for whichever NPC that can speak can be checked by just looking at that speaking NPC's class itself.	The speaking NPC class has multiple responsibilities which go against the **Single Responsibility Principle** as it stores the possible monologues, handles logic for conditional monologues and evaluates these conditions. Having all of these multiple tasks in a single class makes it difficult to extend.

Comparison:

Design A was selected for long-term flexibility and maintainability. Using a shared interface and abstract class ensures consistent behavior across all NPCs that are able to speak. If a future addition introduces a system which is required by all NPCs that can say monologues, then all NPCs using **MonologueSpeaker** can automatically support it without modification.

3. Creating Conditional Attack Behavior for NPCs that can attack (like Guts)

Design A:

Introduced an **AttackCondition** interface and concrete implementations such as **TargetAboveHpCondition** and **TargetHasCapabilityCondition**. These are passed into an **AttackBehaviour** class which encapsulates logic for deciding whether Guts should attack.

Pros	Cons
Easily extendable by adding new conditional classes if there are any new attack conditions that are to be added in the future without modifying existing code, adhering to the **Open Closed Principle**.	Introducing separate condition classes for different systems (e.g., MonologueCondition, AttackCondition) may lead to **redundancy and duplicated logic** when the same underlying condition (such as checking HP thresholds or capabilities) is needed in multiple contexts. This may result in **similar or identical condition implementations across multiple interfaces**, which can increase maintenance overhead and violate the **DRY (Don't Repeat Yourself)** principle if not carefully abstracted or reused.
Separating attack conditions from NPC logic follows the **Single Responsibility Principle**, as each class has one job whereby the AttackBehaviour decides only whether an attack is to be made and this behaviour itself does not overlap with others and is uniquely its own class.	
Encourages reuse of condition logic, eg. there may be another NPC that may attack when the HP of another actor is > 90, which only requires updating the **TargetAboveHpCondition** for that creature and redoing	

AttackBehaviour does not need to be done, thus following the **DRY principle**.	

Design B:

Directly implement the conditional attack logic in the **AttackBehaviour** class itself.

Pros	Cons
Appears to be centralised as all the logic for NPC attacking will be in one class itself.	Violates **Open Closed Principle** as any time any attack logic needs to be changed, the AttackBehaviour class itself must be edited.
	God class is encouraged as when this class grows to accommodate new edge cases, the class will include many if-conditions, making it hard to maintain.

Comparison:

Design A was chosen because it offers greater flexibility and extensibility. If future requirements include creatures that attack under certain other conditions, they can be done by creating new classes of such conditions that implement the **AttackCondition** interface.

REQ 4:

A. Implementing a flexible purchasing system for multiple weapons and merchants

Design 1: Use a single generic Purchasable interface implemented by all items/weapons and handle purchase logic for each merchant using if-else or instanceof inside the weapon subclasses.

Pros	Cons
Simple and easy to understand	Violates the Open Closed Principle: every time a new merchant or weapon is introduced, the existing item logic needs to be modified
Minimizes the number of interfaces and classes	Results in tight coupling and high connascence between weapons and merchants, reducing reusability
	Introduces downcasting and instanceOf, which violates Liskov Substitution and leads to poor design

Design 2: Introduce a separate XXXPurchasable interface (e.g. BroadswordPurchasable) for each purchasable item and a matching PurchaseXXXAction class.

Pros	Cons
Strict adherence to the Single Responsibility Principle: each interface governs only one item's behaviour	Requires creating a new interface and action class for each new weapon introduced, increasing class count
Follows Open Closed Principle: to introduce a new weapon or merchant, no existing classes need to be changed	Increases project complexity due to many specific interfaces and action classes
Enables delegation of item purchase logic to the appropriate merchant class, improving maintainability	

Design 2 was chosen because it adheres strictly to SOLID principles and offers strong extensibility. By introducing separate XXXPurchasable interfaces and PurchaseXXXAction classes per item, logic is cleanly separated and easily extended. Although this increases the number of classes, each class has a single responsibility and remains easy to maintain. New items or merchants can be added without modifying existing code, supporting scalability. The design avoids coupling item logic with merchant behaviour and eliminates branching conditions. This results in a predictable, maintainable structure that handles unique pricing and effects per merchant without requiring instances of downcasting.

B. Refactoring WeaponItem into an abstract class

The WeaponItem class was refactored into an abstract class to allow shared behaviour and structure across all weapons while enabling the introduction of unique concrete weapon types such as Broadsword, Katana, and DragonslayerGreatsword. Each weapon defines its specific damage, hit rate, and on purchase effect via constructor.

Design 1: Keep WeaponItem as a concrete class and create each weapon (e.g. Broadsword) by configuring it through the WeaponItem class only.

Pros	Cons
Simple implementation with fewer classes and faster initial development.	Cannot easily customize unique features for each weapon without adding conditionals inside a shared class.
Reduces boilerplate class declarations.	Difficult to override weapon purchase effects cleanly (with separation of concerns)
	Violates the Single Responsibility Principle as one class handles all weapon logic.

Design 2: Make WeaponItem abstract and let each weapon subclass define its own properties and associated purchase effect.

Pros	Cons
Encourages code reuse and consistent structure via inheritance which adhered to DRY principle.	Requires more boilerplate for new weapon creation.
Adheres to Single Responsibility Principle as each weapon encapsulates its own logic.	
High extensibility as new weapons can be added easily by extending WeaponItem without modifying existing code, which adhered to Open Closed Principle.	

Design 2 was chosen because it offers better separation of concerns and future maintainability. It allows each weapon to cleanly define its behaviour while sharing common functionality through the abstract WeaponItem superclass. This aligns with the requirement to support multiple merchants selling the same weapon with different effects.

C. Applying weapon purchase effects using PurchaseWeaponEffect interface with its classes.

A new system is introduced to handle the unique effects that occur when a player purchases a weapon. A new interface called PurchaseWeaponEffect was created, along with concrete implementations such as AttributesModifierEffect, MaximumAttributesModifierEffect, and SpawnCreatureEffect. These were used to encapsulate the logic for what happens when a weapon is bought, such as modifying attributes or spawning a creature near the player.

Design 1: Hardcode the weapon purchase effects inside each merchant class, which is Sellen and MerchantKale.

Pros	Cons
Easy to implement in the short term when there are not as many merchants or weapons in the game.	Violates the Open Closed Principle because modifying weapon purchase effects requires modifying the merchant classes directly.
Centralizes weapon purchase logics within the merchant classes (they know what they are selling and how to sell weapons).	Leads to logic duplication if multiple merchants sell the same weapon but with different effects.
	Tight coupling between merchant classes and weapons' purchase logic.

Design 2: Create the PurchaseWeaponEffect interface and assign an appropriate effect to each weapon. Let the merchant call getPurchaseWeaponEffect().apply(...) during the purchase process.

Pros	Cons
Fully adheres to the Open Closed Principle, allowing new purchase effects to be added without modifying existing merchant class codes.	Increases the number of classes (those stored in the game.weapons.effects), which may increase project complexity.
Supports loose coupling between merchants and weapon purchase effects logic.	Requires slightly more setup(effects storage) for each new weapon.
Highly extensible and maintainable because new merchants or weapons can be added by defining or reusing existing effect classes.	

Design 2 was chosen because it strictly follows SOLID principles, especially the Open Closed Principle, by allowing new weapon purchase behaviours (effects) to be introduced without modifying the existing logic of merchant class' methods. Instead of implementing and storing custom purchase logic within merchant classes, each weapon encapsulates its own purchase effect via the PurchaseWeaponEffect interface and its implementations. This separates the "how are weapons being sold" from "who sells it," enabling merchants to remain generic and unaware of specific weapon effects.

This decoupling allows for better scalability, in which new merchants selling the same weapon with different effects can be supported simply by assigning different effect objects to the weapon instance. Likewise, new weapons can be introduced with new effects without impacting existing merchant logic. This modular design is especially beneficial when expanding game features in the future.

Although this approach increases the number of small classes (which is one effect class per new weapon purchase behaviour), each class serves a clear purpose, improving readability and maintainability. The use of polymorphism and interface driven design ensures that logic is organized around responsibilities, not actors, and each class can evolve independently. This ultimately reduces the risk of unclear game features' structure and will improve code readability among the project team in implementing future requirements.

Appendix:

Changes made for Assignment 1: Remove RotResistantCreature and NonRotResistantCreature classes, add the counter attribute to creatures that are not immune to rot manually.