FIT2099 Assignment 2 Design Documentation
Applied Session 33 (Group 4)

Group members:

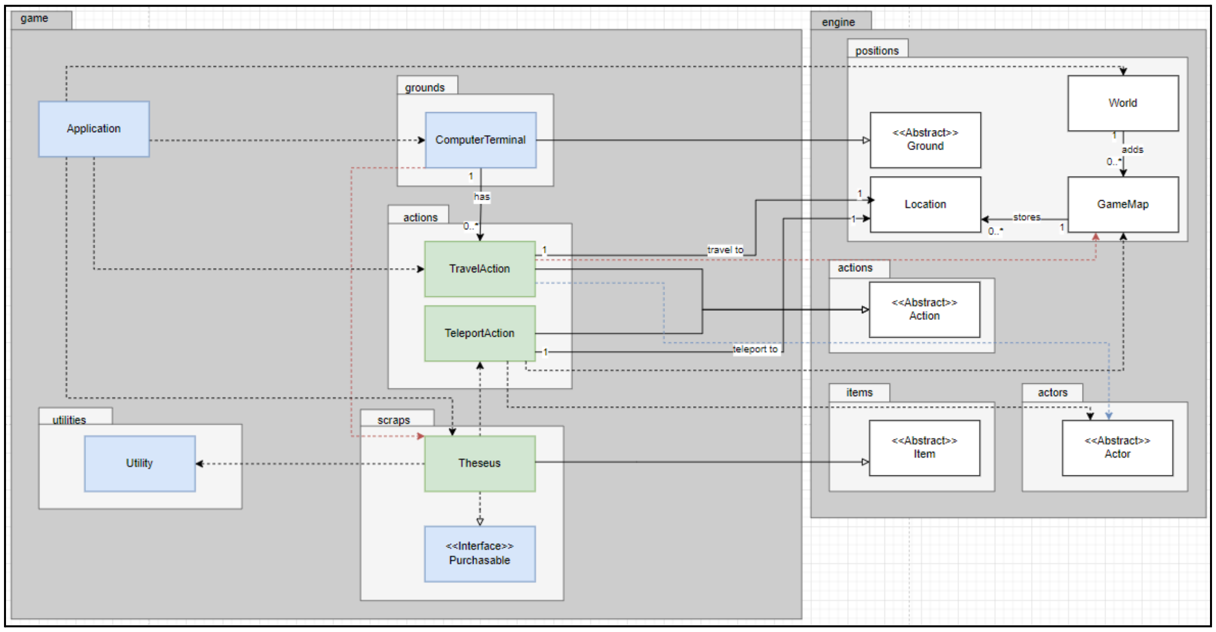
1. Claire Zhang
2. Jin Ruo Yew
3. Anh Nguyen
4. Ke Er Ang (Chloe)

Design Diagrams

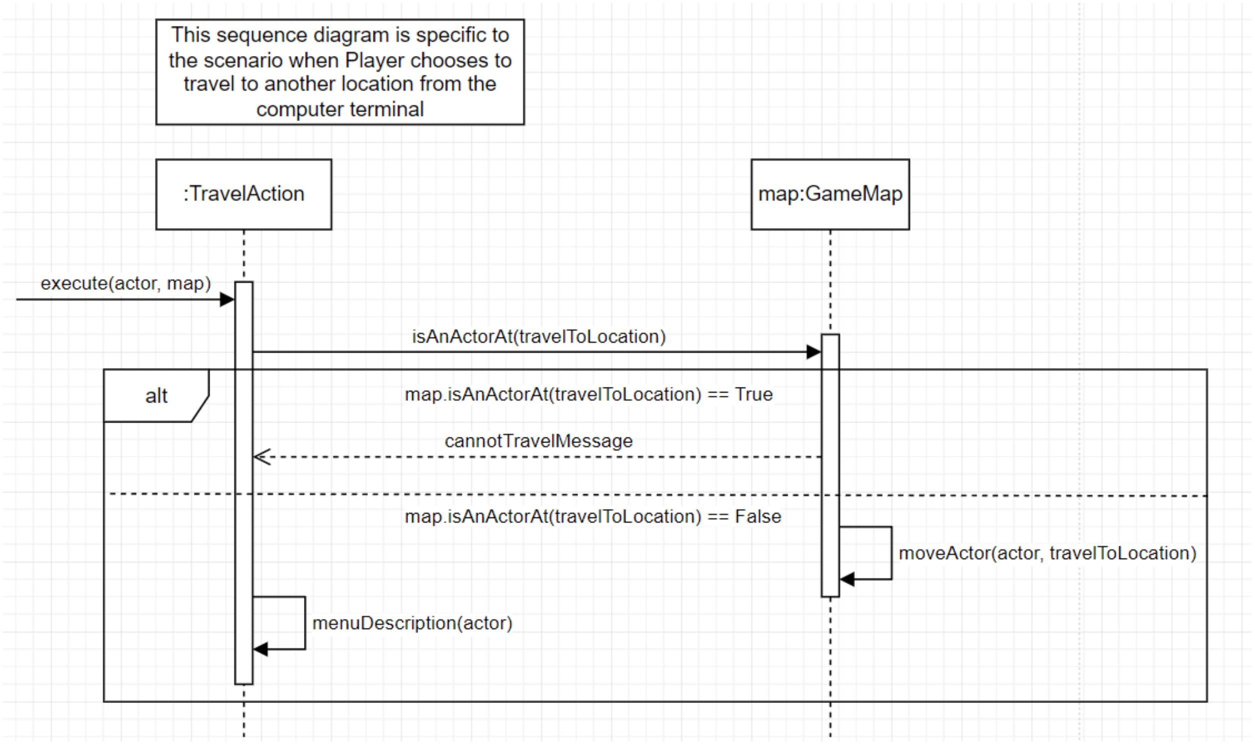
*Note: For all UML diagrams, newly created classes are coloured in green, updated/modified classes from base code are coloured in blue.

Table of content:

REQ 1 UML Diagram:

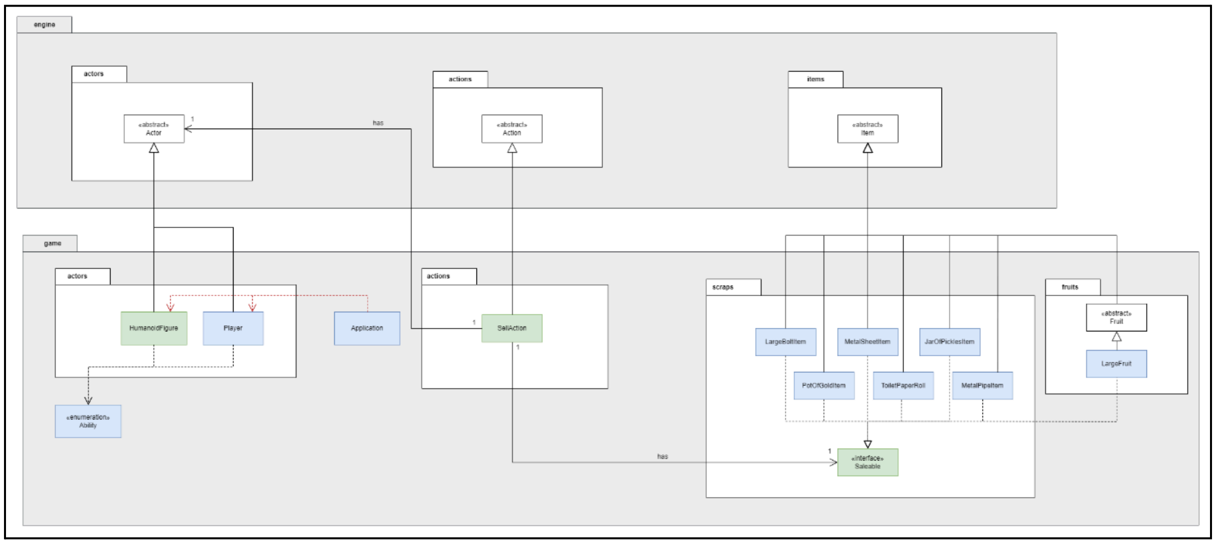


REQ 1 SEQ Diagram:

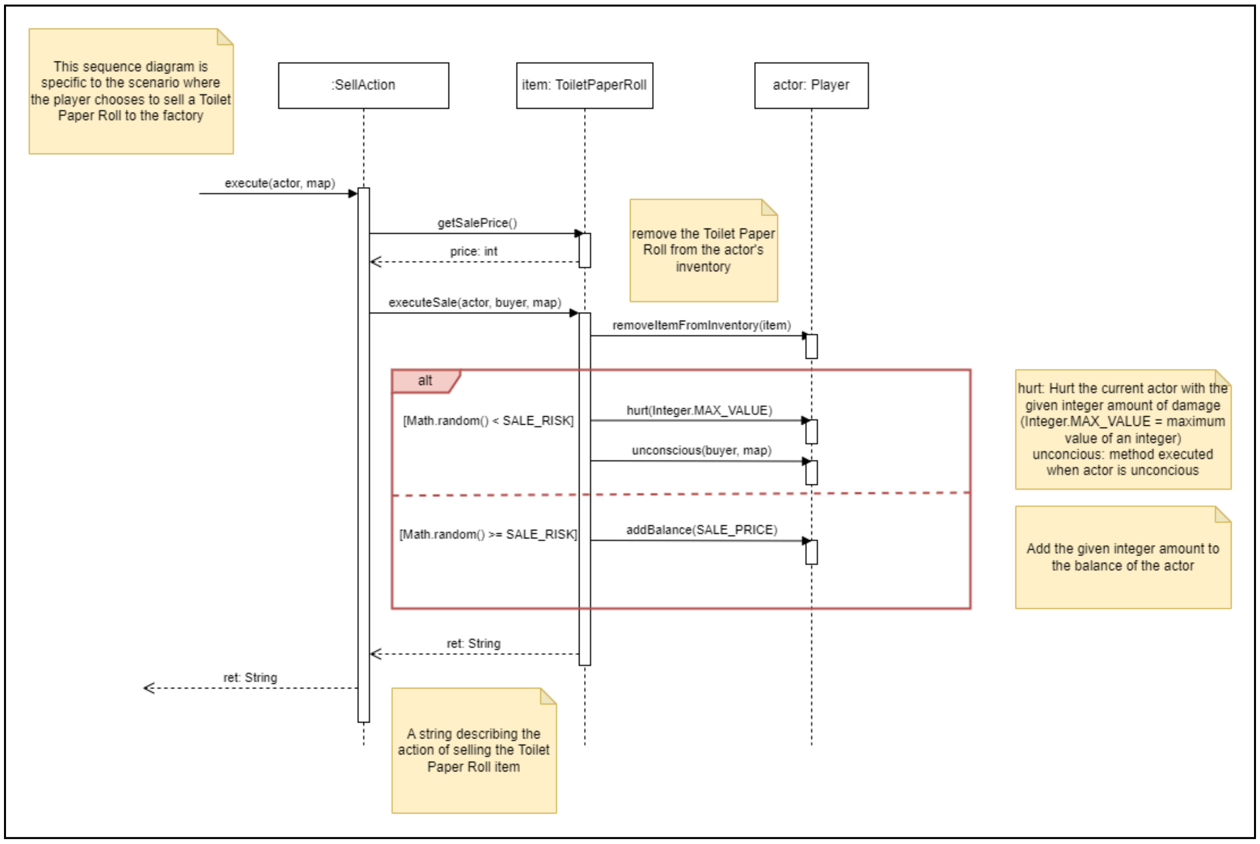

This sequence diagram is specific to the scenario when Player chooses to travel to another location from the computer terminal

:TravelAction
map:GameMap
execute(actor, map)
isAnActorAt(travelToLocation)
alt
map.isAnActorAt(travelToLocation) == True
cannotTravelMessage
map.isAnActorAt(travelToLocation) == False
moveActor(actor, travelToLocation)
menuDescription(actor)

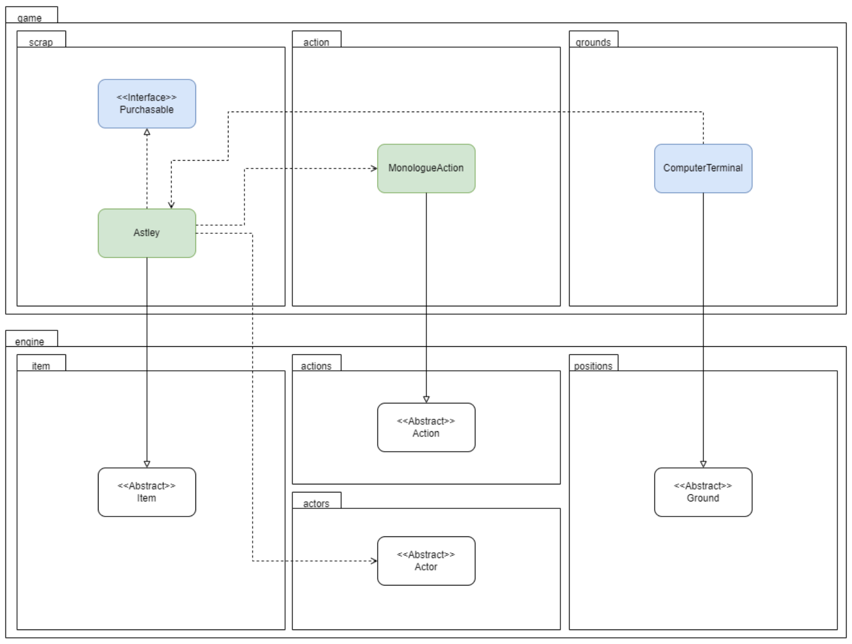
REQ 2 UML Diagram:



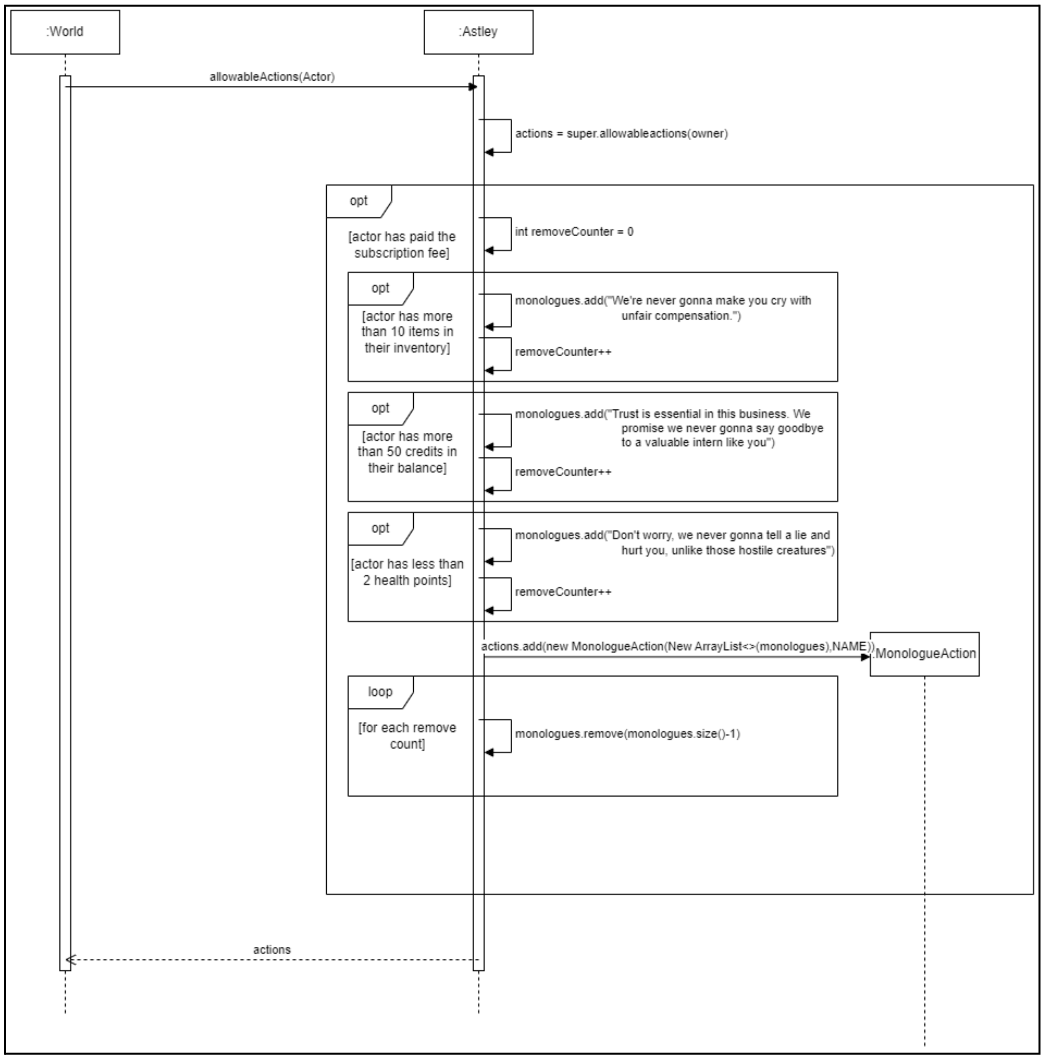
REQ 2 SEQ Diagram:


This sequence diagram is specific to the scenario where the player chooses to sell a Toilet Paper Roll to the factory
:SellAction
item: ToiletPaperRoll
actor: Player
execute(actor, map)
getSalePrice()
price: int
remove the Toilet Paper Roll from the actor's inventory
executeSale(actor, buyer, map)
removeItemFromInventory(item)
alt
[Math.random() < SALE_RISK]
hurt(Integer.MAX_VALUE)
unconscious(buyer, map)
hurt: Hurt the current actor with the given integer amount of damage (Integer.MAX_VALUE = maximum value of an integer)
unconcious: method executed when actor is unconcious
[Math.random() >= SALE_RISK]
addBalance(SALE_PRICE)
Add the given integer amount to the balance of the actor
ret: String
ret: String
A string describing the action of selling the Toilet Paper Roll item

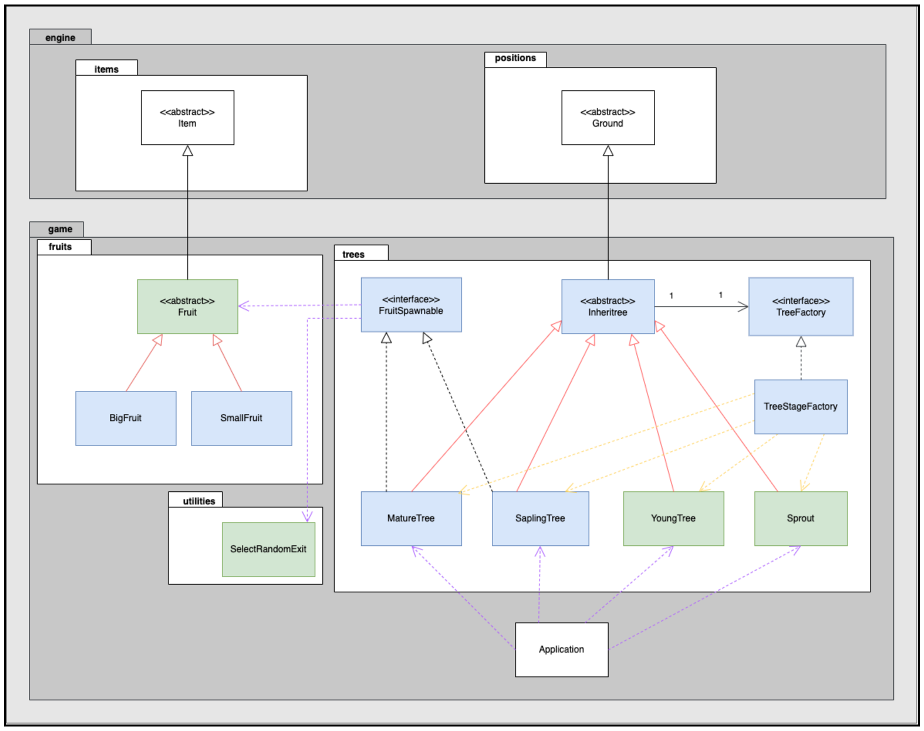
REQ 3 UML Diagram:



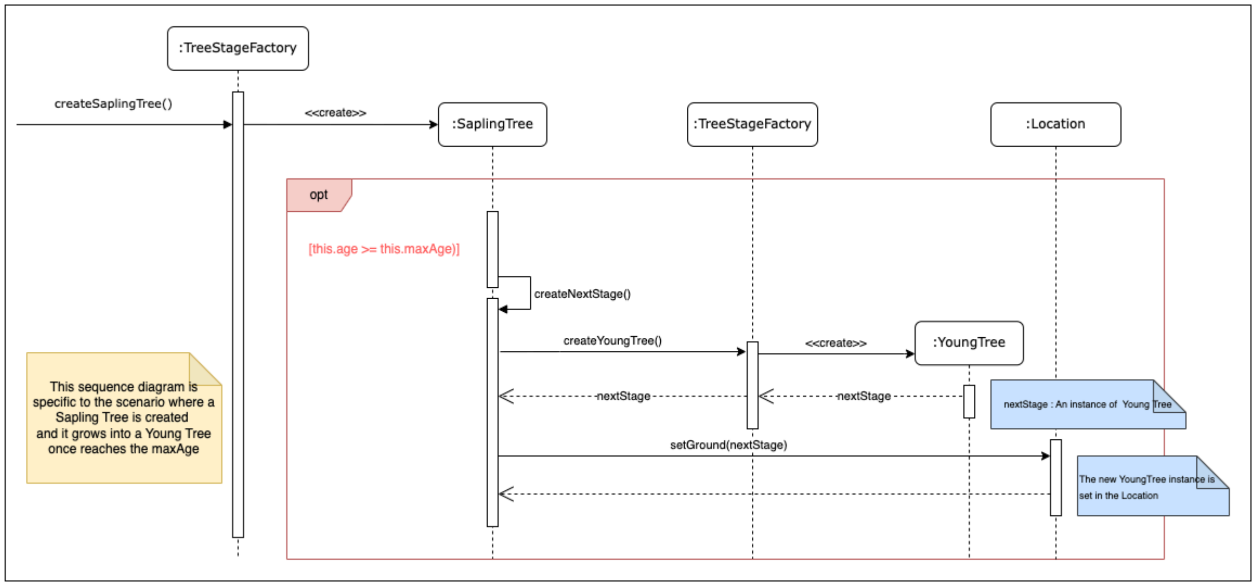
REQ 3 SEQ Diagram:


:World
:Astley
allowableActions(Actor)
actions = super.allowableactions(owner)
opt
[actor has paid the subscription fee]
int removeCounter = 0
opt
[actor has more than 10 items in their inventory]
monologues.add("We're never gonna make you cry with unfair compensation.")
removeCounter++
opt
[actor has more than 50 credits in their balance]
monologues.add("Trust is essential in this business. We promise we never gonna say goodbye to a valuable intern like you")
removeCounter++
opt
[actor has less than 2 health points]
monologues.add("Don't worry, we never gonna tell a lie and hurt you, unlike those hostile creatures")
removeCounter++
actions.add(new MonologueAction(New ArrayList<>(monologues),NAME))
:MonologueAction
loop
[for each remove count]
monologues.remove(monologues.size()-1)
actions

REQ 4 UML Diagram:



REQ 4 SEQ Diagram:


:TreeStageFactory
createSaplingTree()
<<create>>
:SaplingTree
:TreeStageFactory
:Location
opt
[this.age >= this.maxAge)]
createNextStage()
createYoungTree()
<<create>>
:YoungTree
nextStage
nextStage
nextStage : An instance of Young Tree
setGround(nextStage)
The new YoungTree instance is set in the Location
This sequence diagram is specific to the scenario where a Sapling Tree is created and it grows into a Young Tree once reaches the maxAge

REQ1

The goal of this requirement is to add travel to other maps actions to the computer terminal and a new device called a Theseus that can be bought from the computer terminal that can teleport the actor to any location on a map.

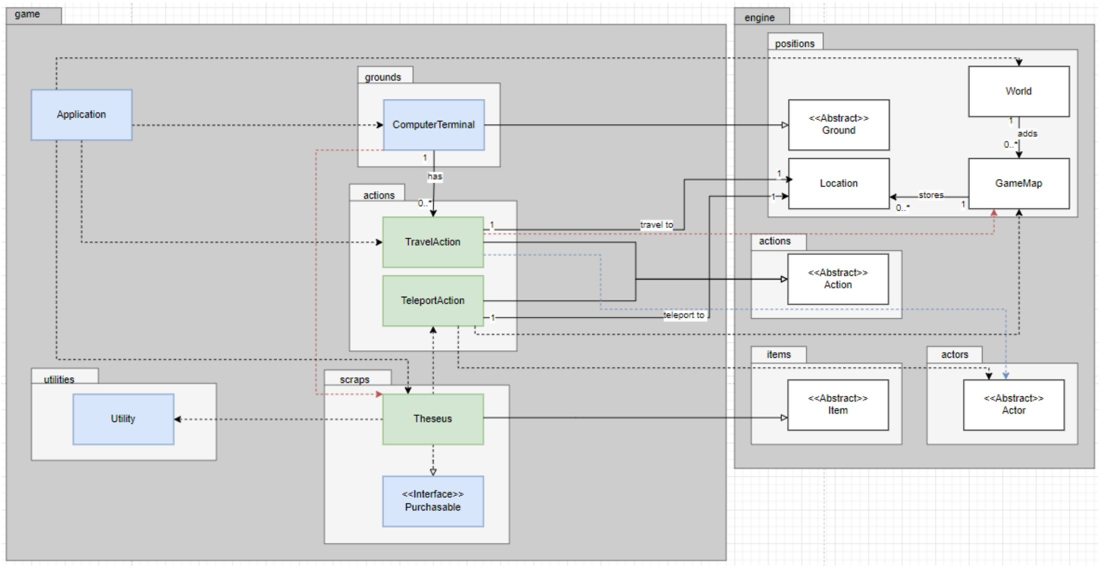
Final design:

ComputerTerminal & TravelAction

In the final design, TravelAction inherits the abstract Action class and executes the logic required for an actor to travel to a different map. The ComputerTerminal has been extended to have an ArrayList of TravelActions available to that ComputerTerminal. The travel actions are manually added to each concrete ComputerTerminal subclass, one line at a time in Application.

Theseus

The Theseus is a concrete class that inherits Item class and implements Purchasable interface. After A2, Purchasable interface has been simplified to having only 2 methods. TeleportAction inherits the Action class and executes the logic required for an actor to teleport across the map. Theseus creates an instance of TeleportAction when an actor is standing next to/on the Theseus, making this action available to the Actor.



Pros	**Cons**
Simple implementation, and each class has its own responsibilities. (SRP)	Manually adding each TravelAction instance to each ComputerTerminal subclass instance can cause Application class to be harder to maintain as it will be long. (Long method)
The TravelAction ArrayList in ComputerTerminal enables easy extensibility of new travel actions as	

the multiplicity is 0..*, as compared to a fixed array of travel actions. The size of the array would need to be constantly changed if more travel actions are possible in the future. (OCP and avoids primitive obsession)	
The new KISS Purchasable interface avoids null returns for the Theseus implementation, ensuring more predictable behaviour and maintainable code. (LSP/Polymorphism)	

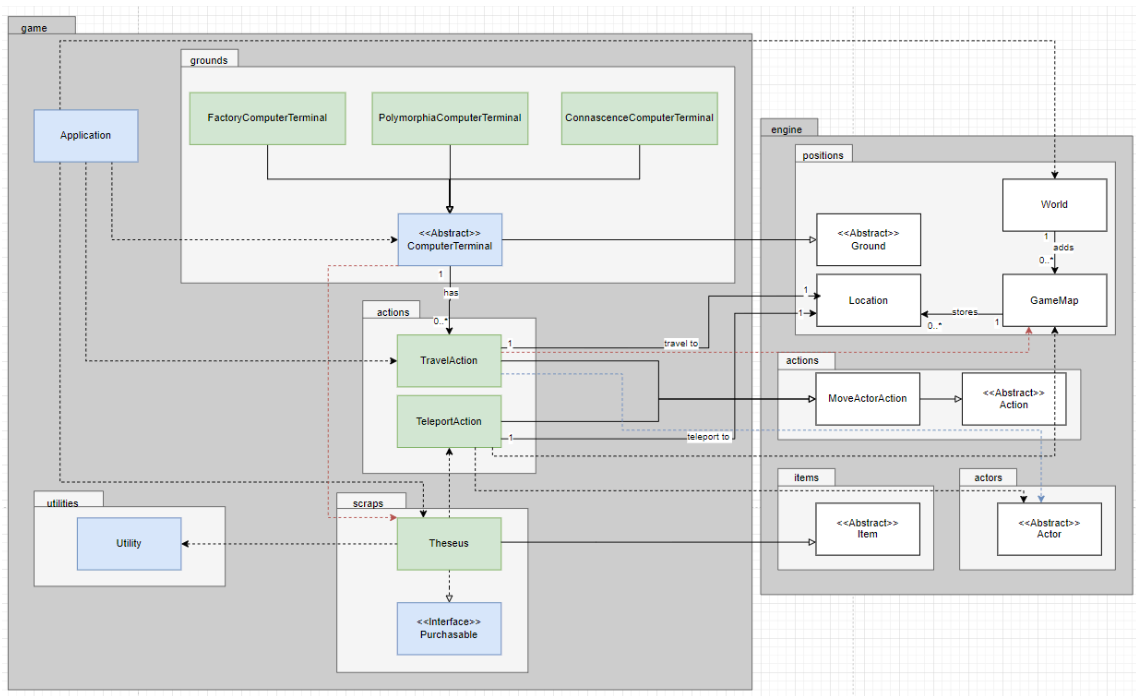
Alternative Design:

ComputerTerminal & TravelAction

In this alternative design, each map has a concrete ComputerTerminal that extends from the ComputerTerminal abstract class. Unlike the final design, the Application class in this design has a nested for-loop that loops over each ComputerTerminal subclass instance, and loops over all possible TravelActions, only adding these TravelActions to the ComputerTerminal subclass instance if the travel action moves the Actor to a different map by checking that the ComputerTerminal's ToString is not equal to the TravelAction's mapName.

Theseus

The Theseus has a similar implementation as the final design, however the difference here is that TravelAction and TeleportAction abstract from MoveActorAction, instead of straight from Action abstract class. Additionally, it uses the original Purchasable interface.



Pros	**Cons**
Travelling to additional maps can be done easily by adding a new instance of TravelAction and injecting the location to travel to and the new map's name to the travelActions ArrayList in the Application. (OCP)	Only adding a TravelAction to a ComputerTerminal if the ComputerTerminal's toString is not equal to the TravelAction's mapName creates Connascence of Values, as these strings are linked. This can be problematic if future changes to these strings break the linkage, causing bugs and requiring extensive refactoring. (CoV)
Extending from ComputerTerminal allows the concrete ComputerTerminal subclasses to avoid repeated code that adds TravelActions and TeleportAction. (DRY)	Connascence of Identity is present as each concrete ComputerTerminal points to the same instance of TravelAction (for example, both FactoryComputerTerminal and RefactorioComputerTerminal have the same TravelAction to Polymorphia), increasing the likelihood of bugs and harder extension. (CoI)
Future maps that have a ComputerTerminal can be instantiated anywhere ComputerTerminal is expected, without modifications to the code. (LSP/Polymorphism)	All methods of MoveActorAction need to be overridden in TravelAction and TeleportAction, as they need to check if an actor is in the location and the return strings are slightly different, so they may not replace MoveActorAction wherever MoveActorAction is expected. (Violates LSP/Polymorphism)
	The Theseus implements Purchasable, but does not use some of the methods or returns null. This violates ISP as it makes the codebase harder to maintain and understand. (overengineering)
	Having separate concrete computer terminals for each map may be overengineering, as each one currently does not have any additional logic other than the ToString. (overengineering)

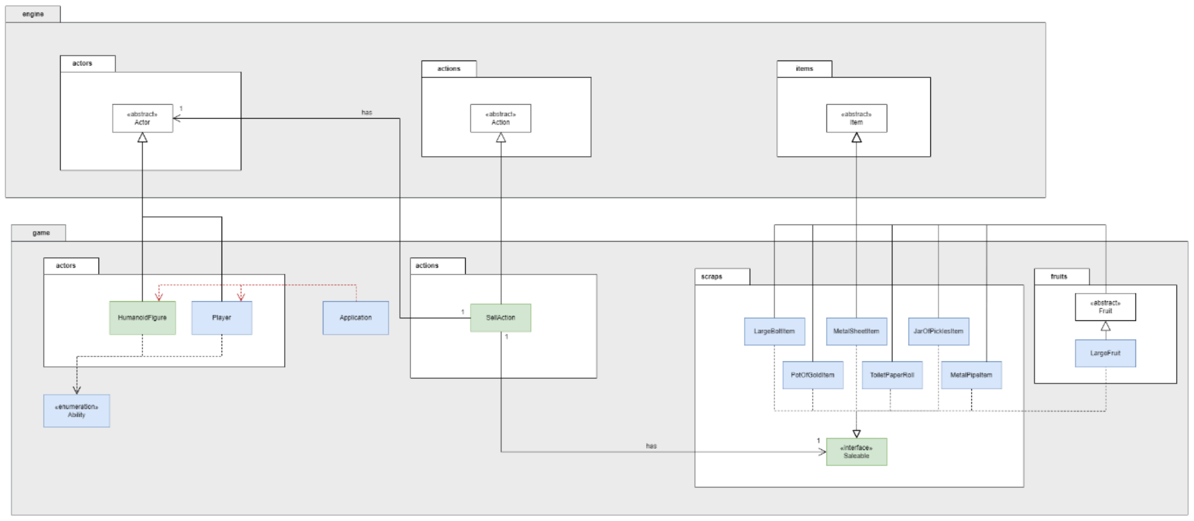
REQ 2

Final Design:

The goal of this requirement is to implement selling items collected by the player to the factory and add a new humanoid figure at the factory's spaceship parking lot, which the player can sell scraps they have collected to if the player is within the surroundings of the entity.

Large Bolt, Metal Sheet, Large Fruit, Jar of Pickles, Metal Pipe, Pot of Gold and Toilet Paper Roll which are concrete classes created in previous assignments by extending abstract Item class from engine, now also implement an interface Saleable, which encapsulates methods that items that can be sold must implement. A concrete class SellAction is created by extending abstract Action class from engine, which is used to execute logic of selling items.

The new Humanoid Figure is implemented as a concrete class that extends Actor. When the player is within an exit away from the Humanoid Figure, the saleable items collected by the player will add a new SellAction to the allowable actions of the player to add an option to sell the item to the figure.

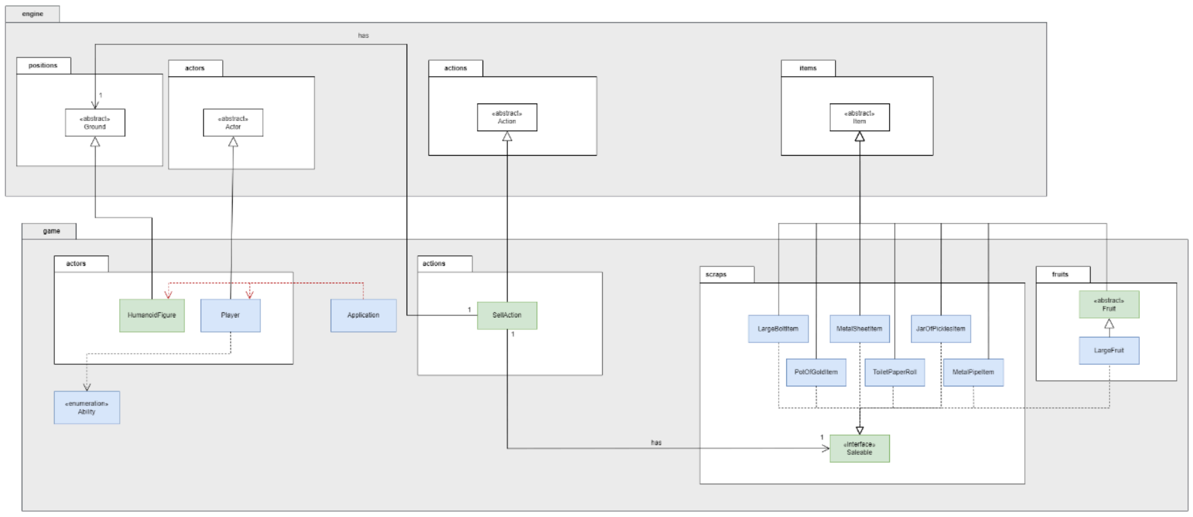


Pros	Cons
Using an interface Saleable to encapsulate logic of items that can be sold allows us to remove dependencies of SellActions to different concrete Saleable items, diverting the dependency onto the interface (DIP).	executeSale method of the Saleable interface takes in two parameters of type Actor, which represent the seller and buyer. If the position of the arguments are swapped when calling the function, it may cause errors as the buyer does not have the item to be removed from its inventory (Connascence of Position).
Dependencies of classes that interact with items that can be sold are diverted to the Saleable interface, do not need to modify code	May cause shotgun surgery as duplicated logic is present in the execution of selling items, changing a part of the selling logic would lead

in SellAction class if new items that can be sold are added (OCP).	to multiple small changes to methods in different classes.
Higher code reusability, using a SellAction class for every item that can be sold avoids repetitive code of similar sale execution logic, extending abstract Actor class to create Humanoid Figure avoids repeating logic for checking if the figure is an exit away, we just reuse the checking mechanism with an extra added check for capability to by items(DRY).	
SellAction class is only responsible for handling item selling transactions between the seller and buyer, the item itself defines its own selling logic (SRP).	
Humanoid Figure can be easily extended to perform more actions, since base methods for executing allowable actions are already implemented in abstract Actor class.	

Alternative Design:

In this design, the main difference with the final design is the implementation of Humanoid Figure. Humanoid Figure is implemented as a concrete class that extends abstract Ground class from engine. When an actor is an exit away, the figure adds a new SellAction for every saleable item in the player’s inventory to the list of allowable actions. The implementation of other features is the same as the final design used.



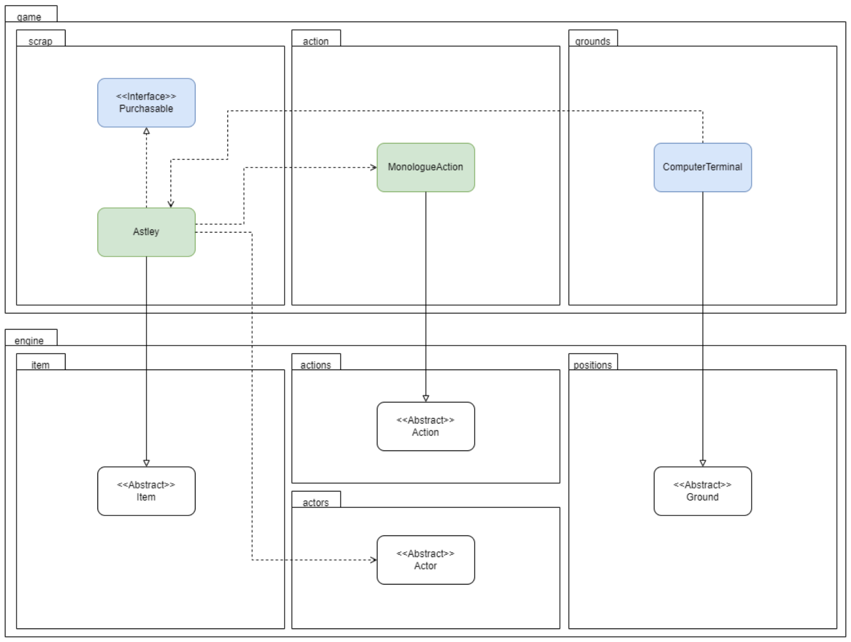
Pros	**Cons**
Since the Humanoid Figure does not move, creating the class by extending Ground allows us to fully utilize methods in the parent class, while making the Humanoid Figure responsible for detecting if an entity is an exit away. This reduces the responsibility put on other classes for checking (SRP).	When creating new SellAction with items in the player's inventory, we need to downcast the type of the item to type Saleable to be able to create the SellAction.
	Humanoid Figure does behave and have a few more responsibilities than regular Ground classes, overriding existing base class methods might result in violation of LSP, where Humanoid Figure might change base methods to an extent where it is a completely different class.
	Harder to extend in the future when we want to allow the Humanoid Figure to perform other actions.

REQ 3

The goal of the requirement is to add a new item named Astley that can monologue/talk to the player if the player pays the subscription for the item.

Final design:

The subscription requirement was implemented as a method inside the Astley class alongside the decision making for which monologues could be selected. Thus, monologueAction's only responsibility is randomly selecting a monologue from a list and displaying that chosen monologue.



Subscription requirement implemented as a method in Astley class

Pros	**Cons**
Prevents over-engineering and keeps the codebase lean by avoiding unnecessary complexity of creating a subscriptionAction that may or may not be needed in the future. (YAGNI)	Keeping all decision logic within the Astley class can lead to a god class, where the class takes on too many responsibilities, making it harder to manage and maintain. However, all decision logic is exclusive to the Astley class. (God class and could violate SRP)
Keeps the design simple and straightforward, by keeping the method in the only class that needs it. (KISS)	
Ensures that all related functionalities of the Astley class are contained within the class,	

leading to a more modular and cohesive code structure. (Encapsulation)	

MonologueAction only randomly selects a monologue from an ArrayList of monologues

Pros	**Cons**
The MonologueAction class focuses solely on managing monologue actions, making it more cohesive and easier to manage. (SRP)	Potential for duplicated code if multiple classes need to implement similar monologue decision logic but with different numerical values (Potentially violating DRY)
Keeping the decisions of which monologues can be randomly selected inside Astley instead of in monologueAction allows for extensibility without needing multiple subclasses of monologueActions.	
Improves reusability by ensuring that the MonologueAction class can be used with any set of monologues without modification	

Alternative design:

A subscriptionAction interface and an AstleySubscriptionAction is created to implement the subscription of the Astley

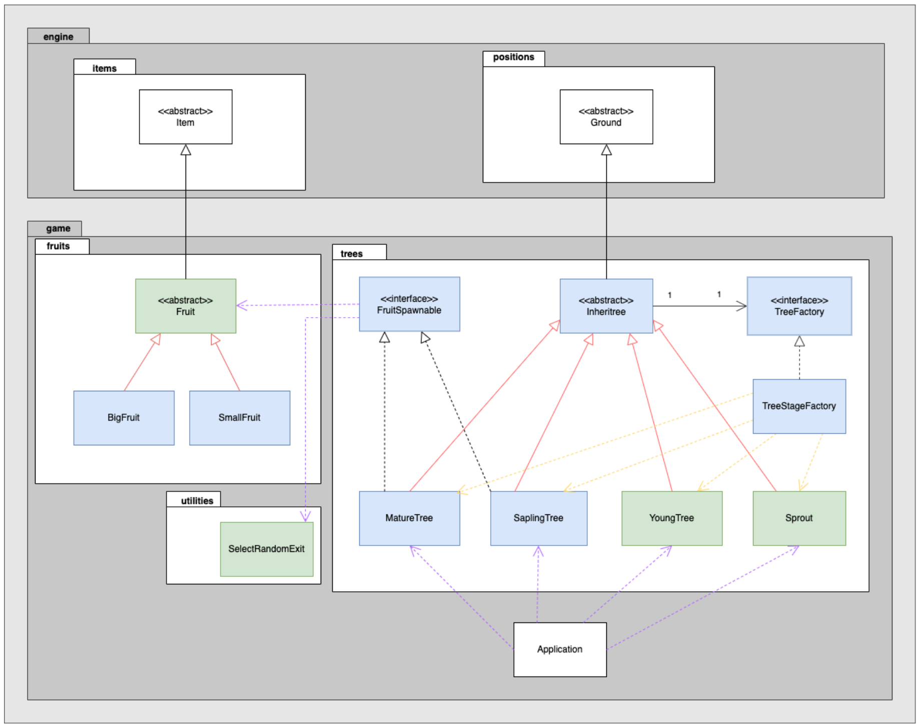
Pros	**Cons**
By creating a SubscriptionAction class, the subscription logic is separated from the Astley class, ensuring that each class has a single responsibility. (SRP)	If no other items in the future needs a subscription this level of abstraction would be excessive (YAGNI + KISS)
	Depending on the implementation either the subclasses would have very similar logic or the subclasses may be forced to implement unused methods (Potentially violating DRY and ISP)

An abstract monologueAction is created with the same execute and menuDescription methods as the one in the final design, however, with an added “monologueDecision” method that edits the arrayList of monologues depending on the logic inside the method.

Pros	**Cons**
Both Astley and AstleyMonologueAction classes will only have to focus on a single responsibility. (SRP)	If no other items/creatures in the future have the ability to monologue this level of abstraction would be excessive (YAGNI + KISS)
	The abstract MonologueAction class may force all subclasses to implement the monologueDecision method, even if they don't need it. (ISP)

REQ 4

Final Design:
In this design, the TreeStageFactory is central to the creation of various tree stages, it is responsible for generating instances of different tree stages. This ensures that all tree instances are created in a consistent and controlled manner. Inheritee abstract class serves as the blueprint for all other tree stages. . It encapsulates the common properties and behaviours of trees, such as managing the tree's age and handling the transition to the next stage. It uses dependency injection to get a reference to TreeStageFactory. The MatureTree, Sapling Tree, YoungTree and Sprout class extends the abstract Inheritree class and implements the specific logic for transitioning to the next stage by overriding the createNextStage method to use TreeStageFactory to create the next stage.

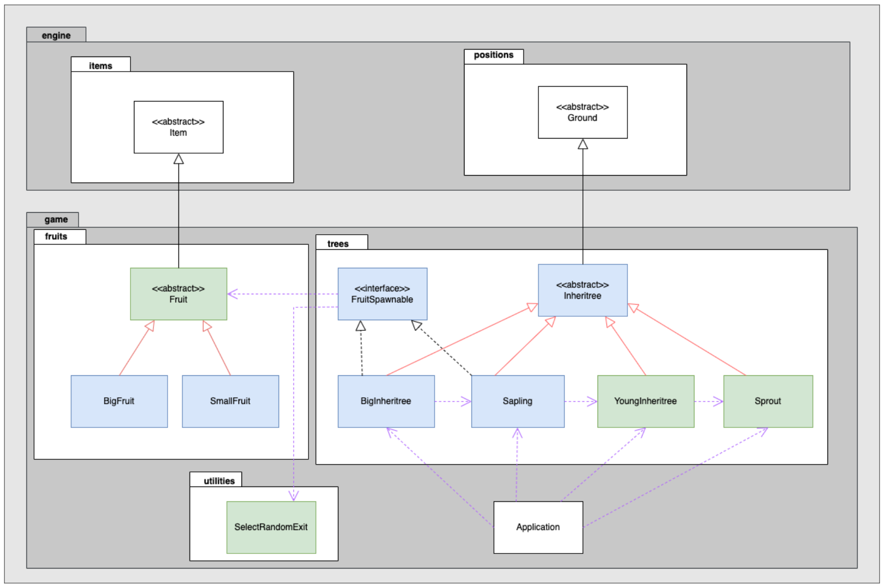


Pros	**Cons**
The creation logic is moved to the factory, hence, concrete classes are only responsible for the specific behaviours of one tree stage. (Single Responsibility Principle)	The use of a factory and dependency injection adds layers of abstraction, which can make the code harder to follow and add to the complexity of the system.
Concrete classes now depend on the TreeFactory interface; they do not need to know the details of	

how the tree stages are created, promoting loose coupling and flexibility.	
The system is open for extension but not for modification. Additional tree stages can be added easily without changing the existing code. New tree stages can be added by updating the factory. (Open-Closed Principle)	
In the previous design, tree classes need to know the names of other tree classes they transition to directly. This design the tree class uses Tree Factory to create the next stage, hence, reducing the need to know specific class names directly. (Connascence of Name)	
Having a tree factory interface makes the tree classes depend on an interface rather than a concrete class, making the code more modular and also allow easy testing as the factory can be replaced with a mock object.	

Alternative Design:

The goal is to design a system that is extensible and flexible to simulate growth of trees and fruit production of the trees. Each tree extends the abstract Inheritee class which encapsulates the general behaviour of the trees in the system. The abstract Inheritree class takes care of the ageing and transitioning of the tree. Each concrete tree class specifies its own transition logic by instantiating the next stage directly within its constructor. Trees that can produce fruits implement the FruitSpawnable class which provides a default method to produce fruits. This interface allows different trees to produce fruits based on the specific spawn chance and fruit type.



Pros	**Cons**
It is straightforward as the object is instantiated directly, making it easier to understand and follow.	The direct instantiation of next stages within constructors creates a high coupling between classes.
The abstract Inheritree class for every tree avoids code duplication. (DRY principle)	Each class is responsible for its own creation and life cycle management, violating Single Responsibility Principle.
The default method in FruitSpawnable interface ensures that the fruit spawning logic will not be repeated across tree classes that can produce fruits (DRY principle). Also it ensures consistent produce fruit behaviour across fruit-producing trees, which is beneficial for the maintainability of the system.	The use of multiple layers and interfaces might introduce complexity that could complicate the onboarding process for new developers.
The FruitSpawnable class ensures that trees that don't have fruit spawning capabilities are not required to implement the unnecessary method. (Liskov Substitution Principle)	High connascence of name and type, as each class needs to know the exact class names and types of the stages it transitions to. For example, if renaming SaplingTree to something else, have to update every instance which it is referenced directly.
The logic of searching for an exit to drop a fruit has been implemented in the SelectRandomExit utility class, hence, has avoided code duplication and ensured logic consistency across all classes. (DRY principle)	When creating the new tree object, there position of the parameter in the constructor has to be adhered to strictly, causing connascence of position.