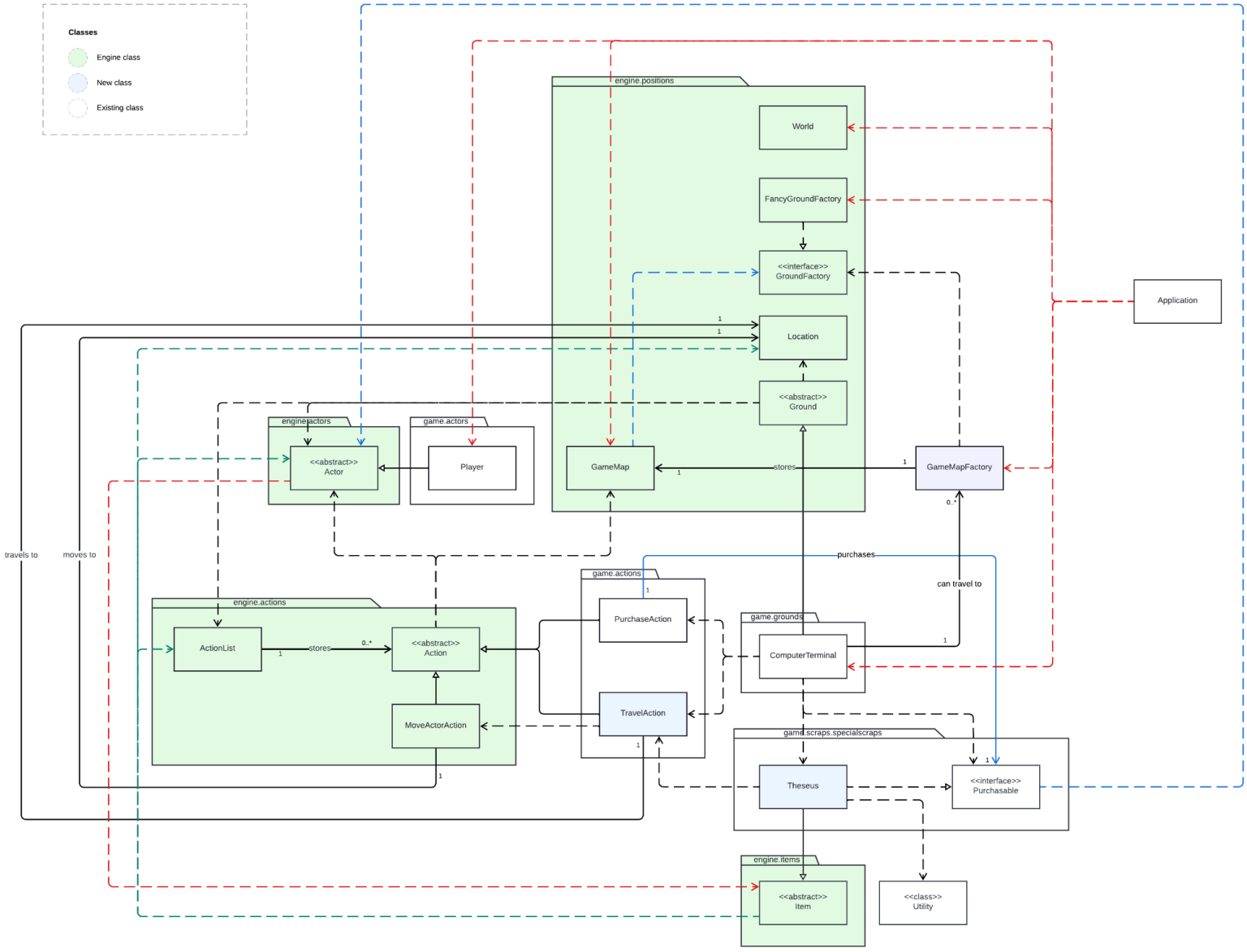
Design Rationale

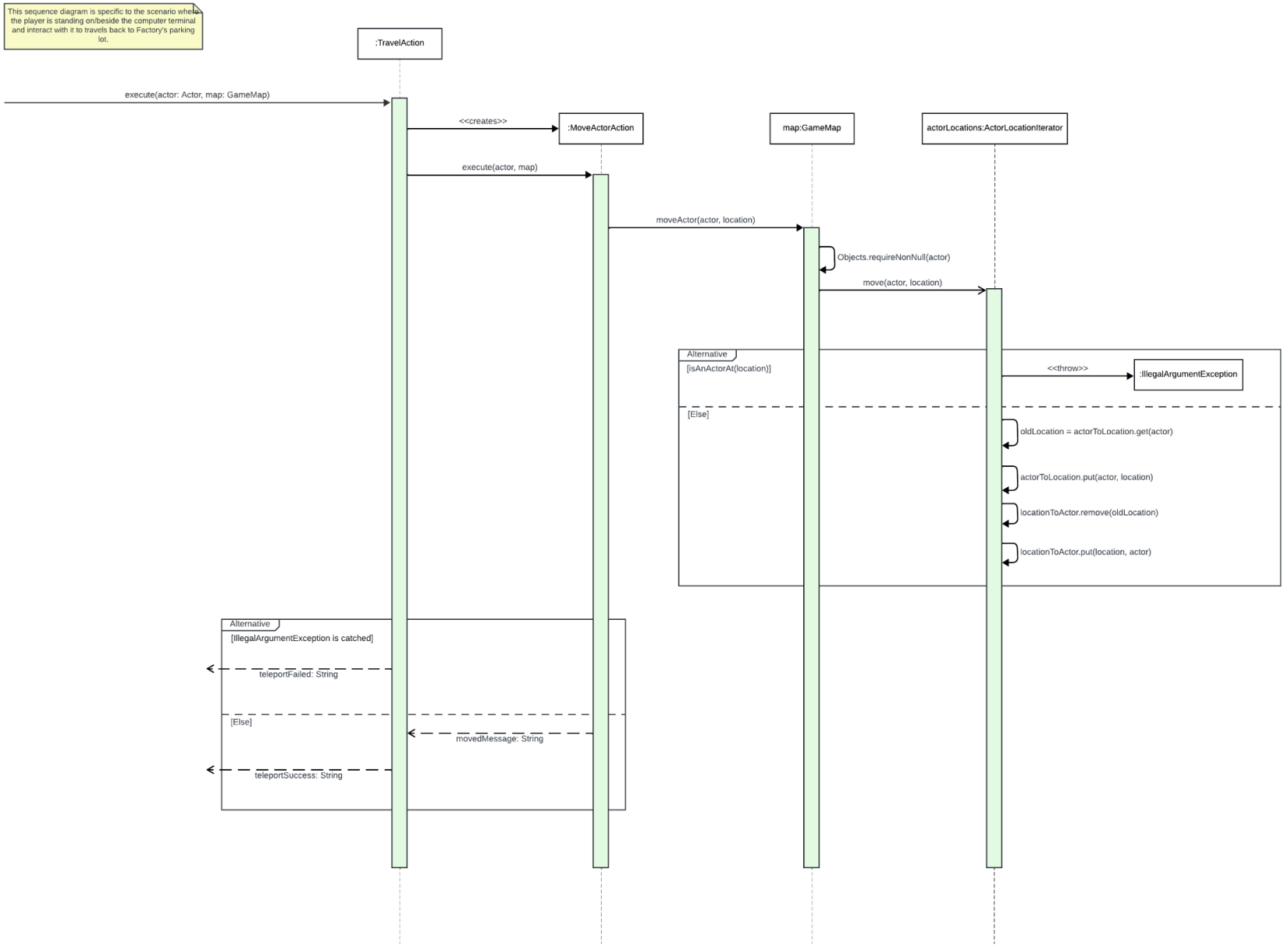
Design Goal

The design aims to break down the system into smaller, more manageable classes to promote modularity and encapsulation. In this system, each class will have a clear and distinct responsibility so it is easier to understand and maintain. By adhering to the SOLID principle and DRY principle, the system should allow for easy extension and promote reusability. This involves identifying common functionalities and abstracting them into reusable components as well as anticipating potential changes in order to enhance code flexibility and adaptability to changing requirements and future extensions.

Req 1 - The Ship of Theseus



Requirement 1 Class Diagram



Requirement 1 Sequence Diagram: This sequence diagram is specific to the scenario where the player is standing on/beside the computer terminal and interact with it to travels back to Factory's parking lot.

<table>
<tr><th>Classes Modified / Created</th><th>Roles and Responsibilities</th><th>Rationale</th></tr>
<tr><td>GameMapFactory</td><td>A concrete class creating and managing a GameMap.

Relationships:
1. It is associated with GameMap to store the map it is managing.
2. It depends on GroundFactory interface to create a GameMap.</td><td rowspan="3">Alternative Solution:
1. Multiple maps feature:
- Store a HashMap in the Computer Terminal class to handle the location and map name it teleports to. In the Application class, pass in these two arguments when instantiating the ComputerTerminal object.
- Create a concrete class for each map to handle the details of the specific map. This class will store the map name, teleport location and also the string demonstrating the structure of the map.

2. Teleport feature:
- Override the allowableActions() method in ComputerTerminal class by adding the MoveActorAction to the ActionList before returning it if the new location to teleport to has no actor standing on it.

Finalised Solution:
1. Multiple maps feature:
Create a concrete class GameMapFactory to manage the details of the GameMap, including the map name and the location to teleport to through computer terminal. In ComputerTerminal class, store a list of GameMapFactory for each map it can teleport to. The map string is declared in the Application class.

2. Teleport feature:
Create a concrete class TravelAction which inherits from the Action abstract class to implement the details of the teleport action. The TravelAction class implements the abstract method execute() by creating and executing the MoveActorAction to teleport the actor to the new location. It catches the exception error thrown and returns a teleport failure message.

Reasons for Decision:</td></tr>
<tr><td>TravelAction
(extends Action)</td><td>Class representing the action to travel to a new location within the map or across two maps.

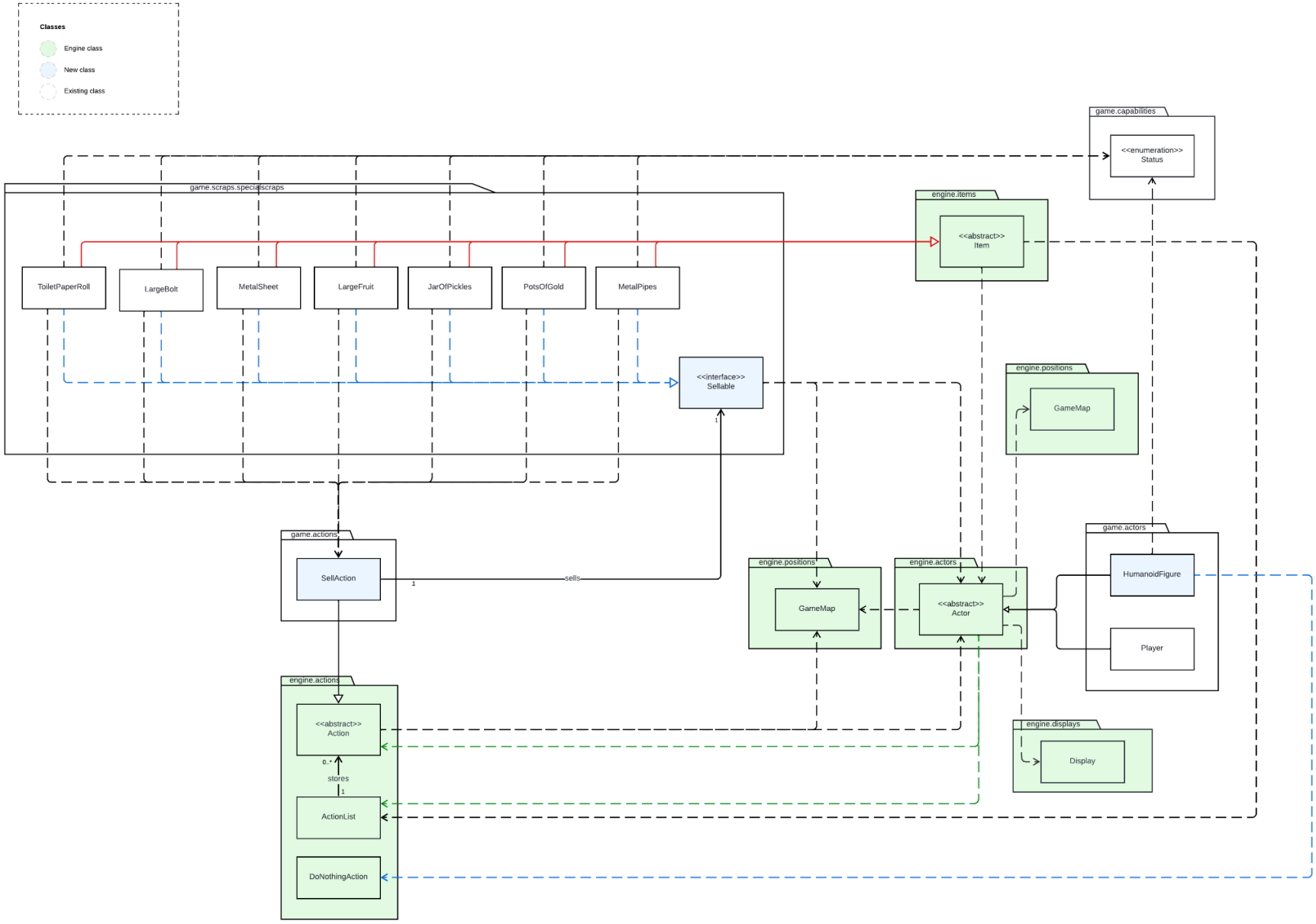
Relationships:
1. It is associated with Location to store the new location to send the intern to.
2. It depends on MoveActorAction to move the actor to the new location.</td></tr>
<tr><td>ComputerTerminal
(extends Ground)</td><td>Class representing the computer terminal where intern can interact with it.

Relationships:
1. It depends on GameMapFactory class to store the map where it</td></tr>
</table>

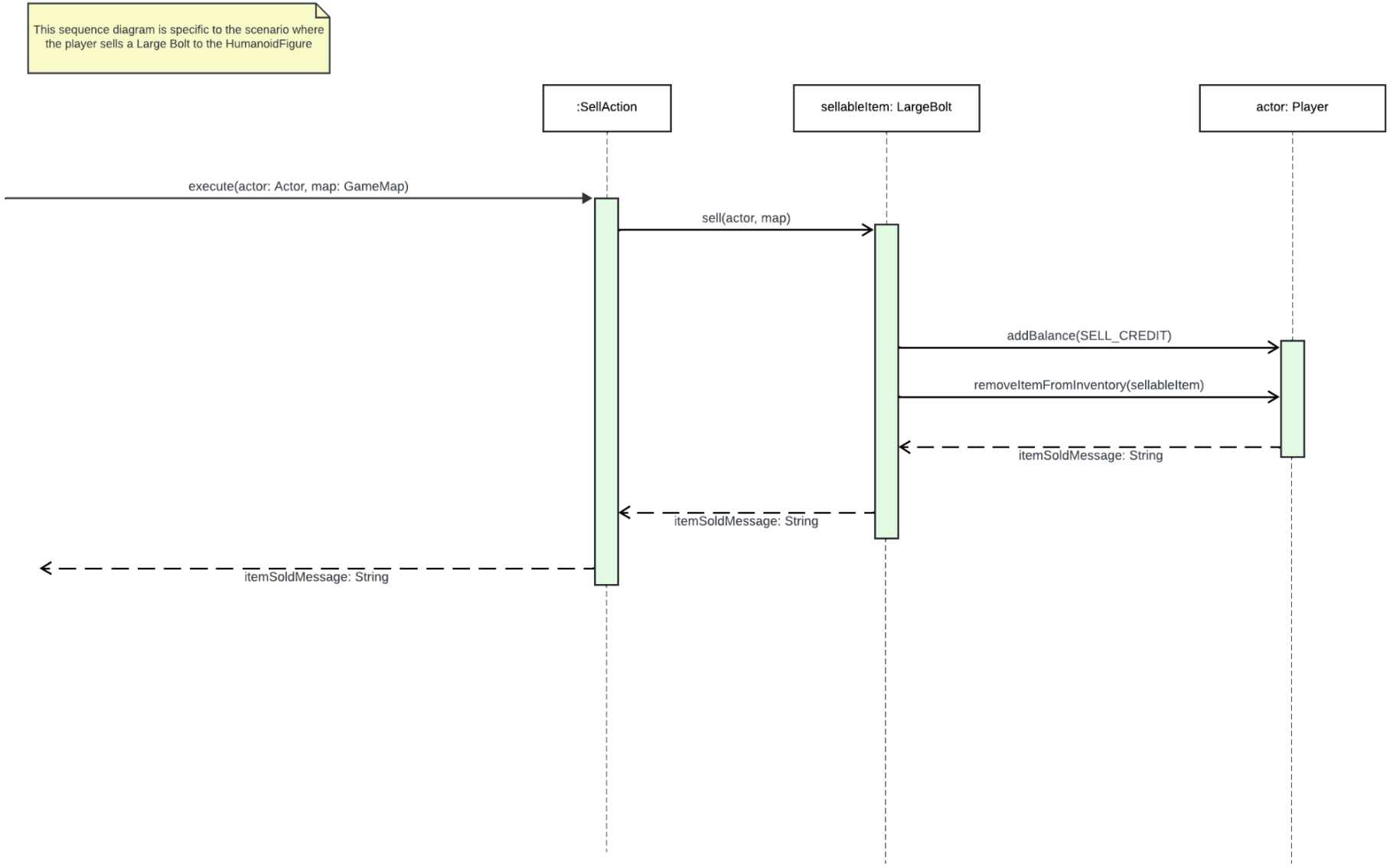
	can teleport the intern to. 2. It depends on TravelAction class to teleport intern to the specified map. 3. It depends on PurchaseAction to allow the intern to make the purchase with it. 4. It depends on Theseus class to allow the intern to purchase THESEUS from it.	1. By using a single class to manage the details of the map, the dependency of the map name across multiple modules can be reduced. This approach centralises the map-related information so any changes to the map name are only made in one place. This minimises the risk of errors such as spelling mistakes which can occur if multiple modules refer to the map name directly. Additionally, declaring the map string in the Application class keeps the map string closer to the code that sets the actors and ground in the map, which requires specific locations to avoid the related value spread across different parts of the code. This current setup better protects the invariant and reduces the risk of errors. *(Connascence of Value)* 2. Based on the requirement, the computer terminal should allow the intern to travel to another map and return a failure message if the new location teleported to contains an actor. However, the alternate approach only provides the options of valid teleport for the player without the teleport fail scenario. Thus, the current approach creates a TravelAction to provide all the teleport map options in the menu and do the checking after the selection and before teleporting the intern. 3. Changing the parameters from location and map name to only the GameMapFactory shifts the responsibility of knowing the parameter order from the Application to the GameMapFactory, effectively reducing connascence of positions. *(Connascence of Position)* 4. A concrete class is created for the THESEUS to implement all the related details instead of directly implementing them in the ComputerTerminal class to ensure that the ComputerTerminal class focuses only on its primary function to enhance code maintainability. (eg. ComputerTerminal has no responsibility to teleport the intern within the current map) *(Single Responsibility Principle)* 5. The Purchasable interface is small enough and contains only the methods related to purchase. Hence, subclasses that implement it are not forced to implement
Theseus (extends Item)	Class representing the THESEUS that can teleport the intern within the current map. Relationships: 1. It implements Purchasable interface to make itself purchasable by the intern through computer terminal. 2. It depends on TravelAction class to allow the intern to be teleported within the current map. 3. It depends on Utility class to generate the random location to teleport.	

		methods they are not required to (eg. Theseus can be purchased but not consumable). *(Interface Segregation Principle)* 6. Storing a list of GameMapFactory instead of multiple lists of type factory classes of different maps prevents the modification of the ComputerTerminal class when a new map is introduced. *(Open-closed Principle)* 7. By implementing the Purchasable interface, the ComputerTerminal no longer has to depend on the THESEUS class, instead, it only depends on the abstraction Purchasable which in turn reduces the coupling between the two classes. *(Dependency Inversion Principle)* **Limitations and Tradeoffs:** 1. The creation of GameMapFactory class might lead to overengineering since the main usage of the class is only managing the map name and teleport location. This could lead to extra effort in maintaining the class without significant benefits.

Req 2 - The Static Factory



Requirement 2 Class Diagram



Requirement 2 Sequence Diagram: Scenario where the player sells a Large Bolt to the HumanoidFigure

Classes Modified / Created	Roles and Responsibilities	Rationale
HumanoidFigure (extends Actor)	Class representing the HumanoidFigure as an Actor, who can purchase items on behalf of the factory. Spawned only in the parking lot. Relationships: 1. It extends Actor in order for another Actor to engage with it in order to sell items. 2. Has the (enum) status of *BUYER*, which prevents it from being attacked (it is not *HOSTILE_TO_INTERN*) and enables for a sale transaction to occur.	**Alternative Solution:** Create a SellAction class which inherits from the Action abstract class. Implement all the details of selling different items in the execute() method using multiple if statements with instanceof() to check which item to sell and perform the logic of selling that item. **Finalised Solution:** A 'HumanoidFigure' class was created, which extends Actor, and has the '*BUYER*' enum status. A 'Sellable' interface was created with one method: sell(otherActor, map), which left as a blueprint for items to implement their respective logic for the sale transaction to occur (probabilities, 'killing'). Within each item as specified in REQ2, they all implement the sell() method, which has the unique logic for that item. This entails calculating and adding the credits for the sale to the actor who sold the item and removing the item from their inventory, with some classes having more unique requirements as outlined in the specifications. A SellAction class was created (extending Action) to represent the action of selling a sellable item, an attribute in SellAction through dependency injection. It overrides the execute() method by invoking the sell() method in the sellable item class to perform the sell action. Within each (sellable) item's class, the method allowableActions(Actor otherActor, Location location), checks for whether another actor (in this case, the HumanoidFigure) has the *BUYER* status. If so, the SellAction() for that item is added to the list of actions the player can choose from. **Reasons for Decision:** 1. Using the finalised approach, when a new sellable item is introduced, the modification on the SellAction class can be avoided. The new class only needs to implement the Sellable interface and the sell() method to be invoked by the
Sellable	Interface representing an item which can be sold. Relationships: 1. It has a dependency with the Actor class (otherActor), which represents the actor attempting to sell an item. 2. It has a dependency with GameMap (map) to represent the map that otherActor is in when they chose to sell (SellAction) the item.	
SellAction (extends Action)	Class extending Action which represents the sell action. Relationships: 1. Extends Action so that 'sell' action can be an option. It overrides the execute() and menudescription() method.	

	2. Association with Sellable. Via a dependency injection, an item which is Sellable, is an attribute of SellAction, so that such an item can be sold.	SellAction class. This reduces the risk of introducing bugs into the SellAction class when new sellable item is introduced which might break the entire logic of the action and affect the selling capabilities of other items. *(Open-closed Principle)* 2. SellAction() is only responsible for returning strings corresponding to the sell action, and only manages actions related to the sale of an item. Furthermore, Sellable interface has only the single responsibility of allowing its child classes to implement the sell() method, with each implementation for each sellable item containing the unique conditions for the sale of that item. *(Single Responsibility Principle)* 3. Any instance of a class that implements Sellable interface (i.e. any Item that uses sell()), can be used wherever an item of interface Sellable is expected within the program. Consequently, sellable items can be substituted with the Sellable type in attributes of the SellAction class without breaking their expected behaviour as well as the whole system. *(Liskov Substitution Principle)* 4. The Sellable interface, having only sell() method, is small enough and contains only the methods needed by the sellable item which implements it, in other words, subclasses that implement it are not forced to implement methods they are not required to. This enhanced the clarity and maintainability of code compared to having a large interface for all sellable items. *(Interface Segregation Principle)* 5. For a sale to occur, we created a Sellable() interface and got the items themselves to implement the sell() method. This removes the direct dependency from the SellAction and the item they're trying to sell. Conversely, the SellAction class is able to stores the item of single type Sellable. *(Dependency Inversion Principle)* Additionally, the sellableItem is passed to the SellAction using constructor injection which helps in prevents the
LargeBolt (extends Item) (implements Sellable)	Class representing a Large Bolt. Relationships: 1. Extends the item class so it can have the sellable functionality. It overrides the allowableActions() method to add a SellAction(). 2. Implements Sellable so that the unique constraints for its purchase can be performed for a sale. This is done in the sell() method. 3. Dependency with (enum) Status. In the allowableActions() method, it checks for whether the otherActor can purchase the item (this). 4. Dependency with SellAction. In allowableActions(), if the otherActor has enum *BUYER,* then a new SellAction corresponding to that item is created.	
BigFruit (extends Item) (implements Sellable)	Class representing a Big Fruit. Relationships: 1. Extends the item class so it can have the sellable functionality. It overrides the allowableActions() method to add a	

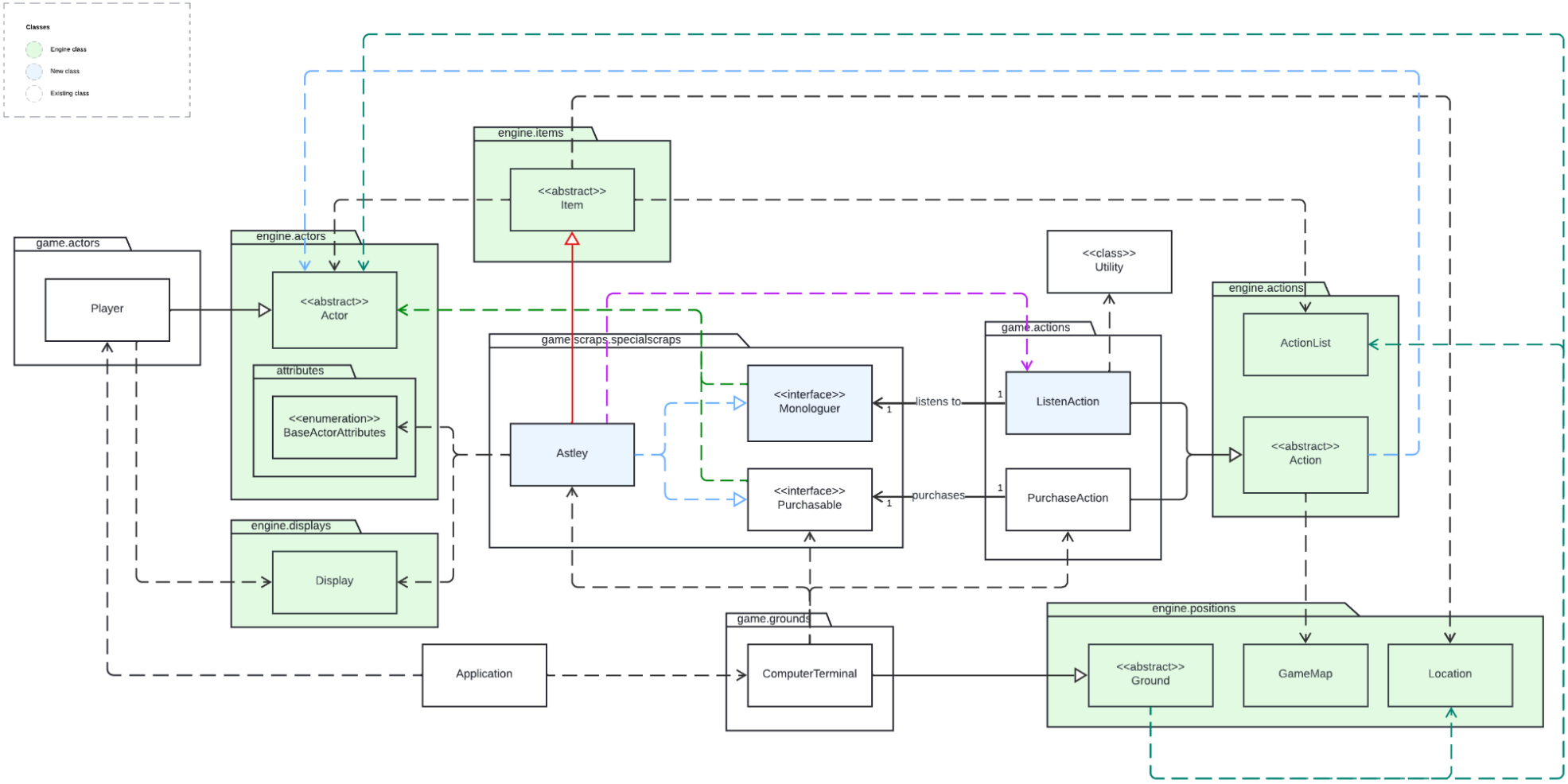
	SellAction(). 2. Implements Sellable so that the unique constraints for its purchase can be performed for a sale. This is done in the sell() method. 3. Dependency with (enum) Status. In the allowableActions() method, it checks for whether the otherActor can purchase the item (this). 4. Dependency with SellAction. In allowableActions(), if the otherActor has enum *BUYER,* then a new SellAction corresponding to that item is created.	high-level module SellAction tight coupling with the low-level module item class. 6. Each of the unique values such as the number of credits and the probabilities of special cases are labelled with an appropriate name and stored as a constant to avoid magic numbers. *(Connascence of Meaning)* **Limitations and Tradeoffs:** 1. Since each Sellable Item implements its own sell() method with its respective selling logic, this can become challenging to manage if the game were to scale (say, there were 2000 items which could be sold), making an excessive amount of classes required to maintain.
MetalSheet (extends Item) (implements Sellable)	Class representing a Metal Sheet. Relationships: 1. Extends the item class so it can have the sellable functionality. It overrides the allowableActions() method to add a SellAction(). 2. Implements Sellable so that the unique constraints for its purchase can be performed for a sale. This is done in the sell() method. 3. Dependency with (enum) Status. In the allowableActions() method, it checks for whether the otherActor can purchase the item (this). 4. Dependency with SellAction. In allowableActions(), if the otherActor has enum *BUYER,* then a new SellAction	

	corresponding to that item is created.	
JarOfPickles (extends Item) (implements Sellable)	Class representing a Jar of Pickles. Relationships: 1. Extends the item class so it can have the sellable functionality. It overrides the allowableActions() method to add a SellAction(). 2. Implements Sellable so that the unique constraints for its purchase can be performed for a sale. This is done in the sell() method. 3. Dependency with (enum) Status. In the allowableActions() method, it checks for whether the otherActor can purchase the item (this). 4. Dependency with SellAction. In allowableActions(), if the otherActor has enum *BUYER,* then a new SellAction corresponding to that item is created.	
MetalPipe (extends Item) (implements Sellable)	Class representing a Metal Pipe. Relationships: 1. Extends the item class so it can have the sellable functionality. It overrides the allowableActions() method to add a SellAction(). 2. Implements Sellable so that the unique constraints for its purchase can be	

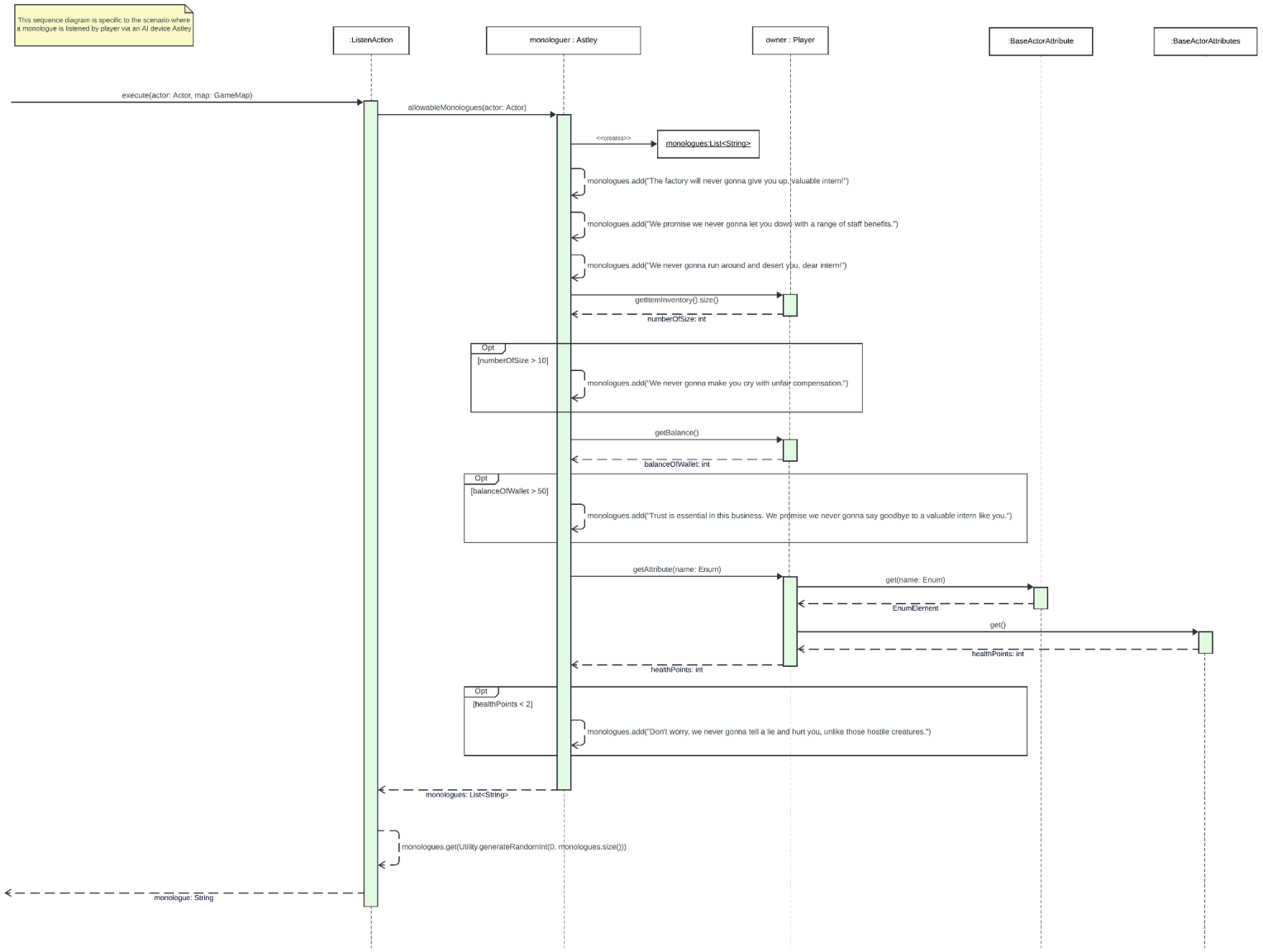
	performed for a sale. This is done in the sell() method. 3. Dependency with (enum) Status. In the allowableActions() method, it checks for whether the otherActor can purchase the item (this). 4. Dependency with SellAction. In allowableActions(), if the otherActor has enum *BUYER,* then a new SellAction corresponding to that item is created.	
PotOfGold (extends Item) (implements Sellable)	Class representing a Pot of Gold. Relationships: 1. Extends the item class so it can have the sellable functionality. It overrides the allowableActions() method to add a SellAction(). 2. Implements Sellable so that the unique constraints for its purchase can be performed for a sale. This is done in the sell() method. 3. Dependency with (enum) Status. In the allowableActions() method, it checks for whether the otherActor can purchase the item (this). 4. Dependency with SellAction. In allowableActions(), if the otherActor has enum *BUYER,* then a new SellAction corresponding to that item is created.	

ToiletPaperRoll (extends Item) (implements Sellable)	Class representing a Toilet Paper Roll. Relationships: 1. Extends the item class so it can have the sellable functionality. It overrides the allowableActions() method to add a SellAction(). 2. Implements Sellable so that the unique constraints for its purchase can be performed for a sale. This is done in the sell() method. 3. Dependency with (enum) Status. In the allowableActions() method, it checks for whether the otherActor can purchase the item (this). 4. Dependency with SellAction. In allowableActions(), if the otherActor has enum *BUYER,* then a new SellAction corresponding to that item is created.	

Req 3 - dQw4w9WgXcQ



Requirement 3 Class Diagram



Requirement 3 Sequence Diagram: This sequence diagram is specific to the scenario where a monologue is listened by player via an AI device Astley

<table>
<tr><th>Classes Modified / Created</th><th>Roles and Responsibilities</th><th>Rationale</th></tr>
<tr><td>Astley
(extends Item, implements Purchasable and Monologuer)</td><td>Class representing a portable AI device which can be purchased from a computer terminal, and able to monologue.

Relationships:
1. It extends the Item class to achieve the portable functionality.
2. It implements the Purchasable interface as it can be purchased.
3. It implements the Monologuer interface as it can monologue.
4. It depends on the ListenAction class to enable an actor to listen to the monologue of the device.</td><td rowspan="3">Alternate Solution:
Create a class for each monologue object and create a ListenAction class. In the ListenAction class, check the type of object that triggers the action, and return a string from the list of monologue options of that object by using instanceof operator

Finalised Solution:
Create an interface Monologuer which defines a specific method allowableMonologues() for storing the monologue options. Then, create a new class for Astley, inherits from Item class and implements Purchasable and Monologuer interface as it can be purchased from computer terminal and can monologue. Astley overrides tick() method from Item class in order to handle subscription fee payment logic. Lastly, create a concrete class ListenAction which inherits from the Action class, with an attribute ‘monologuer’ to store the monologue object. In the override execute() method, the allowableMonologues() method is invoked to randomly choose a string from all monologue options.

Reasons for Decision:
1. By separating the listen action details from the Astley class, we ensure that each class focuses on a single responsibility, making it easier to understand, extend, and modify each class independently. The ListenAction class is responsible for managing the monologue options, ensuring that only a single monologue option is output at any given time. Meanwhile, the Astley class is responsible for storing a list of monologue options, each with different constraints, at every tick. (Single Responsibility Principle)

2. By implementing the logic of managing monologue options in the ListenAction class, we can avoid repeated code, as there's no need to duplicate the logic in each monologue object class in the same way. (DRY Principle)

3. Each object that can monologue has its own specific set of monologues and conditions. Hence, a Monologuer interface is created instead of an abstract class</td></tr>
<tr><td>Monologuer</td><td>An interface for some object that can monologue.

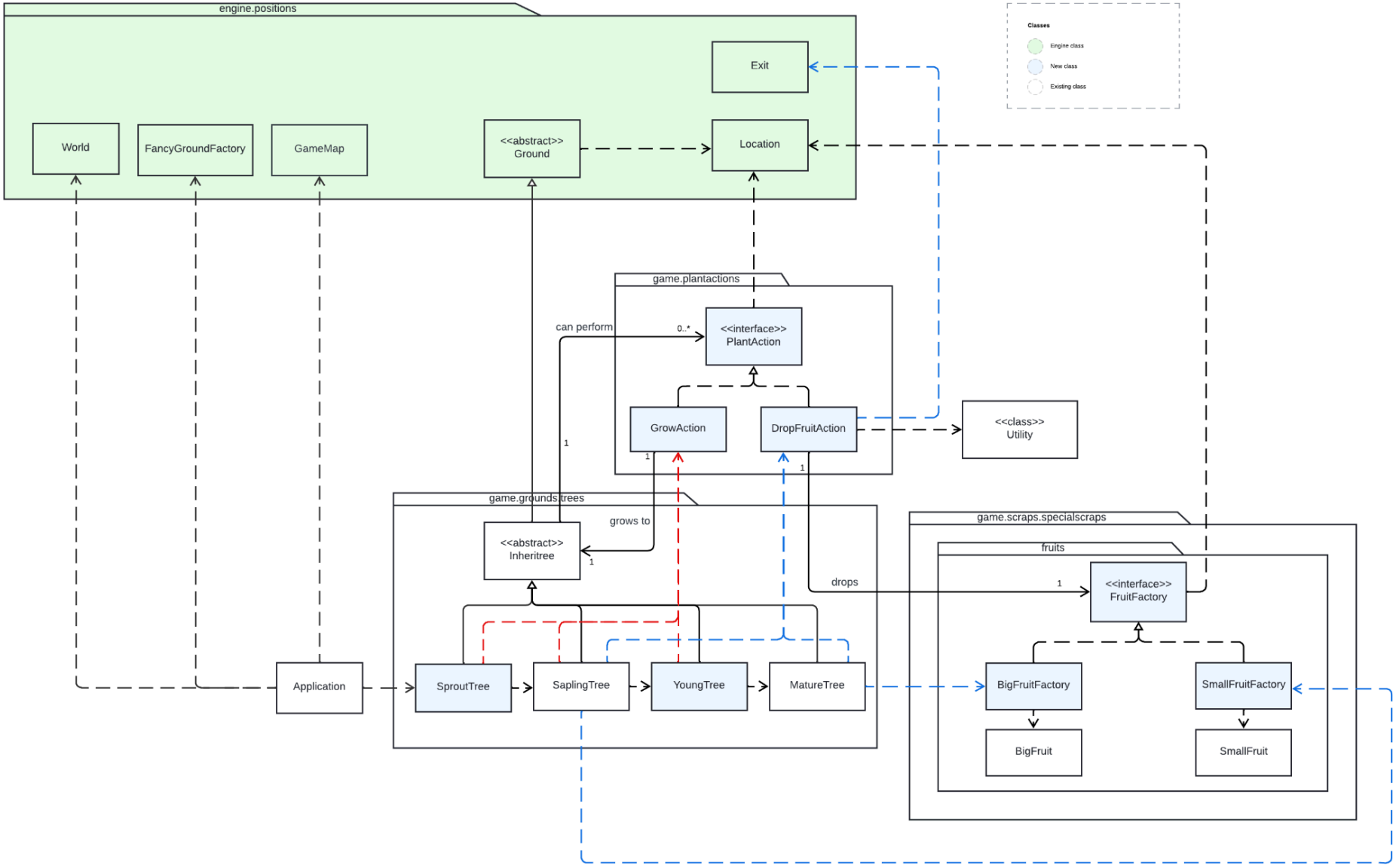
Relationships:
1. It depends on the Actor class to know which actor listens to the monologue object.</td></tr>
<tr><td>ListenAction
(extends Action)</td><td>Class representing the listen action.

Relationships:
1. It extends the Action class to</td></tr>
</table>

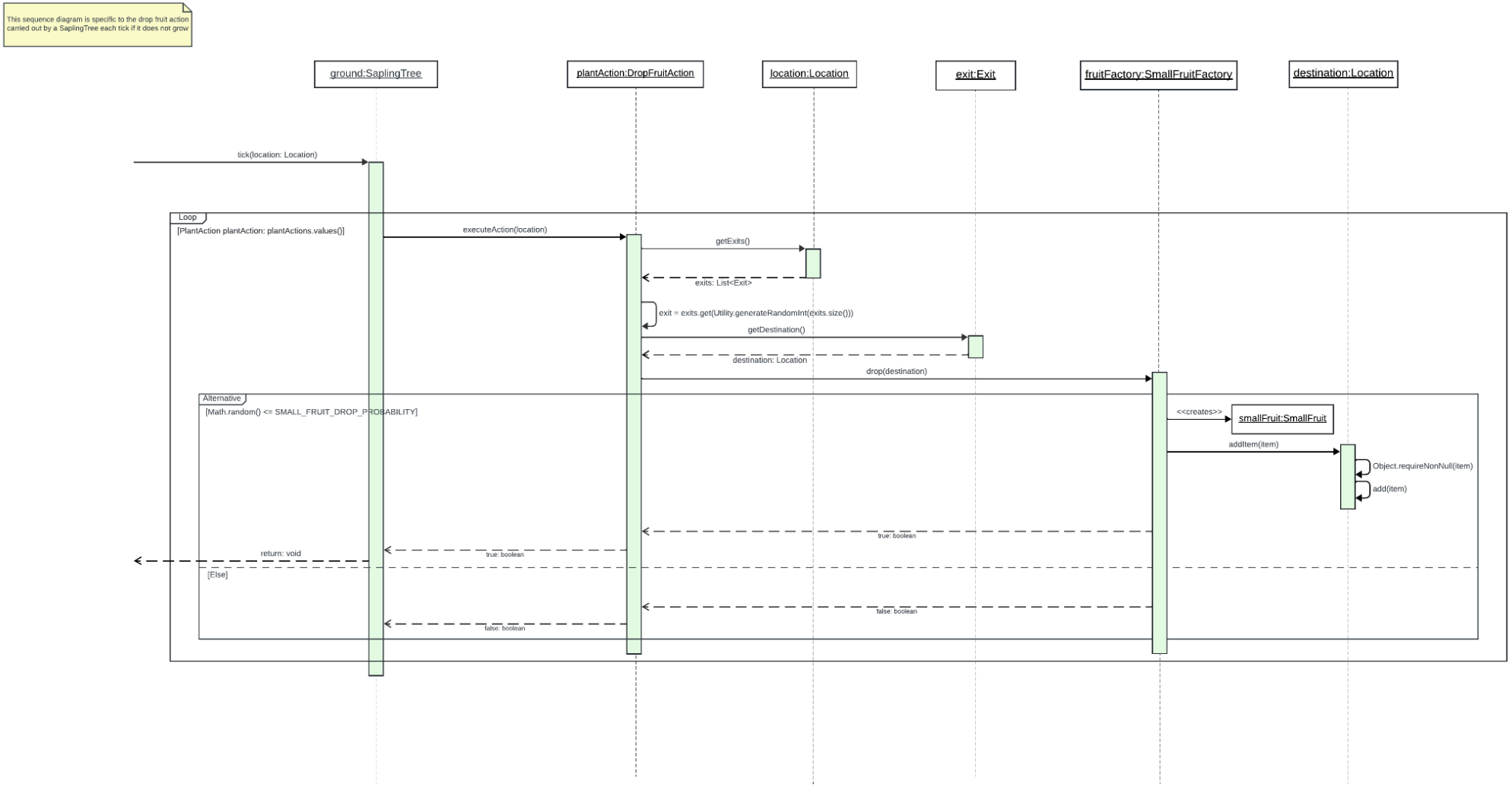
	perform an action. 2. It is associated with the Monologuer interface to perform the listen action on the monologue object.	which requires expensive maintenance of common attributes and methods. By using the Monologuer interface, each monologue object stores its own monologue options along with its condition constraint. Hence, when a new monologue object is introduced with a different set of monologues and conditions, it only needs to create a new monologue object class by implementing a Monologuer interface with the allowableMonologues() method, without any modification to the ListenAction class. *(Open-closed Principle)* 4. By having the Monologuer interface, only object classes that can monologue need to implement it. This forces objects that can monologue to implement all non-default methods in the Monologuer interface with their own code. Consequently, those objects can be substituted with the Monologuer type in attributes of the ListenAction class without breaking their expected behaviour as well as the whole system. *(Liskov Substitution Principle)* 5. The Monologuer interface is small enough and contains only the methods needed by the monologue object which implements it, in other words, subclasses that implement it are not forced to implement methods they are not required to. This enhanced the clarity and maintainability of code compared to having a large interface for all monologue objects (eg. including purchasing, consuming methods). *(Interface Segregation Principle)* 6. By using the Monologuer interface, multiple checks for the type of monologue object are not required, as all the monologue options are implemented by the subclasses. Consequently, the ListenAction class no longer needs to depend on every monologue object subclass, decoupling it from the specific details of monologue options *(Dependency Inversion Principle)*. Additionally, by having an attribute 'monologuer' in the ListenAction class and passing in the monologue object using constructor injection, it reduces the dependency of the ListenAction class on different monologue object classes *(Dependency Injection)*.

		7. The amount of credits to purchase (some objects have additional information, such as the amount of credits to pay for subscription, tick required to pay for subscription) is stored as attributes in the monologue object class. These attributes are given meaningful names to avoid using 'magic numbers' and enhance code maintainability. Hence, developers understand the meaning of these values and can easily change them in one place without affecting other parts of the codebase. *(Connascence of Meaning)* **Limitations and Tradeoffs:** 1. Having the Astley class implement multiple interfaces, such as Purchasable and Monologuer, raises concerns about the clarity of its responsibilities and the potential for design complexity which might require additional time and effort for developers to locate its specific functionality and navigate the codebase. 2. The monologue options are stored in a list, with each option manually added after checking certain conditions, reflecting a brute force implementation. As more monologue objects are introduced, the same monologue option may appear in multiple objects, leading to repeated code and inefficiencies.

Req 4 - Static factory's staff benefits



Requirement 4 Class Diagram



Requirement 4 Sequence Diagram: This sequence diagram is specific to the actions carried out by a SaplingTree, a Sapling Inheritree each tick

Classes Modified / Created	Roles and Responsibilities	Rationale
SproutTree (extends Inheritree)	Class that represents a Sapling tree, the first stage of growth for an Inheritree. Relationships: 1. Extends from abstract class Inheritree. 2. Depends on the next stage, SaplingTree.	**Alternate Solution:** Refactor inheritree abstract class to just include all four stages as one tree, keeping track of current tree and drop fruit or grow depending on the situation and the type of tree currently on ground. **Finalised Solution:** Create two new classes for SproutTree and Young Tree that inherits the Inheritree abstract class. Refactored previous Sapling Inheritree to depend on YoungTree and the new SproutTree and YoungTree to depend on SaplingTree and MatureTree respecticely **Reasons for Decision:** 1. There should not be a single class that represent all four stages of trees despite the similarities that both grows and drop fruit as growing times are different and a class should only represent one object. *(Single Responsibility Principle)* 2. With these newly created class, it ensures that there is not a requirement to modify previous parts of the code when adding new trees with similar features, instead the new stage class only need to inherit from the Inheritree to perform the growing and dropping fruit. *(Open-closed Principle)* 3. The transition from one tree to another uses the setGround() method that accepts Ground objects proves that Liskov's Substitution principle have been adhered to whereby the children classes were successfully upcasted into items of the parent class *(Liskov substitution Principle)* **Limitations and Tradeoffs:** 1. With multiple trees grow and drop fruit each tick, this design may cause confusion and issues keeping track of the classes. However, the alternative as described in the alternate solution proves to be a worst design implementation.
YoungTree (extends Inheritree)	Class that represents a Young tree, the third stage of growth for an Inheritree. Relationships: 1. Extends from abstract class Inheritree 2. Depends on the next stage, MatureTree	
PlantAction	Interface representing possible actions	**Alternate Solution:**

<table>
<tr><td></td><td>that can be taken by plants each tick

Relationships:
1. It is a parent class that does not extend or implement any other class, but is extend and implemented upon</td><td rowspan="3">1. Have Inheritree consist of the grow() and the dropfruit() method by simply returning null. Each stage of tree overrides this method if they can grow to the next stage.
2. Create Droppable and Growable interfaces which are implemented by each stage of the tree if they have the capabilities.
3. In each stage of the tree class, simply override the tick() method to implement the capabilities of growing and dropping fruits based on the stage of the tree.

Finalised Solution:
Create a PlantAction Interface that branches out into GrowAction and DropFruitAction. The Inheritree stores a TreeMap of PlantActions with different priorities. In each stage class, add the relevant action to the TreeMap to perform this action on each tick.

Reasons for Decision:
1. By creating GrowAction and DropFruitAction that specifically only execute a particular action that can be taken by a tree at any particular tick, this ensures the Single Responsibility Principle whereby each class only has one responsibility (Single Responsibility Principle).

2. When a new action is introduced, the details of the action is implemented in the new action class without having to modify the existing tree class. This new action class only needs to implement the PlantAction interface to allow itself to be added to the TreeMap list of PlantActions. (Open-closed Principle).

3. Either using the Droppable and Growable interfaces or simply overriding the tick() method to implement the capabilities might lead to duplicate code as the same code logic is repeated in each and every subclass only with different dropping probabilities and number of tick to grow. With the finalised approach, duplicate code can be avoided as the code logic is implemented in each PlantAction class. (DRY Principle)</td></tr>
<tr><td>GrowAction
(extends from PlantAction)</td><td>Class that represents the action of a plant growing in a single tick

Relationships:
1. Extends from PlantAction
2. Is a dependency for multiple plants</td></tr>
<tr><td>DropFruitAction
(extends from PlantAction)</td><td>Class that represents the action of a plant dropping fruit

Relationships:
1. Extends from PlantAction
2. Is a dependency for multiple plants
3. Has an association with a FruitFactory that is dependency injected during construction</td></tr>
</table>

		4. As a PlantAction object can be successfully passed into the TreeMap of plantActions, the child classes are being upcasted without breaking their expected behaviour *(Liskov substitution Principle)*. 5. Implement the grow() and dropFruit() method in the Inheritree class by simply returning null forces every stage class to implement both capabilities which they might not required to. To resolve this, PlantAction interface is introduced with GrowAction nor DropFruitAction to ensure that the trees only drop and grow if they are able to *(Interface Segregation Principle)*. 6. The creation of PlantAction interface allows the Inheritree class no longer have to depends on each and every Action class by having multiple attributes of types GrowAction and DropFruitAction. Instead, this high-level module Inheritree class only depends on abstractions (ie. the PlantAction interface). *(Dependency Inversion Principle)* 7. This particular design and the use of a TreeMap also potentially avoids the Connascence of Execution if each tree implements their respective grow() and dropfruit() method and then use if and else statements to pick between either. This might cause the dropFruit() method to execute before the grow() method and leave the loop early, causing errors in the tick method for the Inheritrees *(Connascence of Execution).* **Limitations and Tradeoffs:** 1. Excessive amount of code to implement actions that can be taken by a tree in a particular tick as opposed to simply implementing the methods. However, this ensures that particular actions that are being carried out are similar and can be collected in a treemap of plantActions.
FruitFactory (Interface)	Interface that represents a Fruit Factory used to create new fruit instances in the map	**Alternate Solution:** 1. Each stage class creates its Fruit object and drops the fruit implemented in its drop() method.

<table>
<tr><td></td><td>Relationships:
DropFruitAction is associated with this FruitFactory
It is a parent class that does not extend or implement any other class, but is extend and implemented upon</td><td rowspan="3">Finalised Solution:
Similar to the creature and their spawners created in Assignment 2, the FruitFactory interface helps with creating FruitFactory that adds a particular Fruit Instance to the map.

Reasons for Decision:
Every implementation of the FruitFactory only has the sole responsibility of adding one type of Fruit to the map. This satisfies SRP which is against a single FruitFactory spawning multiple types of fruits. (Single Responsibility Principle)
With the possibility that different fruits have different dropping probability, FruitFactory is implemented as an interface instead of an abstract class. With this, when new fruit is introduced, its FruitFactory only implements the required methods to create the new fruit instance to drop from the tree. (Open-closed Principle)
BigFruitFactory and SmallFruitFactory can be substituted with FruitFactory in attributes of the DropFruitAction class without breaking the whole system, successfully adhering to Liskov’s Substitution Principle whereby parent classes can be substituted by their children classes in a system (Liskov substitution Principle) .
The FruitFactory interface is small enough and contains only the methods needed by BigFruitFactory and SmallFruitFactory and neither are forced to implement methods they are not required to (eg. SmallFruitFactory are not required to drop BigFruit and vice versa). (Interface Segregation Principle)
With the use of the FruitFactory interface, DropFruitAction class no longer needs to depend on BigFruitFactory and SmallFruitFactory classes but only the FruitFactory interface, ensuring that such a high-level module is not affected by the low level modules (Dependency Inversion Principle).</td></tr>
<tr><td>BigFruitFactory (extends FruitFactory and has a dependency with BigFruit)</td><td>Class that represents a Big Fruit Factory used to create new Big fruit instances in the map.

Relationships:
Extends FruitFactory
Has a dependency on BigFruit as it spawns a BigFruit</td></tr>
<tr><td>SmallFruitFactory (extends FruitFactory and has a dependency with SmallFruit)</td><td>Class that represents a Small Fruit Factory used to create new small fruit instances in the map

Relationships:
Extends FruitFactory
Has a dependency on SmallFruit as it spawns a SmallFruit</td></tr>
</table>

		6. To ensure that class attributes and values used in the code can be easily understood hence improving code maintainability, the probability of dropping BigFruits and SmallFruits in BigFruitFactory or SmallFruitFactory respectively are given meaningful names to ensure that developers understand the meaning of these values and can them as desired. *(Connascence of Meaning)* **Limitations and Tradeoffs:** 1. This implementation tradeoff application ease for maintainability in the event that the application is extended for the benefits listed above. This is because the number of FruitFactory requires scales linearly with each new FruitFactory to be introduced to the system, potentially causing the system tedious to maintain.

Assignment 3 - Contribution Log

FIT2099 - CL_Lab26Team2 - Assignments' Contribution Logs

Task/Contribution(~30 words)	Contribution type	Planning Date	Contributor	Status	Actual Completion Date	Extra notes
Meeting discussion						
Revised Assignment 2 UML and sequence diagrams and discuss changes required based on feedback	Discussion	18/05/2024	EVERYONE	DONE	18/05/2024	
Revised Assignment 2 code design and discuss reflecting changes based on feedback	Discussion	18/05/2024	EVERYONE	DONE	18/05/2024	
Distribute tasks related to the changes required for A2	Discussion	18/05/2024	EVERYONE	DONE	18/05/2024	
Distribute tasks after reviewing the assignment brief	Discussion	18/05/2024	EVERYONE	DONE	18/05/2024	
Briefly create a UML diagram for each task	Brainstorm	18/05/2024	EVERYONE	DONE	18/05/2024	
Discuss the type of classes used, ie concrete, abstract, interface or enum	Brainstorm	18/05/2024	EVERYONE	DONE	18/05/2024	
Refactoring of assignment 2	Implementation	21/05/2024	EVERYONE	DONE	21/05/2024	
Requirement 1						
Choice in the computer terminal to return to factory	Implementation	21/05/2024	Wei Xin Soh	DONE	21/05/2024	
Interact with computer terminal to return to factory, switching maps	Implementation	21/05/2024	Wei Xin Soh	DONE	21/05/2024	
Computer terminal sells THESEUS ^, portable teleporter	Implementation	21/05/2024	Wei Xin Soh	DONE	19/05/2024	
THESEUS can be placed on the map	Implementation	21/05/2024	Wei Xin Soh	DONE	19/05/2024	
If THESEUS is placed on the map, the Intern is able to take the Teleport Action	Implementation	21/05/2024	Wei Xin Soh	DONE	19/05/2024	
THESEUS can be used by the Intern to teleport to any location on the map randomly	Implementation	21/05/2024	Wei Xin Soh	DONE	19/05/2024	
Javadoc	Documenting	02/06/2024	Wei Xin Soh	DONE	31/05/2024	
Requirement 2						
Humanoid Figure that cannot be hurt in the factory's spaceship parking lot	Implementation	21/05/2024	Jeevan Vetteel Tharun	DONE	22/05/2024	
Intern can sell Large bolts for 25 credits, Large fruits for 30 credits, Metal pipes for 35 credits	Implementation	21/05/2024	Jeevan Vetteel Tharun	DONE	22/05/2024	
Metal Sheets can be sold for 20 credits but with a 60% chance to only be sold for 10 credits	Implementation	21/05/2024	Jeevan Vetteel Tharun	DONE	22/05/2024	
Jars of pickles can be sold for 25 credits, but 50% chance sold for 50 credits	Implementation	21/05/2024	Jeevan Vetteel Tharun	DONE	22/05/2024	
Pots of gold can be sold for 500 credits, but 25% chance be taken without pay	Implementation	21/05/2024	Jeevan Vetteel Tharun	DONE	22/05/2024	
Toilet paper rolls can be sold for 1 credit, but 50% chance the Humanoid figure kills Intern instantly	Implementation	21/05/2024	Jeevan Vetteel Tharun	DONE	30/05/2024	
Javadoc	Documenting	02/06/2024	Jeevan Vetteel Tharun	DONE	02/06/2024	
Requirement 3						
Intern can buy AI Device from the Computer Terminal	Implementation	21/05/2024	Ying Shan Khor	DONE	19/05/2024	
If the AI terminal is in the intern's inventory, the intern pays the factory 1 credit every 5 tick	Implementation	21/05/2024	Ying Shan Khor	DONE	19/05/2024	
If the intern does not pay 1 credit every 5 tick, the intern cannot interact with the device	Implementation	21/05/2024	Ying Shan Khor	DONE	19/05/2024	
Ticking is paused when the device is not in the intern's inventory but will resume shortly	Implementation	21/05/2024	Ying Shan Khor	DONE	21/05/2024	
Option to listen to the device monologue if the intern pays and the device is in the intern's inventory	Implementation	21/05/2024	Ying Shan Khor	DONE	19/05/2024	
The device chooses randomly from possible dialogue options	Implementation	21/05/2024	Ying Shan Khor	DONE	19/05/2024	
New dialogue option unlocked when the intern has more than 10 items in their inventory	Implementation	21/05/2024	Ying Shan Khor	DONE	19/05/2024	
New dialogue option unlocked when the intern carries more than 50 credits in their wallet	Implementation	21/05/2024	Ying Shan Khor	DONE	19/05/2024	
New dialogue option unlocked when the intern's health point is below 2	Implementation	21/05/2024	Ying Shan Khor	DONE	19/05/2024	
Javadoc	Documenting	02/06/2024	Ying Shan Khor	DONE	30/05/2024	
[Survival Mode] Requirement 4						
Creation of Second Moon Refactorio	Implementation	21/05/2024	Suet Yuen Chin	DONE	20/05/2024	Require of the integration with Requirement 1 multiple maps
Sprout Inheritree that cannot produce fruits but grow into Sapling after 3 turns	Implementation	21/05/2024	Suet Yuen Chin	DONE	20/05/2024	
Sapling that drop fruits can grow into Young after 6 turns	Implementation	21/05/2024	Suet Yuen Chin	DONE	20/05/2024	
Young Inheritree that lays dormant grows into Mature after 5 turns	Implementation	21/05/2024	Suet Yuen Chin	DONE	20/05/2024	
Behaviour Tree for Inheritrees	Implementation	21/05/2024	Suet Yuen Chin	DONE	20/05/2024	
Fruit Factory for Big Fruits and Small Fruits	Implementation	21/05/2024	Suet Yuen Chin	DONE	20/05/2024	
Javadoc	Documenting	02/06/2024	Suet Yuen Chin	DONE	01/06/2024	

Requirement 1					
UML Diagram	Documenting	23/05/2024	Wei Xin Soh	DONE	27/05/2024
Sequence Diagram	Documenting	26/05/2024	Wei Xin Soh	DONE	27/05/2024
Design Rationale	Documenting	28/05/2024	Wei Xin Soh	DONE	27/05/2024
Create handover videos	Implementation	03/06/2024	Wei Xin Soh	DONE	03/06/2024
Requirement 2					
UML Diagram	Documenting	23/05/2024	Jeevan Vetteel Tharun	DONE	30/05/2024
Sequence Diagram	Documenting	26/05/2024	Jeevan Vetteel Tharun	DONE	30/05/2024
Design Rationale	Documenting	28/05/2024	Jeevan Vetteel Tharun	DONE	30/05/2024
Create handover videos	Implementation	03/06/2024	Jeevan Vetteel Tharun	DONE	03/06/2024
Requirement 3					
UML Diagram	Documenting	23/05/2024	Ying Shan Khor	DONE	21/05/2024
Sequence Diagram	Documenting	26/05/2024	Ying Shan Khor	DONE	21/05/2024
Design Rationale	Documenting	28/05/2024	Ying Shan Khor	DONE	28/05/2024
Create handover videos	Implementation	03/06/2024	Ying Shan Khor	DONE	03/06/2024
Requirement 4					
UML Diagram	Documenting	23/05/2024	Suet Yuen Chin	DONE	21/05/2024
Sequence Diagram	Documenting	26/05/2024	Suet Yuen Chin	DONE	21/05/2024
Design Rationale	Documenting	28/05/2024	Suet Yuen Chin	DONE	28/05/2024
Create handover videos	Implementation	03/06/2024	Suet Yuen Chin	DONE	03/06/2024
Finalisation					
Uploading handover videos	Implementation	01/06/2024	EVERYONE	DONE	01/06/2024
Code Testing and Debugging	Code review	02/06/2024	EVERYONE	DONE	02/06/2024
Design Rationale Document Check	Code review	02/06/2024	EVERYONE	DONE	02/06/2024