MONASH University

FIT2099 - Object-oriented design and implementation

Assignment 2:
Design Rationale

Team Members:
Desmond Chong Qi Xiang(33338248)
Maliha Tariq (33473692)
Ong Chong How (33363102)
Yoong Qian Xin (32846517)

MONASH University

REQ1: The Ancient Woods

*In the UML Diagram, all the classes in green are the new classes we added ourselves in Assignment 2 for aligning the specified requirements.

ForestEnemySpawnableGround

The ForestEnemySpawnableGround is designed as an abstract class which extends the EnemySpawnableGround abstract class. It is used to represent ground terrain such as Hut and Bushes that can spawn enemies.

Abstraction:

Abstraction provides a level of indirection, allowing the code to be more flexible and adaptable to different environments and enemy types.

Polymorphism:

Polymorphism simplifies the code by allowing different spawnable grounds to exhibit behaviour specific to their types, improving maintainability and extensibility.

SOLID Principles:

- **Open-Close Principle (OCP):**

The design is open for extension, allowing the addition of new ground types which can spawn enemies in forest by creating new concrete classes which extend ForestEnemySpawnableGround. The generic ForestEnemySpawnableGround abstract class is closed for modification.

Pros:	Cons:
Code Reusability: Two of the above-mentioned ground classes will inherit the EnemySpawnableGround abstract class because they share common attributes. This promotes code reusability and reduces code duplication (DRY), making this design easier to maintain and more efficient.	**Complexity:** It does introduce a level of complexity. If the hierarchy of classes becomes extensive, it might be challenging to understand and maintain the code.

Future Extension:

This design allows for easy extension without modifying existing code. If a new ground type needs to spawn enemies in the future, we can create a new class that extends EnemySpawnableGround without modifying the abstract class itself.

FollowBehaviour

FollowBehaviour is a concrete class which implements the Behaviour interface, as Behaviour interfaces have the basic methods needed based on the specific behaviour needs.

SOLID Principles:

- **Open-Close Principle (OCP):**

The design is open for extension, allowing the addition of new behaviour types which can be done by the enemy by creating new concrete classes which implements Behaviour. The Behaviour interface is closed for modification.

Pros:	Cons:
Code Reusability: FollowBehaviour classes will implement the Behaviour interface because they share common methods. This promotes code reusability and reduces code duplication (DRY), making this design easier to maintain and more efficient.	**Complexity:** It does introduce a level of complexity. If the hierarchy of classes becomes extensive, it might be challenging to understand and maintain the code.

Future Extension:

This design allows for easy extension without modifying existing code.

REQ2: Deeper into the Woods

Runes

The Runes class has been meticulously designed to stand out as a distinctive item within a text-based game. It builds upon the foundation of the base item class while introducing specialised functionalities for interacting with other entities in the game world, especially in the context of transactions and acquiring them by defeating enemies. This explanation will shed light on the key decisions taken during the Runes class development, highlighting how it adheres to the principles of Object-Oriented Design (OOD).

Polymorphism:

One of the core tenets of Object-Oriented Design (OOD) has been thoughtfully integrated into the Runes class. This becomes apparent through the utilization of the Consumable interface, which demonstrates a high degree of flexibility in the implementation of the consumeBy and playTurn methods, enabling actors to effectively consume runes. Additionally, the enemy unconscious method exemplifies the concept of polymorphism, further showcasing the adaptability of the Runes class. This adaptability fosters the creation of distinct attributes for various item types, thereby enhancing the complexity and richness of interactions within the game.

Abstraction:

The abstraction of Runes is achieved through the interface Consumable and ConsumeAction in allowableActions method. This provides a high-level representation of the item's functionality to the actors in the game without needing to concern about the details of item behaviour.

Don’t Repeat yourself Principle(DRP):

The Don't Repeat Yourself Principle (DRP) has been effectively applied in this scenario. To ensure that all enemies in the game drop runes when defeated by other actors, a new attribute called "runesNumDropped" has been introduced for the Enemy class. By overriding the unconscious method and having it drop a Runes object based on the "runesNumDropped" attribute, the need for each individual enemy to implement this behavior has been eliminated. This approach minimizes code duplication and promotes a more efficient and maintainable design.

SOLID Principles:

- **Single Responsibility Principle (SRP):**

The Runes class rigorously follows the Single Responsibility Principle (SRP) by maintaining a singular, well-defined purpose: serving as the in-game currency, facilitating actor consumption, and effectively managing the balance when actors use it.

- **Open-Closed Principle (OCP):**

The class complies with the OCP by welcoming extensions without necessitating modifications to existing code. This way of design can allow a varying number of runes dropped by after different enemies died. Therefore, whenever an enemy is created, the enemy can have a different number of runes without modifying the runes class to check which enemy and it can consumed by actor who defeated the enemy.

- **Liskov Substitution Principle (LSP):**

The Runes class, which inherits from the Item class, upholds the principles of the Liskov Substitution Principle (LSP). It expands upon the capabilities of the Item class while retaining its core compatibility, ensuring that it can be both picked up and dropped by actors without any issues.

- **Dependency Inversion Principle (DIP):**

The Runes class strictly adheres to the Dependency Inversion Principle (DIP) by implementing the Consumable interface. This design choice allows actors to consume runes based on higher-level code without the need to explicitly check whether the item is a Rune and then take the appropriate action. Instead, the DIP-compliant design ensures that actors can interact with runes through a consistent interface, promoting a more flexible and modular system.

Pros:	Cons:
Avoid Stronger dependency: In our design, we opted for an int attribute named "runesNumDropped" within the Enemy class instead of using a direct association with a Runes object. This choice was made to minimise the level of coupling between the Runes and Enemy classes. We create a Runes object dynamically within the unconscious method, which introduces a form of dependency between the two classes. However, it's important to note that this dependency is relatively weaker than a direct association, making our design more flexible and maintainable. **Modularity:** The Runes class encapsulates the processes related to updating the balance after its consumption, which simplifies any future modifications or expansions aimed at adjusting the actor's wallet balance. **Maintainability:** By adhering to SOLID principles, this class ensures a high level of maintainability, reducing the likelihood of errors as the codebase evolves relevant to Runes functionality over time.	**Potential of connascence of position:** The inclusion of the "runesNumDropped" integer attribute could potentially introduce a connascence of position issue, as developers using this class might inadvertently misplace the value of this attribute with the "hitpoints" integer attribute

Future Extension:

Different currency:

The design of the Runes class has been structured in a way that allows developers to easily extend it when introducing different currencies in various game maps. This approach ensures the continued functionality of the basic Runes while accommodating the incorporation of new currency systems as needed.

Puddle

The Puddle class has been thoughtfully crafted to serve as a unique terrain feature within a text-based game. It extends the capabilities of the base ground class by introducing specialised functionalities for interactions with other entities in the game world, particularly in terms of allowing actors to consume it and providing healing and stamina restoration. This explanation will provide insight into the pivotal design choices made during the development of the Puddle class, emphasising how it aligns with the principles of Object-Oriented Design (OOD).

Polymorphism:

One of the fundamental principles of Object-Oriented Design (OOD) has been carefully incorporated into the Puddle class. This is evident in the way it utilizes the Consumable interface and extends the base ground class, which showcases a significant level of flexibility in how the consumeBy and allowableActions methods are implemented. This design choice allows the Puddle to be consumed by an Actor when the actor steps on it, highlighting the adaptability and modularity of the class.

Abstraction:

The abstraction of the Puddle class is achieved through a combination of mechanisms, including the utilisation of the Consumable interface, the extension of the abstract ground class, and the implementation of the ConsumeAction within the allowableActions method. This approach offers a high-level representation of the puddle's functionality to the actors in the game, freeing them from the need to delve into the intricacies of puddle behaviour and allowing for a more abstract and simplified interaction with the puddle.

Don’t Repeat yourself Principle(DRP):

The Don't Repeat Yourself Principle (DRP) has been successfully employed in this situation. Instead of creating a separate class exclusively for the Puddle, which is a unique ground type that can be consumed by actors in the game, we chose to have the Puddle class implement the Consumable interface. This approach allowed us to preserve the core functionality of both the basic ground and basic consumable entity by overriding the consumedBy and allowableActions methods, all without the need to duplicate code from the Ground class and ConsumeAction. This design promotes code reuse and maintains a more efficient and maintainable codebase.

SOLID Principles:

- **Single Responsibility Principle (SRP):**

The Puddle class strictly adheres to the Single Responsibility Principle (SRP) by having a clear and singular purpose: it serves as an in-game ground element, enables actor consumption, and efficiently manages the updating of actor stamina and health. This focused design approach ensures that the class has a well-defined role without taking on unnecessary responsibilities.

- **Open-Closed Principle (OCP):**

The Puddle class's adherence to the Open-Closed Principle (OCP) is evident in its extension of the Ground class and the addition of new functionality, allowing it to be consumed by actors and update the status of actors without the need to modify the existing Ground class. This design approach aligns perfectly with the OCP principle, as it follows the principle of "open for extension, but closed for modification," promoting the modularity and extensibility of the codebase.

- **Liskov Substitution Principle (LSP):**

The Puddle class, inheriting from the Ground class, maintains compliance with the Liskov Substitution Principle (LSP). It enhances the functionalities of the Ground class while preserving its fundamental compatibility, guaranteeing that actors can step on it seamlessly without any compatibility issues.

- **Dependency Inversion Principle (DIP):**

The Puddle class rigorously follows the Dependency Inversion Principle (DIP) by incorporating the Consumable interface and extending the Ground class. This design decision empowers actors to interact with and consume puddles based on higher-level code without the necessity of explicitly checking whether the ground is a Puddle object and then determining the appropriate action. Instead, this DIP-compliant design establishes a consistent interface, facilitating actors' interactions with puddles and promoting a more adaptable and modular system.

Pros:	Cons:
Modularity: The Puddle class encapsulates the processes of consume and healing, stamina restoration for actors, simplifying modifications and expansions of the game's trade system. **Flexibility of modification:** The incorporation of polymorphism through the Ground parent class and the Consumable interface enhances the flexibility of the Puddle class. This flexibility allows for the modification of ground or consumption functionality within the Puddle class without impacting the core implementations in the Ground parent class or the ConsumeAction, underscoring the versatility and adaptability of the design. **Maintainability:** By adhering to SOLID principles, the Puddle class ensures that developers will find it easier to maintain and troubleshoot any errors or issues that may arise in the Puddle functionality. This adherence to SOLID principles promotes a more robust and manageable codebase, making it more straightforward to identify and address problems when they occur.	**Flexibility of effect:** The current implementation of the Puddle class has limited flexibility, as it only provides a fixed healing of 1 point and restores 1% of an actor's stamina. This uniformity makes it challenging to differentiate the effects of various puddle objects in the game. It might be beneficial to explore variations where some puddles could have different effects, such as increasing enemy damage or offering more significant healing to actors.

Future Extension:

New functionality:

The Puddle class can be extended to introduce new functionality, like a "PuddleEnemy" variant, where the puddle might have the potential to drown an actor, offering diverse and dynamic gameplay experiences.

BloodBerry

The BloodBerry class has been carefully crafted to serve as a unique item within a text-based game. It extends the capabilities of the base item class by introducing specialized functionalities for interacting with other entities in the game world. These functionalities include the ability to be dropped, picked up, and consumed by actors, as well as the power to increase the maximum health of the actor. This explanation will provide insights into the key decisions made during the development of the BloodBerry class, emphasizing how it aligns with the principles of Object-Oriented Design (OOD).

Polymorphism:

The BloodBerry class seamlessly incorporates a fundamental principle of Object-Oriented Design (OOD). This is evident in its use of the Consumable interface and extension of the Item class, which showcases a remarkable degree of flexibility in implementing the consumeBy and allowableActions methods. This flexibility empowers actors to efficiently consume BloodBerry items. Furthermore, the extension of the Item class allows for the application of polymorphism, granting the ability to pick up and drop all child classes of items, thereby enhancing the versatility of the design.

Abstraction:

The abstraction of the BloodBerry class is successfully achieved through the utilization of the Consumable interface and the implementation of ConsumeAction within the allowableActions method. Furthermore, the extension of the Item class enables actors to perform pickup and drop actions without having to worry about the intricacies of those actions. This design approach offers actors a high-level representation of the item's functionality in the game, freeing them from the need to delve into the specifics of item behavior and enhancing overall usability and clarity.

Don’t Repeat yourself Principle(DRP):

The Don't Repeat Yourself Principle (DRP) has been effectively applied in this scenario. Rather than duplicating code to enable BloodBerry to be consumed, picked up, and dropped by actors, we have leveraged inheritance from the Ground parent class for pickup and drop functionality. Additionally, we've implemented the Consumable interface and ConsumeAction to handle the consumption functionality. This approach minimizes code duplication and promotes a more efficient and maintainable codebase.

SOLID Principles:

- **Single Responsibility Principle (SRP):**

The BloodBerry class strictly adheres to the Single Responsibility Principle (SRP) by having a clear and singular purpose: it serves as an in-game item element, enables actor consumption, and efficiently manages the updating of actor maximum health. This focused design approach ensures that the class has a well-defined role without taking on unnecessary responsibilities.

- **Open-Closed Principle (OCP):**

The BloodBerry class demonstrates a strong adherence to the Open-Closed Principle (OCP). This is evident in its extension of the Item class and the introduction of new functionality, allowing it to be consumed by actors and update their maximum health, all without the necessity of modifying the existing Item class. Additionally, the implementation of the Consumable interface and the overriding of the consumedBy method do not require any changes to the existing code in the ConsumeAction class. This design approach perfectly aligns with the OCP principle, which emphasizes being "open for extension but closed for modification," thereby enhancing the modularity and extensibility of the codebase.

- **Liskov Substitution Principle (LSP):**

The BloodBerry class, as an inheritor of the Item class, upholds the principles of the Liskov Substitution Principle (LSP). It expands upon the capabilities of the Item class while retaining its core compatibility, ensuring that actors can pick up or drop it without encountering any compatibility issues. This inheritance maintains a consistent and reliable interface for interacting with items in the game.

- **Dependency Inversion Principle (DIP):**

The BloodBerry class perfectly aligns with the Dependency Inversion Principle (DIP) by incorporating the Consumable interface and extending the Item class. This design approach allows actors to seamlessly interact with and consume items using higher-level code, without the necessity to explicitly verify whether the item is a BloodBerry object and then determine the appropriate actions. Instead, this DIP-compliant design establishes a standardized interface, streamlining actors' interactions with BloodBerry items and ultimately boosting the system's flexibility and modularity.

Pros:	Cons:
Modularity: The "BloodBerry" class encapsulates the processes of consumption and health increase, providing a well-organized structure that simplifies any future modifications or expansions related to altering the attributes or behavior of BloodBerry items. **Ease of Maintenance:** By adhering to SOLID principles and separating concerns, the "BloodBerry" class becomes easier to maintain and troubleshoot. Developers can work on enhancements or bug fixes within the class without causing unintended side effects in other parts of the game.	**Predictability:** The fixed health increase of 5 points for all BloodBerries may make the game somewhat predictable. Players might find it less exciting if all BloodBerries offer the same benefit. Introducing some randomness or diversity in the health bonus from BloodBerries could add an element of surprise to the game. **Limited Interactivity:** The "BloodBerry" class, in its current form, doesn't offer much interactivity beyond being consumed. Adding additional features or interactions related to BloodBerries, such as the ability to combine them with other items or use them in specific quests, could enhance player engagement and gameplay depth.

Future Extension:

New Functionality:

The "BloodBerry" class can be extended to introduce different varieties of BloodBerries, each with unique effects. This addition would offer diverse and dynamic gameplay experiences, making the game more engaging and strategic.

REQ3: The Isolated Traveller

Traveller

The Traveller class has been crafted to serve as a distinct character within a text-based game, possessing a unique role and capabilities. It extends the base Actor class while introducing specialised features for engaging with other in-game entities, particularly concerning transactions involving the exchange of items. This design rationale will elucidate the pivotal choices made during the development of the Traveller class, emphasising its alignment with principles of Object-Oriented Design (OOD).

Polymorphism:

A cornerstone of OOD, is effectively employed in the Traveller class. This is manifested through method overriding, where the playTurn method is customised to define a specific behaviour for the Traveller. This flexibility enables the crafting of distinct behaviours for various actor types, enhancing the richness of the game's interactions.

Abstraction:

It is achieved by conceiving the Traveller class as an abstract representation of a character within the game's universe. This abstraction shields the intricate details of how the Traveller interacts with other actors, fostering a higher degree of encapsulation and promoting modularity within the codebase.

SOLID Principles:

- **Single Responsibility Principle (SRP):**

The Traveller class meticulously adheres to the SRP by having a singular purpose: portraying the character of the traveller and managing the intricacies of buying and selling items. It abstains from entangling unrelated tasks.

- **Open-Closed Principle (OCP):**

The class complies with the OCP by welcoming extensions without necessitating modifications to existing code. Introducing new items available for sale can be seamlessly achieved through the creation of new PurchaseAction objects and updates to the allowableActions method, all without altering the existing codebase.

- **Liskov Substitution Principle (LSP):**

The Traveller class, as a subclass of the Actor class, maintains adherence to the LSP. It extends the functionalities of the Actor class while preserving its inherent behaviour and compatibility.

· **Interface and Dependency Inversion:**

The utilisation of the Ability.BUYING capability serves as a quasi-interface mechanism, determining the capacity of an actor to engage in purchasing from the Traveller. This fosters loose coupling and conforms to the Dependency Inversion Principle (DIP) by depending on abstractions rather than specific implementations.

Pros: **Modularity:** The Traveller class encapsulates the processes of buying and selling, simplifying modifications and expansions of the game's trade system. **Flexibility:** Polymorphism and the application of capabilities empower diverse interactions with distinct actor types, enhancing gameplay dynamics. **Maintainability:** By conforming to SOLID principles, the class ensures a high level of maintainability, minimising susceptibility to errors during the codebase's evolution.	**Cons:** **Limited Gameplay Complexity:** The current behaviour of the Traveller is relatively basic, lacking intricate gameplay elements. Future enhancements might introduce dynamic pricing, negotiation mechanics, or additional interaction possibilities to elevate encounters with the Traveller. **Static Pricing:** Item prices are hard-coded, potentially limiting their suitability for intricate in-game economies. Future iterations could implement more advanced pricing mechanisms.

Future Extension:

Dynamic Pricing: Develop a pricing strategy that adapts item costs based on factors such as supply and demand, player reputation, or in-game events.

Expanded Inventory: Enable the Traveller to maintain a dynamic inventory, introducing variety and unpredictability to player interactions.

Diverse Interactions: Create new types of interactions with the Traveller, such as quests, negotiation scenarios, or random events, to infuse encounters with dynamism.

Personality Traits: Introduce personality traits for the Traveller, influencing their behaviour and interactions with players, thus enhancing immersion and engagement in the game world.

PurchaseAction and SellAction

The PurchaseAction and SellAction classes have been crafted to orchestrate actions pertaining to the acquisition and divestiture of items in a text-based game. These actions interface with items that adhere to the Purchasable and Sellable interfaces, respectively. This design rationale elucidates the pivotal design choices made in the development of these classes, underscoring their alignment with the principles of Object-Oriented Design (OOD).

Polymorphism:

A pivotal tenet of OOD, is astutely harnessed in both the PurchaseAction and SellAction classes. These actions are rendered generic, capable of interacting with any item implementing the Purchasable or Sellable interface. By operating at the interface level rather than binding to concrete item classes, these actions attain versatility and reusability.

Abstraction:

Abstraction is achieved by the judicious application of interfaces (Purchasable and Sellable) that proffer a shared contract for items. These interfaces abstract away the particulars of purchasing and selling, affording the capacity to integrate a multitude of item types into the game sans the necessity for substantial code modifications.

SOLID Principles:

- **Single Responsibility Principle (SRP):**

Both PurchaseAction and SellAction are stalwart adherents of SRP, as they diligently concentrate on a solitary responsibility: executing the procurement or disposal of an item. Extraneous concerns remain eschewed, ensuring the codebase's modularity and perspicuity.

- **Open-Closed Principle (OCP):**

These actions exemplify the OCP by being open to expansion whilst steadfastly closed to alteration. They can be seamlessly extended to accommodate fresh item categories that adhere to the Purchasable or Sellable interface, sans necessitating revisions to extant code.

- **Liskov Substitution Principle (LSP):**

PurchaseAction and SellAction dutifully abide by the LSP, interacting harmoniously with any item in compliance with the corresponding interface. They steadfastly uphold anticipated behaviour, irrespective of the concrete item's type.

- **Interfaces and Dependency Inversion:**

The interfaces (Purchasable and Sellable) manifest the essence of the Dependency Inversion Principle (DIP). They assume the role of abstractions upon which the PurchaseAction and SellAction classes rest their dependence. This decoupling fosters a state of nimbleness in the codebase, conducive to both flexibility and maintainability.

Pros:	**Cons:**
Modularity and Reusability: These actions encapsulate the facets of procurement and divestment, endowing them with a quality of reusability across diverse items in the game. **Interface-Driven Design:** By tethering themselves to interfaces, the codebase becomes well-suited to the assimilation of new item categories, thus facilitating the incorporation of additional items into the game with relative ease. **Exception Handling:** The judicious use of exception handling confers a capacity for graceful error management when exceptional scenarios, such as paucity of funds or atypical pricing, come into play.	**Limited Behaviour:** While these actions serve as a generic scaffold for the execution of procurements and disposals, they may not encompass the full spectrum of idiosyncratic item behaviours. Adapting them for highly specialised items may necessitate supplementary code.

Future Extension:

Item-Specific Actions: To accommodate items endowed with distinctive behaviors, contemplation should be given to the creation of dedicated procurement and disposal actions that supersede the generic actions when warranted.

Transaction Logging: The implementation of a transaction logging system can be contemplated, preserving a comprehensive record of all acquisitions and disposals effected by the player. Such a record could furnish invaluable gameplay statistics or serve as a diagnostic resource.

Pricing Strategies: The introduction of more intricate pricing strategies for items, such as the dynamic modulation of prices contingent on in-game occurrences or the law of supply and demand, could augment gameplay depth and complexity.

Purchasable and Sellable

The Purchasable and Sellable interfaces have been conceived with the primary goal of encapsulating the essential characteristics and functionalities of items that can be acquired and disposed of within the game's context. These interfaces serve as a blueprint for defining the behaviour of such items, ensuring a standardised and cohesive approach to handling item transactions.

Abstraction:

Abstraction, a cornerstone of object-oriented design, is meticulously applied in these interfaces. They abstract the intricate details associated with the processes of buying and selling items, providing a structured and high-level representation that items within the game must conform to.

- **Single Responsibility Principle (SRP):**

Both interfaces adhere to the Single Responsibility Principle, affirming that each interface has a singular and well-defined role:

- The Purchasable interface focuses on determining the methodology by which an item can be procured by an actor and computing the purchase price.
- The Sellable interface centres on ascertaining how an item can be divested of by an actor and calculating the selling price.

- **Open-Closed Principle (OCP):**

These interfaces align seamlessly with the Open-Closed Principle, signifying their openness to extension while remaining impervious to modification. This means that any new item introduced into the game can seamlessly implement these interfaces, integrating into the existing framework without necessitating changes to preexisting code.

- **Interfaces and Dependency Inversion:**

The adoption of interfaces in these constructs embodies the principles of the Dependency Inversion Principle (DIP). Game actors, whether it be the player or non-player characters, depend on abstractions, in this case, the interfaces, rather than specific implementations. This abstraction fosters adaptability and facilitates the incorporation of diverse item types into the game, enhancing overall flexibility.

Pros:	Cons:
Modularity: These interfaces champion a modular design paradigm, permitting the inclusion of items into the game's ecosystem with minimal disruption to existing code. **Standardisation:** By adhering to these interfaces, items across the game adhere to a consistent contract, promoting clarity and minimising discrepancies in item-related interactions. **Flexibility:** The utilisation of interfaces accommodates a wide spectrum of item-specific behaviours, making room for diverse items characterised by distinct purchasing and selling rules.	**Limited Behavior Specification:** While these interfaces provide a foundational structure for item behaviour, they may not encompass every conceivable item interaction or behaviour. More intricate and item-specific behaviours may necessitate the incorporation of additional methods or interfaces.

Future Extension:

Item-Specific Behaviours: To cater to items featuring idiosyncratic behaviours, the introduction of supplementary methods or interfaces can be contemplated to handle specialised purchasing and selling logic.

BloodBerry, HealingVial, RefreshingFlask and Broadsword

The BloodBerry, HealingVial, RefreshingFlask and Broadsword class represents an item in the game that is both purchasable and sellable. This design rationale elucidates the key design decisions made for the BloodBerry, HealingVial, RefreshingFlask and Broadsword class, considering object-oriented design principles, item behaviour, pricing strategies, and its role in the game's economy.

Abstraction:

Abstraction is a fundamental principle applied in BloodBerry, HealingVial, RefreshingFlask and Broadsword class, encapsulating the intricate details of item behaviour and economic interactions. It provides a high-level representation of the item's functionality.

Use of Interfaces:

Those classes implements both the Purchasable and Sellable interfaces, which adhere to the principles of interface-based programming:

· Purchasable Interface: The BloodBerry, HealingVial, RefreshingFlask and Broadsword class implements this interface to define the logic for purchasing the item by an actor, considering pricing variations and potential discounts.

· Sellable Interface: By implementing the Sellable interface, those classes enable itself to be sold to other actors within the game, defining its selling price and potential bonuses.

· **Single Responsibility Principle (SRP):**

Those classes adhere to the SRP, encapsulating item behaviour and pricing strategies within separate methods:

· consumedBy handles item consumption, increasing the actor's health, and removing the item from the actor's inventory.

· allowableActions methods determine the actions that actors can perform with the item, ensuring separation of concerns.

· purchasedBy and soldBy methods handle buying and selling interactions, respectively, adhering to their distinct responsibilities.

· **Open-Closed Principle (OCP):**

Those classes follow the OCP, allowing for extensions without modifying existing code. Additional items with unique purchasing and selling behaviours can be introduced seamlessly by extending this class.

Pros:	Cons:
Modularity: Those item classes embodies a modular design, supporting easy integration of new items with distinct economic behaviours.	**Complexity:** Those item classes introduce complexity due to the management of pricing variations, discounts, and bonuses, which may require additional testing and debugging.
Player Health: The class enhances player health and gameplay dynamics by providing items that can be consumed.	**Balancing Challenges:** Pricing variations and bonuses need to be carefully balanced to maintain game fairness.
Economic Transactions: The class supports buying and selling interactions, enriching in-game economics and player choices.	

Future Extension:

Diverse Items: Expand the game's inventory by introducing more items with unique purchasing, selling, and consumption behaviours, promoting diverse gameplay experiences.

Item Interactions: Implement interactions between items and other game elements, such as the influence of items on quest outcomes or character relationships.

REQ4: The room at the end of the forest

GreatKnife

This class extends WeaponItem with additional features. It sets up the basic properties of a 'Great Knife'. It has an allowableActions method to provide the actions that the item allows its owner to perform on another actor. The class also includes the logic for purchasing (purchasedBy) and selling (soldBy) the item. The special skill activation logic is included in the activateSkill method.

Polymorphism
Polymorphism is achieved through method overriding and interfaces. For example, the GreatKnife class implements the Sellable, Purchasable, and ActiveSkill interfaces. By doing so, it provides specific implementations of methods defined in these interfaces. This allows objects of the GreatKnife class to be treated as instances of these interfaces, enabling flexibility in how they can be used.

Abstraction
Abstraction is achieved by defining interfaces (Sellable, Purchasable, ActiveSkill) that provide a contract for the methods that implementing classes must implement.

SOLID Principles

Single Responsibility Principle (SRP): The GreatKnife class follows the SRP by focusing on the responsibilities related to representing a weapon with special skills. It handles actions, pricing, and skill activation, which are all related to its primary purpose.

Open-Closed Principle (OCP): The code shows openness to extension, as we have the interfaces (Sellable, Purchasable, ActiveSkill) that can be implemented by other classes to extend the behaviour of weapons.

Interface and Dependency Inversion: The class effectively uses interfaces to define contracts (e.g., Sellable, Purchasable, ActiveSkill), allowing high-level modules to depend on these abstractions rather than specific implementations. This promotes the Dependency Inversion Principle.

Pros:	Cons:
Flexibility: The use of interfaces allows for flexibility in how classes can be used and extended. This is valuable for creating a variety of weapons with different behaviours. **Maintainability**: The code is structured with a focus on encapsulation and clear separation of concerns, which can enhance maintainability. **Modularity**: The code is organized into packages and classes, promoting modularity and making it easier to manage and extend.	**Complexity:** The GreatKnife class introduces a level of complexity to the game due to its implementation of specialized actions and skills. This complexity can make the codebase harder to maintain and extend in the long run.

Future Extension

Adding More Weapons: We can create new weapon classes that implement the Sellable, Purchasable, and ActiveSkill interfaces to introduce a variety of weapons with unique skills and behaviours.

GreatHammer

This class is quite similar to GreatKnife but differs in terms of damage and special skills. It also has an allowableActions method and defines how it's sold (soldBy). The special skill activation logic is in the activateSkill method, and it's different from the GreatKnife.

Polymorphism
Polymorphism is achieved in the GiantHammer class by implementing the Sellable and ActiveSkill interfaces. It provides specific implementations of the methods defined in these interfaces (allowableActions, soldBy, activateSkill, staminaConsumedByActivateSkill, skillAction). This allows objects of the GiantHammer class to be treated as instances of these interfaces, promoting flexibility and extensibility.

Abstraction
Abstraction is achieved through the use of interfaces (Sellable and ActiveSkill) that provide a contract for methods. This abstracts the specific details of how these methods work while providing a high-level structure for their behaviour.

SOLID Principles

Single Responsibility Principle (SRP): The GiantHammer class follows SRP by focusing on responsibilities related to representing a specialized weapon item and its actions. It handles actions, pricing, and skill activation, which are all related to its primary purpose.

Liskov Substitution Principle (LSP): The class adheres to LSP by providing specific implementations of methods defined in the interfaces it implements, ensuring that it can be used interchangeably with other objects implementing those interfaces. The GiantHammer class extends WeaponItem, and it can be used interchangeably with other weapons in the game, adhering to the LSP.

Pros:	Cons:
Modularity: The code is organized into packages and classes, promoting modularity, which makes it easier to manage and extend. **Flexibility**: The use of interfaces allows for flexibility in how classes can be used and extended. This is valuable for creating different weapons with unique skills and behaviours.	**Complexity:** Complexity: The code can be complex due to the inclusion of multiple actions within the skillAction method. While this complexity is necessary to achieve the desired gameplay effect, it makes the code harder to follow and maintain.

Maintainability: The code is structured with a focus on encapsulation and clear separation of concerns, enhancing maintainability. **DRY Principles:** The class generally follows DRY by encapsulating common behaviours in methods like activateSkill and staminaConsumedByActivateSkill. This reduces code duplication and makes maintenance easier	

Ways to extend in the future:

Weapon: We can create more weapon classes that implement the Sellable and ActiveSkill interfaces to introduce a variety of weapons with unique skills and behaviours.

ActiveSkill

The ActiveSkill interface represents a common contract for weapons in the game that have special skills. It defines methods related to activating these skills, consuming stamina, and executing skill actions. This interface plays a crucial role in enabling the implementation of specialized weapon classes like GiantHammer and GreatKnife.

Polymorphism
Polymorphism is achieved through the ActiveSkill interface by allowing different weapon classes (e.g., GiantHammer and GreatKnife) to implement this interface while providing their own unique implementations of the activateSkill and skillAction methods. This allows the game to treat these different weapons uniformly, as they share a common interface.

Abstraction
Abstraction is achieved by defining the ActiveSkill interface with method signatures but without implementation details. It abstracts away the specific implementation of weapon skills, focusing on what these skills should do rather than how they do it. This abstraction allows developers to create various weapon classes with unique skills while adhering to a common contract.

SOLID Principles

Single Responsibility Principle (SRP): The ActiveSkill interface adheres to SRP by defining methods related to weapon skills. It doesn't include unrelated functionalities, ensuring that each class implementing it has a single reason to change.

Open-Closed Principle (OCP): The interface is open for extension (new weapon classes with different skills) but closed for modification. New classes can be created by implementing the interface without altering its existing structure.

Liskov Substitution Principle (LSP): The ActiveSkill interface supports LSP by allowing derived weapon classes to replace the base interface without affecting the correctness of the program.

Interface and Dependency Inversion: The interface represents a higher-level abstraction that both GiantHammer and GreatKnife depend on, promoting dependency inversion.

DRY Principle
Defining the functionalities as an interface helps prevent repetition by enabling multiple classes to implement the same set of methods.

Pros:	Cons:
Reusability: The ActiveSkill interface can be reused for any future weapon classes with special skills. **Modularity**: The interface promotes modularity by separating the definition of weapon skills from their implementations, making it easier to add new weapons with unique skills. **Maintainability**: The code is structured with a focus on encapsulation and clear separation of concerns, enhancing maintainability.	**Complexity:** Managing multiple weapon classes that implement the ActiveSkill interface introduces complexity to the codebase, especially as each weapon has intricate skill mechanics. **Structure:** Implementing classes must provide implementations for all methods, which may not always be necessary

Future Extension:

New Skills: New types of active skills can easily be added by creating new classes that implement this interface.

Stamina Recovery: A future method could allow some skills to recover stamina instead of always consuming it.

Conditional Skills: Methods could be added to the interface to activate skills only under certain conditions (e.g., enemy health below a threshold).

ActivateSkillAction

This class represents an action in the game to activate the special skill of a weapon.

Polymorphism
Polymorphism is achieved through the use of interfaces. The ActiveSkill interface is implemented by various weapon classes, allowing them to have different implementations of the activateSkill method. This enables different weapons to exhibit unique behaviours when their special skills are activated.

Abstraction
This is achieved by encapsulating the activation of weapon skills in the ActivateSkillAction class. It abstracts the details of how each weapon's special skill is executed, allowing for a unified way of activating skills across different weapons.

SOLID Principles

Single Responsibility Principle (SRP): The ActivateSkillAction class has a single responsibility, which is to activate the special skill of a weapon.

Open-Closed Principle (OCP): The code follows the OCP by allowing for the extension of weapon classes to implement the ActiveSkill interface and define their own special skill behaviour without modifying existing code.

Liskov Substitution Principle (LSP): The ActivateSkillAction can work with any class that implements the ActiveSkill interface, demonstrating adherence to LSP.

Interface and Dependency Inversion: The code uses interfaces like ActiveSkill to define a contract for weapon skills, promoting dependency inversion by relying on abstractions rather than concrete implementations.

Pros:	Cons:
Code Reusability: By implementing the ActiveSkill interface, multiple weapon classes can reuse and share common behaviours for skill activation. This reduces code duplication and promotes efficient development. **Modularity**: The code separates the activation of weapon skills into a distinct	**Complexity:** While the code maintains a clear structure, it may become complex if the game requires a large number of weapon classes with unique special skills. Managing interactions between different weapons and their skills can add complexity.

action class, promoting modularity and code organization. **Maintainability**: The code's adherence to SOLID principles enhances maintainability by ensuring that changes or extensions can be made without extensive modifications to existing code.	

Future Extension:
The code is designed to support future extensions by allowing the addition of new weapons with distinct special skills that implement the ActiveSkill interface. Additionally, it can accommodate modifications or enhancements to existing special skill behaviours without altering the ActivateSkillAction class.

REQ5: Abxervyer, The Forest Watcher

Abxervyer, The Forest Watcher

The ForestWatcher class is designed to represent a powerful forest-themed enemy in the game, known as the Forest Watcher. This design rationale explains the key design decisions and principles applied to this class, including object-oriented design (OOD) principles, patterns, and considerations.

Polymorphism:

The ForestWatcher class exhibits polymorphism by extending the Enemy class. This allows it to be treated as an Enemy while also having specialized behaviour.

Abstraction:

Abstraction is achieved through the use of abstract classes and interfaces like WeatherControllable. The ForestWatcher abstracts away the complexities of weather control and enemy behaviour.

DRY Principle (Don’t Repeat Yourself):

The ForestWatcher class adheres to the DRY principle by reusing common functionalities from the parent class, Enemy. It doesn't duplicate methods or attributes already defined in the Enemy class, promoting code reusability.

SOLID Principles:

· **Single Responsibility Principle (SRP):**

The ForestWatcher class adheres to SRP by focusing on representing the Forest Watcher enemy and managing its behaviour. It does not handle concerns unrelated to enemy behaviour.

· **Open-Closed Principle (OCP):**

The ForestWatcher class follows the OCP by allowing for future extensions without modifying existing code. For example, additional behaviours can be added by extending the class without altering its core functionality.

· **Liskov Substitution Principle (LSP):**

The ForestWatcher class complies with LSP as it can be used interchangeably with its parent class, Enemy, without affecting program correctness. It retains expected enemy behaviours while introducing new ones.

· **Interfaces and Dependency Injection:**

  o The WeatherControllable interface is implemented by the ForestWatcher class, enabling it to control weather in the game.

  o Dependency injection is employed through the constructor to provide the WeatherManager and Gate instances, reducing tight coupling.

Pros:	**Cons:**
Encapsulation: Encapsulates the unique behaviour and attributes of the Forest Watcher enemy. **Reusability:** Promotes code reusability by extending the Enemy class. **Ease to extend** Allows for future extensions without modifying existing code. **Flexibility** Utilizes interfaces to separate concerns and enable flexibility in behaviour. **Maintainability** Implements encapsulation and encapsulates complexities, improving maintainability.	**More complex** The class may become complex if additional behaviours and capabilities are introduced. **Complex dependency** Extensive reliance on behaviours and actions could lead to complex behaviour chains.

Future Extension:

· Introducing more sophisticated behaviours based on the game's storyline.

· Customizing the Forest Watcher's abilities, such as unique attack patterns.

· Modifying its interaction with the WeatherManager to control other game elements.

· Adding visual effects or animations to enhance the player's experience during encounters with the Forest Watcher.

· Incorporating sound effects or dialogues to provide additional immersion during boss battles.

Weather (Enum)

Polymorphism:

It enables the representation of diverse weather states through a unified interface. Each weather condition (SUNNY and RAINY) can be treated interchangeably as instances of the Weather enum. It simplifies code, enhances flexibility, and allows for easy addition of new weather conditions without modifying existing code.

Abstraction:

The Weather enum provides an abstraction for different weather conditions in the game. The enum encapsulates the concept of weather without exposing the underlying details. It enhances clarity, reduces complexity, and allows for easy representation of distinct weather states. However, it is limited to representing only two weather conditions. Future extensions might require modifying the enum.

DRY (Don't Repeat Yourself) Principle:

The Weather enum class eliminates redundancy in the code by defining weather conditions in a single location. The enum centralizes the definition of weather conditions, ensuring consistency throughout the codebase. It can reduce errors, promotes maintainability, and avoids duplication of code related to weather conditions.

SOLID Principles:

- **Single Responsibility Principle (SRP):**

The Weather enum has a single responsibility of representing weather conditions, adhering to the SRP.

- **Open/Closed Principle (OCP):**

The enum is open for extension (adding new weather conditions) but closed for modification.

- **Liskov Substitution Principle (LSP):**

Instances of Weather can be substituted for each other without affecting the correctness of the program.

- **Interface Segregation Principle (ISP):**

As an enum, Weather does not implement interfaces, making ISP less applicable.

Pros:	Cons:
Simplicity: The enum provides a simple and straightforward representation of weather conditions. **Readability:** Enhances code readability by clearly defining weather-related concepts. **Ease of Use:** Easy integration into the game logic due to its simplicity.	**Limited Extensibility:** The enum may need modification to accommodate additional weather conditions in the future. **Static Nature:** Lack of dynamic behaviour associated with each weather condition (which might be desirable in certain scenarios).

Future Extension:

Possible Extensions:

- To enhance extensibility, consider converting Weather into an interface or superclass, allowing for the creation of custom weather classes with specific behaviour.

Flexibility:

- Implement a strategy pattern, allowing the game to dynamically assign behaviours to different weather conditions.

WeatherControllable

Abstraction:

The interface provides an abstraction for objects that can be affected by weather conditions. It abstracts away the details of how objects respond to weather changes, focusing on the essential method updateWeatherMode. The WeatherControllable interface enhances modularity and allows different classes to respond to weather independently.

Interface Design:

It declares a contract for classes that need to respond to weather changes. The updateWeatherMode method has a clear signature, specifying the inputs required for updating behaviour based on weather. The interface promotes consistency among implementing classes and ensures clarity regarding the responsibilities of implementing classes.

SOLID Principles:

- **Single Responsibility Principle (SRP):**

The interface adheres to SRP by focusing on a single responsibility—updating behaviour based on weather conditions.

- **Open/Closed Principle (OCP):**

Open for extension as new classes can implement the interface without modifying the existing interface.

- **Liskov Substitution Principle (LSP):**

Instances of classes implementing WeatherControllable can be substituted without affecting the correctness of the program.

- **Interface Segregation Principle (ISP):**

The interface is focused on a single method, aligning well with ISP.

Pros:	Cons:
Flexibility: Allows for a variety of classes to respond to weather changes independently. **Readability:** Clearly defines the method that implementing classes must override for weather-related updates. **Ease of Use:** Integration into the game logic is straightforward due to the simplicity of the interface.	**Prescriptive:** Assumes that all weather-related behaviour can be encapsulated within a single method. More complex scenarios might require additional methods or a different design.

Future Extension:

- **Possible Extensions:**

Consider adding more methods to the interface if different weather-related behaviours need to be addressed separately.

- **Flexibility:**

Allow for optional or default implementations of certain weather-related behaviours to provide a baseline for implementing classes.

WeatherManager

Abstraction:

The WeatherManager class abstracts the complexity of managing weather conditions and their effects on objects implementing the WeatherControllable interface. It provides a high-level interface for switching between weather conditions and updating objects, hiding the underlying implementation details. It enhances modularity, simplifies code for other components, and isolates weather-related concerns.

Polymorphism:

It utilizes polymorphism through the WeatherControllable interface, allowing diverse objects to be managed uniformly. The controlEnemy method iterates over a list of objects, treating them polymorphically by invoking the updateWeatherMode method. It facilitates extensibility, enabling the integration of new classes that implement WeatherControllable without modifying existing code.

DRY (Don't Repeat Yourself) Principle:

The class adheres to the DRY principle by encapsulating weather-related functionalities in a single location. Weather-related logic, such as switching weather conditions and updating objects, is centralized within the WeatherManager. It can reduce redundancy, promotes maintainability, and ensures consistency in how weather-related tasks are handled.

SOLID Principles:

- **Single Responsibility Principle (SRP):**

The class adheres to SRP by managing weather conditions and their effects on objects. It has distinct responsibilities for switching weather and updating objects.

- **Open/Closed Principle (OCP):**

The WeatherManager is open for extension, allowing for the addition of new weather conditions or objects responding to weather without modifying existing code.

- **Liskov Substitution Principle (LSP):**

Objects implementing WeatherControllable can be substituted without affecting the correctness of the program.

· **Interface Segregation Principle (ISP):**

The WeatherControllable interface focuses on a single method, aligning with ISP.

Pros:	**Cons:**
Modularity: Centralized management of weather-related concerns promotes modularity and clean separation of responsibilities. **Flexibility:** Adaptable to different weather conditions and diverse objects responding to weather changes. **Readability:** Clearly defines methods and responsibilities, making the code easy to understand.	**Uniformity:** Assumes uniform response to weather changes for all objects in forestEnemySpawnableGroundList. **Complexity:** May become complex if additional, divergent behaviour is introduced in response to different weather conditions.

Future Extension:

Possible Extensions:

- Introduce a more sophisticated system for weather transitions (e.g., gradual changes).
- Allow for custom behaviour updates for individual objects based on specific weather conditions.

Flexibility:

- Consider parameterizing the switchWeather method to allow for more complex weather conditions beyond sunny and rainy.

Dynamic Weather:

- Enable dynamic changes in weather based on in-game events or user interactions.

Refactoring enemies into ForestEnemy and VillageEnemy

Polymorphism:

- **Increased Flexibility:**

The use of polymorphism allows the game to treat forest enemies and village enemies uniformly, providing a common interface. This flexibility enables the game to easily add new enemy types without modifying existing code.

- **Simplified Code:**

Polymorphism simplifies the code by promoting a consistent approach to interacting with different enemy types. The game can call methods on enemy objects without needing to know their specific types.

Both ForestEnemy and VillageEnemy classes extend the abstract class Enemy. This allows for polymorphic behaviour, enabling the use of a common interface for both types of enemies.

Abstraction:

- **High-Level View:**

Abstraction hides the implementation details of enemy behaviours, allowing developers to focus on high-level design and interactions. It promotes a clear distinction between what an enemy does and how it achieves it.

- **Ease of Maintenance:**

By abstracting common behaviours into base classes, changes or improvements can be made at the base level, affecting all subclasses. This promotes ease of maintenance and reduces the chances of introducing errors.

Modularization:

- **Separation of Concerns:**

Dividing enemies into distinct packages based on their environment promotes a clear separation of concerns. This separation allows for independent development and maintenance of forest-themed and village-themed enemies.

- **Improved Readability:**

Modularization enhances code readability by grouping related classes together. Developers can quickly locate and understand the components related to forest or village enemies.

Encapsulation:

- **Controlled Access:**

Encapsulation ensures that the internal details of enemy classes are hidden. This controlled access prevents unintended interference and promotes a more robust system.

- **Easier Testing:**

By encapsulating the implementation details, it becomes easier to test each enemy type in isolation without worrying about unintended dependencies.

DRY (Don't Repeat Yourself):

- **Code Reusability:**

By creating common base classes for enemies, redundant code is minimized. This enhances code reusability and reduces the likelihood of errors introduced by duplicating logic.

- **Consistency:**

DRY promotes consistency in behaviour across different enemy types, making the codebase more maintainable

SOLID Principles:

- **Single Responsibility Principle (SRP):**

Each class (e.g., ForestEnemy, VillageEnemy) adheres to the SRP by having a clear and singular responsibility, promoting maintainability.

- **Open/Closed Principle (OCP):**

The use of polymorphism adheres to the OCP, allowing the system to be extended with new enemy types without modifying existing code.

- **Liskov Substitution Principle (LSP):**

Subclasses (e.g., ForestKeeper, RedWolf) can be substituted for their base class (e.g., ForestEnemy) without affecting the correctness of the program.

- **Interface Segregation Principle (ISP):**

Interfaces such as WeatherControllable are implemented only by classes that need them, adhering to ISP.

- **Dependency Inversion Principle (DIP):**

High-level modules (e.g., ForestWatcher) are not dependent on low-level modules (e.g., ForestEnemy); both depend on abstractions. This adheres to DIP.

Pros:	**Cons:**
Clear Responsibilities: Each subclass (ForestEnemy and VillageEnemy) has a clear and distinct responsibility, making the codebase more understandable. **Specific Behaviours:** Each type of enemy can have behaviours specific to its environment (forest or village), leading to more realistic and context-specific interactions. **Easier Maintenance:** Modifications and enhancements related to forest or village enemies can be made independently, reducing the risk of unintended consequences. **Focused Testing:** Testing can be more focused on the specific behaviours of each type, leading to better test coverage and more reliable code. **Reduced Complexity:** Each enemy type only includes the features and behaviours relevant to its environment, reducing the overall complexity of each class. **Easier to control:** If the weather changed, it is easier to change the mode and the conditions	**Code Duplication:** There may be some duplication of code between the ForestEnemy and VillageEnemy classes, especially if they share common characteristics. This can violate the DRY principle. **Hierarchy Complexity:** The class hierarchy becomes more complex with multiple subclasses. Managing this complexity requires careful documentation and understanding. **Missed Opportunities for Abstraction:** If common behaviours between forest and village enemies are not carefully considered, opportunities for abstraction and a common base class may be missed.

under different weather with some affected enemies.	

Future Extensions:

· **New Enemy Types:**

The current design allows for easy addition of new enemy types. Developers can create new subclasses of ForestEnemy or VillageEnemy to introduce unique behaviours.

· **Dynamic Weather Effects:**

The WeatherControllable interface provides a foundation for introducing dynamic weather effects for forest enemies. Future extensions could involve more sophisticated interactions based on changing weather conditions.

· **Environmental Influences:**

Expand the concept of environmental influences beyond weather. For example, different terrains or conditions in the forest could affect forest enemies.

· **Advanced Behaviours:**

Implement more sophisticated behaviours for enemies, such as collaborative strategies, retreat mechanisms, or adaptive learning.

Refactoring spawnable grounds into ForestEnemySpawnableGround and VillageEnemySpawnableGround

Abstraction:

Abstraction provides a level of indirection, allowing the code to be more flexible and adaptable to different environments and enemy types.

Polymorphism:

Polymorphism simplifies the code by allowing different spawnable grounds to exhibit behaviour specific to their types, improving maintainability and extensibility.

DRY Principle:

By avoiding code duplication, maintenance becomes more straightforward, and changes to spawning logic can be applied uniformly across all spawnable ground types.

SOLID Principles:

- **Single Responsibility Principle (SRP):**

Each class has a single responsibility, e.g., EnemySpawnableGround handles common spawning logic, while ForestEnemySpawnableGround and VillageEnemySpawnableGround manage environment-specific behaviours.

- **Open/Closed Principle (OCP):**

The design is open for extension (e.g., adding new spawnable ground types) but closed for modification, thanks to the use of abstract classes and polymorphism.

- **Liskov Substitution Principle (LSP):**

Subclasses (ForestEnemySpawnableGround and VillageEnemySpawnableGround) can be substituted for their base class (EnemySpawnableGround) without affecting the correctness of the program.

- **Interface Segregation Principle (ISP):**

The WeatherControllable interface segregates the responsibilities related to weather control, ensuring that implementing classes only need to provide methods relevant to weather changes.

- **Dependency Inversion Principle (DIP):**

Dependencies are inverted by relying on abstractions (e.g., Spawner and WeatherControllable) rather than concrete implementations, enhancing flexibility.

Pros:	Cons:
Specialization: Specialization allows for clearer and more focused implementations tailored to the unique characteristics of the forest and village environments. **Code Reusability**: Common functionality related to enemy spawning is abstracted into a shared base class, reducing code duplication and adhering to the DRY (Don't Repeat Yourself) principle. **Flexibility and Extensibility:** By defining abstract classes (ForestEnemySpawnableGround and VillageEnemySpawnableGround), the design accommodates future variations, such as new types of environments or enemies, without modifying existing code extensively. **Separation of Concerns:** Separating spawning logic into dedicated classes follows the Single Responsibility Principle, making each class focused on a specific aspect of the game (e.g., spawning enemies in a particular environment). **Clear Interface for Weather Control:** This clear interface ensures that any ground capable of spawning enemies can be easily integrated into the weather system without modifying the existing ground classes. **Enhanced Readability:** Developers can quickly identify the purpose of a class based on its name (ForestEnemySpawnableGround or	**Complexity** Increased number of classes may initially seem complex, but the benefits in terms of modularity and maintainability outweigh this concern. **Hard to understand** Learning curve for developers unfamiliar with the specific design may be present.

VillageEnemySpawnableGround), leading to a more maintainable codebase. **Consistent API:** Consistency in the API simplifies usage for game developers, promoting a standardized approach when working with different types of spawnable grounds.	

Refactoring spawners by using General Types

Abstraction:

Abstraction is achieved by defining a common interface for spawners, allowing for flexibility in the instantiation of various enemy types.

Polymorphism:

Polymorphism enables the interchangeable use of different spawners, facilitating a unified approach to enemy instantiation in the game.

DRY Principle:

Code duplication is avoided, providing a cleaner and more maintainable design. Changes to the spawning logic can be applied uniformly across all spawner implementations.

SOLID Principles:

· **Single Responsibility Principle (SRP):**

The Spawner interface has a single responsibility: spawning enemy actors. Each spawner class adheres to this principle by providing a concrete implementation for a specific enemy type.

· **Open/Closed Principle (OCP):**

The design is open for extension, allowing the addition of new enemy types by creating new spawner implementations. The generic Spawner interface is closed for modification.

· **Liskov Substitution Principle (LSP):**

Subtypes (ForestKeeperSpawner, RedWolfSpawner, etc.) can be substituted for their base type (Spawner) without affecting the correctness of the program.

· **Interface Segregation Principle (ISP):**

The Spawner interface is focused on a single responsibility, adhering to the ISP.

Pros:	**Cons:**
● Improved code organization and readability through the use of a generic interface. ● Enforces a consistent spawning API, making the codebase more maintainable. ● Facilitates the addition of new enemy types without modifying existing spawner logic. ● Promotes a more modular and extensible design.	● Initial setup may require creating multiple spawner classes, but this investment pays off in terms of long-term maintainability. ● Developers unfamiliar with generic types might need some time to adapt.

Future Extensions:

- Introduce new enemy types by creating new implementations of the Spawner interface.
- Expand spawner behaviour by adding methods or parameters to the Spawner interface, allowing for more sophisticated spawning logic.
- Enhance the spawner hierarchy to support more complex scenarios, such as dynamic spawning rates or location-based spawning.