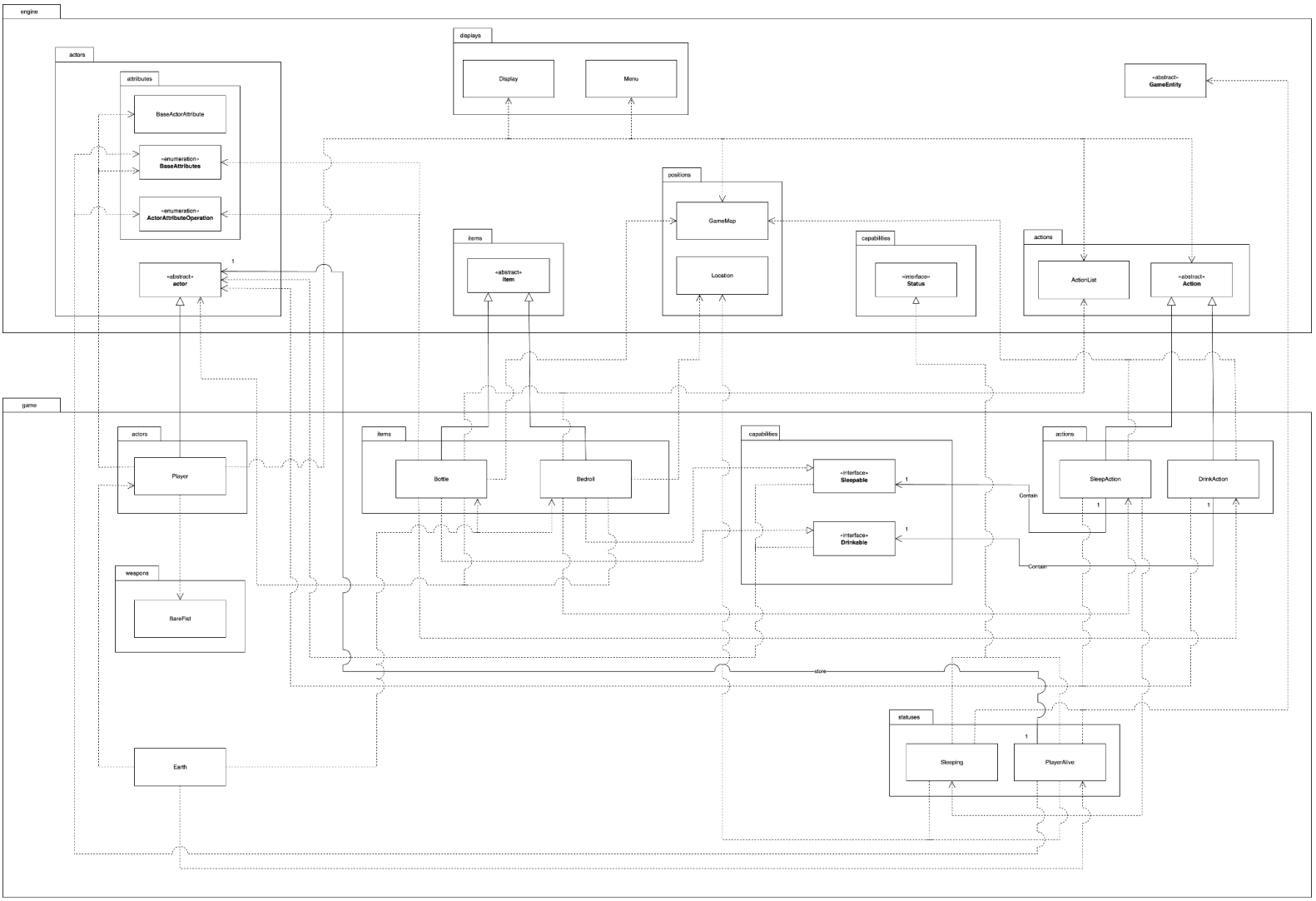
Fauzanda Lathifanka Sunarko (34310509)
REQ 1

Main UML Class Diagram:



Design Rationale :

Main Design (The above UML Class Diagram):

- **Create an interface Drinkable and Sleepable that represent drink and sleep capability, respectively.**
- Create a concrete class Bedroll and Bottle that extend item abstract class that represent item bedroll and bottle, respectively.
- Create a concrete class SleepAction and DrinkAction that extend Action class that represent sleep action and drink action, respectively.
- Add dependency line from each Bedroll and Bottle to SleepAction and DrinkAction, respectively
- Add an association line from SleepAction and DrinkAction to Sleepable and Drinkable, respectively

- **Create the PlayerAlive and Sleeping class to represent statuses, The PlayerAlive class represent a status that the player is alive (where their HydrationLevel, WarmthLevel, and HealthPoints are all above 0)**
- PlayerAlive class is used to automatically reduces HydrationLevel and WarmthLevel in each ticks in normal condition.

Pros	Cons
More flexible for future changes and loose coupling, specifically when there is more than one item that can use DrinkAction or SleepAction but with different effect. The interface Drinkable and Sleepable allows the object (the item) to define the effect of each action independently.	It can be considered overengineered if there will not be many modifications or updates in the future.
Reduce repetition: This adheres to Dont Repeat Yourself and dependency inversion principle as whenever you want to add an item that can do similar action with other item but different effect, you don't need to create another similar concrete action class (in this case, if I want to add a cup item, I dont need to create another DrinkAction class with only a different effect)	More complex as it increases dependency
As a product of the two pros above, this design can reduce the number of classes	
Adding a class to represent status also adheres to Single Responsibility Principle, because the Player class can only represent the player object itself, while the condition of what happened to the player in each turn can be represented by other classes.	

Alternative Design (Without Drinkable and Sleepable interface and Sleeping and PlayerAlive Status):

- Create a concrete class Bedroll and Bottle that extend item abstract class that represent item bedroll and bottle, respectively.
- Create a concrete class SleepAction and DrinkAction that extend Action class that represent sleep action and drink action, respectively.
- Add dependency line from each Bedroll and Bottle to SleepAction and DrinkAction, respectively.
- Add an association line from SleepAction and DrinkAction to Bedroll and Bottle, respectively.

Pros	Cons
Simpler design for the current purpose.	Tight coupling and too rigid, will need to add another action class if another object/item is added with similar action but different effect. This violates Don't Repeat Yourself principle and dependency inversion principle.
Less dependency and easier debugging.	Similarly, Without the PlayerAlive Status, automatic reduction of HydrationLevel and WarmthLevel will need to be written in PlayTurn Method in the Player class, which is not ideal, as if we have a new similar class such as animals that have similar characteristic, it will violates the Don't Repeat Yourself principle, as it needs to implement the same code in this new class.

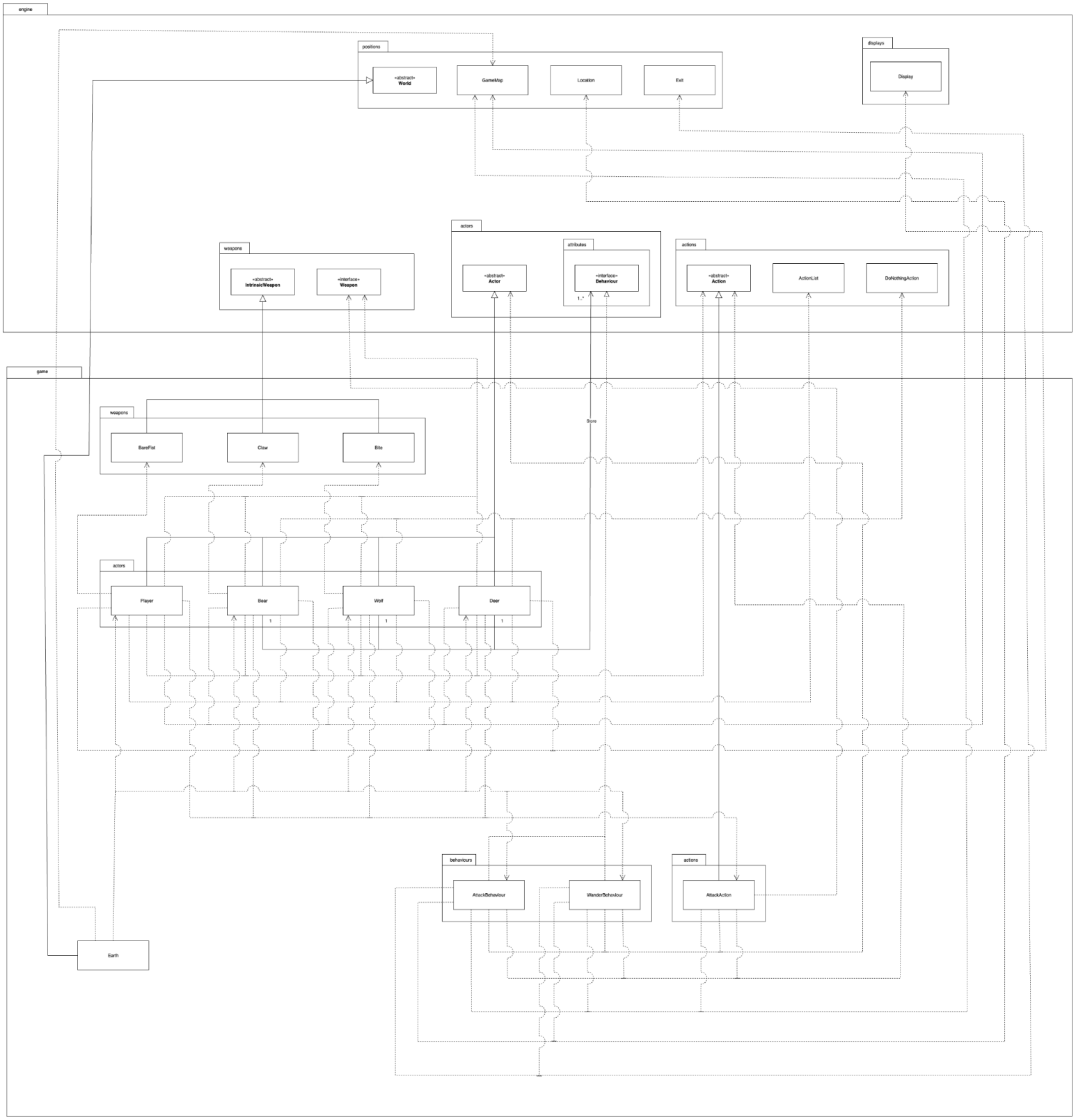
The design that is chosen is the main design with the Drinkable and Sleepable interface:

This design is more flexible toward future changes and it is loose coupling, specifically when there is more than one item that can use DrinkAction or SleepAction but with different effect. The interface Drinkable and Sleepable allows the object (the item) to define the effect of each action independently. For example we want to add another item cup, which if drunk from, will increase thirst level by 2, this effect differs from the bottle effect, which increase thirst level by 4. Using interface Drinkable allows the cup/bottle to define their effect independently. This adheres to Dont Repeat Yourself and Dependency Inversion principles as whenever you want to add an item that can do similar action with other item, you don't need to create another concrete class (in this case, if I want to add a cup item, I dont need to create another DrinkAction class with only a different effect), as a result, it can reduces the number of classes and will not grow too big, therefore making the current design flexible and can easily be updated/extend.

Similarly, the PlayerAlive Status allows automatic reduction of HydrationLevel and WarmthLevel, this allows future flexibility if we want to add actor that have similar characteristic (HydrationLevel and warmthLevel), we just need to add the PlayerAlive status. Therefore it adheres to Don't Repeat Yourself principle.

REQ 2

Main UML Class Diagram:



Design Rationale :

Main Design (The above UML Class Diagram) (Without capabilities interface such as Drinkable and Sleepable):

- Create Bite and Claw class that extends IntrisicWeapon abstract class to represent Wolf and Bear Intrinsic weapon, respectively.
- Create Bear, Wolf, and Dear that extends Actor abstract class to represent the Bear, Wolf, and Dear object, respectively
- Create AttackBehaviour and WanderBehaviour that implements Behaviour interface to represent the wander behaviour and attack behaviour of the non-Player acto (Bear, Wolf, and Deer).
- Add association line from non-Player object/class to Behaviour interface, because non-Player class needs to store the behaviour that they can do.

Pros	Cons
Make use of existing code (which is IntrinsicWeapon class and Weapon interface) which already has the implementation.	
The existing code is flexible enough, that it can be used by many intrinsic weapon object with different effect. This adheres to Don't Repeat Yourself principle, as you don't need to implement interface method in each intrinsic weapon object (Like Bottle that implements Drinkable interface)	

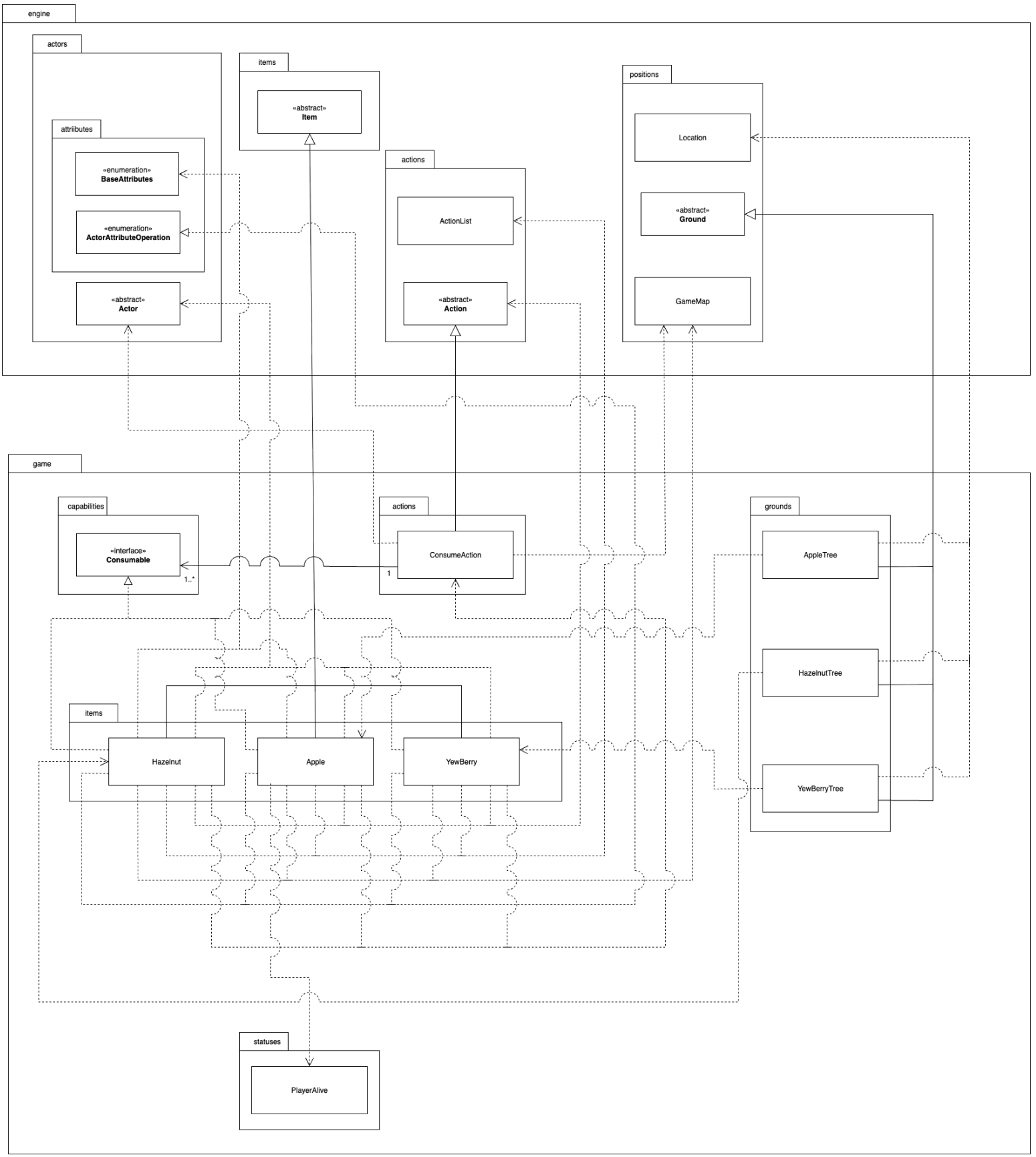
Alternative Design (With capabilities interface such as Drinkable and Sleepable):

- Create Bite and Claw class that extends IntrisicWeapon abstract class to represent Wolf and Bear Intrinsic weapon, respectively.
- **Create Bear, Wolf, and Dear that extends Actor abstract class and implement Attackable interface to represent the Bear, Wolf, and Dear object, respectively**
- Create AttackBehaviour and WanderBehaviour that implements Behaviour interface to represent the wander behaviour and attack behaviour of the non-Player acto (Bear, Wolf, and Deer).
- **Create an Attackable Interface as a contract for the attack implementation that implements this interface.**
- Add association line from non-Player object/class to Behaviour interface, because non-Player class needs to store the behaviour that they can do.
- **Add association line from AttackAction to Attackable interface.**

Pros	Cons
More flexibility, in term of major future updates (Future non-player Actor that totally have different attack effect/new effect). Attackable interface can handle this without violating open-closed principle (still adheres to the open-closed	Unnecesarry for the current requirement, as there are no hint that there will be non-player Actor that totally have different attack effect/new effect.

REQ 3

Main UML Class Diagram:



Design Rationale :

Main Design (The above UML Class Diagram) (With Consumable Interface):

- **Create interface consumable that represent consume capabilities, this interface is used a signature for classes that has this capability.**
- Create ConsumeAction that represent the execution of the implementation of consumable interface method.
- Create Hazelnut, Apple, and YewBerry class that extends Item abstract class. This is used to represent Hazelnut, Apple, and YewBerry, respectively.
- Create HazelnutTree, AppleTree, and YewBerryTree class that extends Ground abstract class. This is used to represent Hazelnut Tree, Apple Tree, and YewBerry Tree, respectively. Each tree drops their respective fruits within a certai time period.

Pros	Cons
More flexible for future changes and loose coupling, specifically when there is more than one item that can use ConsumeAction but with different effect. The interface Consumable allows the object (the item/fruits) to define the effect of each action independently.	It can be considered overengineered if there will not be many modifications or updates in the future.
Reduce repetition: This adheres to Dont Repeat Yourself as whenever you want to add an item that can do similar action with other item but different effect, you don't need to create another similar concrete action class (in this case, if I want to add a cup item, I dont need to create another DrinkAction class with only a different effect)	More complex as it increases dependency
Adheres to Dependency inversion principle as a ConsumeAction will only have an association with the consumable interface (Abtract), this means that you can extend the code (Adding new consumable item) without changing the action class.	
As a product of the two pros above, this design can reduce the number of classes	

Alternative Design (Without capabilities interface Consumable):

- Create ConsumeAction that represent the execution of the implementation of consumable interface method.
- Create Hazelnut, Apple, and YewBerry class that extends Item abstract class. This is used to represent Hazelnut, Apple, and YewBerry, respectively.

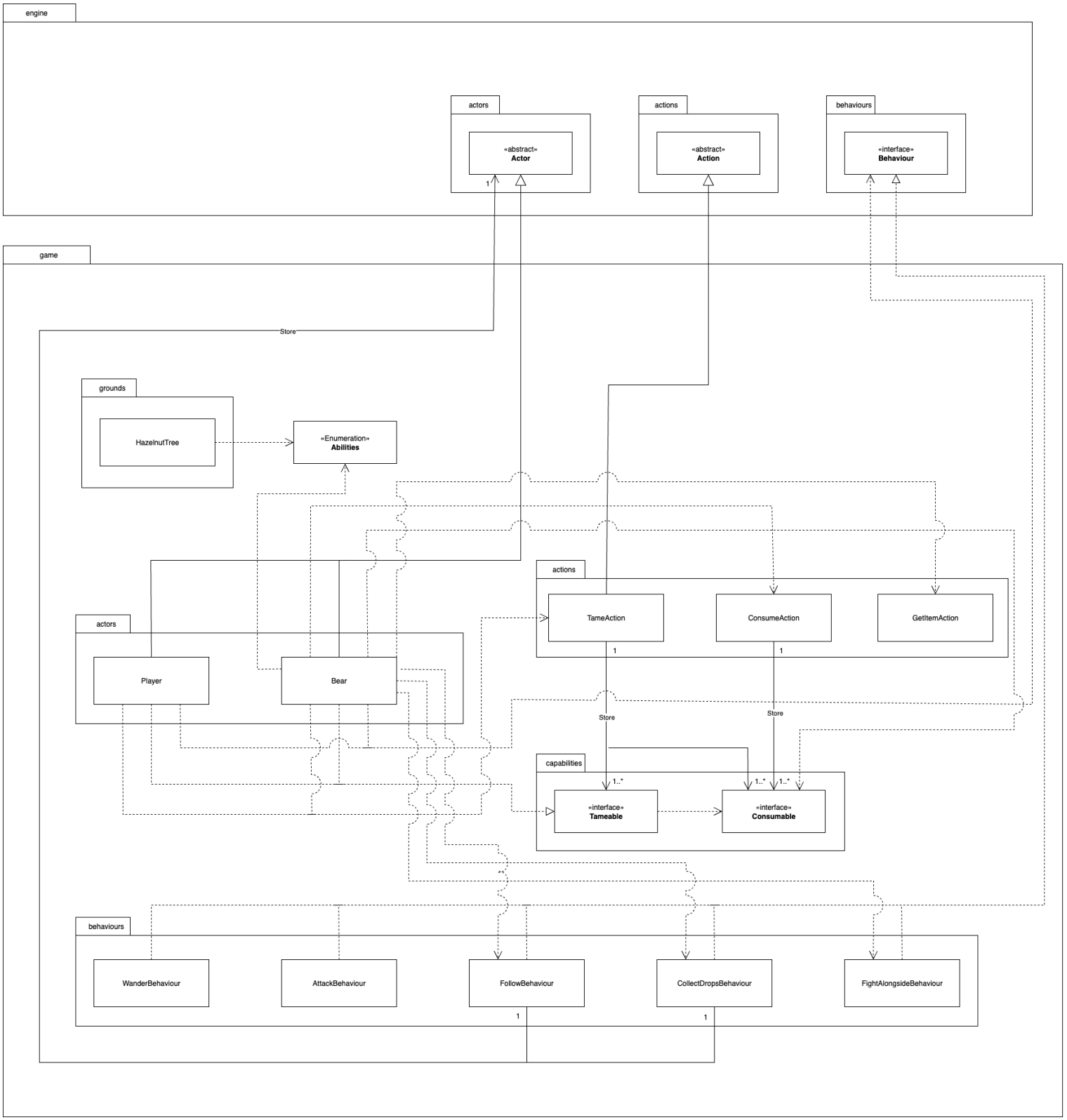
- Create HazelnutTree, AppleTree, and YewBerryTree class that extends Ground abstract class. This is used to represent Hazelnut Tree, Apple Tree, and YewBerry Tree, respectively. Each tree drops their respective fruits within a certai time period.
-

Pros	Cons
More simpler (less complex) can easily be understand	Violates the Don't Repeat Yourself principle, as it will need to create an action class for every item that can be consumed that have different effect
	Violates Dependency inversion principle, as every consume action class will have an association with their respective item class. Concrete class will have an association with another concrete class, this is inefficient because this means that it will be harder to extend the code, as you need to modify and debug the current code when extending it
	Harder to extend and modify, because this means that whenever we want to extend (add a new consumable item), we will need to add many more related class.

-

REQ 4

Main UML Class Diagram:



Design Rationale :

Main Design (The above UML Class Diagram) (With Tameable Interface and GetItemAction):

- **Create interface Tameable that represent tame capabilities, this interface is used as a signature for classes that have this capability.**
- **Create a GetItemAction that extends Action abstract class, this is used to represent get item where the tamed beast/non-player previously picked up from tamed beast/non-player actor action.**
- Create TameAction that extends Action abstract class, this is used to represent tame action that the player/explorer can do to a non-player actor. In this case the explorer can only tame a bear using a Hazelnut.
- Create a FightAlongsideBehaviour class that implements Behaviour interface. This is used to generate action for the tamed beast/non-player actor to fight anyone in their exits except for their owner.
- Create a CollectDropsBehaviour class that implements Behaviour interface. This is used to generate action for the tamed beast/non-player actor to collects drops, in this case the bear can only collect Hazelnut, because they love it.
- Create a FollowBehaviour class that implements Behaviour interface. This is used to generate action for the tamed beast/non-player actor to follow their owner.
- Add CAN_TAME enum in the Abilities Enumeration, to represent ability where a player can tame another non-player actor that is tameable.
- Add TAME_BEAR enum in the Abilities Enumeration, to represent ability of an item that can be used to tame a bear.

Pros	Cons
More flexible for future changes and loose coupling, specifically when there is more than one item that can use TameAction but with different behaviour. The interface Tameable allows the object (the beast) to define the behaviour of each action independently.	It can be considered overengineered if there will not be many modifications or updates in the future.
Reduce repetition: This adheres to Dont Repeat Yourself as whenever you want to add a tameable beast that can do similar action with other tameable beast but different behaviour, you don't need to create another similar concrete action class (in this case.	More complex as it increases dependency
Adheres to Dependency inversion principle as a TameAction will only have an association with the tameable interface (Abtract), this means that you can extend	

the code (Adding new tameable beast) without changing the action class.	
As a product of the two pros above, this design can reduce the number of classes	
Adheres to Single Responsibility Principle, as in this design, the CollectDropsBehaviour only responsible for collecting the drops that the tamed beast encounter, but is not responsible for giving it to their owner, the GetItem action is responsible for giving it to their owner, it is its sole responsibility	

Alternative Design (The above UML Class Diagram) (Without Tameable Interface and GetItemAction):

- Create TameAction that extends Action abstract class, this is used to represent tame action that the player/explorer can do to a non-player actor. In this case the explorer can only tame a bear using a Hazelnut.
- Create a FightAlongsideBehaviour class that implements Behaviour interface. This is used to generate action for the tamed beast/non-player actor to fight anyone in their exits except for their owner.
- Create a CollectDropsBehaviour class that implements Behaviour interface. This is used to generate action for the tamed beast/non-player actor to collects drops, in this case the bear can only collect Hazelnut, because they love it.
- Create a FollowBehaviour class that implements Behaviour interface. This is used to generate action for the tamed beast/non-player actor to follow their owner.
- Add CAN_TAME enum in the Abilities Enumeration, to represent ability where a player can tame another non-player actor that is tameable.
- Add TAME_BEAR enum in the Abilities Enumeration, to represent ability of an item that can be used to tame a bear.

Pros	Cons
More simpler and have less dependency for the current game requirement	Violates the Don’t Repeat Yourself principle, as it will need to create an action class for every beast/Non-player actor that can be tamed that have different behaviour
	Violates Dependency inversion principle, as every Tame action class will have an association with their respective beast/non-player actor class. Concrete class will have an association with another concrete class, this is inefficient because

	this means that it will be harder to extend the code, as you need to modify and debug the current code when extending it
	Violates the Single Responsibility principle, because the CollectDropsBehaviour class will need to handle/be responsible for two action which are, pickup the drops action and give it to the owner action.
	Harder to extend and modify, because this means that whenever we want to extend (add a new tameable beast), we will need to add many more related class.