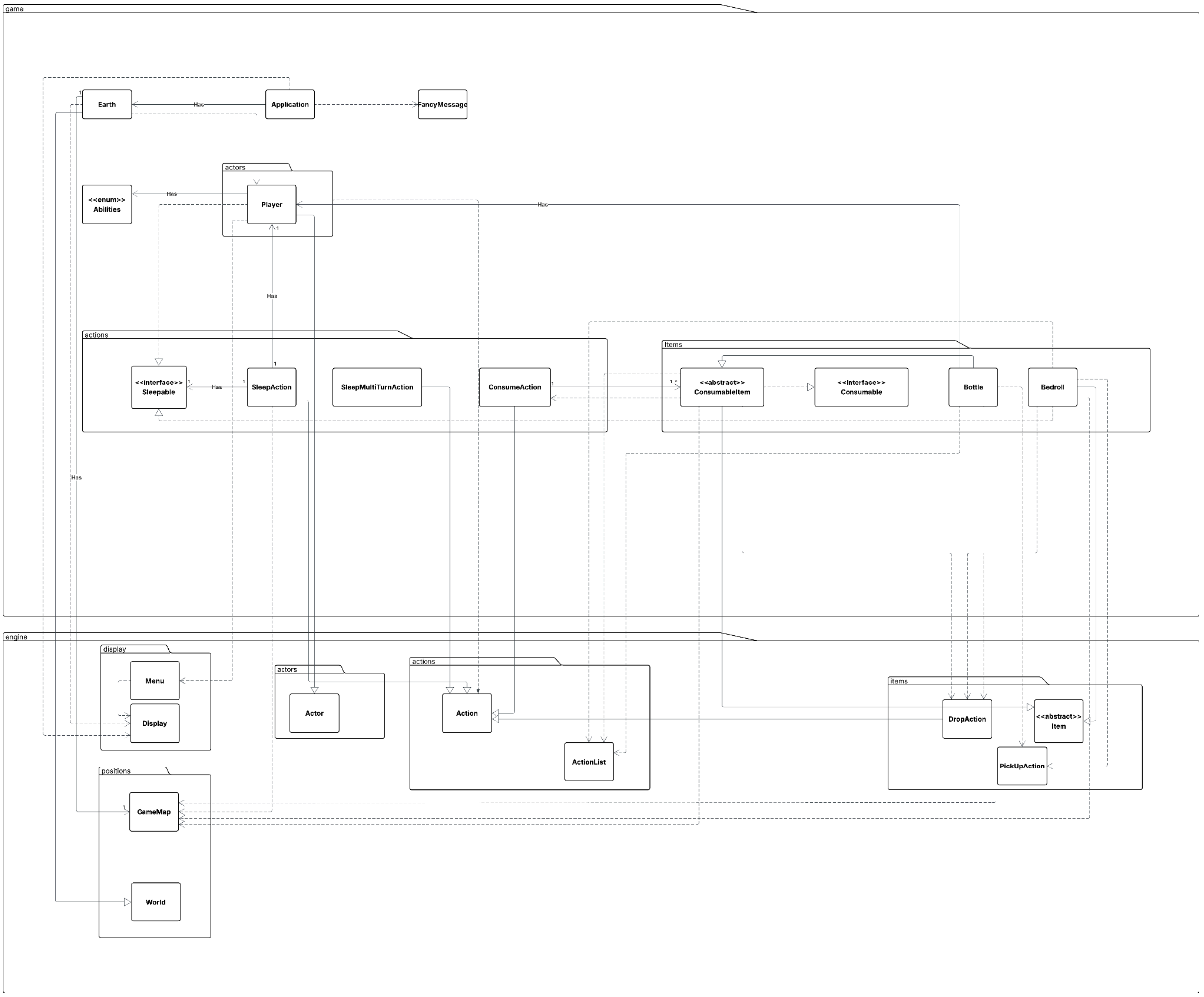
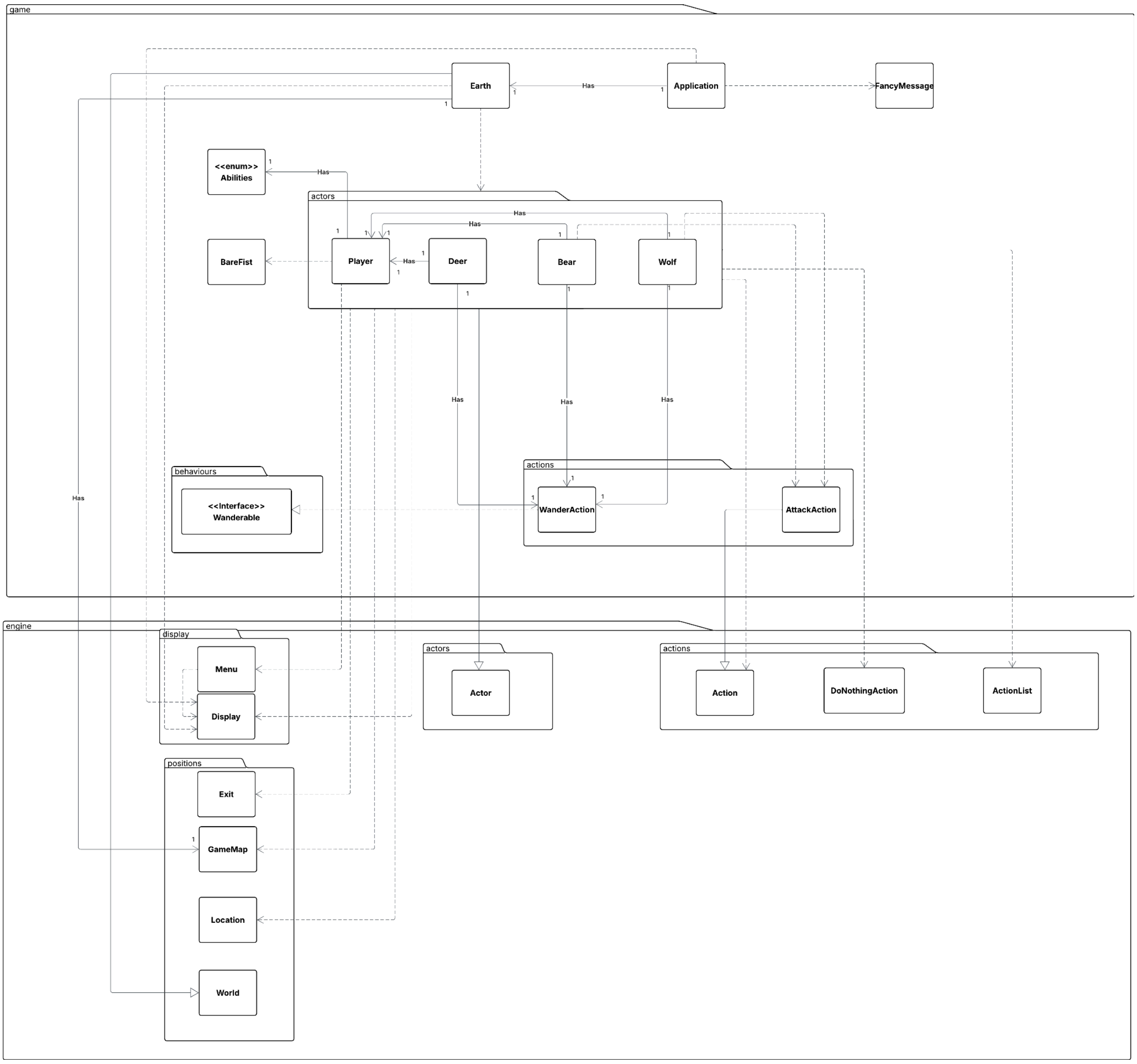
FIT2099 Assignment 1 Design Documentation

Design Diagrams

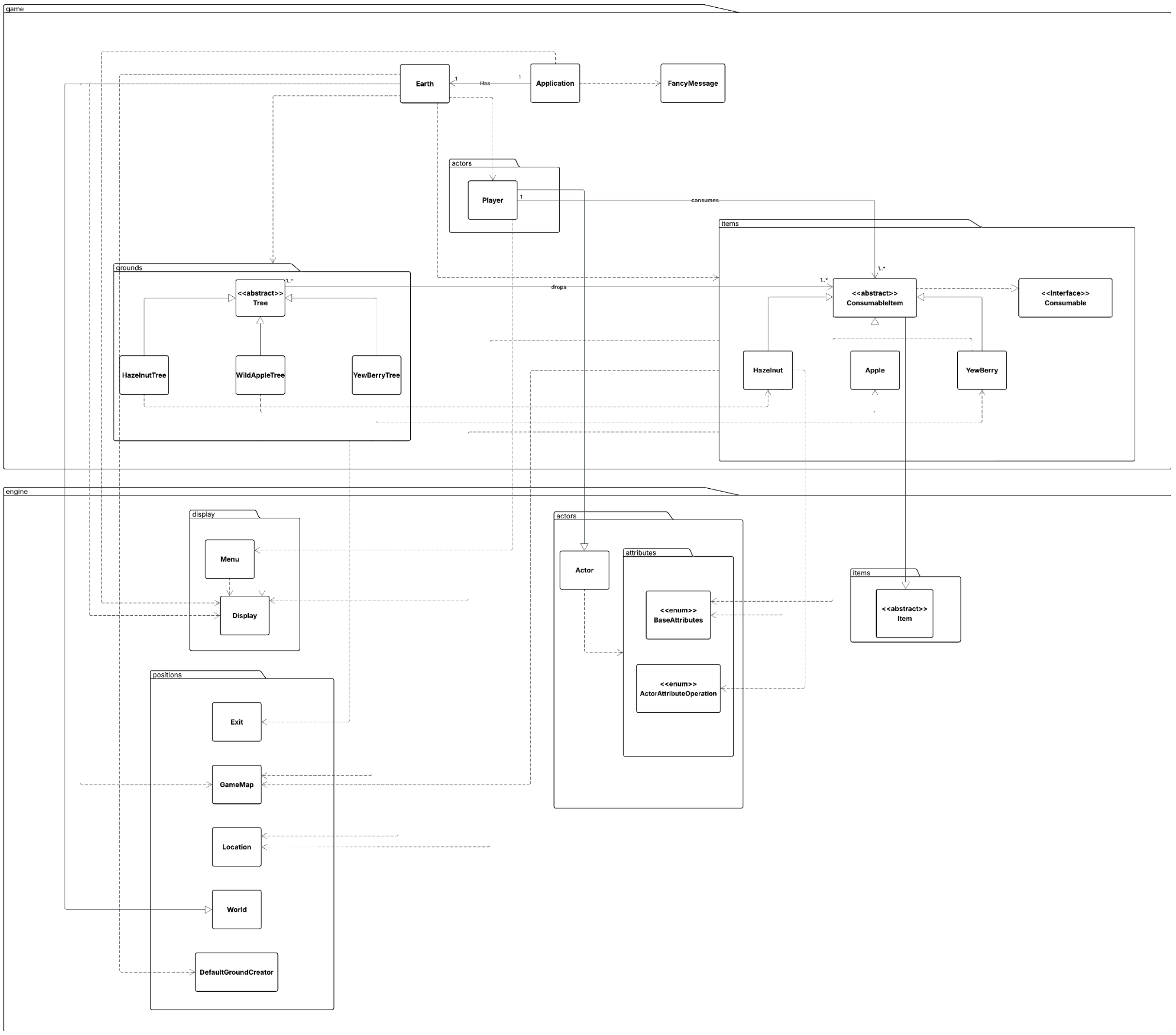
REQ 1 UML Class Diagram:



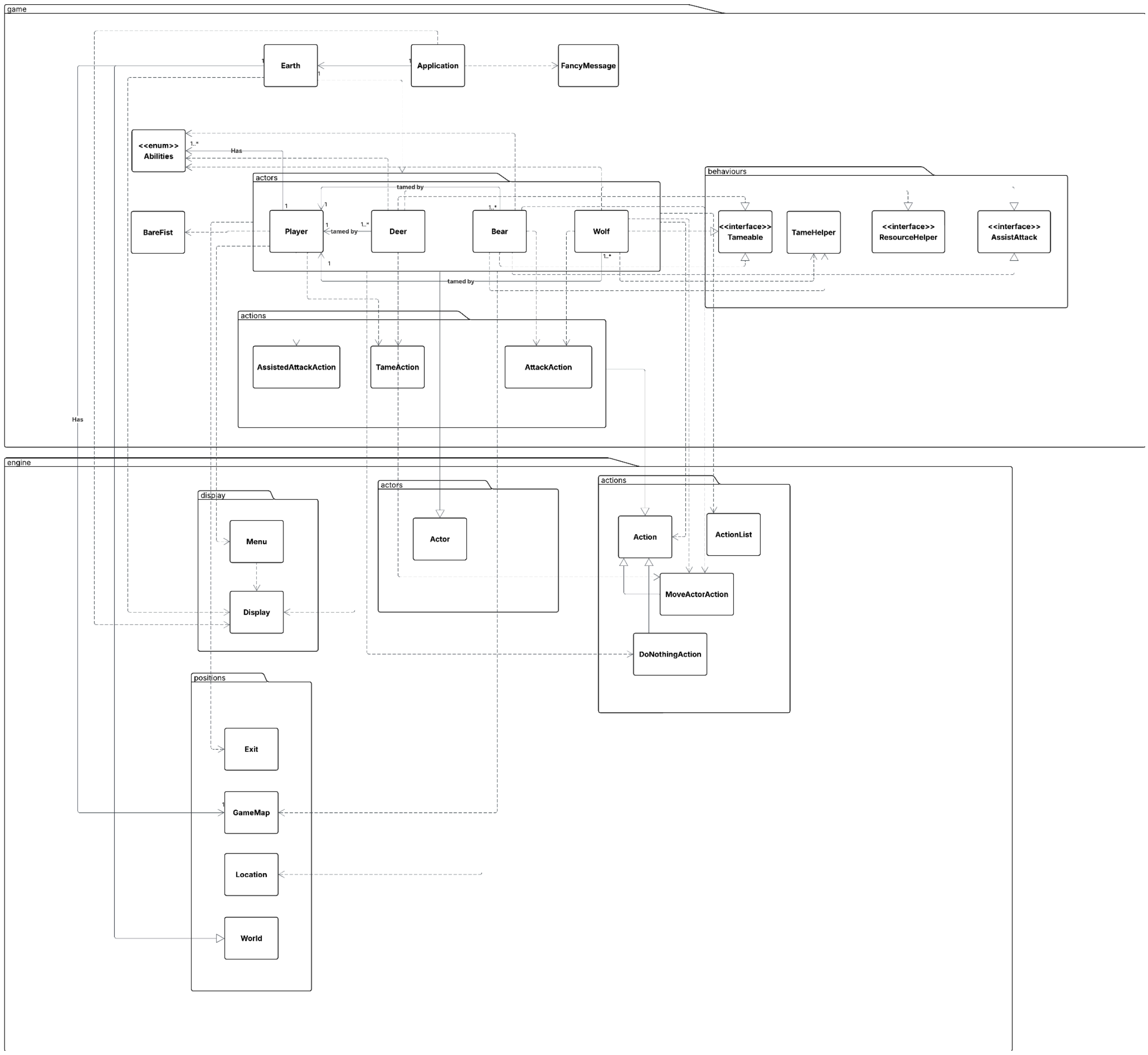
REQ 2 UML Class Diagram:



REQ 3 UML Class Diagram:



REQ 4 UML Class Diagram:



Design Rationale

REQ 1

The goal of this requirement is to set up the Explorer as the player's character, with survival rules dependent on hydration and warmth. The Explorer starts the game with 100 HP points, a hydration level of 20, and a warmth level of 30. Every turn, both hydration and warmth decrease by one, and the game finishes when either hits zero. To support survival, the Explorer starts with two items: a bedroll and a bottle. The bedroll allows the Explorer to sleep for six to ten turns(randomly chosen) while maintaining hydration and warmth, but it must first be dropped on the ground before the choice to sleep is accessible. The bottle contains five drinks, each restoring four points of hydration. Importantly, the Explorer's statistics - hit points, hydration, and warmth - are presented at each round, keeping the player informed of their survival condition.

Two designs were considered:

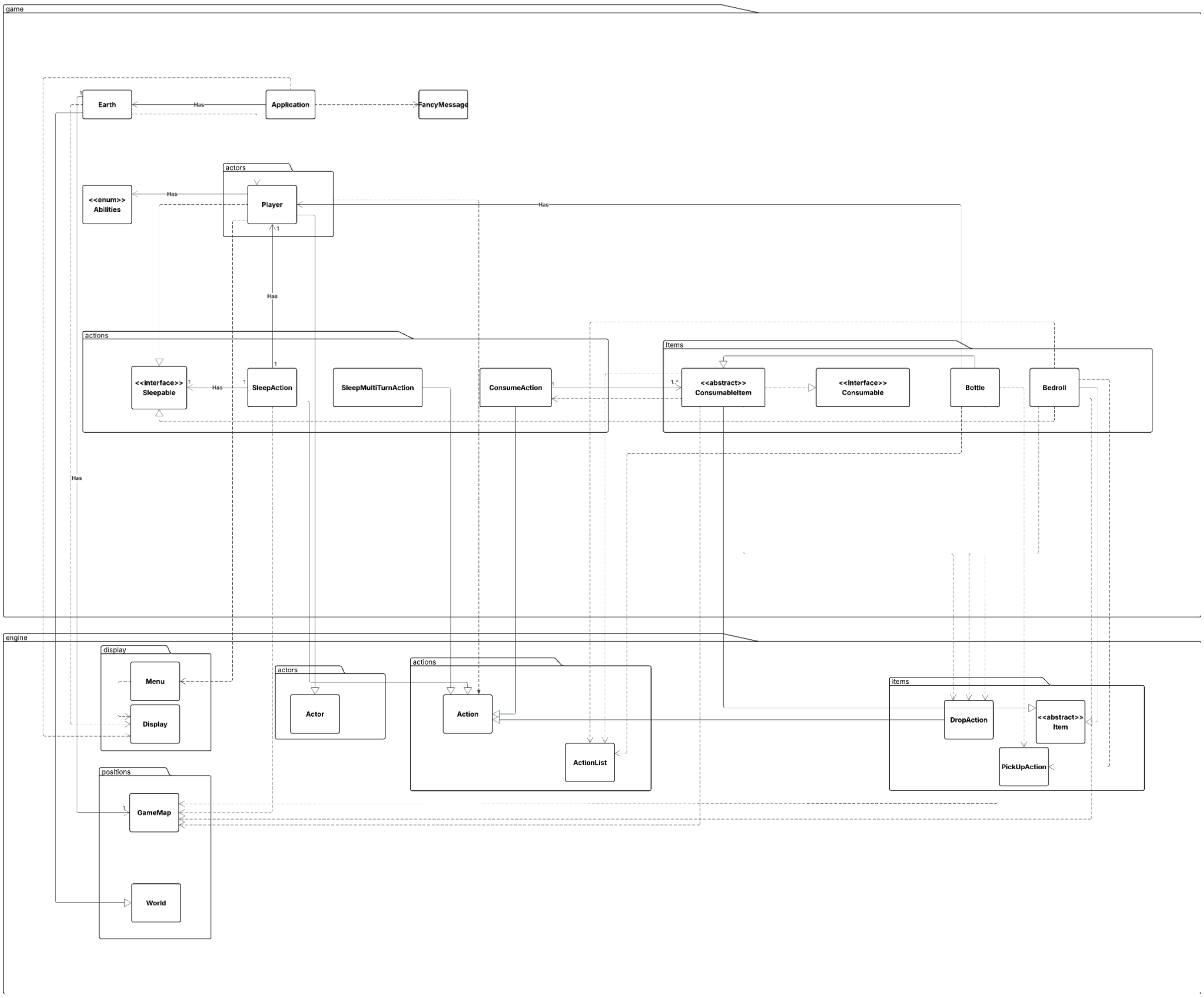
Design 1: The Explorer class is directly in charge of sleeping and drinking mechanisms, updating hydration, temperature, and bottle capacity internally whenever the player sleeps or drinks.

Pros	Cons
The implementation is simple and easy because all functionality is contained in a single class.	Violates the DRY principle: any new drinkable or sleepable things would need duplicating logic in Explorer.
Explorer manages all state changes directly, making it easier to track the game's progress.	Explorer items are tightly coupled, making the system difficult to extend or maintain.
The mechanisms are centralized, making debugging simple.	Adding new functionality requires changing the Explorer class itself, which violates the Open/Closed Principle.

Design 2 (Chosen): Action and interface classes manage the mechanics: ConsumableItem implements Consumable and uses ConsumAction to drink; Sleepable is implemented by objects such as Bedroll, which uses SleepAction and SleepMultiTurnAction for single- and multi-turn sleep. The Explorer class only shows accessible actions depending on their current condition.

Pros	Cons
Centralizes logic in action classes to reduce repetition and adhere to DRY principles.	Structure is more complex, requiring an understanding of how actions, interfaces, and items interact.
Explorer does not directly control item mechanics, which reduces coupling.	Explorer status changes like hydration and warmth are handled by actions, not directly by the Explorer.
New items and sleep/drink processes can be added without modifying Explorer (Open/Closed Principle).	The initial setup is more complicated since items, actions, and interfaces must be properly connected before the game can work.
Improves cohesion and follows the Single Responsibility Principle (SRP).	
Polymorphism is supported, which means that any object that implements the	

appropriate interface can be utilized without modifying Explorer.	



The diagram above shows the final design for requirement 1, which uses Design 2 from above.

When I implemented the Explorer's survival features, I chose to split the logic for hydration, warmth, and sleeping into separate Action and Interface classes rather than putting it directly into the Explorer. The primary objective for this design decision was to improve maintainability, extensibility, and consistency while avoiding duplicated functionality. By delegating hydration restoration to Consumable objects via ConsumeAction and sleep behavior to Sleepable objects via SleepAction and SleepMultiTurnAction, the Explorer class is only required to monitor its state and make accessible the actions that may be executed. This allows for a better separation of issues and a more modular framework.

This design has good cohesiveness since each class has a single, well-defined responsibility: the Explorer maintains its state, objects like Bottle or Bedroll describe properties, and actions like ConsumeAction or SleepAction explain how the state changes. It also supports minimal coupling because the Explorer interacts with abstractions such as

Consumable and Sleepable rather than hardcoding the survival logic. Therefore, new items or survival mechanics may be added without modifying the Explorer itself.

Single Responsibility Principle (SRP): The Explorer only monitors its own characteristics (health, hydration, and warmth), whereas items and Action manage consumption and sleeping behavior. Each class has a particular purpose.

The Open/Closed Principle (OCP) states that new consumables or sleepable things can be added by implementing the appropriate interfaces rather than changing current classes. For example, adding a new food item would simply need extending ConsumableItem, not rewriting the Explorer.

The Liskov Substitution Principle (LSP) states that because all consumables use the Consumable interface and all sleepable objects use the Sleepable interface, one object can be replaced with another without causing the system to fail. For example, substituting a bottle with another drinking item remains valid under the same contract.

Interface Segregation Principle (ISP): Interfaces are kept minimal and role-specific, so classes only implement what they require. For example, the Consumable interface is just concerned with being eaten, whereas the Sleepable interface exclusively describes sleeping behavior.

Dependency Inversion Principle (DIP): The Explorer relies on abstractions (Consumable, Sleepable, and Action) rather than real implementations. This reduces the system's reliance on specific item types and makes it more responsive to potential modifications.

The disadvantage of this approach is that it adds complexity by having several layouts and action classes. Developers must properly setup and link these abstractions to minimize misunderstanding. However, the long-term benefits much exceed the expense, since the system becomes modular, testable, and scalable for future survival mechanics.

Overall, this technique results in a codebase that is clean, extendable, and follows object-oriented design principles. By separating state management and behavior, the game eliminates redundancy, simplifies maintenance, and allows new survival features to be added with minimum interruption to the present system.

REQ 2

The purpose of this need is to introduce animals that the Explorer may interact with, such as deer, wolves, and bears, each with distinct hit points, random movement habits, and attack capabilities. These species present obstacles and dynamic interactions in the wilderness, forcing the explorer to make tactical decisions for survival.

Two main designs were considered:

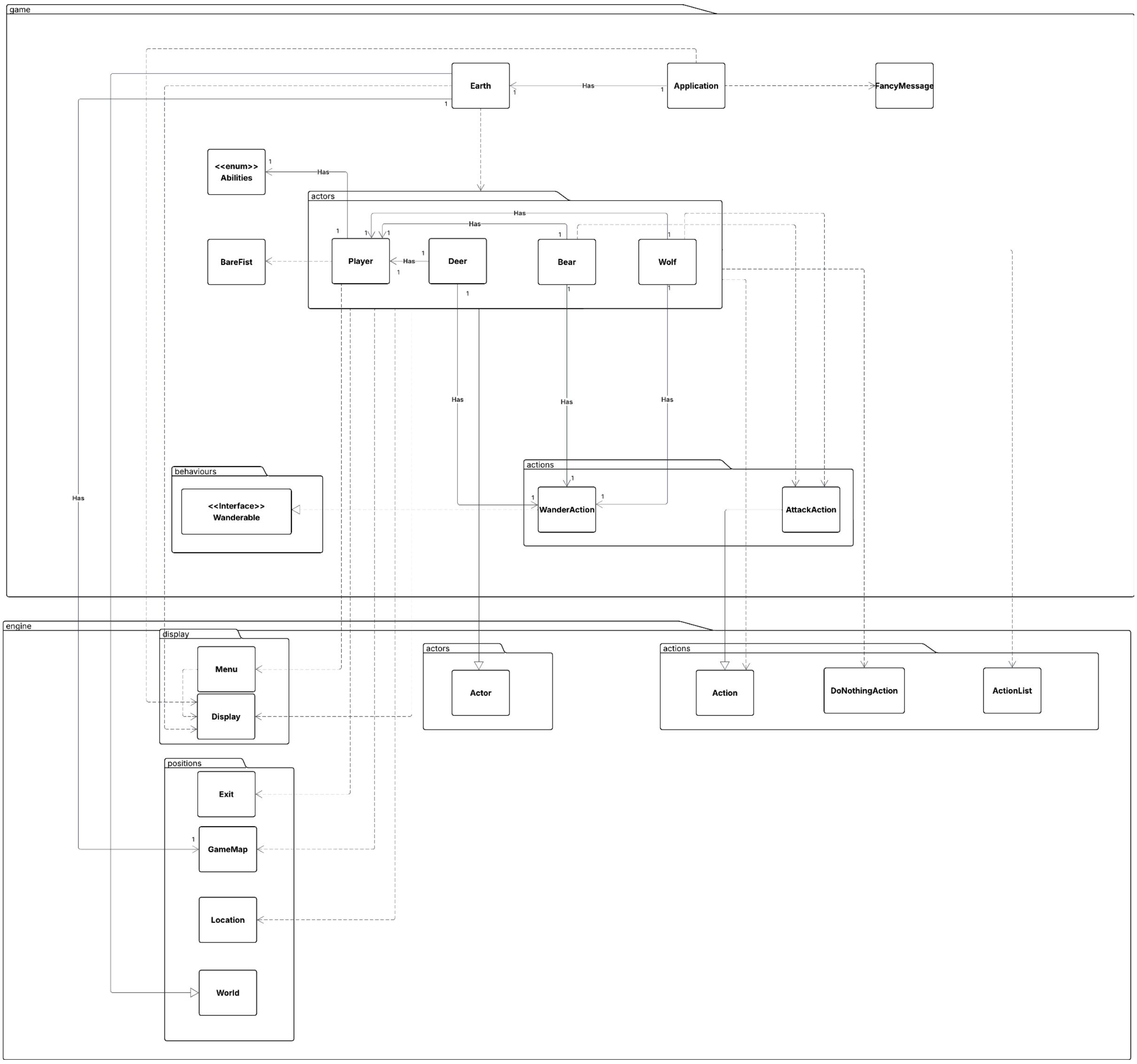
Design 1: Each animal class (wolf, deer, and bear) has its own movement and attack logic. While this solution is simple to implement, it results in redundant code across animal types, which makes maintenance and scalability difficult.

Pros	Cons
All logic for wander and attack is contained within the animal class, making it simple to understand what an animal does by reading only that class.	Wander and attack code are reused across numerous animal types, making the codebase more difficult to maintain.
Implementation is quick for a small number of animals since no additional classes or interfaces are required.	Any change in behavior requires changing each animal type manually, which increases the likelihood of creating inconsistencies or bugs.
Good for prototyping since it allows you to observe immediate outcomes without having to consider abstract concepts.	Adding more animals is difficult and error-prone since each new class requires its own copy of comparable logic, which limits scalability.

Design 2 (Chosen): Use action and interface classes to manage behaviors. Combat is handled by AttackAction, whereas Wanderable with WanderAction is responsible for random movement. Each animal class just keeps its own state (hitpoints, damage, and abilities) and displays possible actions. This method divides responsibilities (SRP), and enables the addition of new animals or changes in movement/attack behavior without modifying current classes (OCP).

Pros	Cons
Wander and attack are divided into Action and Interface classes, so each animal class just needs to keep its own state (hitpoints, damage, abilities) and list available actions.	There is more initial setup since you must specify interfaces and action classes, which adds complexity at the beginning.
Adding new animals or modifying the way movement or attack functions is much easier since action classes may be implemented without affecting current animal classes.	Developers must understand how the Action classes interact with animals, as behavior is no longer incorporated directly in each class.
Reduces the coupling between animals and core game classes, increasing modularity and making it easier to manage, test, and expand their behavior.	
Encourages cleaner code organization and adherence to the Single Responsibility	

Principle (SRP), which makes the system more manageable in the long run.	



The final requirement 2 design, which makes use of Design 2 mentioned above, is shown in the diagram above.

When creating the animal behaviors, I decided to divide the logic into distinct Action and Interface classes rather than putting wandering and attacking directly into each animal type (such as Deer, Wolf, and Bear). The main reason for this option was to promote maintainability, extensibility, and consistency between animals while eliminating duplicated logic. By centralizing fighting in AttackAction and random mobility in WanderAction (via the Wanderable interface), each animal type may simply specify its properties, such as hitpoints, abilities, and damage, while leaving behavioral execution to reusable components.

This design achieves high cohesiveness because each class has a specific responsibility: animals specify their state, whereas actions describe how behaviors are performed. It also encourages minimal coupling since animals are no longer permanently connected to their own movement or attack rationale. Instead, they rely on actions and interfaces that may be reused or changed without affecting current animal types.

Single Responsibility Principle (SRP): Animal classes are only in charge of maintaining state (hitpoints, abilities, and stats), whereas AttackAction and WanderAction handle particular actions. Each class serves a certain purpose.

The Open/Closed Principle (OCP) states that new animals or behaviors (such as a particular attack or a different movement style) can be introduced by extending or creating new actions, without modifying the code of current creatures.

Liskov Substitution Principle (LSP): Because animals have the same interfaces (e.g., Wanderable), one kind of animal may be changed with another on the map without affecting functionality. For example, replacing a deer with a wolf has no effect on how movement or conflict are processed.

Interface Segregation Principle (ISP): Role-based interfaces, such as Wanderable, guarantee that animals only do actions that are relevant to them. A deer that simply wanders does not need the AttackAction class, but a wolf or bear can engage in fighting behavior.

Dependency Inversion Principle (DIP): The system is based on abstract specifications of behavior (interfaces and actions) rather than hardcoded logic within each animal. This makes the design more adaptable and easier to extend in the future.

The biggest disadvantage of this strategy is the increased initial complexity. Developers must carefully configure interfaces and actions, as well as realize that behaviors such as wandering or fighting are executed by external classes rather than the animals themselves. However the system becomes more modular, easier to maintain, and capable of supporting new features without requiring major adjustments.

Overall, this approach produces a codebase that is cleaner, more modular, and adheres to object-oriented design guidelines. Redundancy is reduced by separating animal state and behavior, making it easier to add new animals or behaviors.

REQ 3

The goal of this requirement is to add flora with which the Explorer can interact, such as wild apple trees, hazelnut trees, and yew berry trees. Each tree variety has a distinct gameplay impact through the items that it drops: apples restore health and reduce thirst, hazelnuts enhance maximum health, while yew berries are toxic, killing the Explorer instantaneously. This system presents both possibilities and threats, promoting strategic choices about what to eat.

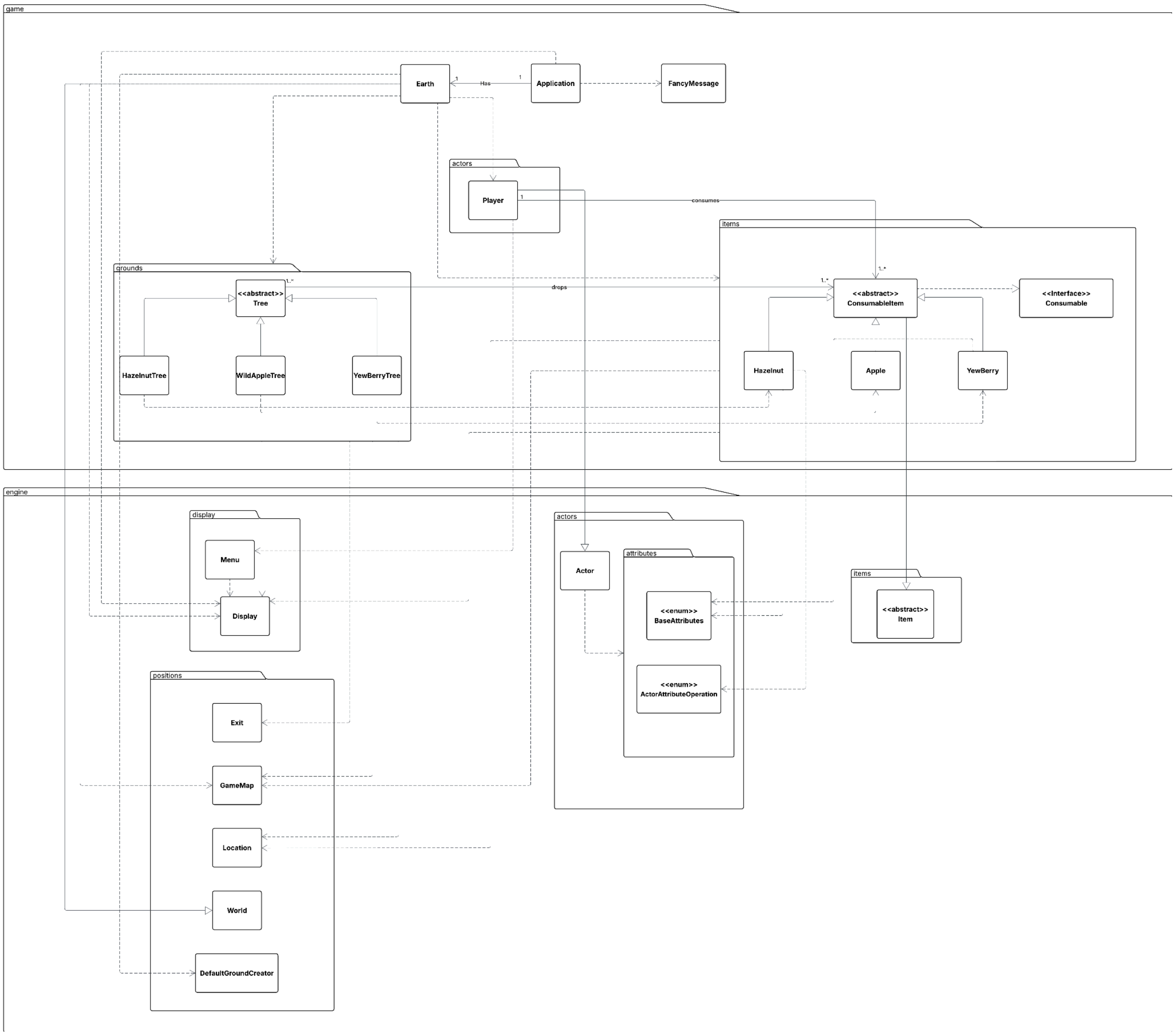
Two main designs were considered:

Design 1: Each tree type (e.g., AppleTree, HazelnutTree, YewBerryTree) would have direct control over both its item drop turns and the effects of its items when consumed by the Explorer.

Pros	Cons
Very straightforward, given the tree class itself contains all of the logic.	Adding a new tree or changing the spawn behavior requires modifying numerous classes, which raises the possibility of inconsistencies or bugs.
Putting fruit-dropping logic directly in each tree class enables fast testing of gameplay aspects such as fruit spawn rates and Explorer interactions, without the need for elaborate abstractions or reusable components.	Each tree class handles its own fruit drop logic, which creates code duplication and complicates system maintenance, testing, and extension.
	Tree classes that manage both their own state and fruit-dropping behavior are more difficult to maintain, comprehend, extend, and test. This violates the Single Responsibility Principle since each class manages numerous issues rather than focusing on just one.

Design 2 (Chosen): Use an abstract Tree class to manage shared mechanics such as drop timings and tile selection. Each subclass (such as WildAppleTree, HazelnutTree, and YewBerryTree) defines the type of object dropped. Each item also has its own class (Apple, Hazelnut, YewBerry) that defines its effects when consumed. This separates tree mechanics from item behavior, which increases modularity and maintainability.

Pros	Cons
Centralized drop logic reduces redundancy and makes maintenance easier.	To properly add a new Tree or Fruit, developers must understand how trees and objects differ.
Adding new trees or items is simple. Subclasses simply specify drops, whereas items describe effects.	The initial setup is more complicated due to the number of classes.
The responsibilities for the tree and each item are separated, which improves modularity, testability, and scalability.	



The completed design for requirement 3, which applies Design 2 as described earlier, is illustrated in the diagram above.

When creating the flora system, I chose to put the drop mechanisms in an abstract Tree class and divide each consumable item into its own class (such as Apple, Hazelnut, or YewBerry), rather than having each tree subclass manage drop time and item effects separately. Maintainability, extensibility, and adherence to object-oriented design principles were all important factors while making this selection. By centralizing the drop logic, I avoided duplicating comparable methods across several tree subclasses, according to the DRY (Don't Repeat Yourself) concept. Each tree subclass now just describes which item it generates and when it will drop, while the corresponding item class handles the Explorer's impact when eaten.

This approach promotes strong coherence since the responsibility for producing fruits and controlling their impacts is clearly divided. Tree classes are no longer required to comprehend item effects, letting them concentrate only on their function in handling drop mechanisms. This also supports minimal coupling because the game loop interacts with trees and objects using well-defined actions rather than allowing trees to directly affect Explorer status.

The Single Responsibility Principle (SRP) states that tree classes handle drop mechanics and item classes handle consumption effects. Each class has a distinct, well-defined responsibility.

The Open/Closed Principle (OCP) allows new tree kinds or items to be introduced by extending current classes without modifying the basic drop or impact logic.

Dependency Inversion Principle (DIP): The system relies on abstractions (tree behavior and item impacts) rather than specific implementations, eliminating direct dependencies and allowing for flexible extension.

The Liskov Substitution Principle (LSP) states that any subclass of Tree or item may be substituted with another without changing the way drops or effects work. For example, replacing WildAppleTree with HazelnutTree works perfectly within the same drop structure.

Interface Segregation Principle (ISP): Future expansions might use smaller, role-based interfaces to ensure that trees and objects only implement methods related to their function and avoid superfluous obligations.

The biggest disadvantage of this technique is that it requires slightly more starting complexity than having each tree manage drops and effects individually. Developers must realize that item effects are handled by the item classes rather than the trees themselves. However, the system's long-term maintainability, scalability, and clarity outweighs its initial complexity.

Overall, this design results in a modular, expandable, and manageable codebase. Centralizing drop logic in the abstract Tree class and separating item effects into separate classes improves readability, reduces redundancy, and makes it easier to add new flora or consumable items, all while maintaining clear responsibilities and adhering to object-oriented design principles.

REQ 4

The purpose of this requirement is to include a taming mechanism that will allow the Explorer to befriend creatures like wolves, bears, and deer. Once tamed, animals show additional supporting behaviors that provide game variety and strategic complexity. For example, wolves and bears can help the Explorer in fight, but deer can gather resources such as fruits and give them to the Explorer. Furthermore, all tamed animals will follow the Explorer and will not fight their tamed companions, making cooperation more natural and cooperative.

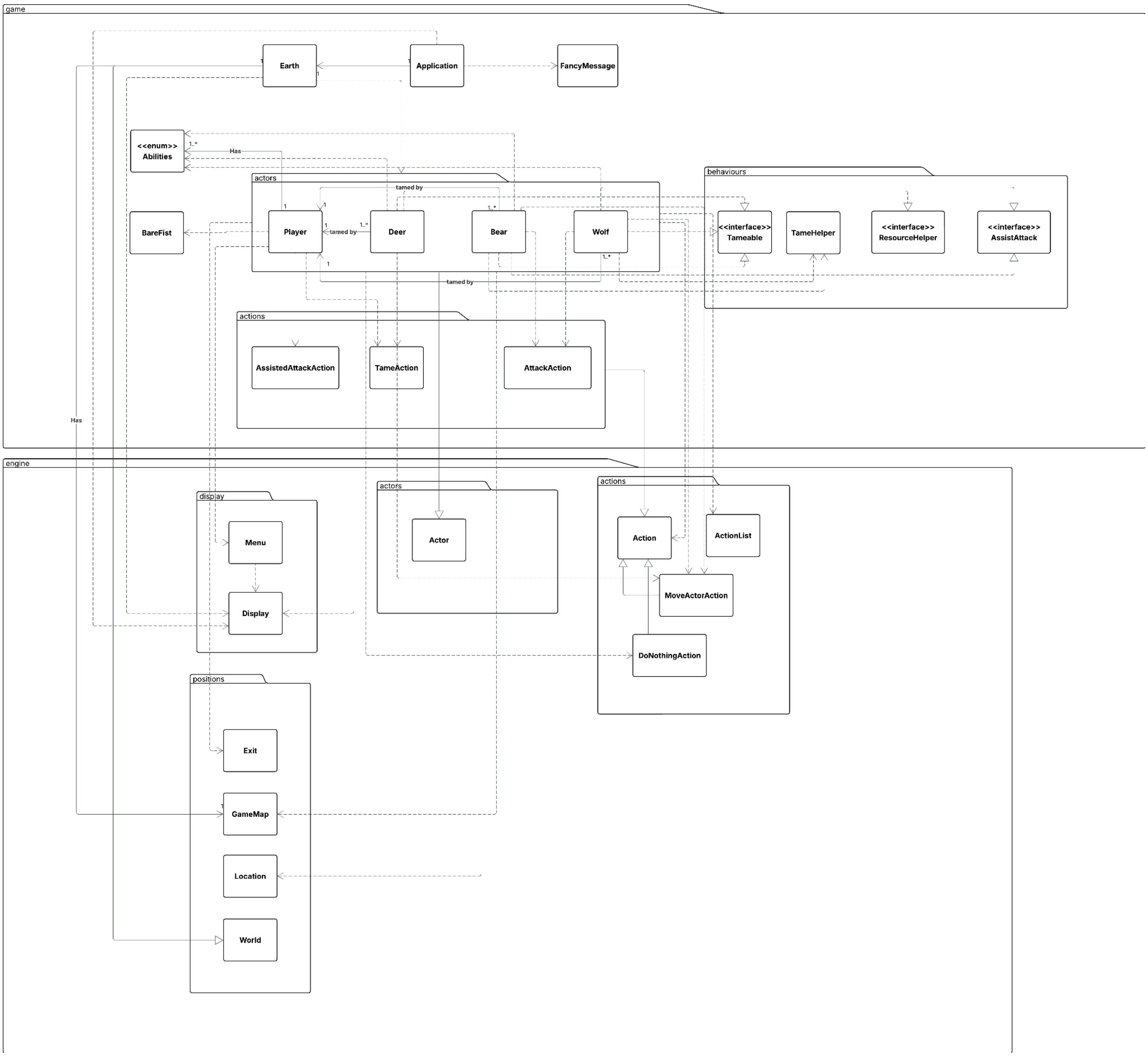
Two main designs were considered:

Design 1: Put taming logic directly into each animal type (such as wolves, bears, and deer), handling taming states and post-tame behaviors inside the animal.

Pros	Cons
Since all taming states and behaviors are put together under the animal class, implementation is straightforward.	Code duplication among animal types: logic for resource collecting, follow, or help attack would need to be repeated several times.
It's simple to observe all of an animal's behaviors in one location.	By making animal classes manage state, taming transitions, and various behaviors, it violates SRP.
Taming effects may be quickly prototyped without requiring additional abstractions.	Difficult to scale and maintain: introducing new tame behaviors requires modifying current animal classes, which raises the possibility of errors.

Design 2 (Chosen): Use a modular design with role-based interfaces (AssistAttack, ResourceHelper) and their corresponding actions (AssistAttackAction), a Tameable interface, and a TameHelper class for controlling tame state. This assigns behaviors to reusable components and keeps taming mechanisms apart from animal definitions.

Pros	Cons
Animals only implement abilities that are relevant to them (wolves and bears use AssistAttack, deer use ResourceHelper).	More difficult to set up since additional classes and interfaces must be written and coordinated.
TameHelper helps tame state logic, which reduces duplication and enables extension.	Developers must learn to track behavior using helper and action classes rather than a single animal class.
Assist behavior is contained under AssistAttackAction, which makes battle logic reusable and consistent.	
Allows for long-term scalability: new tameable animals or behaviors may be introduced without modifying existing code.	
Clear separation of concerns increases maintainability, testing, and adherence to the SOLID principles.	



The finalised design for requirement 4, implementing the previously described Design 2, is depicted in the diagram above.

When creating the taming system, I chose to combine tame state in the TameHelper class and delegate post-tame behaviors to defined interfaces and actions rather than embedding them directly in each animal subclass. Maintainability, extensibility, and adherence to object-oriented design principles were major motivators. By separating concerns, wolves and bears can utilize the AssistAttack interface with AssistAttackAction to add extra damage to the Explorer's attacks, whilst deer use the ResourceHelper interface to acquire and deliver fruits to the Explorer's inventory. All tamed animals usdde the Tameable interface, which ensures a uniform interaction model.

It increases cohesiveness since each class serves a defined purpose: animals keep stats, TameHelper maintains taming status, and action classes handle behaviors such as helping in fighting or gathering resources. Coupling is decreased since animals no longer integrate their own tame logic behaviors; instead, the Explorer and game loop may rely on reusable abstractions.

Single Responsibility Principle (SRP): Animals maintain their attributes, TameHelper and TameAction handles tame transitions, AssistAttackAction handles assist damage, and ResourceHelper performs item collecting.

The Open/Closed Principle (OCP) states that new tameable animals or behaviors can be added by extending interfaces and adding new actions rather than changing existing classes.

The Liskov Substitution Principle (LSP) states that any tameable animal can be replaced for another in the taming system since they all share the same Tameable Contract.

The Interface Segregation Principle (ISP) states that animals only implement appropriate behaviors (wolves and bears do not implement ResourceHelper, and deer do not implement AssistAttack.)

Dependency Inversion Principle (DIP): The Explorer and main game loop use abstractions (Tameable, TameAction, AssistAttack, ResourceHelper) rather than real animal implementations to promote scalability.

The biggest disadvantage of this technique is the added complexity of handling so many helper and action classes, which requires careful planning. However, the benefits of modularity, reusability, and adherence to SOLID principles exceed the costs, resulting in a system that is easier to extend and maintain over time.

Overall, this approach creates a versatile, modular taming system. Animals can obtain cooperative skills such as following, helping in fighting, and gathering resources by abstracting tame behaviors into specialized interfaces and action classes, while keeping responsibilities clearly segregated and the framework easily extendable.