Design Rationale & Design Diagrams - Assignment 3

Design Rationale

Requirement 1 (Done by Tong Zhi Enn)

Requirement

This requirement includes a new feature that allows Player to travel between different Moons, and a new teleport item that allows Player to teleport to a random location within the map.

The Player can travel to different moons by interacting with the computer terminal on the ship, they are only allowed to choose the travel destination that they are not currently on. (e.g. Player is only given the option to choose Moon B and Moon C if they are currently located on Moon A while interacting with the computer terminal)

The new item THESEUS (‘^’) is now available to purchase for 100 credits on the computer terminal, where this item allows Player to teleport on to a random location within a single map.

Changes to Design

Modified AlienBug class and SuspiciousAstronaut class from Assignment 2 feedback

- ➔ Instead of hardcoding the value of the spawn rate, it is now defined as a constant which makes it easier to update the values without having to search through the entire code.
    - private static final int ALIEN_BUG_SPAWN_PROBABILITY
    - private static final int SUSPICIOUS_ASTRONAUT_SPAWN_PROBABILITY

Added New ability in Ability enum class:

- ➔ TRAVEL
    - To indicate that certain actor(e.g. Player) has the ability to travel to different moons
- ➔ TELEPORT
    - To indicate that certain actor(e.g. Player) has the ability to teleport with a teleport item

Modification in Player class:

- ➔ 2 new HashMap arraylist to store *worldList* and *teleportDestination*
  - Getter methods for both of the attributes is also created
- ➔ Now implements Traveller interface

Modification in ComputerTerminal class:

- ➔ Added a new BuyAction for Theseus
- ➔ Allow actor that has the ability to travel to different moons

Modification in Application class:

- ➔ Added new maps as arrayList and then store them in worldList hashmap and teleportDestination hashmap for easier access
- ➔ Inserted computer terminal onto the map directly

Design Goal

The design choice was to allow Player to travel between moons by interacting with the ComputerTerminal. Each new moon is added as an arrayList respectively. This will lead to having multiple arrayList for every new moon map added, hence, two hashmap objects will be created to store the new moons arrayList into worldList and teleportDestination for flexibility and easier access with the use of key and value pair. worldList will be responsible for storing all the moons that were created with the name of the map and GameMap object as the key value pair. Then, travelDestination will be responsible for storing the destination map and the spawn location of the Player as the key value pair.

A Traveller interface is created where the Player class can implement it, this interface can also act as one of the capabilities of the Player as this interface allows the Player to travel between Moons. This interface includes a default method that moves the Player to desired destination. Then a new action class called TravelAction is created extending the abstract Action class, this class will be responsible for executing the travelling action of the Player by moving the Player to a different map. Thus, Single Responsibility Principle(SRP) is adhered to in this approach as the Traveller Interface and the TravelAction class only has a single responsibility that only handles solely about travelling between maps. DRY principle is also adhered to as inheritance is applied to avoid redundant codes. A new ability called TRAVEL was added for the Player class, with this, the Player will be given the option to travel to other maps interacting with the Computer Terminal.

For teleporting, Theseus is created as an Item object that can be purchased from the ComputerTerminal where it inherits the abstract Item class and inherits the Purchasable interface. This approach adheres to the DRY principle as they have common attributes. Since Theseus is an item that allows the Player to teleport, it will check if the Player has the ability TELEPORT before executing the action, because not all actors on the map are able to use Theseus. Then the TeleportAction class is created by inheriting the abstract Action class. To ensure that Theseus will be working as intended, it will randomly teleport the Player to a random location in the current map as well as in the wall or on the tree, but teleportation will fail as intended if the generated random location has another actor.

Pros

- Future new maps can be stored in the HashMap created in Application class for easier data retrieval
    - HashMap<String, GameMap> worldList
    - HashMap<GameMap, Location> travelDestination

- The design choice of Traveller interface adheres to Single Responsibility Principle (SRP) leading to easier code maintenance as it only has one singular purpose. This interface represents a single responsibility of a travelling actor, in this case, an actor that implements the interface has the functionality to travel between maps.

- The design choice of creating Theseus class adheres to the Open-Closed Principle(OCP) where it extends from the Item abstract class, without modifying the existing Item class itself.

- The usage of hasCapability() in some class to check for certain condition in the if else statement, instead of using instanceof that which violate the principle of OOP and making the code smells which is also violating the Liskov Substitution Principle(LSP) by using it to check if a certain object belongs to certain subclass.

Cons

- To add a new map in the current implementation, we have to add them as a list of strings manually, for each map into the application class. This makes it look very cluttered as we also add the scrap items into the maps within the same class.
  So a new class may need to be created just to store the maps in the future implementation for easier readability and maintenance.

- May consist of connascence of type and name or position in some classes, such as between TravelAction and ComputerTerminal class. This can be seen when the TravelAction constructor is called in the ComputerClass class. If I were to change the type or name of parameters, I would need to change the content of my constructor called in the ComputerClass.

Requirement 2 (Done by Yin Shi)

Requirement

1. Humanoid figure: once the intern arrives at the factory's spaceship parking lot and within the surroundings of the humanoid figure, they can sell the scraps intern collected. Also the intern can not hurt the humanoid figure since it is a part of the factory.

2. Add the sell price for items that can be sold to the humanoid figure, in the process of selling some of the scarps to the humanoid figure, the price may have a certain chance of changing. When selling toilet paper rolls, there is a 50% chance that an intern will be killed instantly by a humanoid figure. Other items can not be sold to the factory.

Changes to Design

1. Change "Sellables" abstract class to interface with method (getSellCredits, getOriginalSellCredits, getItem, getKillRate)

2. Separate the selling scraps to humanoid figure action and picking up pot of gold action, re-define the picking up pot of gold action to PutGoldToWalletAction, re-define the selling scraps to humanoid figure action to SellAction.

3. Create a new Enum in item package, which gives some of the items the capability of CAN_BE_SOLD indicating that type of item can be sold to humanoid Figure.

4. For the items can be sold to Humanoid figure, make these classes implement the Sellable interface, add related attributes like sellcredits, discountRate, and override the get credits methods.

Design Goal

Fix disadvantages in assignment 2.

1. Two abstract classes Sellables and Consumables for items can cause several issues and violate some SOLID principles. (interface segregation principle (ISP) and Liskov Substitution principle (LSP) if there is a class that needs to be both 'Sellable' and 'consumables' it might need unnecessary complexity and be hard to maintain.)

Sellable class should also be re-defined to a different function. So, make the Sellables to become an interface which later used by the items can be sold to the humanoid figure.

2. The SellAction is not the accurate expression, as in the assignment 2, it should be named to the PutGoldToWalletAction which is specific for putting pot of gold in wallet without selling it to the humanoid figure. (Single responsibility principle). Also, the action can be done after

picking up the pot of gold, however if the player puts the gold in the wallet, the pot will disappear, and the player's wallet will increase by 10 credits. So, add addToWallet attribute, which is set to be false in the beginning, and override the allowableAction to make the player have access to action when the addToWallet is false, the attribute is set to be true after executing the PutGoldToWalletAction.

Design goal for the assignment 3.

1.

Set the sellable interface with different methods including (getSellCredits, getOriginalSellCredits, getItem, getKillRate)

Some items have a capability of being sold to the humanoid figure, so for future implementation, create a new interface of sellables which these item classes implement it. (LSP, ISP)

2.

For the item classes (LargeBolts, MetalSheets, LargeFruits, PickleJar, MetalPipe, Gold, ToiletPaperRolls). They all have a sell price when selling to the humanoid figure, so sellCredit is added to be an attribute. For some of the items, they also have a certain chance to change the sell price. So, the chance and the changed price are also added to be one of the attributes. These attributes can be gotten by using the methods overridden from the Sellable interface. (LSP). For the toilet paper roll, it has a 50% rate to kill the player instantly. This is achieved by adding default method to sellables interface, with the killRate set to be 0%, while in toilet paper roll class, the method is overridden to return a proper kill rate of 50%. It adheres the OCP principle which adds new behaviour (like the kill rate for the toilet paper roll) without modifying existing code.

3.

SellAction is the action that extends from the action abstract class, it will show a string in the menu indicating the action player can choose and give out the original price. After the player chooses this action, the execute method is called and updates the statuses, which add balance to the player and remove the item from inventory. However, if a player sells toilet paper rolls to humanoid figures, there is a 50% chance to remove the current player. (open/close principle)

4.

The humanoid figure extends from the ground abstract class, because it is part of the factory. Also humanoid figure cannot move in the map, so instead of extending the actor abstract class, extending the ground abstract class is the simplest way. (open/close principle). This class overrides the allowableActions method, it traverses through the current player's inventory and finds the items with CAN_BE_SOLD capability, then downcast item to Sellables type, and create a new sell action. (Liskov Substitution Principle (LSP): Objects of

a superclass should be replaceable with objects of a subclass without affecting the correctness of the program.)

Pros

- Separate the PutGoldToWalletAcion and SellAction, they are responsible for different actions, and have different execution methods. It adheres to the single responsibility principle.
- Usage of Sellables interface. For those items that can be sold, these classes implement the Sellables interface. (Interface segregation principle)
- Adding proper attributes, overriding the method in interface in items class greatly simplified the SellAction class. (DRY)
- Humanoid figure extends from the ground abstract class and overrides the allowableActions to create new SellAction. (open/close principle)

Cons

- Consumable abstract class violates the (LSP) it can also be an interface, which can support future implementation.
- Although the Humanoid figure is implemented as a ground because it can’t move and is a part of the factory, probably it is actually an actor like bugs.

Requirement 3 (Done by Amadea Theola Benedicta)

Requirement

The new software update allows interns to purchase an AI device named Astley for 50 credits from the computer terminal. Interns must pay a subscription fee of 1 credit every 5 ticks to use Astley's services while it is in their inventory. If the subscription fee is not paid, interaction with Astley is disabled until payment is made. The subscription pauses when Astley is not in the inventory and resumes when picked up again. With Astley in their inventory, interns can listen to random monologues, some of which are conditional based on inventory size, credit amount, and health points. The design should be adaptable for other actors or items that may provide similar monologue features.

Changes to Design

There are no modifications made to the previous design. However, the previous design is improved with new classes such as Monologue, ListenAction, and Astley. Also, recent improvements to the software that enables the player to listen to monologues encourages up to date implementations of interfaces, for instance, MonologueProvider. This up-to-date interface promotes extensibility, flexibility, and reusability for further improvements where another actor or item can monologue.

Design goals

A Monologue class is responsible for the message that is going to be passed on by the item or actor who can monologue, with conditions that must be fulfilled for the listener to receive the message. The class will handle the testing of condition fulfillment by the audience, assumed to be an actor, and return the monologue message. This implementation streamlines the Single Responsibility Principle (SRP) as the class only manages the attributes of monologue, such as passing on the message and condition testing. Although the monologue feature for an object can be handled by this class alone, the functionalities are broken into multiple classes to avoid creating God classes.

For an actor or item to monologue, the messages are conditional based on the bearer of the item or the audience of the speaking actor. To promote extensibility and flexibility of this feature, an interface called MonologueProvider is created. Classes implementing this interface can have their own designated messages alongside their requirements, enabling the monologuing object to return a message based on the listener’s state. Using an interface for this implementation instead of a new abstract class ensures subclasses can have the monologue feature without altering their base code, adhering to the Liskov Substitution Principle (LSP). This interface design avoids connascence of position by encapsulating data access.

As a new AI device is introduced, this object is sold for 50 credits each and has the ability to monologue certain statements based on the state of the actor. To handle all the features of this object, a class called ‘Astley’ is made based on the AI device’s name. Since this device requires the owner to pay a subscription fee for the listening monologue service, the balance

will be deducted every 5 ticks the item is carried. When put down, the item won't prolong the subscription until picked up. Furthermore, if the owner's balance can't pay for the subscription, the listening service will be disabled.

The listening service that the service provided is represented by the class extending from the engine's Action class called ListenAction. This class takes the parameter of MonologueProvider instance in its constructor method and will print the message of the spokesperson on the UI display. The implementation of passing the message follows the OCP (Open-Closed Principle), as the listen action is not implemented inside the device's class. However, the design has a connascence of type where the attribute type for message has to be agreed with String since the ListenAction class has the attribute of message set as String.

Pros

1. Separation of Concerns: Each class has a single responsibility, making the system easier to maintain and extend.
2. Extensibility: The use of interfaces allows for easy addition of new features without modifying existing code.
3. Adherence to SOLID Principles:
    - SRP: Each class handles a distinct part of the functionality.
    - OCP: New monologues or devices can be added without changing existing code.
    - LSP: Subclasses can replace base classes without altering the program's correctness.
    - ISP: Interfaces ensure that classes depend only on the methods they use.
    - DIP: High-level modules do not depend on low-level modules; both depend on abstractions.

Cons

1. Initial Complexity: Setting up a robust class structure initially requires more effort.
2. Potential Over-engineering: There is a risk of over-complicating the design if not managed carefully.

Requirement 4 (Done by Woon Chong)

Requirement

In the requirement, new stages of plants are added, namely Sprout and Young Tree. In the assignment, a new moon Refactorio was introduced where the Inheritrees that inhabit the moon go through the growth phase of Sprout → Sapling → Young Tree → Mature Tree. Sapling and Mature Tree are the only tree stages that can bear fruits, similar to the Inehritrees in Polymorphia.

Changes to Design

Tree concrete class changed into an abstract class where classes to represent separate trees on the different moons can extend from it. PolymorphiaTree and RefactorioTree concrete class would represent the Inheritrees in the respective moons. This would adhere to the Dependency Injection Principle as we have the concrete classes depend on the Tree abstract class. When adding new Inheritrees in to other future maps, we can easily extend the game by adding the new class and extending from the abstract class, which also adheres to the Open Closed Principle where we can add more Inheritrees in other maps without needing to modify the Tree abstract class.

The Inheritree abstract class has changed where it implements code that would otherwise be overridden by the subclasses, which would have similar implementation. This would follow the DRY principle to avoid duplication of code and instead have the subclass refer to its super class for the method, which is upcasting and follows the Liskov Substitution Principle as other classes refer to Inheritree abstract class and would create code smells if we were to ask the abstract class to perform its children class' methods.

Design Goal

To overcome the challenge, I decided to go for the approach where the Inheritrees on both maps would have their separate growth paths, where Polymorphia would have:
Sapling → 5 ticks → Mature Tree

And Refactorio would have:
Sprout → 3 ticks → Sapling → 6 ticks → Young Tree → 5 ticks → Mature Tree

To do this, I created separate concrete classes to represent trees in Polymorphia and Refactorio separately, namely PolymorphiaTree and RefactorioTree. These two concrete classes would extend from the refactored Tree abstract class where low-level classes extend from high-level class, which follows the rule of Dependency Injection Principle, where other classes can refer to the Tree abstract class without the need to manually add new Tree classes for other maps in the future. This closely aligns with the Open Closed Principle as well as we are able to add more MapTree concrete classes without the need to modify the Tree abstract class.

Another challenge to overcome was to ensure that each tree growth stage does not implement a redundant method that would return null. This was necessary for the growFruit

method where Sprout and YoungTree are unable to bear fruits. For this, the Inheritree abstract class is able to handle such issues as the subclasses will refer to the parent class, i.e. the Inheritree class, where the parent class will manage the fruit spanning mechanic as well as ensure that the plant is able to bear fruits or not. This follows with the Open Closed Principle as future tree growth stages can be added without the need to override a redundant method if they are unable to bear fruits. Another principle that is being satisfied is the Single Responsibility Principle where Inheritree does not manage all of its tree growth stages and instead we have a separate concrete class that represents each tree growth stage and its own behaviours.

Some growth states may not grow to other states, i.e. they could be the last growth stage in for that tree's growth process. In order to follow the Open Closed Principle, I created two constructors where one would take in no parameters and the other is an overloaded constructor that takes an integer as a parameter, which represents the maturePeriod. If the tree stage is the last stage, we can call the constructor with no parameter to use the default maturePeriod of 0, and we can call for the other constructor for otherwise, with the integer representing the number of ticks before the tree matures to its next stage. This will allow for different patterns of tree growth processes as it will not require us to modify any of the existing tree growth classes and will also allow for future tree growth states, such as a dying tree or dead tree etc.

Pros

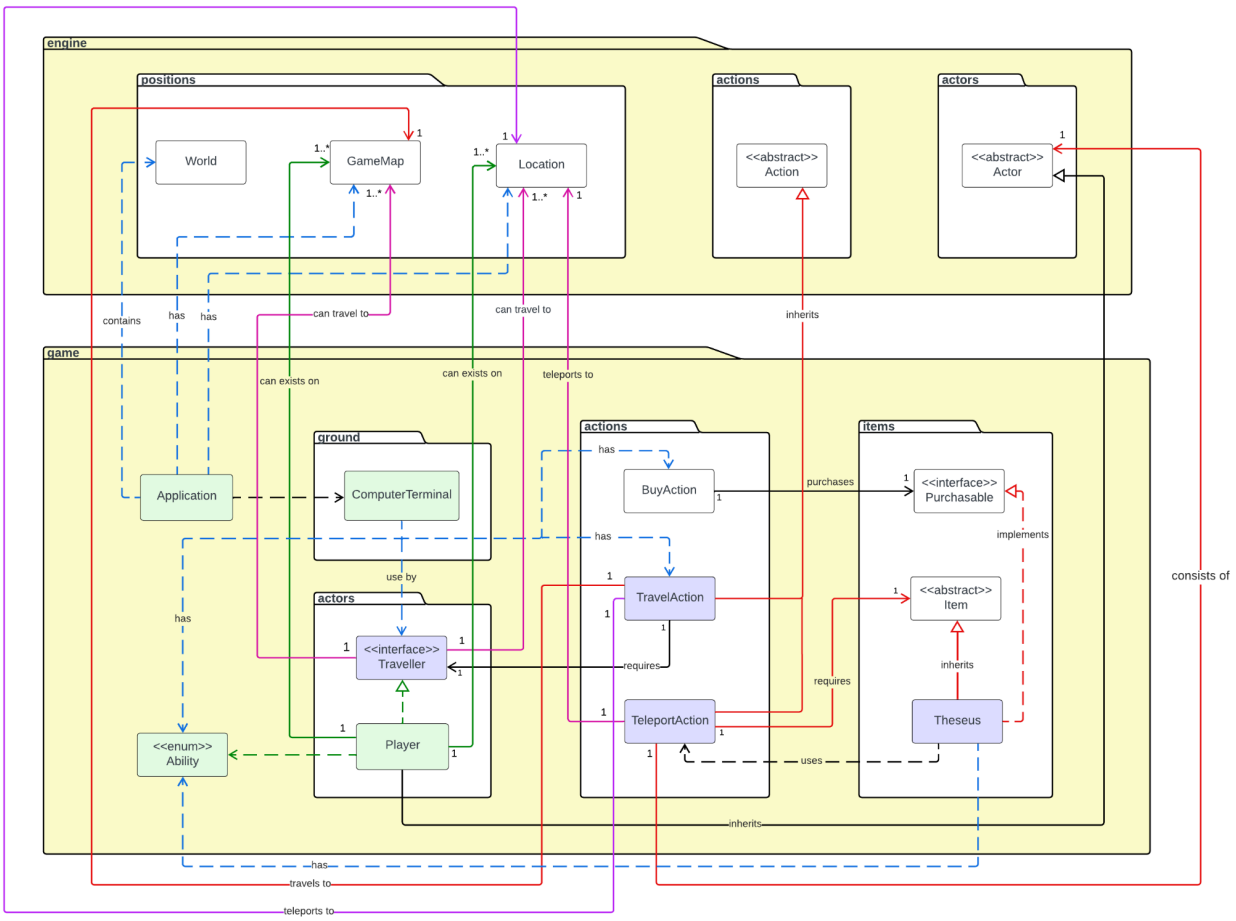
- Inheritree abstract class allows for future tree growth state to be added extensively where they are able to inherit all behaviours of an Inheritree by extending from the abstract class
    - Tree growth states that do not bear fruits will not need to implement a redundant method
        - This avoids duplication of code for trees that don't bear fruits
- Tree abstract class allows for other concrete classes to extend from it for other Tree objects for different maps
    - With their own subclasses, each map has their own class to represent the Tree's behaviour in the map which allows for individual growth processes

Cons

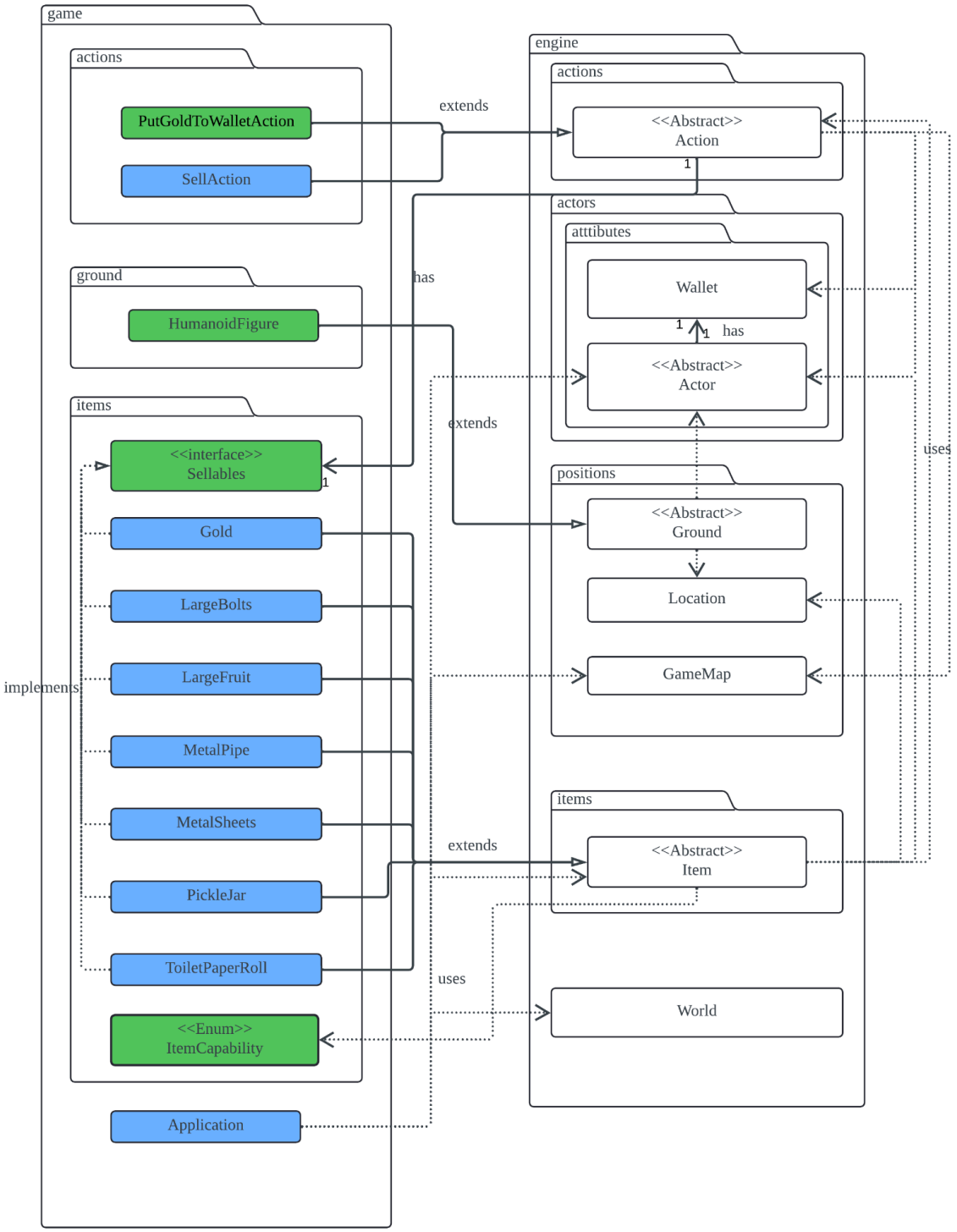
- Growth process for the tree in each map is hardcoded which may require modifications to existing classes if one decides to add more tree growth stages to the tree growth process
    - Manually adding tree growth stages to replicate the growth process of the tree requires proper order else the growth process of the tree is broken
        - Connascence of Timing

UML Designs (Class Diagram)

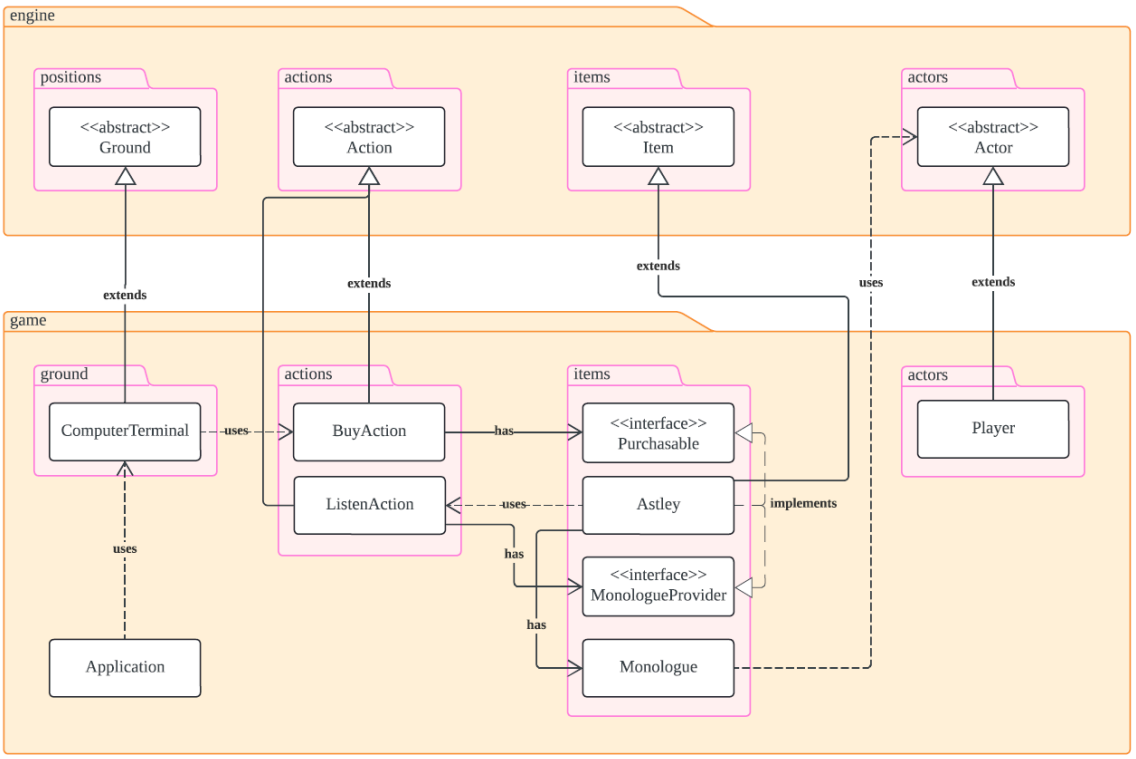
Requirement 1



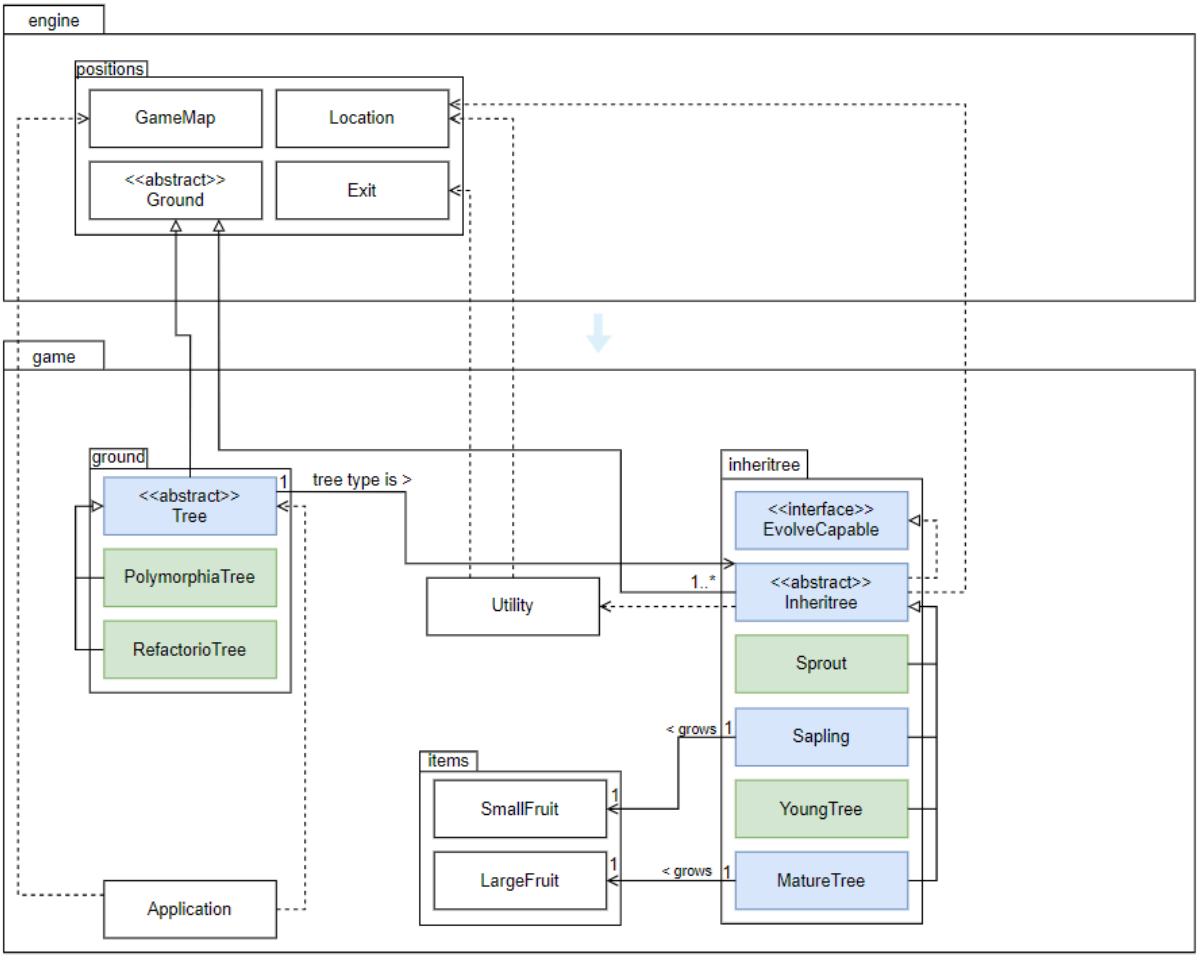
Requirement 2



Requirement 3



Requirement 4

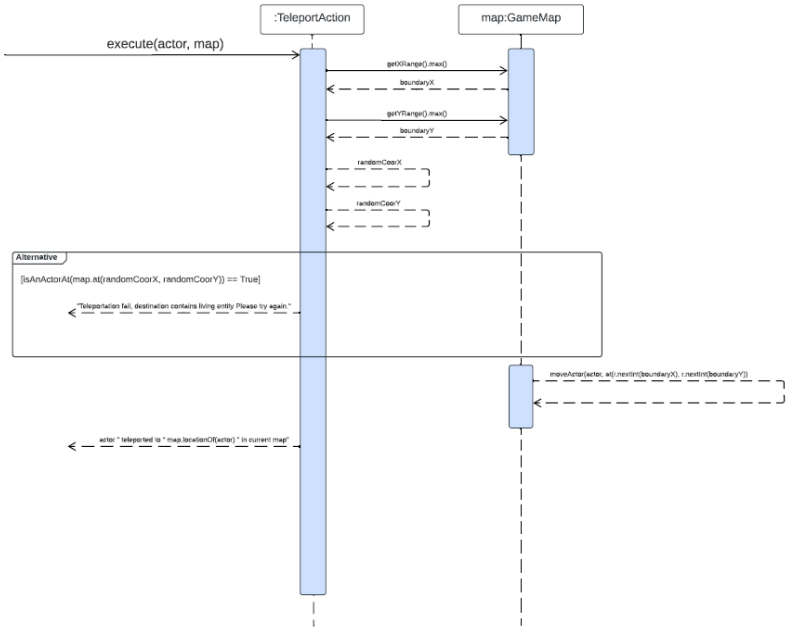


Sequence Diagrams

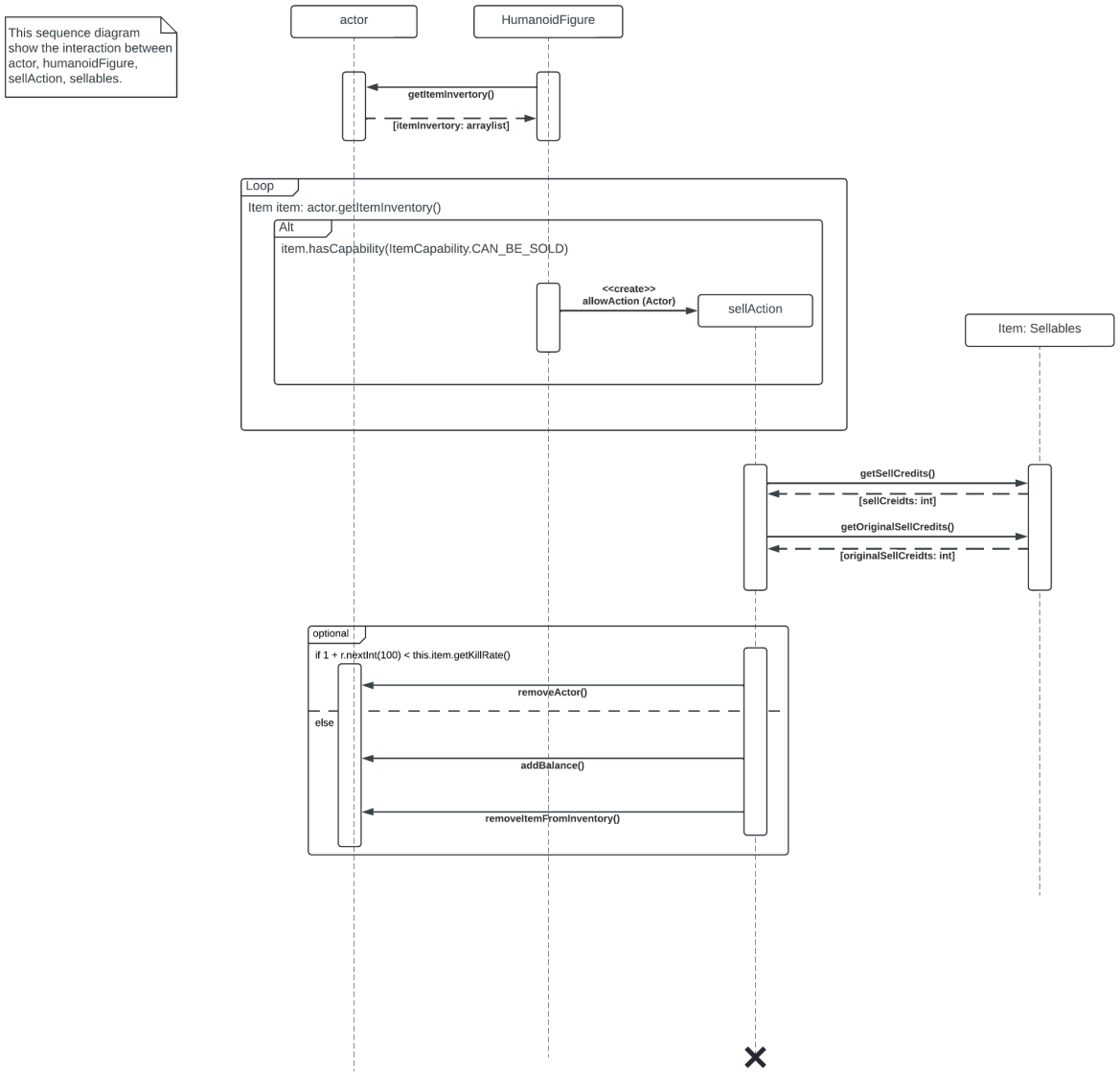
Requirement 1

Method execute() in TeleportAction class

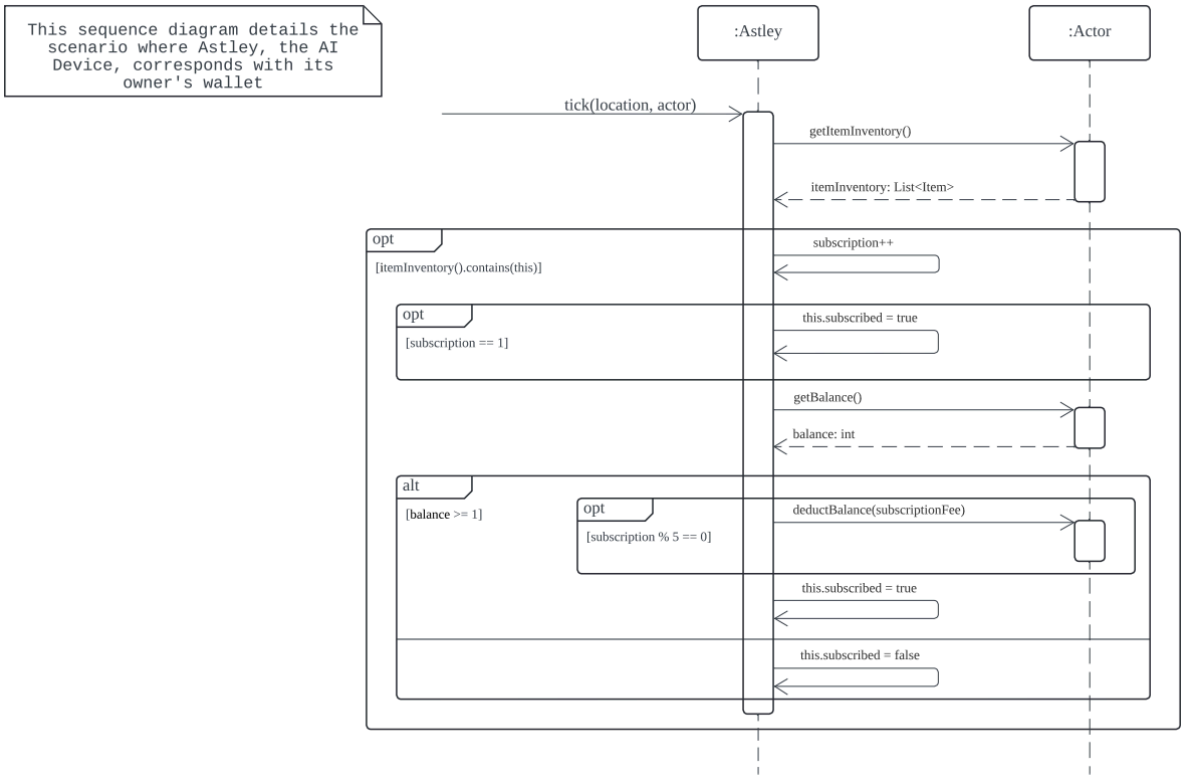
UML sequence diagram that shows the scenario of the process of actor teleporting in the current map



Requirement 2



Requirement 3



Requirement 4