Design Rationale

By: MA_AppliedSession1_Group3

<u>REQ 1: The Ancient Woods</u>

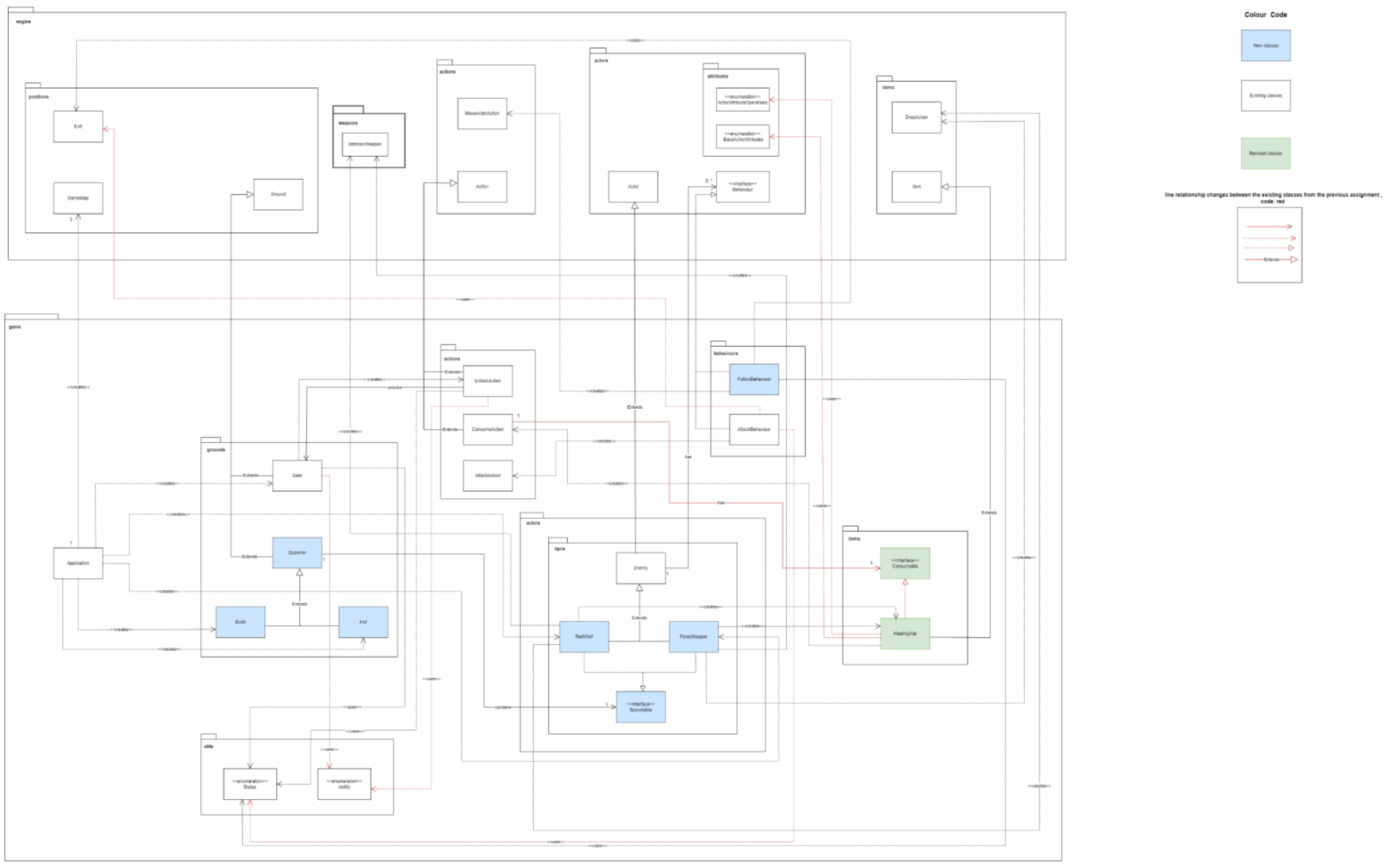


Figure 1: UML class diagram of REQ1 - The Ancient Woods

The provided diagram shows the entire system implementation for Requirement 1, The Ancient Woods. The primary goal of this design is to create a new game map, which is known as Ancient Woods, a gate to travel to the new game map, creating the Bush and Hut in the Ancient Woods to spawn new types of enemies, which are Forest Keeper and Red Wolf at every turn of the game, dropping healing vial item by an enemy once it is defeated. The Red Wolf and Forest Keeper follow the Player until either is unconscious. Thus, new classes are introduced in the extended system: ForestKeeper class, RedWolf class, Bush, Hut, an abstract class, Spawner, an interface class, Spawnable, and FollowBehaviour class. Those newly created classes will interact with the existing classes in the system.

To achieve the goal of the design, we decided to introduce a new Spawner class as an abstract class that extends the abstract Ground class from the engine. All the grounds in the game that can spawn different enemies during each turn will extend this Spawner abstract class because they share common attributes and methods. Grounds in the current implementation, Bush and Hut, will extend this Spawner abstract class as both the Bush class and Hut class can spawn enemies. Since we treated both Bush and Hut classes as types of ground, and both can spawn different enemies, we plan to create a new abstract Spawner class that extends the abstract Ground class, as we considered the Spawner abstract class as a new type of ground which spawn the enemies. Besides, this approach ensures that when additional requirements are introduced in future game implementations that apply to all spawner classes (ground classes that can spawn enemies), these new features can be added to the abstract Spawner class, preventing code repetition (DRY). Importantly, this approach adheres to the Open/Closed Principle (OCP) since the Spawner abstract class extends the functionality of the abstract Ground class from the engine without modifying existing code, as well as Liskov Substitution Principle (LSP), where a subclass can do anything its base class do and it does not disrupt the behaviour of our system. Additionally, the abstract Spawner class has an association relationship with the Spawnable interface class to spawn different enemies at certain chances during each turn of the game. This design choice reduces multiple dependencies (ReD) on various classes that may be implemented in the future, such as the RedWolf and ForestKeeper classes.

We also introduced two new enemies' classes into the extended system, RedWolf class and ForestKeeper class. Both classes extend with the existing Enemy abstract class, mentioned in the previous assignment 1, since both share common attributes and methods. This design choice aligns with the Don't Repeat Yourself (DRY) principle. Since all enemies, regardless Red Wolf or Forest Keeper in the Ancient Woods, can follow the player, it means they share the same behaviour: follow behaviour. Therefore, we had introduced a new FollowBehaviour class, which implement the Behaviour interface class from the engine. The FollowBehaviour class will handle any follow actor actions that are executed by the enemies in Ancient Woods, so this FollowBehaviour class creates MoveActorAction class to perform follow actor action, indicating there is a dependency relationship between FollowBehaviour class and MoveActorAction class from the engine. Our design is not only can reduce multiple dependencies between enemies and their behaviours, but it also aligns with the Dependency Inversion Principle (DIP) where follow behaviour depends on abstractions instead of concrete implementations. We also have the FollowBehaviour class to have a dependency relationship with the Status enum class to check every actor it encounters before performing any actions. Therefore, Red Wolf and Forest Keeper can check the status of an actor they want to follow is accurate, which is the Player. Alternatively, if the Status enum class does not be used, we will need to implement if-else statement and instanceof operator (checking class type) in FollowBehaviour class to ensure that enemies follow the which involves extra dependencies and violates the Open/Closed Principle (OCP) because when new enemies are added, the existing code of the

FollowBehaviour class have to be modified to ensure the condition of enemies not following other actors except the Player holds true at all times. Besides, the FollowBehaviour class has dependency with the Exit class to determine its surrounding contains any actors. In addition, the RedWolf and ForestKeeper classes also create (dependency relationship) DropAction and HealingVial since they will drop healing vial once defeated.

Moreover, the Spawnable interface class, which will be implemented by RedWolf and ForestKeeper classes, is introduced. We decided to design Spawnable as an interface class as it stores a collection of abstract methods for spawning new enemies itself to be implemented in the Enemy subclasses in the later implementation. Instead of defining an abstract method to spawn itself in the Enemy abstract class, which is potentially leading to unnecessary implementations in some subclasses, we can have Enemy subclasses that can spawn itself, which are introduced in future game implementations, can implement it. This design decision aligns with the Interface Segregation Principle (ISP), where the classes that implement this Spawnable interface only need interface methods. This approach also makes our system more extensible and maintainable.

The Application class creates (dependency relationship) a new GameMap class and Gate class, so the Application class has a dependency relationship with the GameMap class and the Gate class. Besides, the Application class also creates (dependency relationship) the Bush class and Hut class as well as the instance of RedWolf and ForestKeeper class.

On the other hand, our design may increase the system's complexity as many new classes are introduced and increase the developers' difficulty if any errors occur during development.

<u>Modification done on classes created in Assignment 1 for REQ 1</u>

A few changes have been made to classes from the previous Assignment 1 interacting with new classes in the requirement 1, specifically, the Healing Vial class and the Consumable interface class. In the last assignment, we have Consumable class as an abstract class. To make our system more extendibility and satisfy this requirement, we decide to modify Consumable class to an interface class. This interface will be implemented by classes that only need to be consumed. This aligns with the Interface Segregation Principle (ISP). Besides, HealingVial class that extends from Consumable abstract class previously will now be implementing the Consumable interface class since it is a consumable item. HealingVial class will no longer extend from the previous Consumable abstract class, now it extends the Item abstract class from the engine directly.

Design Rationale

By: MA_AppliedSession1_Group3

<u>REQ 2: Deeper into the woods</u>

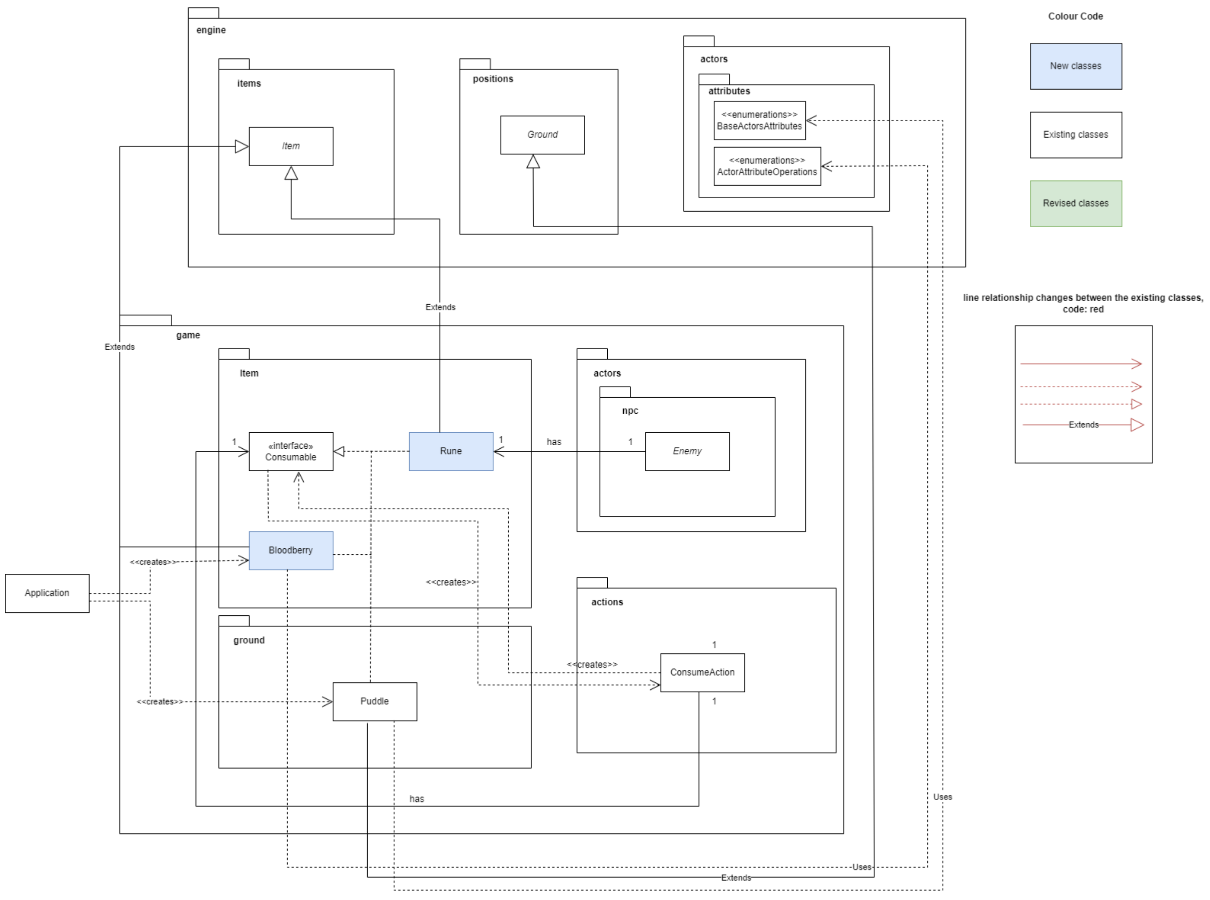


Figure 1: UML class diagram of REQ2: Deeper into the woods

In requirement 2, the “Rune”, “Bloodberry”, and the revised “Puddle” classes are created to follow the Single Responsibility Principle of the SOLID principle, which is only responsible for consumption. For example, “Rune” class focuses on adding the balance of the player, “Bloodberry” class focuses on increasing the maximum health of the player, “Puddle” class focuses on healing and increasing stamina of the player.

“Rune” and “Bloodberry” extend the “Item” class to perform polymorphism (“Puddle” extends “Ground” class), and hence prevent code smells in the design. Besides, these three classes implement the interface “Consumable”, this aligns with the Open/Closed Principle as these classes must utilise the “consume” method in the “Consumable” interface, while the extension for the “consume” method is allowable. This means that these classes might have different implementations with the “consume” method. Also, with OCP, it ensures the extensibility of the programs, for example, if a new item/ground which can be consumed by the player is added, it can implement the “Consumable” interface.

The implementation of the “Consumable” interface and extension of the “Item” abstract class also aligns with the Liskov Substitution Principle because the “Rune” “Bloodberry”, and “Puddle” classes can perform the same functionalities expected from the abstraction class and interface. Moreover, the “Consumable” interface is small enough (only contains “consume” method) for the implementation of these classes, it does not violate the Interface Segregation Principle.

With the “Rune” class passed into the constructor of the “Enemy” class, dependency injection is applied. When the subclasses of the “Enemy” class require a “Rune” object, it can be directly created in the argument of the subclasses’s constructor with the value of the “Rune”. For instance: “super("Forest Keeper", '8', 125,new Rune(50));”, where “50” is the value of the “Rune”, this value can be modified according to different subclasses of the “Enemy”. This helps the design follow the dependency inversion principle.

However, if the abstractions (“Consumable” and “Item”) experience any changes, the subclasses will be affected and they might need to implement the changes even if some are unnecessary. This would introduce connascence which will lead to more chances of bugs.

Design Rationale

By: MA_AppliedSession1_Group3

REQ 3: The Isolated Traveller

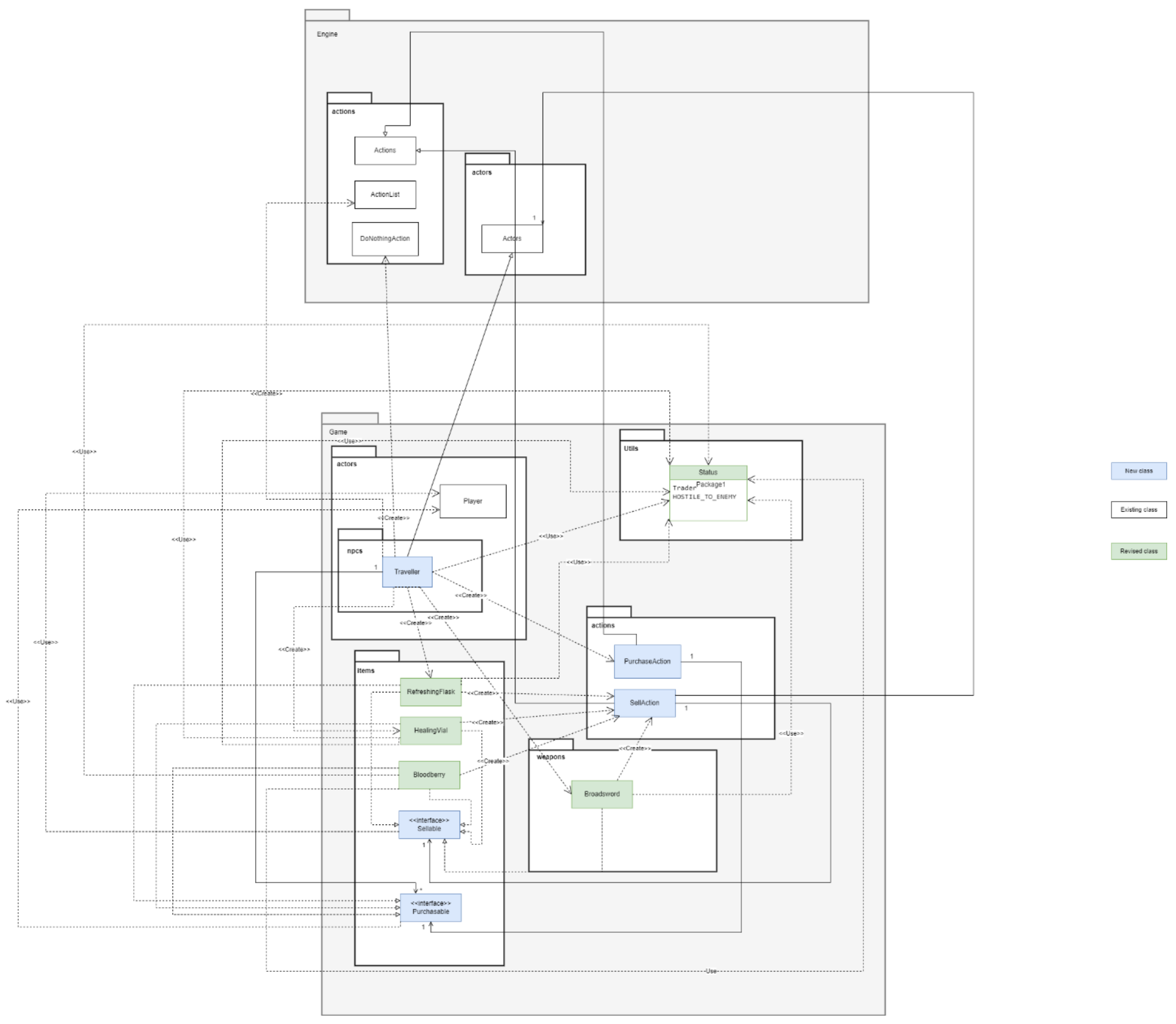


Figure 3: UML class diagram of REQ3 - The Isolated Traveller

The provided diagram shows the entire system implementation for Requirement 3, The Isolated Traveller. The primary goal of this design is to create an actor, which is known as Isolated Traveller, allows the player to purchase items from the traveller's inventory. The player can also sell items from their inventory to the traveller.

Purchasable part: Thus, we have implemented an interface, "Purchasable" for Healing vials, Refreshing flasks, Broadswords which will be purchased by the player. The class which implements getPurchasingPrice() and purchasedBy(Actor actor). Interface Segregation Principle (ISP): The Purchasable interface has a clear and specific purpose: it represents items that can be purchased in a game. This adheres to the ISP, which suggests that interfaces should be client-specific, with only the methods relevant to the client (in this case, items that can be purchased). This aligns with the Open/Closed Principle as these classes must utilize the "purchasedBy(Actor actor)" method in the "Purchasable" interface, while the extension for the "purchasedBy(Actor actor)" method is allowable. This means that these classes might have different implementations with the "purchasedBy(Actor actor)" and "getPurchasingPrice()" method. Moreover, the "Purchasable" interface only contains two methods for the implementation of these classes, it does not violate the Interface Segregation Principle.

PurchaseAction, extends Action class and have a parameter of Purchasable items(the item will be purchased), Encapsulation: The class encapsulates the item to be purchased (Purchasable item) as a private field, ensuring that this data is kept within the class and not directly accessible from outside. The execution of this action relies on the purchasedBy(actor) method of the Purchasable interface to handle the purchase logic. This promotes information hiding and encapsulation.

Traveller, extends Actor, will have an attribute: An ArrayList (purchasables) is used to store the purchasable items. This is an appropriate data structure for holding a collection of items. The configure() method in the Traveller class is responsible for adding purchasable items to the Traveller's inventory, making them available for purchase by other actors. This method is a form of menu configuration, as it sets up the list of items that can be interacted with in the game's trading system. In allowableActions, it will check whether the actor is Player or not, it will add the PurchaseAction to iterate all purchasable items in the purchasables list.

Sellable part: Thus, we have implemented an interface, "Sellable" for Healing vials, Refreshing flasks, Broadsword, Bloodberry which will be sold by player. The concept is similar to the purchasable part, Interface Segregation Principle (ISP) and Open/Closed Principle, Interface Segregation Principle are used. The class which implements getSellingPrice() and soldBy(Actor actor). SellAction is similar to PurchaseAction.

The allowableActions method in the Broadsword class is responsible for determining the actions that can be performed involving the Broadsword when interacting with another actor at a specific location. Thus, if the actor is Trader, the broadsword will create (new SellAction(this)) to list actions to be implemented by the system (using a game engine, like consume).

Code Reusability:

The design promotes code reusability through the use of interfaces and encapsulation. For example, multiple items can implement the Purchasable and Sellable interfaces, allowing for the reuse of common purchase and selling logic.

Modification done on classes created in Assignment 1 for REQ 1

Broadsword: Broadsword extended “SkillWeaponItem” class from A1 because it is unnecessary to create a new class for skill weapon items, we can simply use Ability.SKILL to distinguish which item has skill or not. We make the class less but still work normally.

Design Rationale

By: MA_AppliedSession1_Group3

<u>REQ 4: The room at the end of the forest</u>

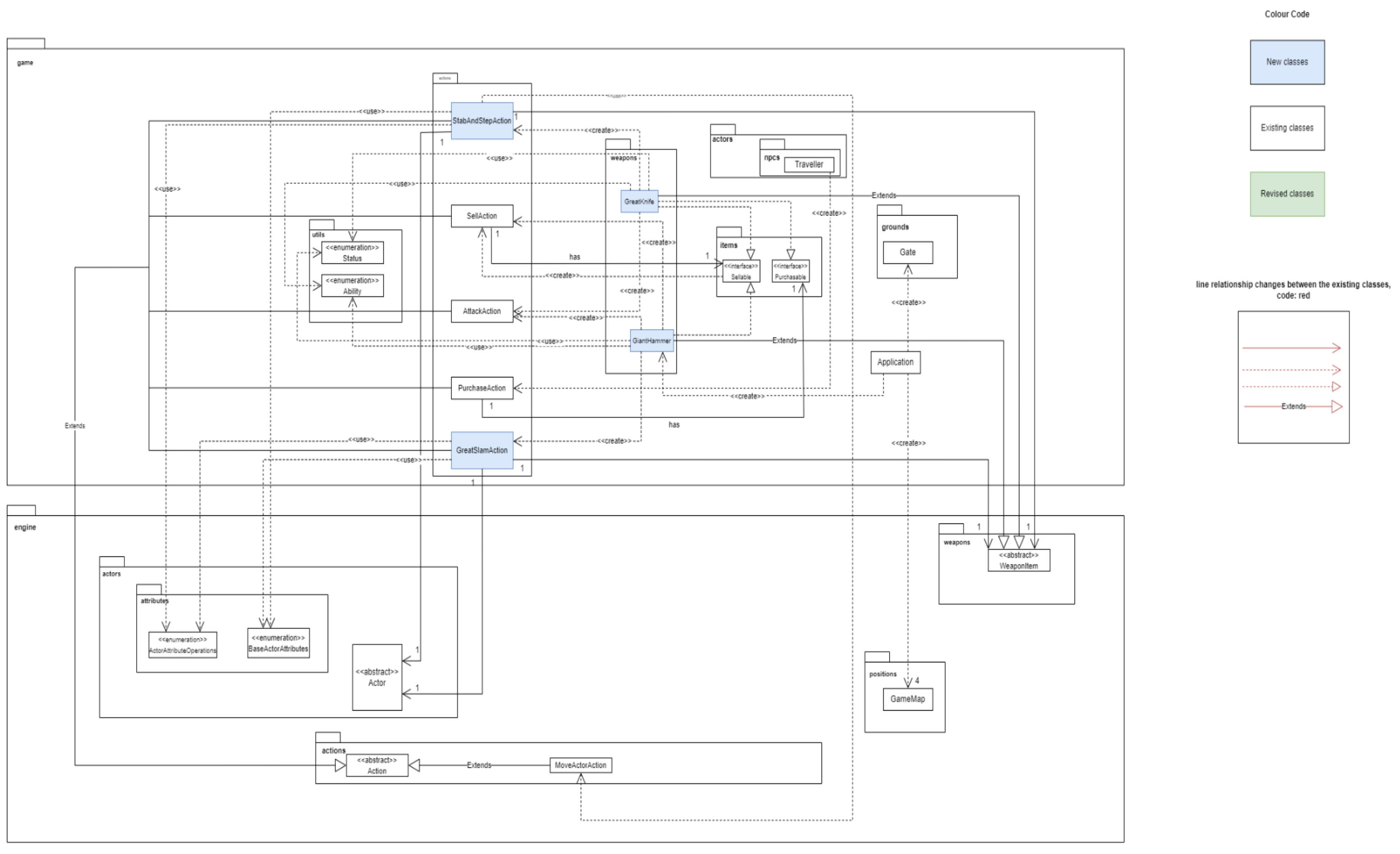


Figure 1: UML class diagram of REQ 4: The room at the end of the forest

The provided diagram shows the entire system implementation for Requirement 4, The Room at the end of the forest. The primary goal of this design is to create a new GameMap, which is Abxveryer Battle Map, and a gate is also created to allow the player to travel to the new game map. A Great Knife, a new weapon offered by the Isolated Traveller with its weapon skill, Stab and Step action, to allow the player to attack enemies and step away to safety within the same turn. A Giant Hammer with its weapon skill, Great Slam action, is created in this requirement and is located in the Abxveryer Battle Map. Both Great Knife and Giant Hammer can be sold, but only Great Knife can be purchased from the traveller. Thus, new classes: GreatKnife class, GiantHammer class, GreatSlamAction class and StabAndStepAction class. These classes will interact with the existing classes in the extended system.

GreatKnife and GiantHammer aren't like Broadsword in that they don't need to take a turn to activate their special skills, but instead they can just apply their specials to the enemy. Players can use these two weapons to attack enemies either normally or with special skills, so we need not only Attack Actions but also Special Skill Actions in our allowableActions(other…) method. For StabAndStepAction and GreatSlamAction, they need to know the target, stamina used and the weapon. These three pieces of information are the attributes of these two action subclasses. It's worth noting that for GreatSlamAction, target refers to the enemy that will be attacked by 100% of the weapon's damage.

Since I made the appropriate adjustments to everyone's code and design prior to this before doing REQ4, it also made my task easy (only implementation not design). Based on the design of REQ3, I just need to have both classes implement the Sellable interface and override the corresponding methods. At the same time, add the SellAction in their respective allowableActions(otherActor...) when the otherActor is TRADER. GreatKnife can be purchased from Trader, so it needs to implement the Purchasable interface and override the corresponding method, and then add this one item to the Traveller's ArrayList.

The core idea inside the execute method of StabAndStepAction is to execute AttackAction followed by MoveActorAction, which is to select the first unoccupied position found. And, for the GreatSlamAction, changing the damage dealt by a weapon really only requires changing the weapon's damage multiplier. First perform an AttackAction to attack the target, then set the damage multiplier to 0.5f, this ensures that the damage of the attack from now on is only 50% of the original damage. To get the “secondary” enemies, we need the main target’s location so that we can get its surroundings. This means that it’s better to store the main target’s location before attacking it. Otherwise, if the main enemy dies after the attack, trying to get the location at that point will report an error because the enemy is no longer in the game. Finally, check the surroundings and perform an AttackAction on every enemy in the surroundings. At this stage, we only care if there's an actor in a position, we don't care if the actor is an enemy or not, because according to the rules, the player himself also takes damage.

We encapsulate the logic required for that specific action in the specific sub action class. This is a good practice of Single Responsibility Principle (SRP). Also, new action can be introduced without affecting others. Changing the implementation of one action class does not affect other action classes. This approach makes the system more robust and easier to extend, thus adhering to OCP.

Design Rationale

By: MA_AppliedSession1_Group3

<u>REQ 5: Abxervyer, The Forest Watcher</u>

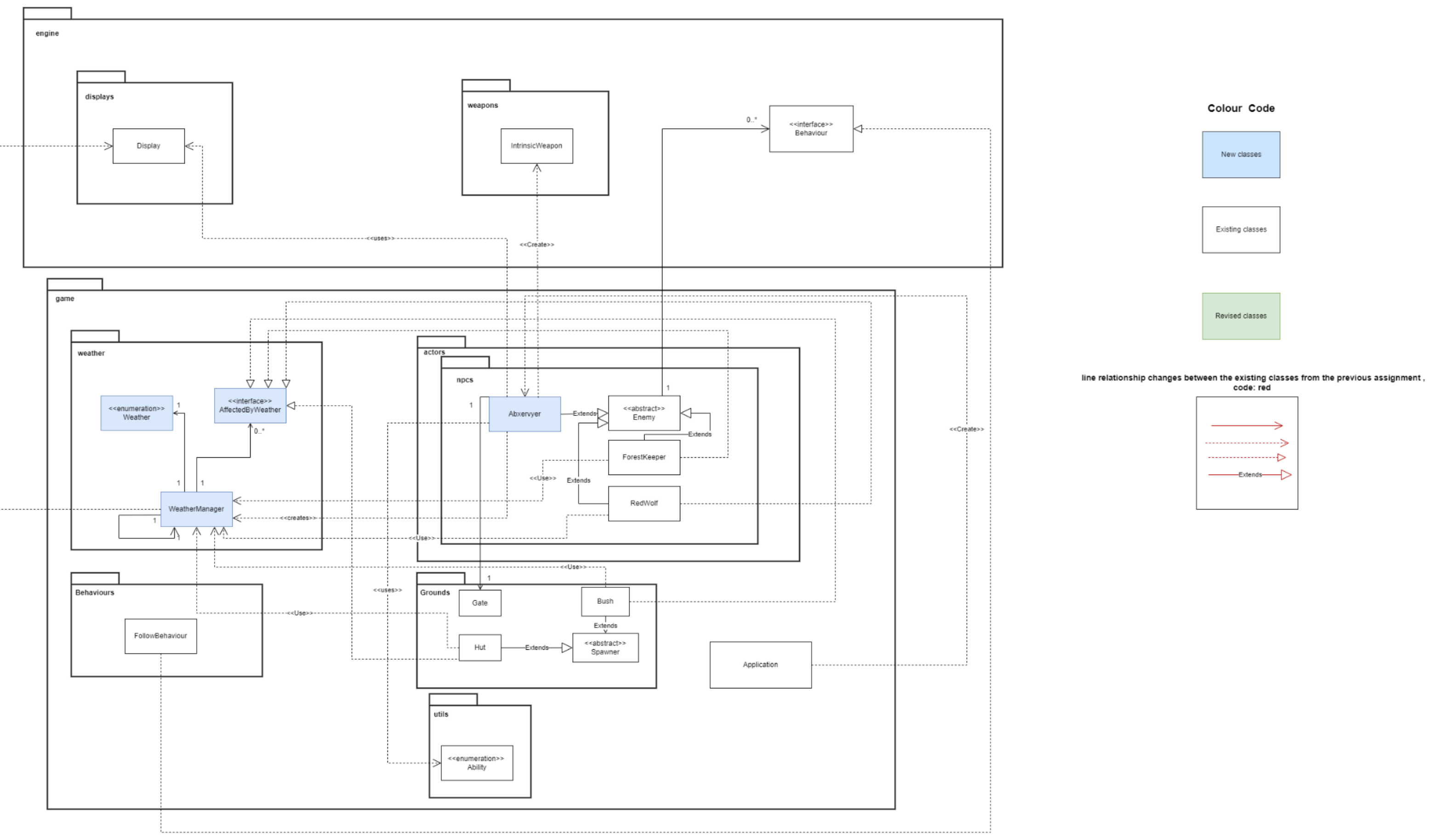


Figure 1: UML class diagram of REQ 5: Abxervyer, The Forest Watcher

The provided diagram shows the entire system implementation for Requirement 5, Abxevyer, The Forest Watcher. The primary goal of this design is to create a new boss enemy, Axervyer, the Forest Watcher. The boss can control weather in the forest, and some grounds such as hut and bush, as well as enemies such as Red Wolf and Forest Keeper will be affected by the weather. Thus, new classes: Abxervyer class,WeatherManager class, Weather class and AffectedByWeather interface class will be created and these classes will interact with the existing classes in the extended. system

Most of the implementation of the boss class is similar to the other enemies and does not require much explanation. It can also follow the player, so it is also necessary to add a FollowBehaviour to its 'behaviours' at construction time. according to the rules, when the boss is defeated, it drops Runes and its current location becomes a gate. The logic for both should be written in the dropItem method, adding Runes (5000) to the boss location, and setting the Ground at this location to Gate (which needs to be specified at boss creation time)

We can add a new static variable "BOSS_DEFEATED" to the FancyMessage class, so we can display this string in the overridden unconscious method of the boss class.

Before that, what all enemies do at the start of their turn can be generalized as "choose what can be done based on priority until an action has been performed." What about on the boss's turn, where things change slightly, as the boss will change the weather, so when the boss starts the turn, it will change the weather first, and then perform what an enemy would do normally. So, the boss needs to override the playTurn method in the Enemy class, and within the method, the first part is to change the weather, and the second part is to call Enemy's playTurn. The boss's turn is randomized and not necessarily right after the player (i.e. the boss isn't necessarily the first enemy that can start the turn), and the weather change is something done at the very beginning of the boss's turn, so it's possible to have both aggressive red wolves and non-aggressive red wolves, etc., in a single turn.

While we can change the spawn rate of Spawner (Ground type) using setSpawnPercentage, we can't go back to the normal rate. Therefore, we need to add a new attribute 'iniSpawnPercentage' to Spawner, and add the corresponding getter method to represent the original spawn rate, i.e., when we want to set the rate back to the original, we just need to setSpawnPercentage(getIniSpawnPercentage()).

The above are the points that our group discussed in a meeting prior to writing REQ5. Then comes the most important part, the implementation related to changing the weather.

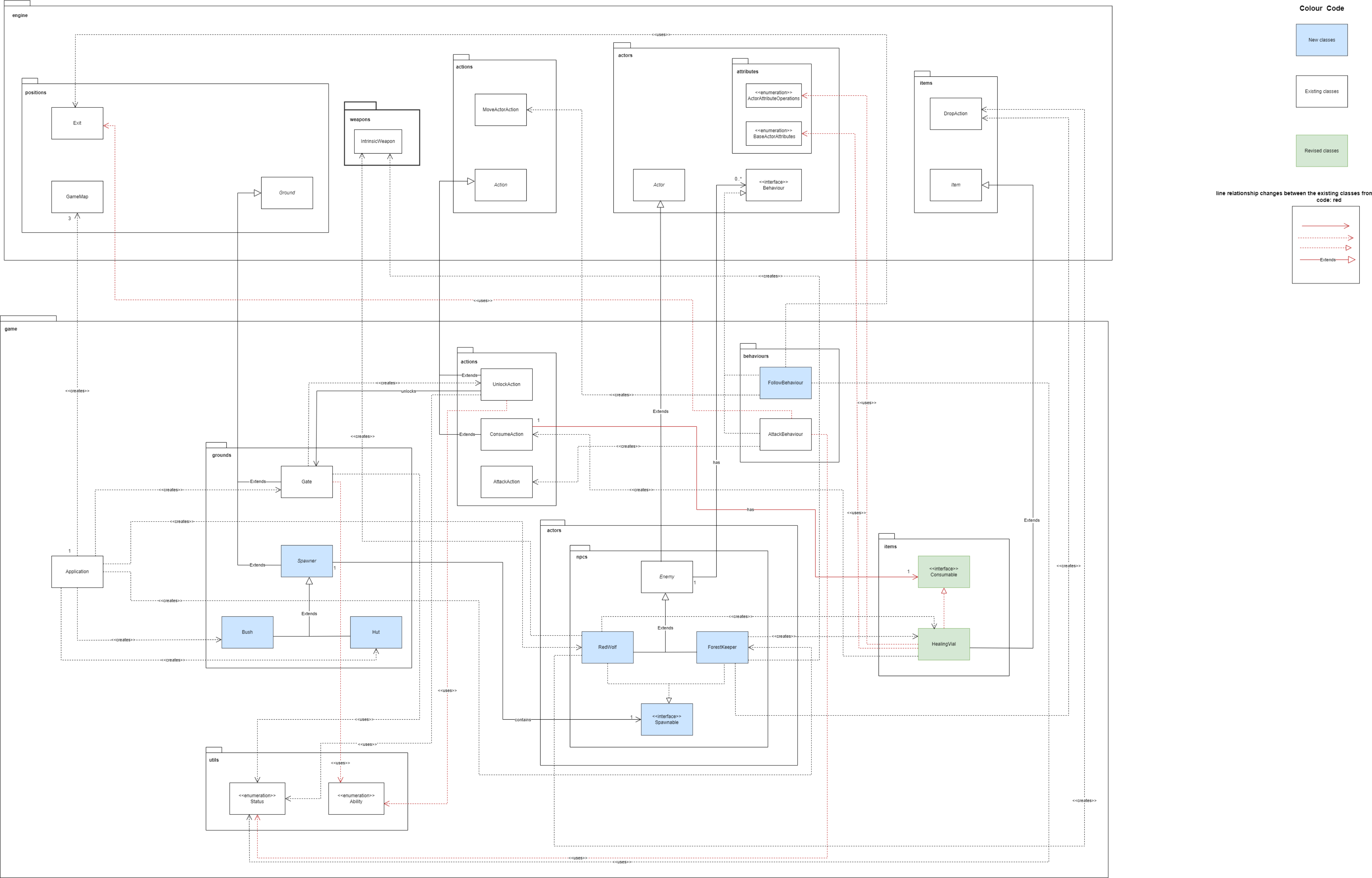
REQ5 indicates that there are two weathers, Sunny and Rainy, and at first, we wanted to use a boolean value for the weather. With a boolean, we’re limited to only two states (true or false). If we ever want to expand the types of weather (e.g., sunny, rainy, snowy), we’ll need to change the data type and refactor the code. And the change will be huge. Also, a boolean value doesn't provide much information about the state itself. It doesn't convey the meaning or context. Thus, instead of using boolean values to toggle the weather, we decided to introduce an enum class to represent the type of weather. Enums allow us to define a set of named constants, which can represent different states or options. This makes it easy to extend and add more weather types in the future. Enums provide a clear and self-documenting way to represent states. We're effectively encapsulating that set of values in one place. This helps to keep the responsibility of managing those constants separate from other parts of our code. Therefore, it follows Single Responsibility Principle (SRP) and Open/Closed Principle (OCP).

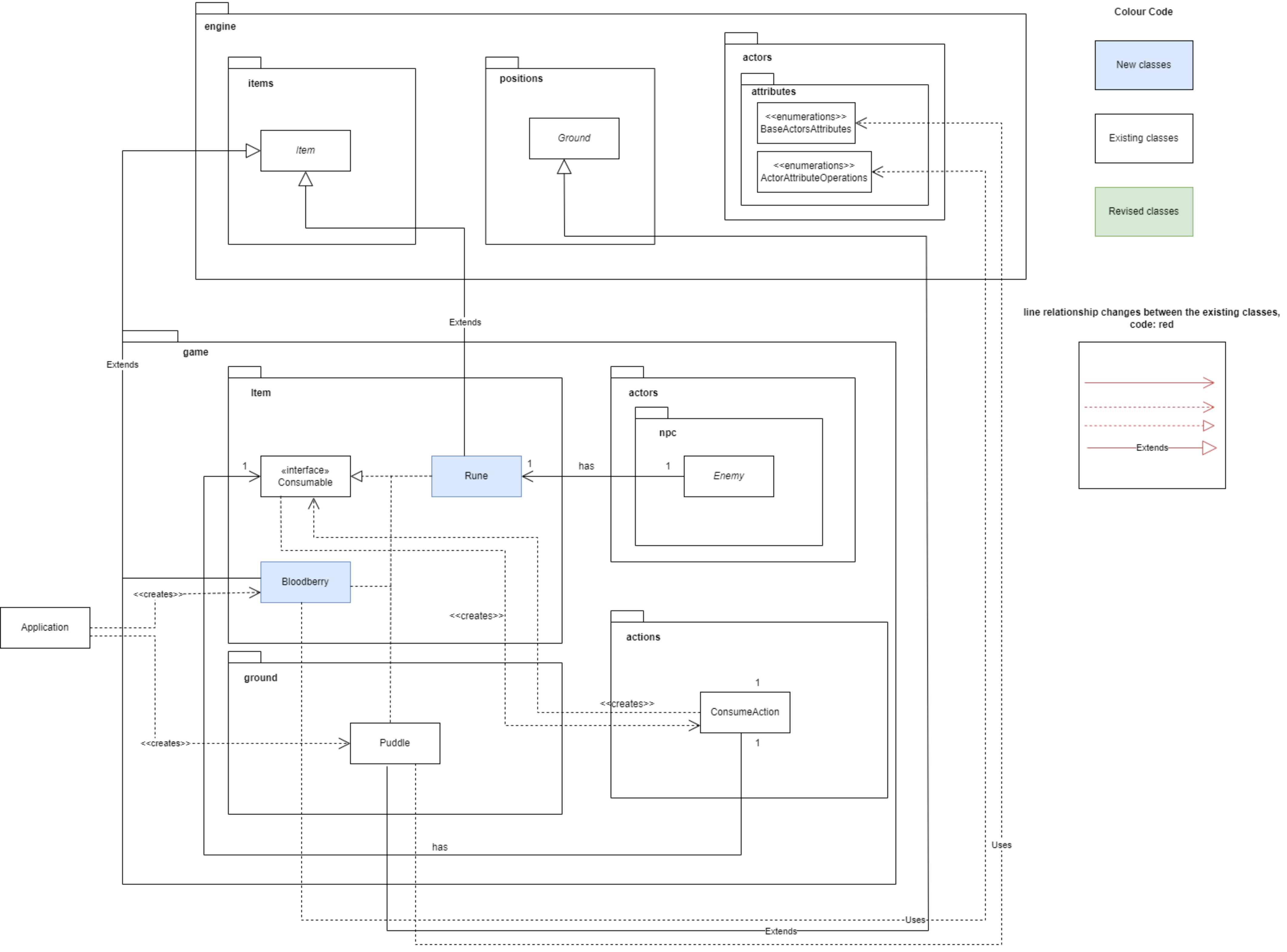
We can't put all the logic for controlling the weather in the boss class, such as keeping track of the number of turns, changing the weather every three turns, and displaying the current weather every turn. By including weather control logic in the boss class, we're increasing its complexity and coupling it with an unrelated responsibility. This can make the code harder to understand and maintain. To address this, we could consider creating a separate WeatherManager class that is responsible for managing the weather and the boss can interact with this manager class. By separating concerns in this way, we adhere to the Single Responsibility Principle.

There is a method run in WeatherManager that affects entities like RedWolf, ForestKeeper, Hut, and Bush according to the current weather. Along this line of thought, we know that the manager needs to be able to access all of the entities that will be affected by the weather. Therefore, we need an interface AffectedByWeather, and all entities that implement this interface are considered to be individuals that are affected by the weather (this interface is equivalent to a marker). Let the WeatherManager class have an ArrayList attribute, a collection that holds all the entities that implement the affectedByWeather interface. In this way, the weather manager will be able to access all affectable entities.

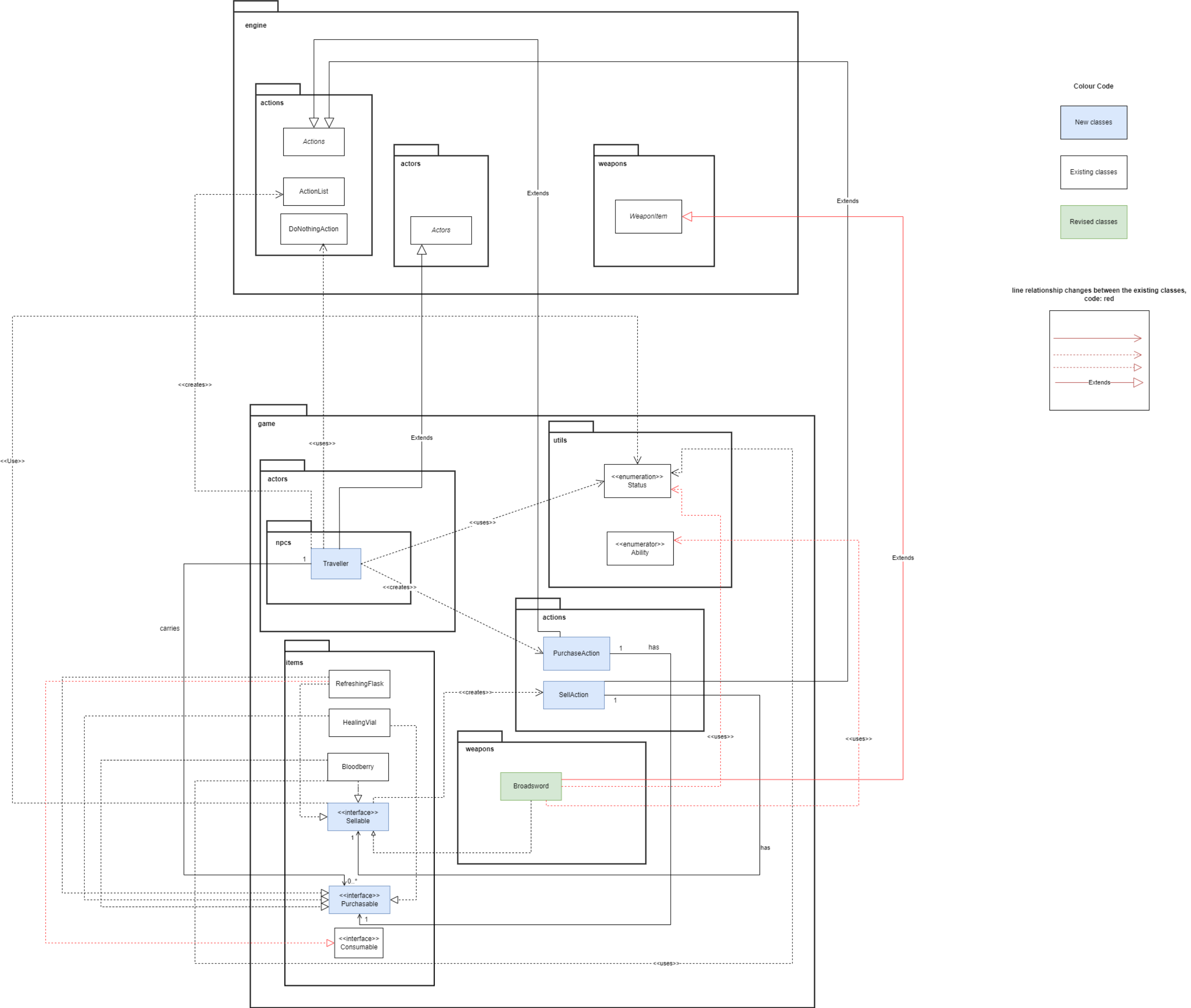
How to add all the objects in the game that implement the AffectedByWeather to this ArrayList? We can have a method, registerWeather(), to add an AffectedByWeather “object” to the ArrayList. And this method can be called in RedWolf’s constructor, ForestKeeper’s constructor, Bush’s constructor and Hut’s constructor. This ensures that whenever a new object is introduced into the game, it will be added to the manager's ArrayList. In order to call register (), we need to create the WeatherManager object, and we need to make sure that we are creating the same object, we need to use the singleton pattern. This can be useful for centralizing weather management and ensuring that all parts of the game are interacting with the same WeatherManager. At this point, we realized that unregisterWeather() is also necessary. When an enemy affected by weather dies, it needs to be removed from the list. And this can be done in the overridden unconscious (actor, map) and unconscious(map) methods.

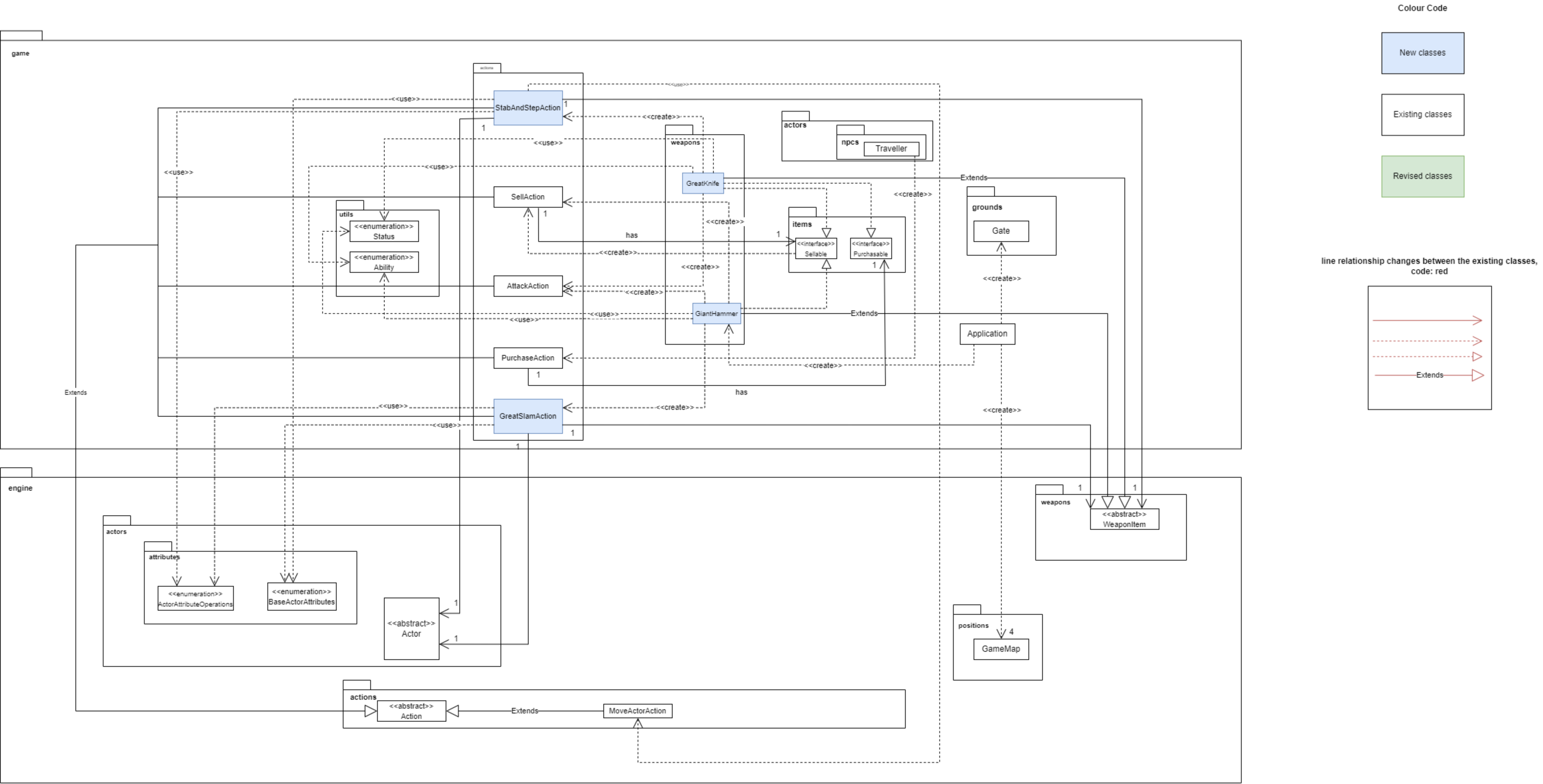
Let’s back to the AffectedByWeather interface, it has two methods, affectedBySunny() and affectedByRainy(), which will be implemented by the Hut class, Bush class, RedWolf class and ForestKeeper class as those classes can This means that all four concrete classes need to override these two methods. Different types of entities can have their own unique reactions to changes in weather. This design allows for easy addition of new entities affected by weather without modifying the WeatherManager class directly. This design aligns with the Open/Closed Principle (OCP) and Interface Segregation Principle (ISP). Thus, our system can lead to more flexible and maintainable code.

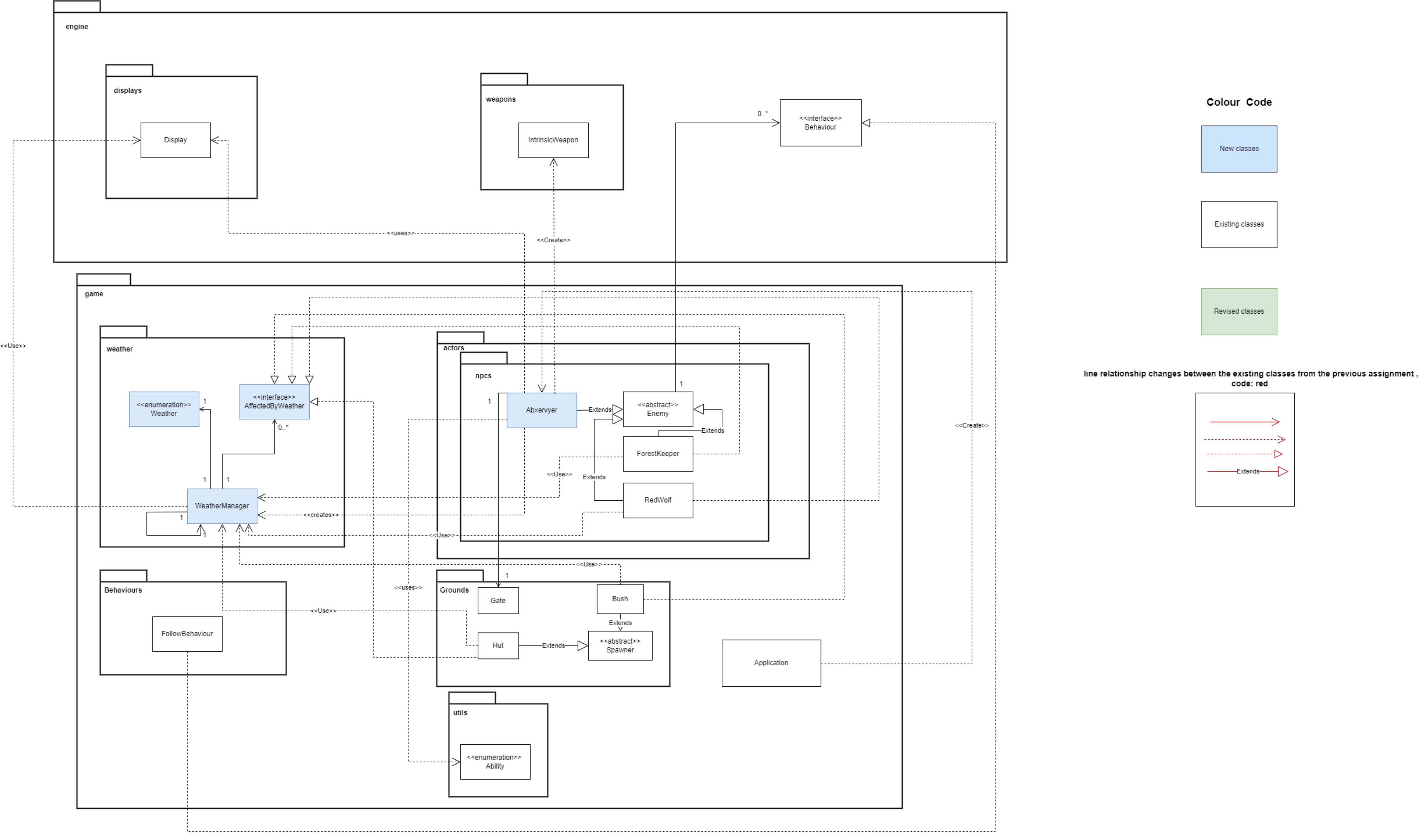

engine
positions
Exit
GameMap
Ground
weapons
IntrinsicWeapon
actions
MoveActorAction
Action
actors
attributes
<<enumeration>> ActorAttributeOperations
<<enumeration>> BaseActorAttributes
Actor
<<interface>> Behaviour
0..*
items
DropAction
Item
game
actions
UnlockAction
ConsumeAction
AttackAction
behaviours
FollowBehaviour
AttackBehaviour
grounds
Gate
Spawner
Bush
Hut
actors
npcs
Enemy
RedWolf
ForestKeeper
<<interface>> Spawnable
items
<<interface>> Consumable
HealingVial
utils
<<enumeration>> Status
<<enumeration>> Ability
Application
<<uses>>
<<creates>>
Extends
unlocks
has
contains
1
3
Colour Code
New classes
Existing classes
Revised classes
line relationship changes between the existing classes from the previous assignment , code: red

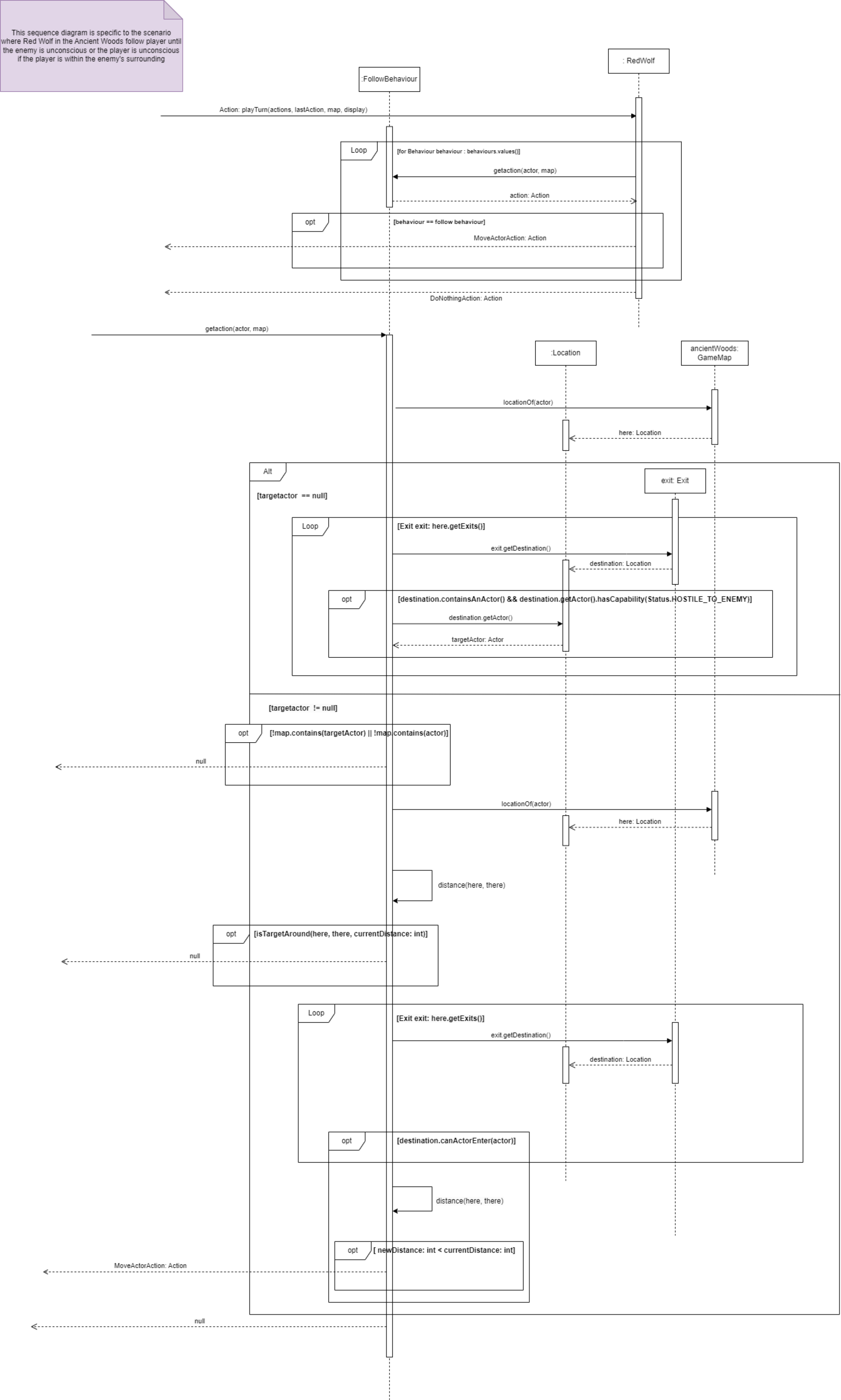


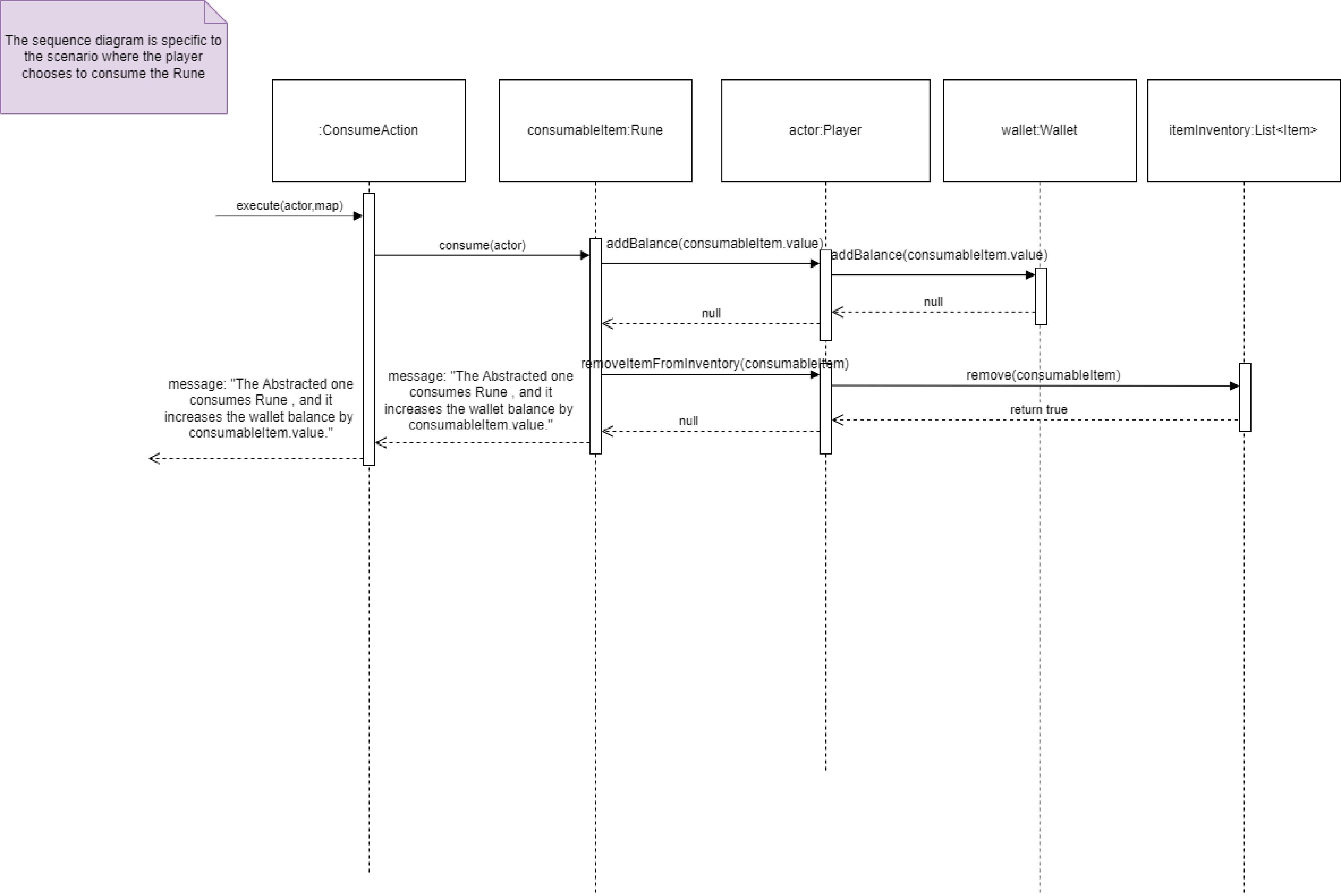
engine
items
Item
positions
Ground
actors
attributes
<<enumerations>> BaseActorsAttributes
<<enumerations>> ActorAttributeOperations
Colour Code
New classes
Existing classes
Revised classes
line relationship changes between the existing classes, code: red
Extends
game
Extends
Item
«interface» Consumable
Rune
1
1
has
1
actors
npc
Enemy
Bloodberry
<<creates>>
<<creates>>
Application
ground
Puddle
<<creates>>
actions
ConsumeAction
<<creates>>
1
1
has
Uses
Uses
Extends


engine
actions
Actions
ActionList
DoNothingAction
actors
Actors
weapons
WeaponItem
Extends
Extends
Extends
Extends
Colour Code
New classes
Existing classes
Revised classes
line relationship changes between the existing classes, code: red
Extends
<<creates>>
<<uses>>
<<Use>>
game
actors
npcs
Traveller
1
carries
utils
<<enumeration>> Status
<<enumerator>> Ability
<<uses>>
<<creates>>
actions
PurchaseAction
1
has
SellAction
1
<<creates>>
<<uses>>
<<uses>>
weapons
Broadsword
items
RefreshingFlask
HealingVial
Bloodberry
<<interface>> Sellable
1
has
0..*
<<interface>> Purchasable
1
<<interface>> Consumable
<<uses>>

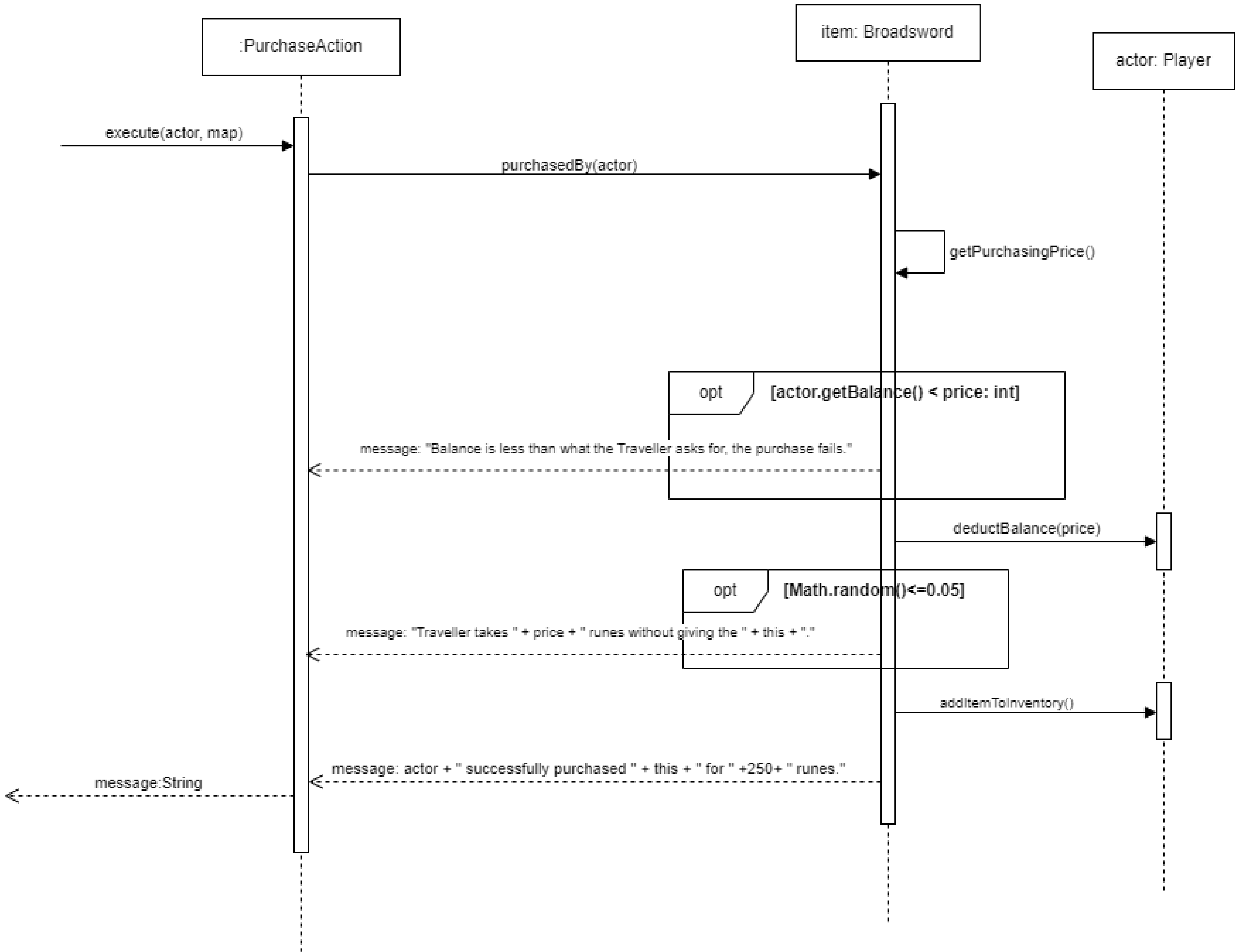

Colour Code
New classes
Existing classes
Revised classes
line relationship changes between the existing classes, code: red
Extends
game
actions
StabAndStepAction
SellAction
AttackAction
PurchaseAction
GreatSlamAction
utils
<<enumeration>> Status
<<enumeration>> Ability
weapons
GreatKnife
GiantHammer
actors
npcs
Traveller
items
<<interface>> Sellable
<<interface>> Purchasable
grounds
Gate
Application
<<use>>
<<create>>
has
Extends
1
4
engine
actors
attributes
<<enumeration>> ActorAttributeOperations
<<enumeration>> BaseActorAttributes
<<abstract>> Actor
actions
<<abstract>> Action
MoveActorAction
weapons
<<abstract>> WeaponItem
positions
GameMap

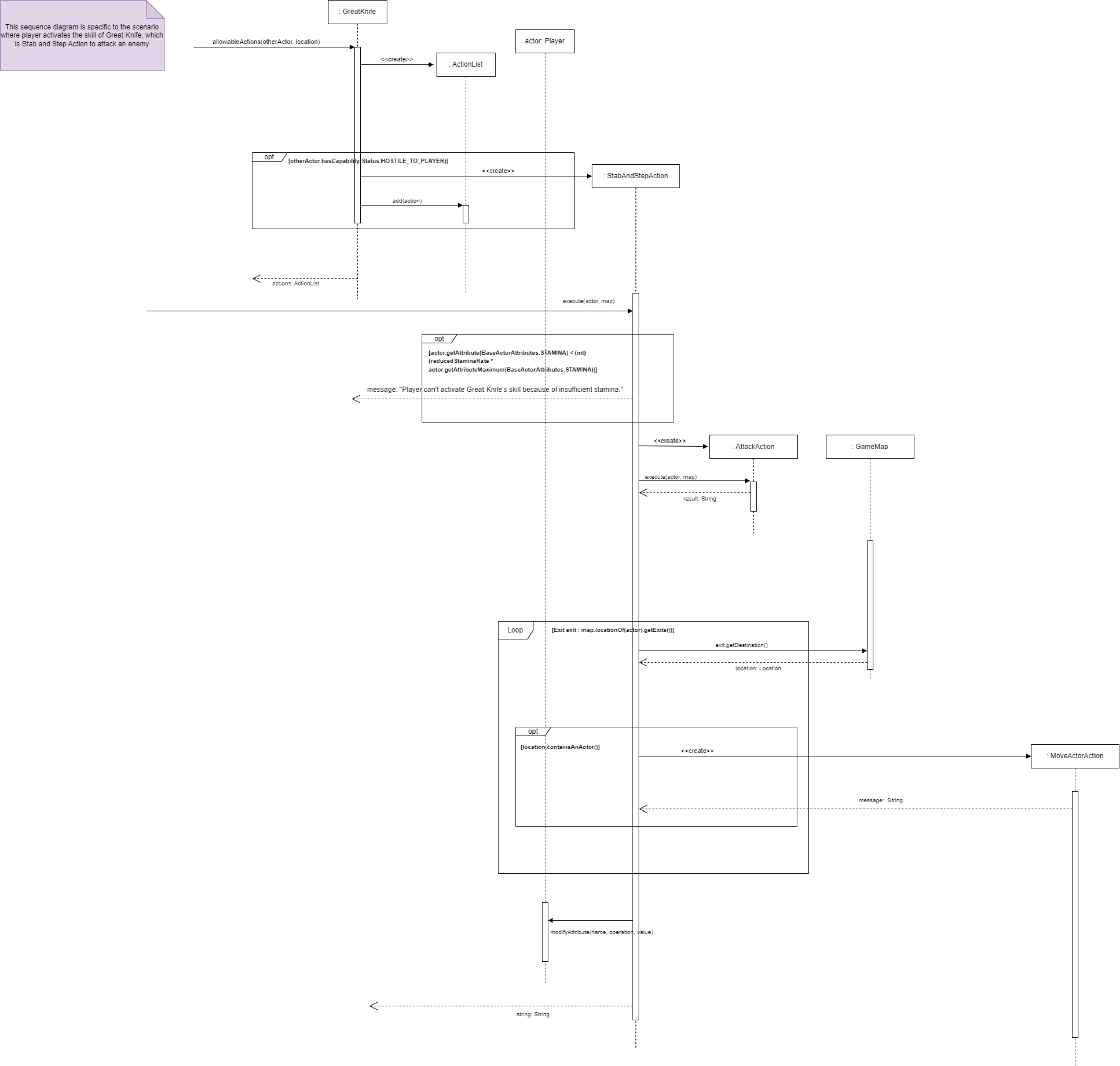

engine
displays
Display
weapons
IntrinsicWeapon
<<interface>> Behaviour
0..*
<<uses>>
<<Create>>
<<Use>>
game
weather
<<enumeration>> Weather
<<interface>> AffectedByWeather
0..*
WeatherManager
1
<<creates>>
actors
npcs
Abxervyer
<<abstract>> Enemy
Extends
ForestKeeper
RedWolf
<<Use>>
<<Create>>
Behaviours
FollowBehaviour
Grounds
Gate
Bush
Hut
<<abstract>> Spawner
<<uses>>
utils
<<enumeration>> Ability
Application
Colour Code
New classes
Existing classes
Revised classes
line relationship changes between the existing classes from the previous assignment , code: red
Extends


This sequence diagram is specific to the scenario where Red Wolf in the Ancient Woods follow player until the enemy is unconscious or the player is unconscious if the player is within the enemy's surrounding
: RedWolf
:FollowBehaviour
Action: playTurn(actions, lastAction, map, display)
Loop
[for Behaviour behaviour : behaviours.values()]
getaction(actor, map)
action: Action
opt
[behaviour == follow behaviour]
MoveActorAction: Action
DoNothingAction: Action
getaction(actor, map)
:Location
ancientWoods: GameMap
locationOf(actor)
here: Location
Alt
exit: Exit
[targetactor == null]
Loop
[Exit exit: here.getExits()]
exit.getDestination()
destination: Location
opt
[destination.containsAnActor() && destination.getActor().hasCapability(Status.HOSTILE_TO_ENEMY)]
destination.getActor()
targetActor: Actor
[targetactor != null]
opt
[!map.contains(targetActor) || !map.contains(actor)]
null
locationOf(actor)
here: Location
distance(here, there)
opt
[isTargetAround(here, there, currentDistance: int)]
null
Loop
[Exit exit: here.getExits()]
exit.getDestination()
destination: Location
opt
[destination.canActorEnter(actor)]
distance(here, there)
opt
[newDistance: int < currentDistance: int]
MoveActorAction: Action
null


The sequence diagram is specific to the scenario where the player chooses to consume the Rune
:ConsumeAction
consumableItem:Rune
actor:Player
wallet:Wallet
itemInventory:List<Item>
execute(actor,map)
consume(actor)
addBalance(consumableItem.value)
addBalance(consumableItem.value)
null
null
removeItemFromInventory(consumableItem)
remove(consumableItem)
return true
null
message: "The Abstracted one consumes Rune , and it increases the wallet balance by consumableItem.value."
message: "The Abstracted one consumes Rune , and it increases the wallet balance by consumableItem.value."

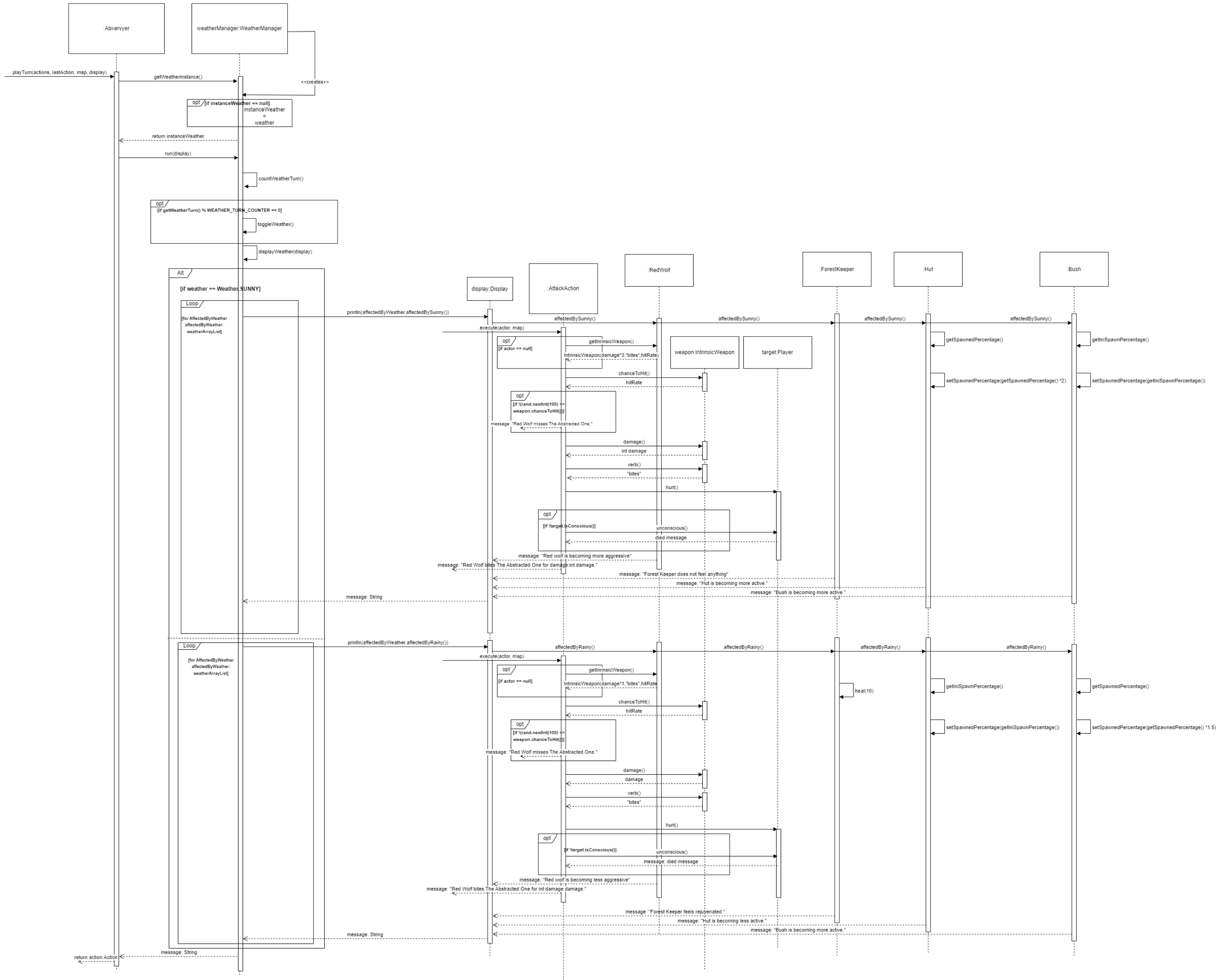
This sequence diagram is specific to the scenario where the player purchases Broadsword from The Isolated Traveller


:PurchaseAction
item: Broadsword
actor: Player
execute(actor, map)
purchasedBy(actor)
getPurchasingPrice()
opt
[actor.getBalance() < price: int]
message: "Balance is less than what the Traveller asks for, the purchase fails."
deductBalance(price)
opt
[Math.random()<=0.05]
message: "Traveller takes " + price + " runes without giving the " + this + "."
addItemToInventory()
message: actor + " successfully purchased " + this + " for " +250+ " runes."
message:String


This sequence diagram is specific to the scenario where player activates the skill of Great Knife, which is Stab and Step Action to attack an enemy

: GreatKnife
actor: Player
: ActionList
allowableActions(otherActor, location)
<<create>>
opt
[otherActor.hasCapability(Status.HOSTILE_TO_PLAYER)]
<<create>>
: StabAndStepAction
add(action)
actions: ActionList
execute(actor, map)
opt
[actor.getAttribute(BaseActorAttributes.STAMINA) < (int)(reducedStaminaRate * actor.getAttributeMaximum(BaseActorAttributes.STAMINA))]
message: "Player can't activate Great Knife's skill because of insufficient stamina."
<<create>>
: AttackAction
: GameMap
execute(actor, map)
result: String
Loop
[Exit exit : map.locationOf(actor).getExits()]
exit.getDestination()
location: Location
opt
[location.containsAnActor()]
<<create>>
: MoveActorAction
message: String
modifyAttribute(name, operation, value)
string: String

The sequence diagram is specific to the scenario where the Abxervyer controls the weather to Sunny or Rainy which affects Hut, Red Wolf, Bush, and the Forest Keeper.


:Abxervyer
weatherManager:WeatherManager
playTurn(actions, lastAction, map, display)
getWeatherInstance()
<<creates>>
opt [if instanceWeather == null]
instanceWeather = weather
return instanceWeather
run(display)
countWeatherTurn()
opt [if getWeatherTurn() % WEATHER_TURN_COUNTER == 0]
toggleWeather()
displayWeather(display)
Alt
[if weather == Weather.SUNNY]
Loop
[for AffectedByWeather affectedByWeather: weatherArrayList]
display:Display
:AttackAction
:RedWolf
:ForestKeeper
:Hut
:Bush
weapon:IntrinsicWeapon
target:Player
println(affectedByWeather.affectedBySunny())
affectedBySunny()
execute(actor, map)
opt [if actor == null]
getInrinsicWeapon()
IntrinsicWeapon(damage*3,"bites",hitRate)
getSpawnedPercentage()
getIniSpawnPercentage()
setSpawnedPercentage(getSpawnedPercentage() *2)
setSpawnedPercentage(getIniSpawnPercentage())
chanceToHit()
hitRate
opt [if !(rand.nextInt(100) <= weapon.chanceToHit())]
message: "Red Wolf misses The Abstracted One."
damage()
int damage
verb()
"bites"
hurt()
opt [if !target.isConscious()]
unconscious()
died message
message: "Red wolf is becoming more aggressive"
message: "Red Wolf bites The Abstracted One for damage:int damage."
message: "Forest Keeper does not feel anything"
message: "Hut is becoming more active."
message: "Bush is becoming more active."
message: String
Loop
[for AffectedByWeather affectedByWeather: weatherArrayList]
println(affectedByWeather.affectedByRainy())
affectedByRainy()
execute(actor, map)
opt [if actor == null]
getInrinsicWeapon()
IntrinsicWeapon(damage*1,"bites",hitRate)
heal(10)
getIniSpawnPercentage()
getSpawnedPercentage()
setSpawnedPercentage(getIniSpawnPercentage())
setSpawnedPercentage(getSpawnedPercentage() *1.5)
chanceToHit()
hitRate
opt [if !(rand.nextInt(100) <= weapon.chanceToHit())]
message: "Red Wolf misses The Abstracted One."
damage()
damage
verb()
"bites"
hurt()
opt [if !target.isConscious()]
unconscious()
message: died message
message: "Red wolf is becoming less aggressive"
message: "Red Wolf bites The Abstracted One for int:damage damage."
message: "Forest Keeper feels rejuvenated."
message: "Hut is becoming less active."
message: "Bush is becoming more active."
message: String
message: String
return action:Action