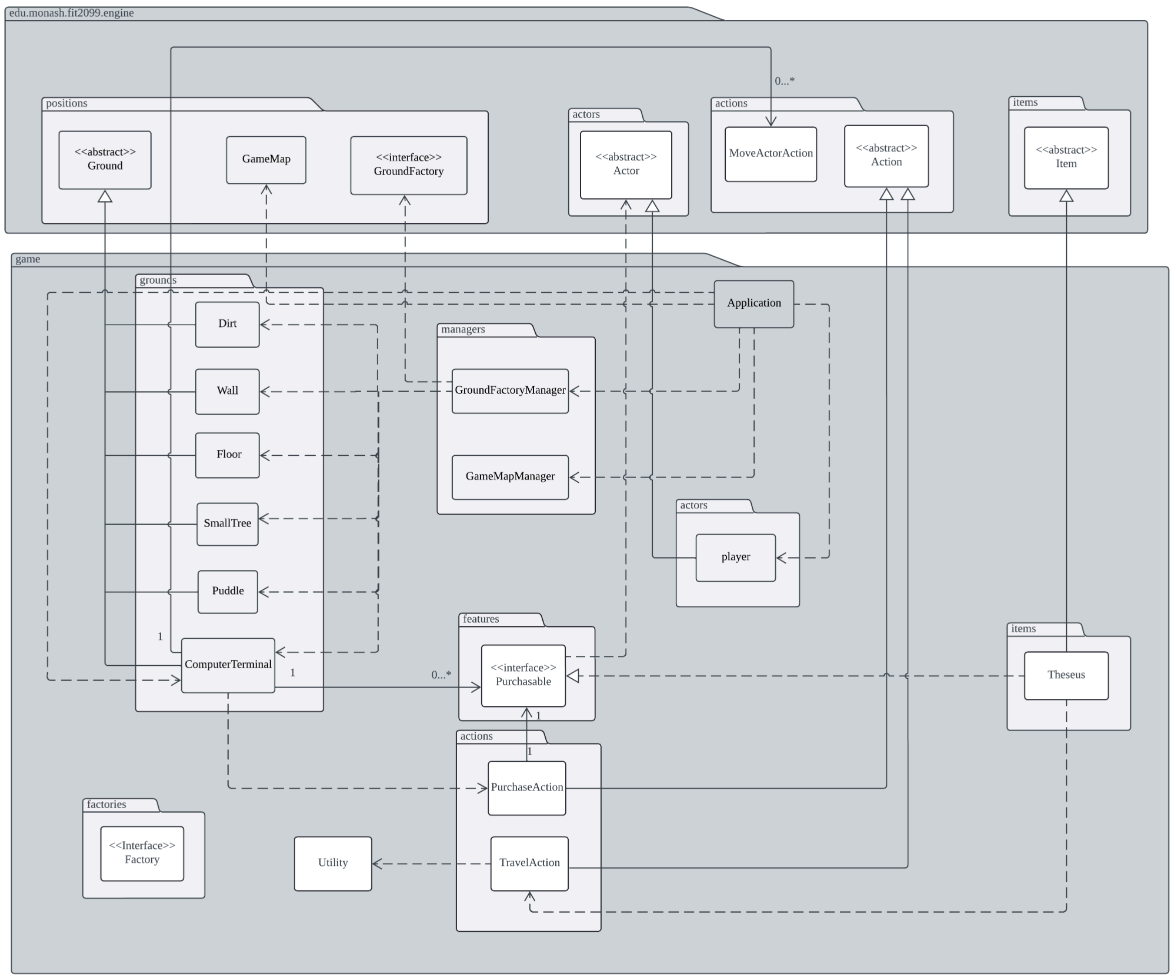
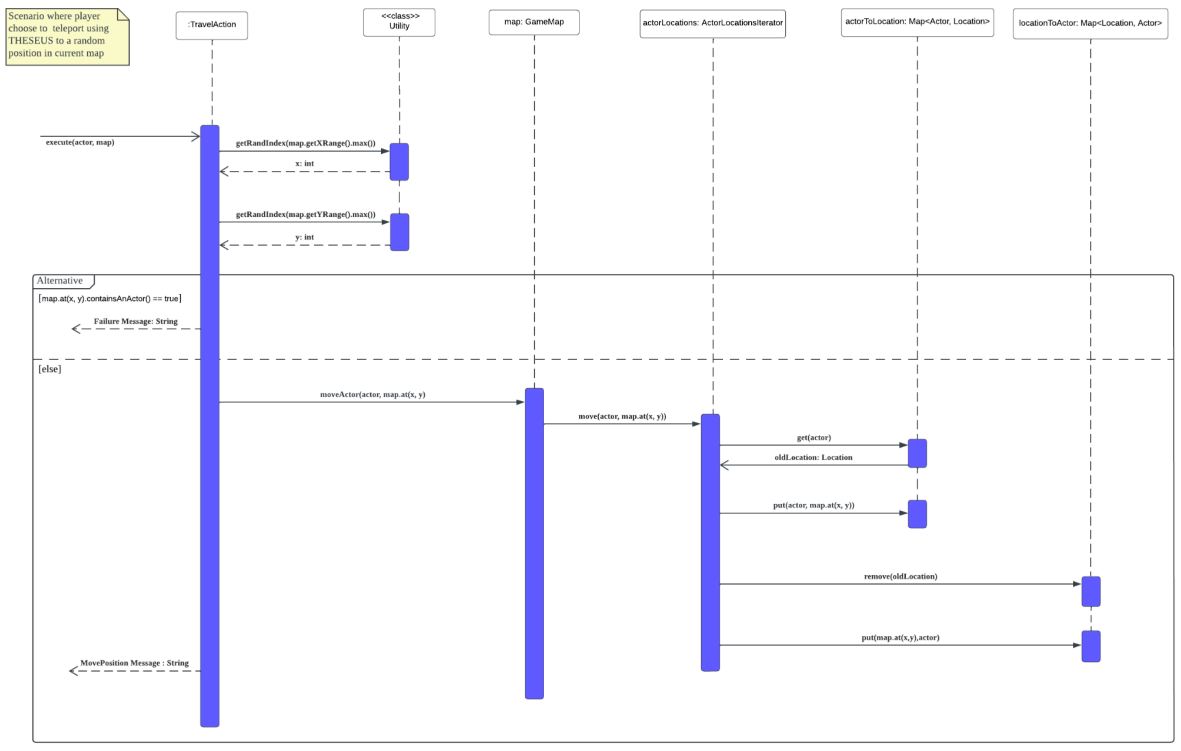
REQ1: The Ship of Theseus

UML Diagram



Sequential Diagram


Scenario where player choose to teleport using THESEUS to a random position in current map
:TravelAction
<<class>> Utility
map: GameMap
actorLocations: ActorLocationsIterator
actorToLocation: Map<Actor, Location>
locationToActor: Map<Location, Actor>
execute(actor, map)
getRandIndex(map.getXRange().max())
x: int
getRandIndex(map.getYRange().max())
y: int
Alternative
[map.at(x, y).containsAnActor() == true]
Failure Message: String
[else]
moveActor(actor, map.at(x, y)
move(actor, map.at(x, y))
get(actor)
oldLocation: Location
put(actor, map.at(x, y))
remove(oldLocation)
put(map.at(x,y),actor)
MovePosition Message : String

Design Goal :

In this requirement, we are required to update a new feature for computer terminal such that the intern can now travel to different planet (other places that not currently in). We are asked to update a new portable teleporter THESEUS for computer terminal. The intern can now purchase THESEUS for 100 credits and it allows the intern to instantly move within a singel map for free. The final goal is to apply SOLID principles and DRY principles in this requirement to implement this two new feature.

Design Decision :

Since we need to travel between different game map, the computer terminal in different game maps should have a list to contain and handle travel actions to different game maps from current game map. We also need a new method to add these travel actions into the list of current computer terminal.

Furthermore, the player can purchase THESEUS at the beginning of the game from the computer terminal for 100 credits. Hence, we should implement a class Theseus that represent THESEUS and it should act like a purchasable item like EnergyDrink, and ToiletPaperRoll such that the player can buy from the computer terminal. We should also add a new travel action for Theseus when the intern interacts with it, the intern will be teleported to a random position in the current map.

Final Design :

Theseus is a concrete class that extends abstract Item class and implements Purchasable interface. It represents the item THESEUS in game and can be bought by the player at the beginning of the game for 100 credits in computer terminal. (SRP) It is responsible for purchase process and allowing teleportation to random positions. Theseus class is open for extension but closed for modification. (OCP) The advantage of this design is that we can add new classes to implement Purchasable interface without modifying the existing Theseus class if we have new features to be added related to the teleportation in the future.

We have created a new TravelAction class that extends Action class. It is a class that manages the travel action such that the player can teleport in the current map. (SRP) It will decide the coordinates where the player will be teleported to then checks if the position is valid. If the position is occupied by another actor, it will return an error message. Else, it will move the player to that position. We are here resuing the getRandIndex method in Utility class when obtaining the random coordinates such that we do not have to implement existing code again. (DRY) If we have other item or actor that can offer similar functionality, we can reuse the TravelAction. Hence, Theseus class is calling TravelAction in order to achieve teleport feature. TravelAction can replace the Action object in the engine when executing the action chosen by the actor without causing an error (LSP).

Inside the ComputerTerminal class, we have implemented a new method addTravel and a list to store MoveActorAction. The advantage is travel actions can be added into the list using addTravel actions for the computer terminal in different game maps. The disadvantage is that we need to call the method in every existing computer terminal when everytime a new game map with computer terminal is added to the game, such that we have the option to travel to the new game map through the computer terminal.

Since we can have multiple computer terminals in a game map. The advantage is we can simply add the new computer terminal at the desirable location in the Application class, then add the MoveActorAction for that specific computer terminal such that decides which places can that computer terminal send the player to. The disadvantage is that we need to do the steps mentioned above everytime we add a new computer terminal in our game map in Application class.

ComputerTerminal is reusing the previously existing factory to create new purchasable item, THESEUS and add it into the purchasable list of computer terminal. (DRY) We are reusing the MoveActorAction given in the engine to move the player across different game maps.(SRP, DRY) MoveActorAction can replace the Action object in the engine when executing the action chosen by the actor without causing an error (LSP).

To handle different game maps and ground factories of corresponding game map, we have created GameMapManager class and GroundFactoryManager class.

For GameMapManager class, it will take care of existing game maps in game. It consists of different methods to retrieve different game map. For instance, getRefactorioMap method will return a string representation of Refactorio game map as well as getPolymorphiaMap and getStaticFactory method. If we have more game maps to be added in the future, we can simply create a new method to store the new game map. We have avoided Application class to be the GOD class such that it does not have to fit in every game map inside especially when there are different game maps.

For GroundFactoryManager class, it is responsible to get the GroundFactory in corresponding game map. For example, getPolyMorphiaGroundFactory will return the GroundFactory for the PolyMorphia game map, this includes every ground entities available in that particular game map. When there are new game maps added in the future, we can simply extend our game by adding a new method to retrieve the GroundFactory for that particular game map.

The methods in both the GameMapManager class and GroundFactoryManager class mentioned above are all static methods. By declaring static methods, we can call the methods without creating an instance of that particular class. The advantage of this design includes global access, object independence and code organization. We have avoided to implement a lengthy code to instantiate an object everytime we need to retrieve the corresponding map and ground factory.

Alternative Design :

An alternative design is that we are using the MoveActorAction class to handle both the travel actions to different game maps and teleport the player to a random position within current map. Although this is following DRY principle by not creating new classes with the same functionality as MoveActorAction. However, it is not suitable for our current scenario.

Since MoveActorAction is a basic action that moves an actor from one location to another on the map. It is initially designed for basic movement, such as moving an actor one step in a certain direction. TravelAction includes an additional logic to check the validity of teleportation destination where MoveActorAction does not require. It will violate SRP if we implement the teleportation feature in MoveActorAction. Although it has the similar functionality as the TravelAction, it is best for us to follow SRP where we have MoveActorAction for player movement and TravelAction for player teleportation.

Furthermore, it is more reasonable to use TravelAction than MoveActorAction when implementing teleportation feature. This is because MoveActorAction takes in several parameters, including a location (destination), string(direction). We have to generate a random location, check the validity and then insert them as parameter for MoveActorAction.

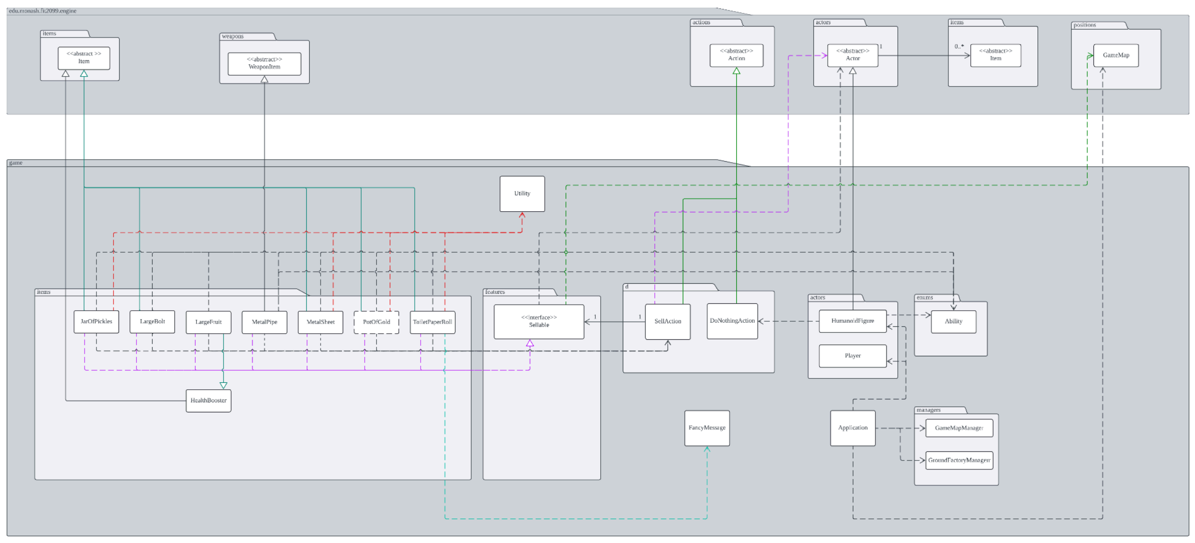
However, when using TravelAction, that takes no parameter, we can complete the generating random position process, check validity within the class, which is better as it follows the encapsulation. This can avoid any unexpected error if every method is scattered and also easier to debug when there is an error.

Conclusion :

The reason for choosing the final design rather than alternative design is that we have created a class with distinct functionality and avoid using current class that not suit our scenario, which is MoveActorAction. (SRP) When we have new item that can offer teleportation, we can simply create a new class and instantiate a new TravelAction. Our current design also supports the feature where a game map can contains multiple computer terminals. It can be simply implemented by adding computer terminal at the desirable location and add the MoveActorAction for that particular computer terminal to the destination. The GameMapManager and GroundFactoryManager class allow future extension on game maps and their respective ground factory.

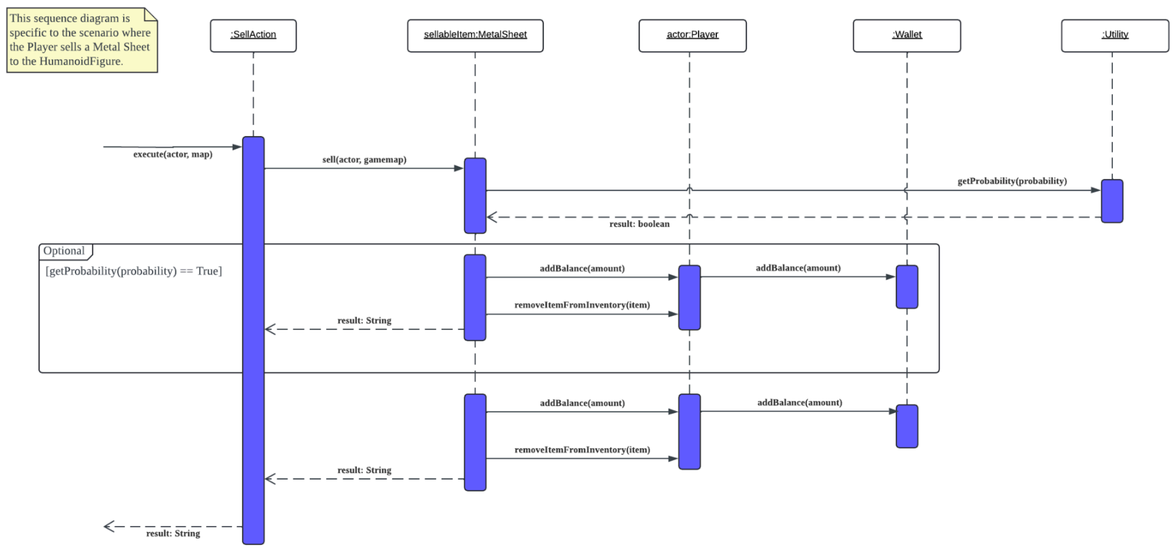
REQ2: The Static Factory

UML Diagram



Sequence Diagram

Chosen Scenario : The Player sells a Metal Sheet to the HumanoidFigure.



Key Points From Assignment 2 which matches with REQ 2

LargeFruit is a class that extends the abstract HealthBooster class and implements the Consumable interface. It can be consumed by player and increase the health of the player.

JarOfPickles is a class that extends abstract Item class in the Engine and implements the Consumables interface. It can be picked up and be consumed by the player as the properties in Item class allow JarOfPickles to perform.

PotOfGold is a class that extends abstract Item class in the Engine and implements Consumables interface. It can be picked up and dropped by the player as the properties in Item class allow it to perform.

MetalPipe is a class that extends WeaponItem abstract class. It can be picked up and dropped by the player and it can be used to attack a desired target when the target is at one of its exits.

Design Goal :

In requirement 2, we will be designing the functionality of the spaceship parking lot, by enabling the intern to sell the scraps collected to a humanoid figure. The sellable scraps are large bolts, metal sheets, large fruits, jars of pickles, metal pipes, pots of gold, and toilet paper rolls. Different scraps are sold at different prices, and there are chances for each product to trigger special events like not being paid, selling at discounted prices, or an instant death. When HumanoidFigure is at one of the exits of Player and Player has those sellable scraps in their inventory, Player can sell the scraps to HumanoidFigure. The goal of this OOP design is to apply the SOLID and DRY principles in the features stated above.

Design Decision :

Having that our goal is to differentiate items to be sold. Therefore, an interface is implemented by item classes. Adding to that, we need to make the player sell items to a humanoid figure, we should first have a selling action. Then, the humanoid figure class is made extending the Actor class. Finally, the item classes implementing features (methods) of sellable items are being sold (selling action) to the humanoid figure.

Final Design :

Sellable interface is an interface representing scraps that are sellable, including the features of sellable items. It contains the sell() method and getSellingItemDetail() method to notify the user the result of selling the item and the details of items being sold, which is to be implemented in the concrete classes representing each sellable items (SRP). This design allows new sellable items to be added without modifying the existing codebase (OCP). New items simply implement the Sellable interface, keeping the system open for extension but closed for modification. High-level modules from the game engine depend on abstractions like Sellable rather than concrete implementations, promoting loose coupling and enhancing modularity (DIP).

Then, the concrete item classes implement the Sellable interface by actually implementing the empty sell() and getSellingItemDetail() methods, in a respective manner. For example, ‘JarOfPickles’ is a concrete class that represents a game item. This class extends the Item abstract class and implements Consumable and Sellable interface. Since it implements the Sellable interface, sell() and getSellingItemDetail() methods are overridden in the class and their concrete implementations are specified in them (SRP, OCP). Using interfaces ensures that classes implementing them are only exposed to the methods they need (ISP).

SellAction is a concrete class that extends the Action abstract class to allow a selling action for the playable actors (the player in this case) across different items that are to be sold in the game system (SRP). SellAction can be replaced with an Action object in the game engine -if made concrete- when the action is being executed by the actor, without causing an error (LSP).

On each concrete item class implementing the Sellable interface, the allowableAction method from the game engine’s Item abstract class is overridden to allow the owner of the item to perform an ‘action’ to another actor . This is implemented by adding sellAction speculated above in the item owner’s ActionList, which results in an option in the user's console to sell the particular item if certain conditions are met (DRY).

‘HumanoidFigure’ is a concrete class extending the Actor abstract class, representing the humanoid figure which buys scraps from players in spaceship parking lots. Although it is an actor, it is not able to move around or attack other actors except for a special case. This is implemented by returning DoNothingAction at its turn, which is an Action class from the game engine (DRY). The usage of DoNothingAction also keeps the HumanoidFigure’s responsibilities focused on interactions rather than movement or attacks (SRP). The humanoid figure has a capability of buying scraps from other actors, represented by having an Ability class object as its attribute. This attribute is later used in each concrete item class to make sure only the humanoid figure is able to buy the sellable items.

Alternative Design :

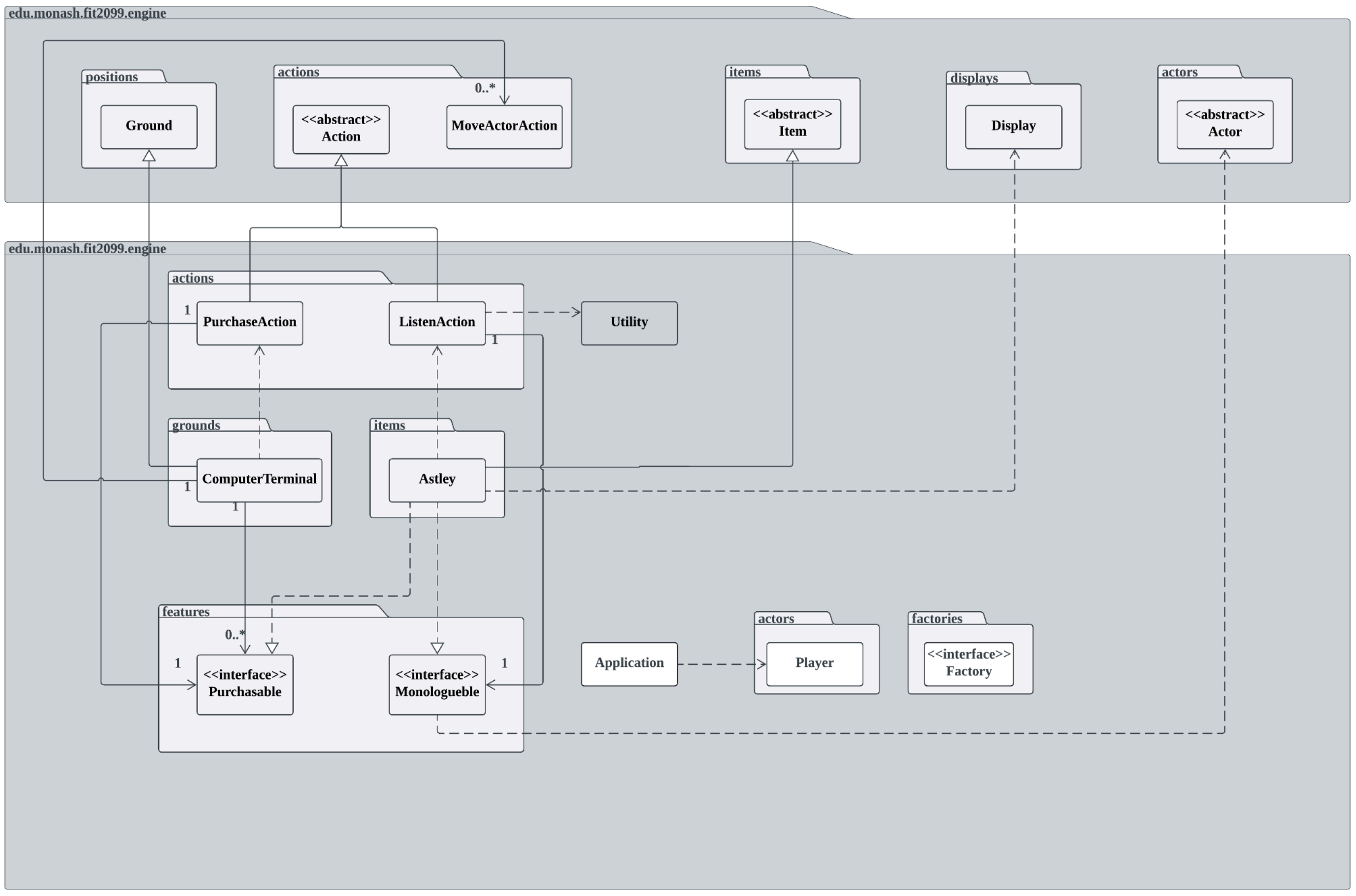
An alternative design is to replace the Sellable interface with an abstract class named SellableItem. This abstract class could include methods to sell items, to describe the items and having attributes like the selling price. Then, each concrete sellable item class could extend the abstract class, overriding the described functionalities above. Adding to that, it should copy the existing features from the previous codebase and even some functionalities that already exist in the game engine. This design is not implemented for the requirement, since it violates the DRY principles.

Conclusion :

The reason for implementing a Sellable interface instead of an abstract class SellableItem is to reduce redundant codes and keep the features modular. The interface-based design aligns with the SRP, OCP, and DIP, ensuring that each class with a singular, focused responsibility, can be extended without modification, and promotes loose coupling. The alternative design would violate DRY principle by using duplicated code. Therefore, the current (interface) design with a more modular, extensible, and maintainable codebase is used.

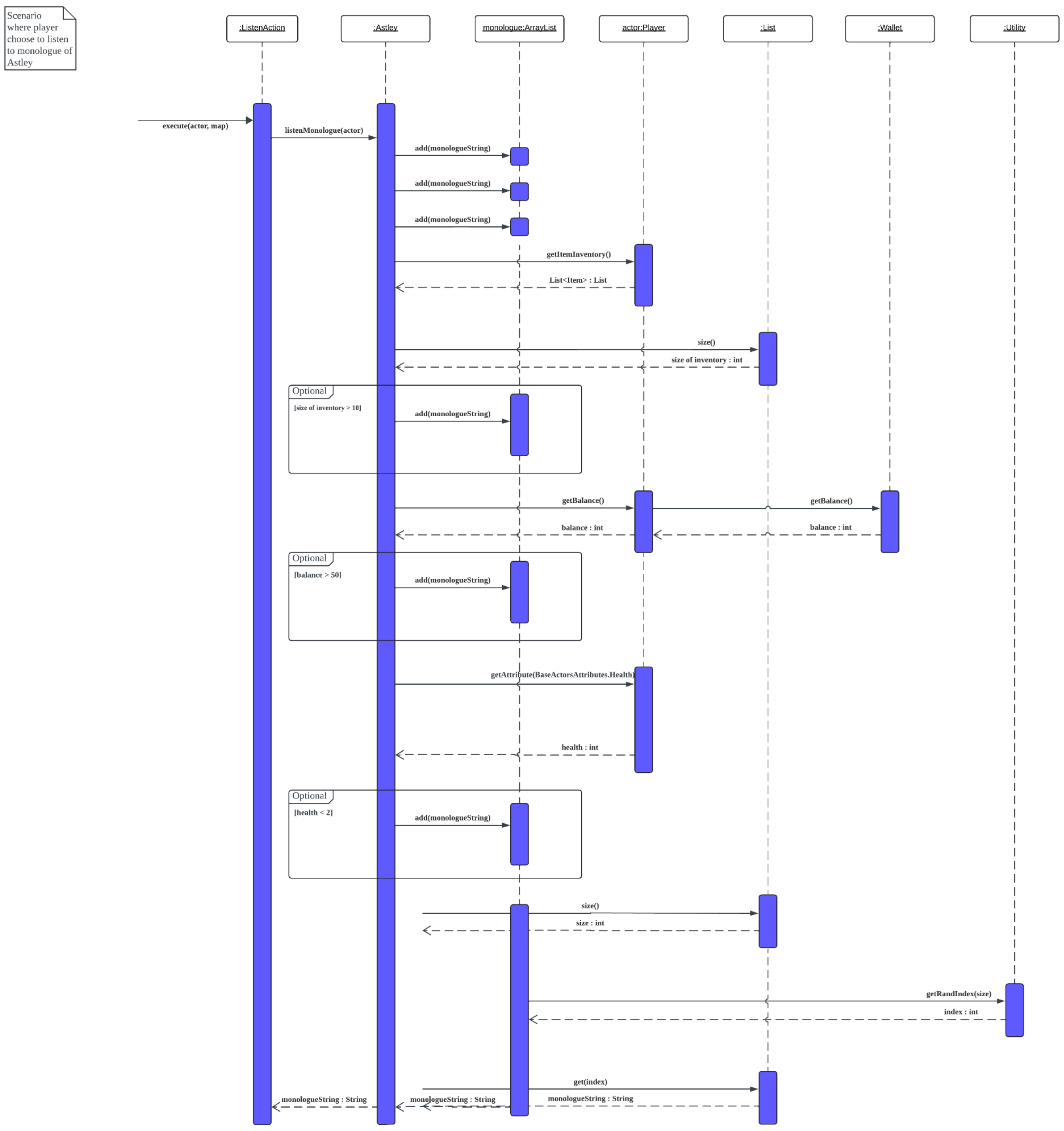
REQ 3 : dQw4w9WgXcQ

UML Diagram



Sequence Diagram

Chosen Scenario : Scenario where player choose to listen to monologue of Astley



Design Goal :

In Requirement 3, we are tasked to create an item, Astley which can produce a monologue. Subscription fee of 1 credit must be paid for each 5 rounds to continue interacting with Astley. If Players fail to pay subscription fee, they will not be able to interact with Astley until they have balance to pay for subscription. Each time Player interacts with Astley, Astley will randomly produce a monologue based on several conditions met.

Design Decision :

Monologueble interface has been created to categorise all items which can produce a monologue together for later use. Astley class has been created to represent Astley which is a new item that can be sold at ComputerTerminal. ListenAction has been created to enable the Player to interact with Monologueble objects.

Final Design :

The SOLID principles used for the code design are Single Responsibility Principle (SRP), Open Closed Principle (OSP), Liskov Substitution Principle (LSP), Interface Segregation Principle (ISP), Dependency Inversion Principle (DIP) and Don't Repeat Yourself (DRY).

Monologueble interface has been created as a single class as we do not want items which implement it to have another method which should not be implemented by the item and ensure each class only has a single responsibility which is process Monologueble object for our case. (SRP , ISP)

Astley concrete class extends from Item abstract class, implements Purchasable and Monologueble interface. Since Astley shares common methods and attributes with Item abstract class, it is compulsory to extend from Item abstract class to avoid redundancy in codes. (DRY) Besides, Astley is designed as a class to increase maintenance of it and ensure each class only has a single responsibility. (SRP)

ListenAction concrete class extends from Action abstract class. This is because ListenAction shares common methods and attributes with Action abstract class, thus it is mandatory to extend from Action abstract class to avoid redundancy in codes. (DRY) Besides, it is designed as a single class instead of merging with another action class. (SRP) The constructor of ListenAction accepts a Monologueble object instead of specific types of items.) With that, ListenAction can be used on different items which have implemented Monologueble without any modification. (OCP, LSP, DIP)

Advantages for this design is that we ensure each class only has a single responsibility without creating a single 'god' class which violates SRP principle. Besides, ListenAction can be widely used by any items, actors or grounds which can monologue as once they implement Monologueble interface, they can create their own monologue to be produced.

Disadvantages for this design is that there will be a sum of classes which need to be maintained. Therefore, when the Astley class turns out to have some errors, we will need to debug it via multiple classes which will be time consuming.

Alternative Design :

Instead of implementing Monologueble interface and make ListenAction accepts a Monologueble object, we can just create a ListenAction() at Astley concrete class and at ListenAction, create a method to insert all strings which can be monologued by Astley and return a random strings which can be monologued by Astley.

Analysis of Alternative Design :

1. Open Closed Principle

    - If there is a new item or actor or ground which can monologue, we will need to modify ListenAction in order to let them monologue. This is definitely a bad design as we will need to modify our code each time a new extension is added.

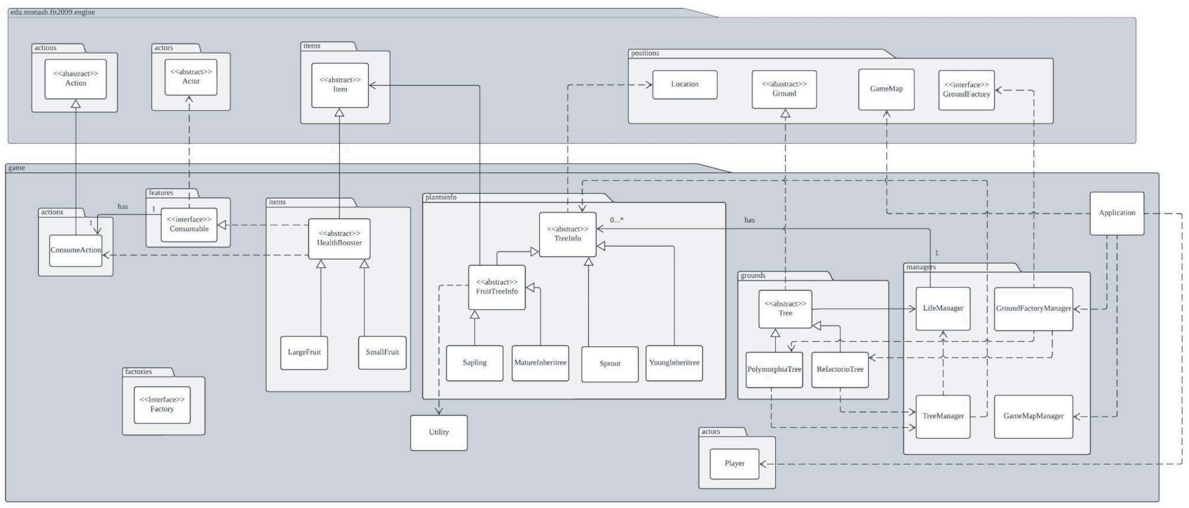
Conclusion :

The reason for choosing this design is that we can let any items, actors or grounds which can monologue complete this action easily by just implementing the Monologueble interface and let ListenAction accept themselves as an object at the constructor. With that, we do not need to repeatedly type out the same thing just to reach one target which is producing a monologue for Player who interacts with them.

For future extension, when there is a new item, ground or actor which can produce a monologue, we can achieve this by just letting them implement Monologueble interface and override methods which are provided in Monologueble interface to produce their monologue. Then, add a ListenAction and the constructor of ListenAction accepts them as an object and Player will then have an option to interact with them and let them produce a monologue to Player.

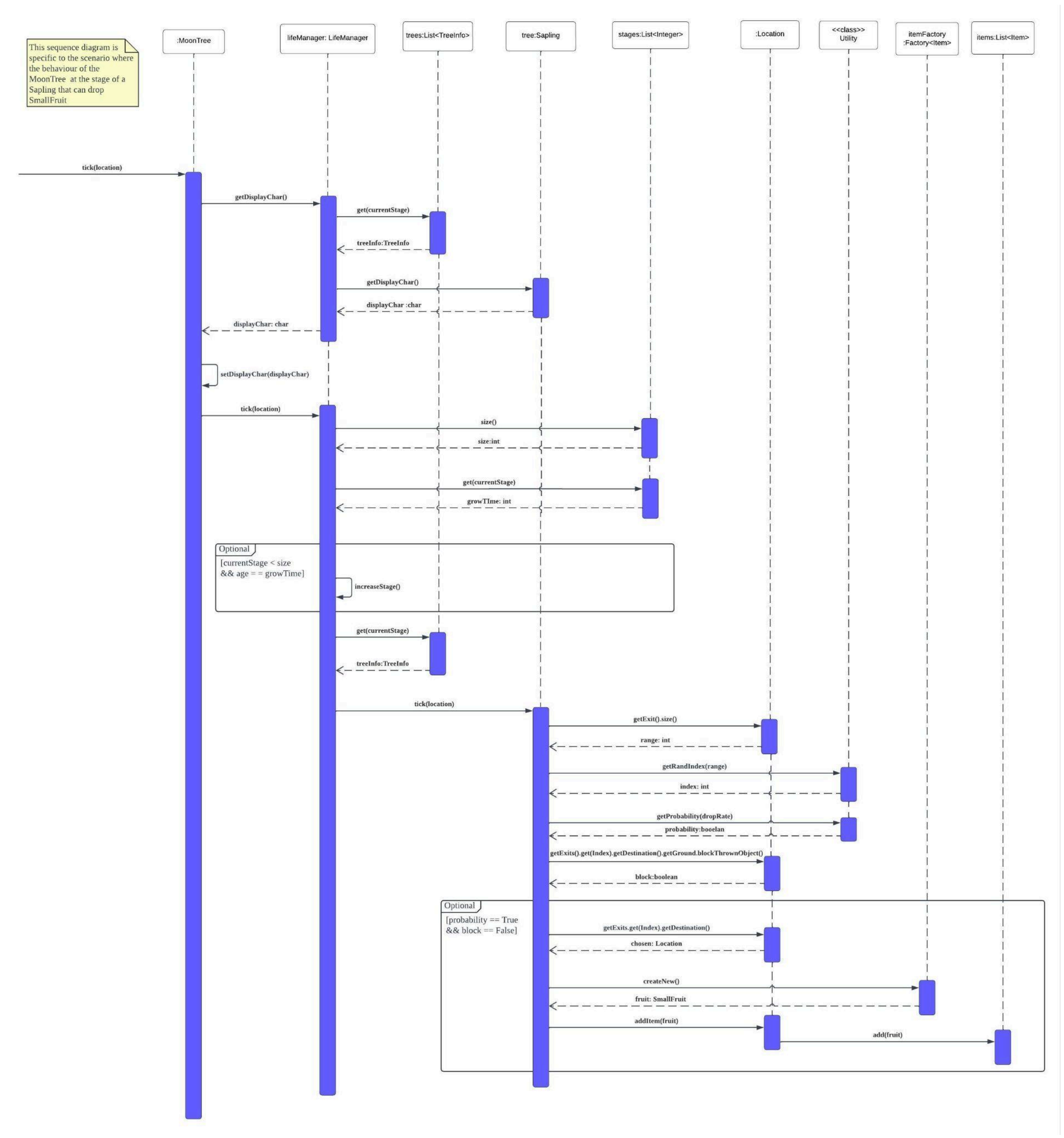
REQ4: [Survival Mode] REQ4: Refactorio, Connascence’s largest moon

UML Diagram



Sequence Diagram

Chosen Scenario : The behaviour of the MoonTree at the stage of a Sapling that can drop SmallFruit



Design Goal :

In this requirement, we are going to redesign the inheritree to enable it to grow in different stages. The stage of the inheritree starts from sprout, sapling, young inheritree and mature inheritree. Sapling and mature inheritree can drop fruits. Sprout and young inheritree cannot drop fruits. The final goal is to apply SOLID principles and DRY principles in this requirement to enable a tree to grow up in different stages.

Modification on previous design:

In the previous design, we had an abstract tree class inherit the Ground class and we assumed a tree always dropped fruits. Since the assumption is different with the description of a tree in this requirement, we need to modify the tree class. We introduced a life manager for the classes to achieve the design goal.

Design Decision :

Since we want to enable a tree to grow in different stages, we should have a life manager class that manages the life cycle of a tree. We need to have a class that records the information of the different stages of a tree. Therefore, the life manager can manage the stage of a tree according to the information of the stage of the tree. TreeInfo class is created and Life Manager class is created.

Final Design :

TreeInfo class is an abstract class created and it has a dependency relationship with the Location. It is abstract as it will not be instantiated and provides a template that stores the information of a tree. TreeInfo class is open for any tree that we want to store the information to extend and closed for modification (OCP). The advantage of this design is that we can easily add a different type of TreeInfo by just extending the TreeInfo. The disadvantage of this design is that we still need to have a dependency relationship with the Location as we need to provide the implementation of the tick method of a tree to interact with the system.

FruitTreeInfo class is an abstract class that extends the TreeInfo class. This class is just a subtype of a TreeInfo that provides a template to store the information of a tree that can drop fruit. The inheritance prevents us from repeating the code (DRY). The FruitTreeInfo class is open for any tree that can drop fruit to extend and closed for modification (OCP). It has a dependency with item class to allow the factory to make a new instance of item (here is fruit). It depends on a higher level of abstraction instead of a lower level of module (DIP) and any subclass of item is allowed to be passed into FruitTreeInfo to generate new instances (LSP). The benefit is we can extend FruitTreeInfo to create more classes about the trees that can drop fruit. The disadvantage is that more classes will be added if there are different types of trees that can drop different types of fruits. The whole system will be hard to manage as it becomes complex.

Sapling and MatureInheritree extend the FruitInfo as both of them can drop fruit (DRY). Sapling represents a tree that can drop SmallFruit and the drop rate is 0.3. MatureInheritree represents a tree that can drop LargeFruit and the drop rate is 0.2 (SRP). The advantage is this design provides a better code quality and the code clean. We don't need to use conditional statements to recognize what should happen. The disadvantage is that adding more class will cause additional complexity to the system.

Sprout and YoungInheritree extend the TreeInfo as both of them cannot drop fruit (DRY). Sapling is used to represent the tree with character ',' and the YoungInheritree represents a tree with character 'y' (SRP). The advantages and disadvantages of this design are similar to the case above.

LifeManager is a class that is added to manage the life cycle of a tree. One LifeManager is associated with 0 or many TreeInfo. Any subclass of TreeInfo can be added to the arrayList of the LifeManager. LifeManager depends on this higher level of abstraction without any lower level of module anymore (DIP). Therefore, any subclass of TreeInfo can be used by LifeManager to execute some function without causing error (LSP). The advantage of this design is that we can pass TreeInfo into the arrayList of TreeInfo to enable trees to have different life cycles. The disadvantage of this design is creating a LifeManager can cause the class creating it to have more dependency on the subclass of TreeInfo without using dependency injection.

TreeManger is created as a static method to get the LifeManager that will be used in the game. Here it is used to apply the dependency injection between the TreeInfo and the Application. Application will just depend on the TreeManger class and get their LifeManager when creating a Tree. The disadvantage of TreeManger is that it will cause a lengthy code when we have more types of LifeManager. However at this stage it works as we only have two types of LifeManager according to the requirement.

Tree is an abstract class that extends the ground. The Tree is associated with a LifeManager and all the behaviour of a tree and its stage will be managed by the LifeManager. Tree is a class to represent a tree that can grow up with different stages(SRP). Tree is abstract as we just provide a template and it is open for extension and closed for modification(OCP). The advantage of having this abstract class is we can extend from the Tree and create a class that can be used by the FancyGroundFactory since we have the static method to get the LifeManager. The disadvantage of this design is we need to manually calculate the accumulated time for the three to grow in the different stages.

PolyMorphiaTree and RefactoriaTree a class that extends the Tree. The inheritance prevents us from repeating the code(DRY). The life cycle of RefactoriaTree is Sprout, Sapling, YoungInheritree and MatureInheritree while the life cycle of PolyMorphiaTree is Sapling and MatureInheritree. It is used to represent this type of life cycle of trees (SRP). The advantage is that adding a new tree is easy for us to do by just creating its LifeManager. Since we use SRP, the types of tree grows will cause the system to become complex.

Alternative Design :

An alternative design is that we do not need to have a LifeManager class and the TreeInfo. We can create a Tree class that extends the Ground and make a subtype of Tree that can drop fruit called FruitTree. Make Sapling and MatureInheritree extends the FruitTree while Sprout and YoungInhertree extends Tree. In the tick method, we need to override each of them to set the ground until their next stage reaches the certain tick of time. This design is not chosen as it violates the Open-closed Principle that we need to modify the previous code when a new tree that can grow in different stages is added, for example from Sapling to MatureTree.

Conclusion :

The reason for choosing the LifeManager version is that it is easy to add different types of tree that can grow in different stages into the game by just creating the LifeManager in the Manger class. We can reuse the TreeInfo class if some of the stages of the tree are the same with the existing TreeInfo.It helps the Tree class to know what to do in the game. Besides that it still maintains the Single Responsibility Principle as it switches the stage of a Tree by using the TreeInfo and no conditional statement is needed.