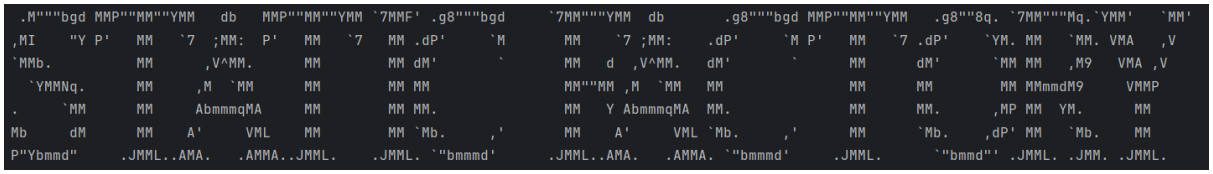

FIT2099 ASSIGNMENT 3 REPORT (DESIGN DIAGRAM AND RATIONALE)

Group: MA_AppliedSession08_Group2

Group Members:

- Lee Quan Hong
- Teh Jia Xuan
- Hoe Jing Yang
- Lee Teck Xian

Assignment 3

Requirement 1: The ship of Theseus

UML Diagram:

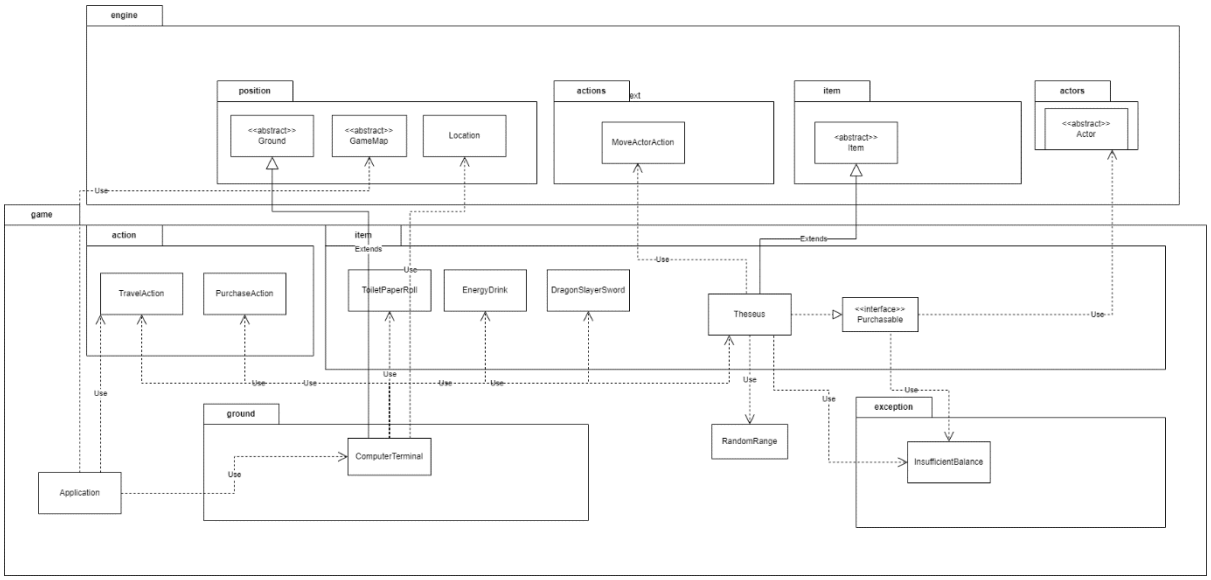

Figure 1 Class Diagram of REQ 1.

Sequence Diagram:

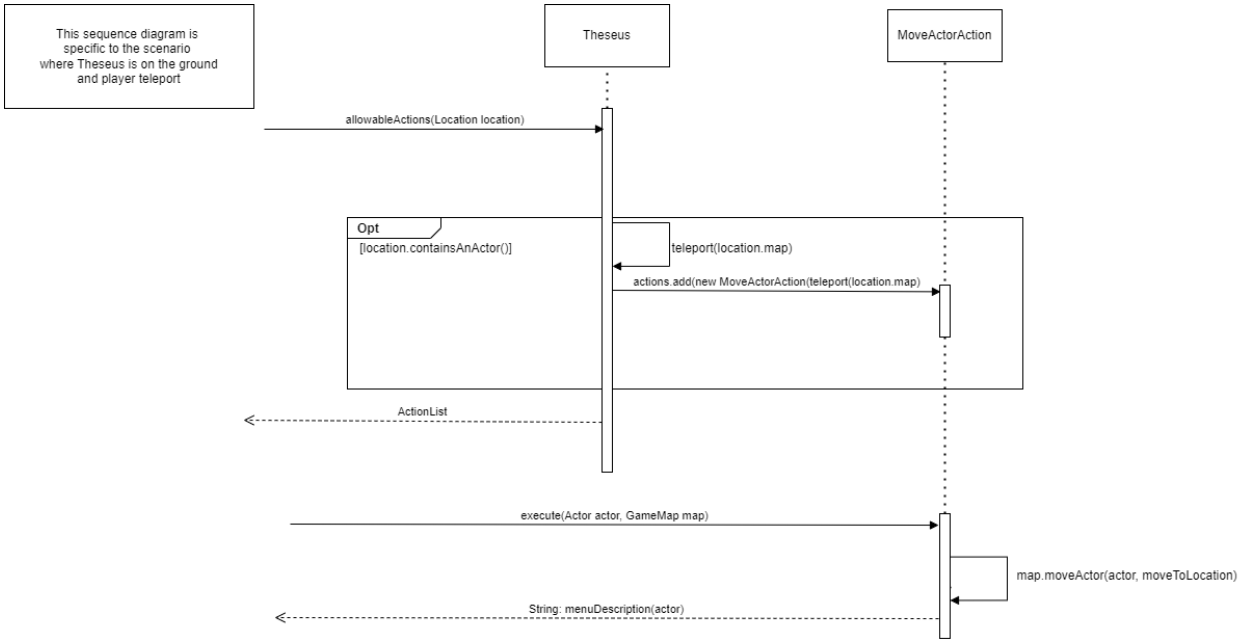

Figure 2 Sequence Diagram of REQ 1.

Design Goal

The design goal for this assignment is to the codebase easy for extension and have good maintainability while adhering closely to relevant design principles and best practices.

Design Decision

In implementing Theseus, I have decided to use the MoveActorAction to teleport the player. In my Theseus class, I have a method called teleport that generates random x and y values to teleport the player. In the allowableActions, I simply add the MoveActorAction with the values returned by the teleport method if the player is standing on Theseus.

Additionally, I created a TravelAction class that inherits from the action abstract class, which allows the player to travel to other maps by utilizing the moveActor method from its parent class. Before it travels it checks whether the player is in the corresponding map. If is in it doesn't allow player to teleport to the corresponding map.

Furthermore, I have added two more game maps. I included them in the game map using the addGameLocation method in the application class. In my computer terminal class, I created a new method called addMapLocation to add travelAction into the list, enabling the computer terminal to use the list and add travel action that is available. In my application class, I initialized each map with a computer terminal and, by using addMapLocation I able to pass the necessary travelAction to the computer terminal. This way, the computer terminal can save the maps that player allows to travel using the list. Moreover, I added a for loop in allowable action to list down all the maps in travelActions list that player can travel when they are in that specific map of the computer terminal.

Alternative Design:

An alternative approach for traveling between maps could involve creating map classes with a public static attribute for all the maps. This way, I can use MoveActorAction in the computer terminal with the public static attribute without needing to pass location attributes in the application class using addMapLocation. The advantage of this approach is that the code in the application class is much cleaner, as we do not need to initialize maps in application class or create the addMapLocation method in computer terminal. However, the disadvantage is that since it is a public static attribute, any class can access it. This means any class can use it to travel to other maps, which is not ideal for security

Additionally, we can use MoveActorAction instead of TravelAction in the computer terminal. The advantage of this is that we do not need to create another class, thereby avoiding code duplication. However, the disadvantage is that in the allowableActions, whenever I need to add a new map, I have to add an if statement to check whether the player is in the map. This makes the code messy and decreases the maintainability of the codebase.

Analysis of alternative design:

The alternative design is not ideal because it violates various Design and SOLID principles:

1. **[Don't Repeat Yourself (DRY)]:**
   - By utilizing MoveActorAction I need to have a lot of if statement in allowableAction everytime I add a new map to the computer terminal
   - This causes a bunch of repeated code

2. **[Open Closed Principle]:**
   - If we utilise MoveActorAction we need to modify previous code to accommodate the new map as we need to add if statement for it to work.

In terms of code smell, this alternative design will lead to a bunch of if statement in allowable action, This results in repeated code snippets across the if statements, which can lead to maintenance challenges and increased risk of bugs due to redundancy.

Final Design:

In our design, we closely adhere to [relevant design principles or best practices, e.g., SOLID, DRY, etc.]:

1. **[DRY]:**
   - **Application:** A TravelAction class is created to avoid code duplication. The execute method checks if the map contains the actor. If not, it executes the moveActor method to move the actor to that map and returns a menu description.

   - **Why:** A travel action class is needed to avoid repeated if statement in the computer terminal's allowableAction. As it helps to check the map contain that actor before it travel to that map.

   - **Pros/Cons:** The pros are by encapsulating the travel logic within the TravelAction class I can avoid repeating if statements in computer terminal. This results in cleaner and more maintainable code. Moreover, it promotes reusability as it can be reused across different parts of the application whenever similar travel functionality is needed.

     The con is if is not designed carefully, there is a risk of the TravelAction class becoming redundant or overlapping with existing functionality like MoveActorAction.

2. **[Single Responsible Principle]:**
    - **Application:** Each class is assigned its own responsibility. In my design, each class is designed with a single specific responsibility.

    - **Why:** TravelAction: This class is responsible for allowing the player to travel to a specified location using its method.

      Theseus: This class provides the player with the option to teleport when standing on it.

      Computer Terminal: This class enables the player to interact with it to perform desired actions.

    - **Pros/Cons:** The pros will be by assigning each class a specific responsibility the design adheres to the single responsibility principle making the code more modular and easier to manage. Moreover, each class has a clear purpose which improves the readability and understanding of the code. Furthermore, class like travel action as it responsibility is to let player to travel so it can be reused in different parts of the application in future projects.
3. **[Open Closed Principle]:**
    - **Application:** Instead of utilizing MoveActorAction to travel player to other map, I have created a separate class called TravelAction.

    - **Why:** This approach simplifies the process of adding new maps for players to travel to. By simply adding the new maps in application and by utilising addMapLocation we able to add map to the travelActions list when player in that particular map, therefore we avoid modifying existing code to accommodate the new maps. If we are using MoveActorAction we need to modify existing code to prevent player travel to the map that they already in. By utilizing TravelAction we can add new maps to computer terminal easily without modifying previous code.

    - **Pros/Cons:** The pros will be adding new maps becomes straightforward as I only need to put it in application class and update the travelActions list in the computer terminal without modifying existing code. Moreover, the design easily accommodates the addition of new maps and make it scalable for future expansion. Since it doesn't modify the existing code it makes the codebase easier for extending functionality and easier to debug in the future.

      The cons will be when introducing a new class like TravelAction it adds to the overall number of classes and it makes the project structure more complex. Furthermore, if TravelAction is not designed carefully there might be an overlap in functionality with existing classes leading to redundancy.

4. **[Liskov Substitution Principle]:**
    - **Why**: By avoiding downcasting, the behavior of the base classes, such as Item and Action, remains consistent. Their child classes can replace the parent classes without altering the program's behavior.

    - **Pros/Cons**: The pros are it ensures that subclasses can replace their parent classes without causing unexpected side effects or breaking existing code. Moreover, it maintains consistent behaviour across the codebase, making it easier to understand and manage. Besides that, it improves the predictability of the program as they wont be having hidden behaviours.

The cons will be Liskov substitution principle may impose constraints on how subclasses can extend or modify the behavior of their based classes which potentially limiting certain design choices. Other than that, it limit the flexibility to implement specific functionalities.

Handling connascence in our code:

The connascence of position is to reduce by centralizing the travel logic in TravelAction so it minimizes the repetition of if statements across the code. So reducing the risk of inconsistencies.

The connascence of meaning is to reduce by replacing numbers with variable. For example in my Theseus class I have replaced 100 with the class attribute of private static final cost = 100.

Code smell:

By keeping the SOLID principle in mind, we have avoided code smells. With travel action class created it can avoid a bunch of if statement in the computer terminal allowable action. Other than that, if I wanted to add new maps in my program, I simply add a travel action in allowable action of computer terminal so I no need to modify my previous code. By creating classes that adhere to Open closed principle and DRY principle, we able to minimize the code smell of classes that need frequent modifications when new functionality is introduced

What has been done and why

Theseus help player to teleport to random locations:

- Theseus inherit item and implement purchasable so it can be purchased through computer terminal. Added a teleport method, to generate random x and y on the map. I have utilised MoveActorAction and pass in teleport method to help player to teleport to random location

Create TravelAction to help player travel to other maps

- Using MoveActor method in the execute method of travel action, if the player is in that corresponding map or actor in that current map location then stop player from travelling to the map.

Add Travel Action to computer terminal TravelAction list

- Add travel action to computer terminal's travelActions list. Depend on which map players are in add in different travel action map. There is a loop in allowableAction, it will add the travel action that contains in the travelActions list

Add game maps in application class

- Pass in travelAction in application to computer terminal's travelActions list by utilizing addMapLocation method in computer terminal

Overall pros and cons of the design

Following the SOLID principles in the implementation enhances code debuggability and extensibility. The utilisation of actions class allow us to minimises redundancy and promote code reusability and extensibility.

The cons will be when there are many maps in the future the application class might be lengthy as we need to add travel action everytime there is a new map and it increases the complexity of the code and it is harder to maintain.

How it can be easily extended

If we need more maps in the future, we can simply initialise in application class and add a travel action in computer terminal 's travel actions list by utilising addMapLocation

If we have other functionality of Theseus we can create a new action of it and add it in its allowable action.

Conclusion:

Overall, our chosen design provides a robust framework for implementing this requirement. By carefully considering relevant factors such as design principles, future extension and constraints we have developed a solution that is efficient, maintainable and extensible. This made a foundation for future enhancements, extensions and optimizations of the program.

Assignment 3
Requirement 2: The Static Factory

UML Diagram

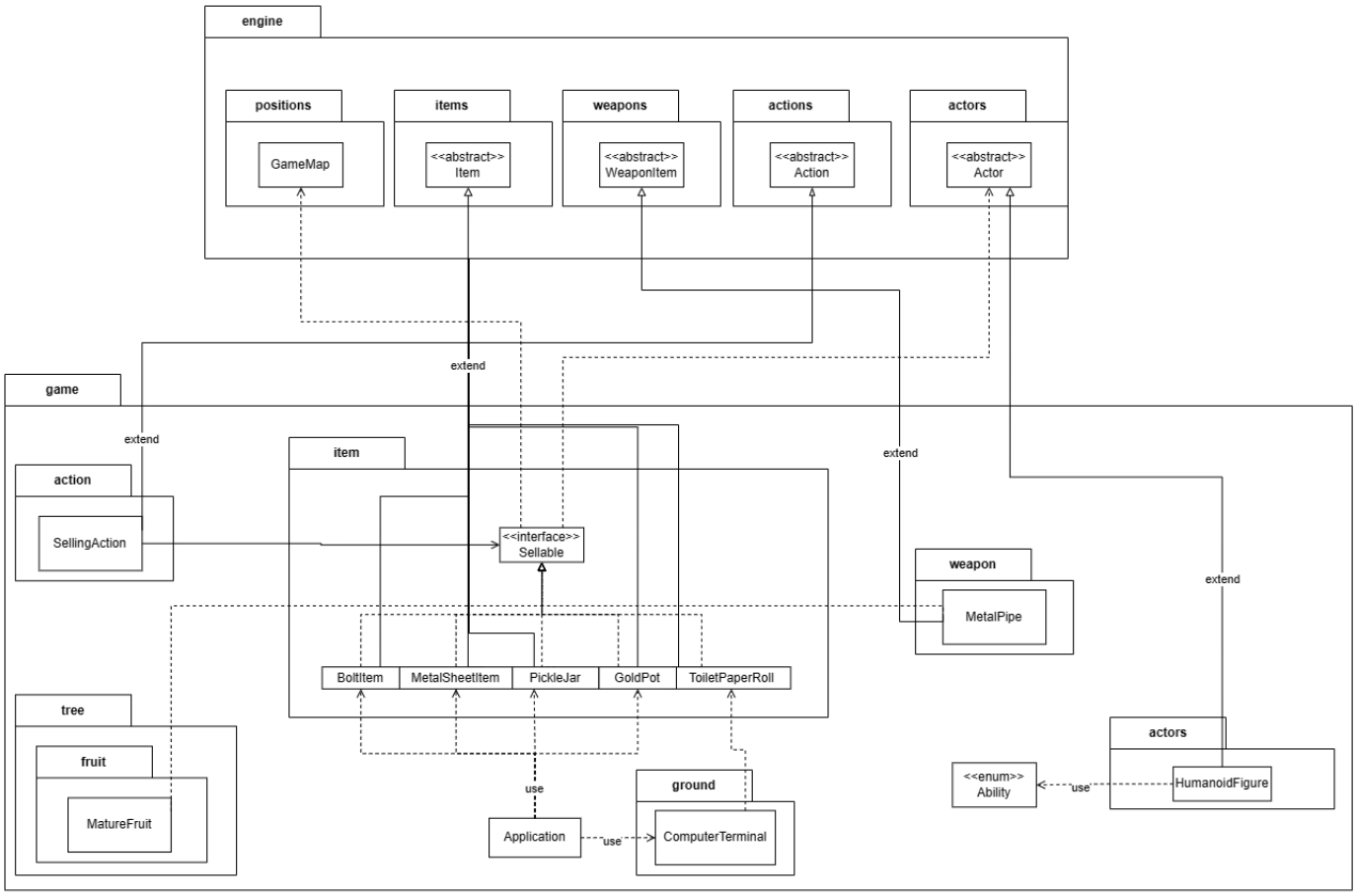

Figure 3: UML Diagram of Requirement 2

Sequence Diagram

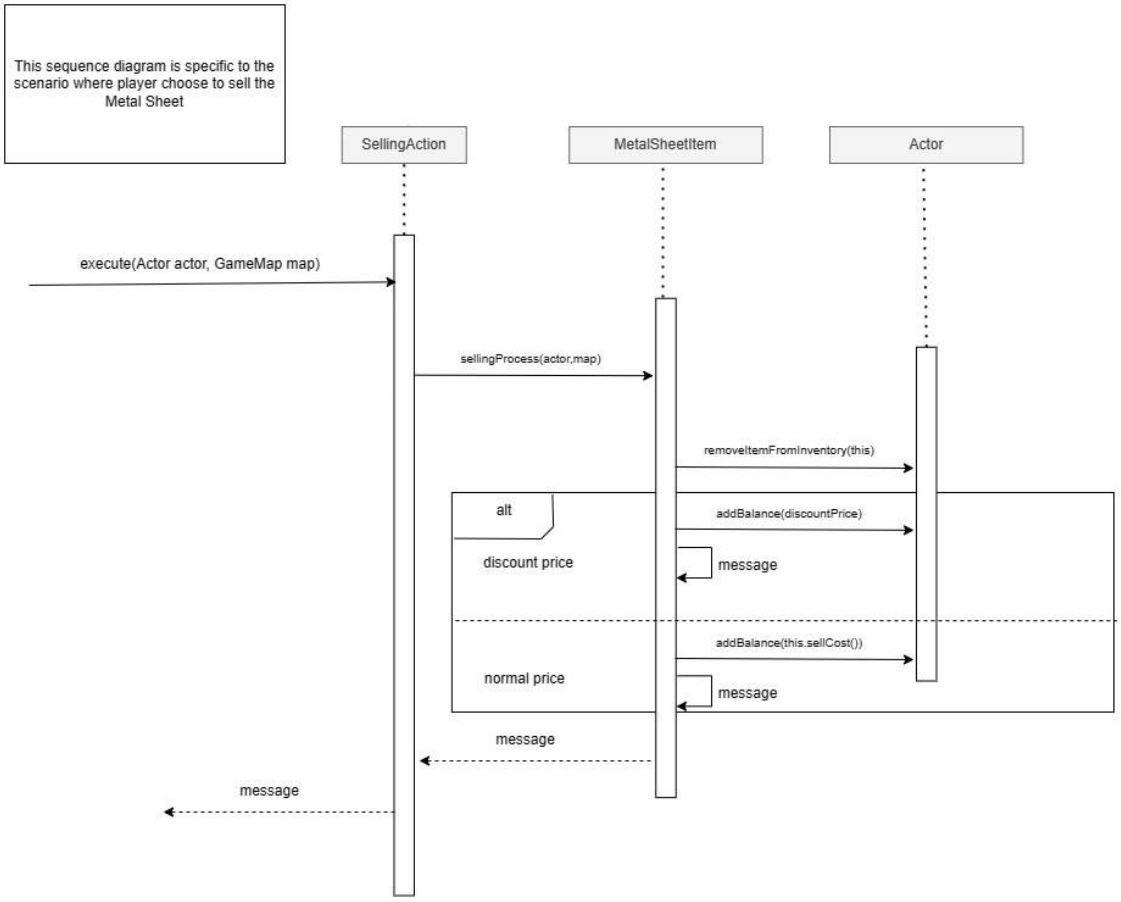

Figure 4: Seqeunce Diagram of Requirement 2

Design Goal

The goal of this design is to implement a selling feature to items easily while adhering to closely to the relevant design principles and best practices.

Design Decision

During the implementation of adding the new feature, I decide to add a new interface “sellable”. This interface standardize the methods needed when the item was selected to be able to sold. There were three method in this interface which were sellCost() which defines the price of the item, eventRate() which defines the rate of triggering an event happens during the trading and sellingProcess() where the process of selling item including setting the selling price and event that will happen during the selling process are all set inside this

method. I also add a player call “humanoid figure” which had the ability to trade item. When the player goes near the humanoid figure, selling option will be available in the action menu of the player. A new ability was also added call “ABILITY_TRADING” where when the other actor has this ability, the player that holds this item can trade with them. By implementing this design, the selling feature can be easily maintained and scalable while being efficient. It could be easily extend for future use in the future.

Alternative Design

An alternative design that can be implemented was directly implement the selling process method in the items without the need of Sellable interface. However, this would violate the SOLID principles.

Analysis of Alternative Design

Dependency Inversion Principle (DIP):

The alternative design would violate the dependency inversion principle by removing the use of interface class. High level module should not depend on low level module but both should depend on abstractions. The interface class acts as an abstraction in this case but since the alternative design eliminates the use of interface class, it violates the DIP.

Final Design

1. Don’t Repeat Yourself (DRY)

- Application:
  - Each sellable item has its own unique action method in their respective class. All of the sellable item have a sellingProcess method in them which will the SellingAction will execute their process and add balance to player’s wallet
- Why:
  - This method allows us to utilise a single SellingAction class and enable each sellable item to execute its own action. This will eliminate the need for every item to create their own action class which will cause the repetition of code.

- Pros/Cons:
  - The benefits of this design is when we want to add a new sellable item in the future, we just need to implement the Sellable class and define its own method without the need to create a new class.

  - The downside of this method is that the respective item class will be lengthy if they have a complex selling process.

2. Single Responsibility Principle

- Application:
  - In this design, every class has its own specific responsibility

- Why:
  - Sellable interface is design to implement on items that are sellable. This interface requires the respective items to define their own selling method which their specific price and event. Each item class that are eligible to selling are responsible to implement the interface class and define their own selling method so it will be unique to the class itself.

- Pros/Cons：
  - The benefit of this design is that each class can have their own specific purpose. It promotes reusability as we can implement the code to other classes when we need a new method with similar functionality.

  - The downside would be that as we add more functions, there will be more and more classes with specific responsibilities which makes the whole codebase lengthy and increase the complexity of the code.

3. Liskov Substitution Principle

- Application:
  - All the sellable item did not override any existing method from item, thus will not change any meaning of the methods.

  - HumanoidFigrue overrides the playTurn method from Actor,the method still has the same meaning as in the super class where it returns the action
  -
  - SellingAction overrides the execute method from Action, the method still has the same meaning as in the super class where it executes the action and returns a string as a result.

- Why:
  - Adhering to the LSP ensures that the inherited methods maintain their meaning in the subclasses. This means having consistent behavior across different classes and following the contracts, facilitating future extensions and maintenance.

- Pro/Cons:

- The benefit of following the LSP ensures that the inherited methods maintain their meaning in the subclass. This allows them to have a consistent behaviour across different classes makes extension and maintaining the code easy in the future.

4. **Interface Segregation Principle**
   - Application
   - Sellable interface has three method. sellCost, eventRate and sellingProcess.
   - These methods are implemented by the classes that are sellable while items that can't be sold doesn't needs them.

   - Why
   - Following the ISP ensures that classes only implements the method if they need it. This eliminates the need for classes that doesn's need the specific method

   - Pro/Cons:
   - The benefit of adhering to ISP ensures that classes only implements the method they need thus eliminates the redundant problem in the code
   - The downside of this principle is that we may add a lot more interface as we add more feature to the code.
5. **Dependency Inversion Principle**
   - Application:
   - SellingAction class depends on Sellable interface, which was an abstraction class instead of concrete class.This allows the SellingAction class to not be dependent on the concrete classes that implement the Purchasable interface.

   - Why:
   - Adhering to the DIP ensures that high-level modules and low-level modules depend on abstractions . Details should depends on abstraction instead of abstraction depends on detail. Codes can be extended and maintained easily by depending on abstractions. Different concrete classes with different responsibilities can easily extend their responsibilities by depending on abstractions.

   - Pro/Cons:
   - The benefit of following the DIP allows us to reduce the dependencies on concrete class by depending on abstraction class thus the code can be extend and maintain easily.

6. **Open-close Principle**
   - Application:
   - SellingAction extends Action while HumanoidFigure extends Actor. These classes extend from their superclasses respectively while adding new functionality without modifying the existing superclass.

   - Why:
   - By following the open-close principle, we can add new functions to existing classes without modifying the existing code. This makes extension and maintenance in the future easier.

   - Pro/Cons:
   - The advantages of following the OCP includes easier to add new functions to existing code, reduce the risk of introducing bug to the existing code and allows the existing code to be reusable as new functions can be add on top of them.

Handling Connascence in our code

Connascence of Name (CoN):

- This is handled by using the interface to define the functions in the class. All the classes that implements the interface must implements the method of the interface using the same name define in the interface.

Connascence of Meaning (CoM):

- This is handled by replacing the numbers in classes with variables. All the price and event rate in the sellable item are replace with private static final int = {their respective price} and private static final double = {their respective event rate}.

Code Smell

By tightly following the the SOLID principles while implementing our design, we have avoided code smells. By following SRP, we manage to reduce the code smell of large class by having smaller class with their own responsibility. By following DRY, we manage to reduce the code smell of duplicate code.

What has been done and why

Modify BoltItem, MetalPipe, MatureFruit, ToiletPaperRoll, PickleJar, GoldPot, MetalSheetItem and BoltItem to implement the Sellable interface

- All the classes that are mentioned above are updated to implements the Sellable interface which allows them to be traded to the HumanoidFigure and define their own price and event in their respective class.

Added Sellable interface

- The Sellable interface class allows sellable items to implements it and define their own methods to handle the selling process.

Added SellingAction extends Action

- The SellingAction extends Action class and provides the functionality to perform selling when the player is near some actor that are having the ability to trade. The SellingAction class takes a Sellable as parameter and returns a string indicates the result of the trading of that item when the actions is performed through the execute method.

Modify Ability Class

- The Ability class was modified to hold another enum which was the "ABILITY_TRADING" which the item holder can trade with another actor if that actor has this ability.

Added HumanoidFIgure extends Actor

- The HumanoidFigure class was created as an actor to allows the player to sell items to it. The HumanoidFigrue has the capability "ABILITY_TRADING" which allows the player to sell items to it when it goes near the HumanoidFigure

Overall Pros and Cons of the Design

Pros

The design follows SOLID principles which allows the code to be debug easily while still maintain the convenience of future extension and scalability.
The utilisation of interface class allows us to minimize the redundancy of code and promote code reusability.

Cons

The cons will be when certain sellable item had a complex selling process and event triggering condition, the item class will be lengthy which will increase the complexity of the code and makes it harder to maintain.

How it can be easily extended

If we need to add more sellable item in the future, we can just simply create the item class and implements the Sellable interface or just implements the Sellable interface to existing item class. The sellable items can define their own price, event rate and selling process in their own respective class.

Conclusion

Overall, our chosen design provides a robust framework for implementing the Requirement 2: The Static Factory in the game. By carefully considering the requirement and adherence to design principles like SOLID and DRY, we have developed a solution that is efficient, scalable, maintainable. This solution provides a solid foundation for future extension, enhancement and makes it very easy for others to understand the functionality of the codes.

REQ3: dQw4w9WgXcQ

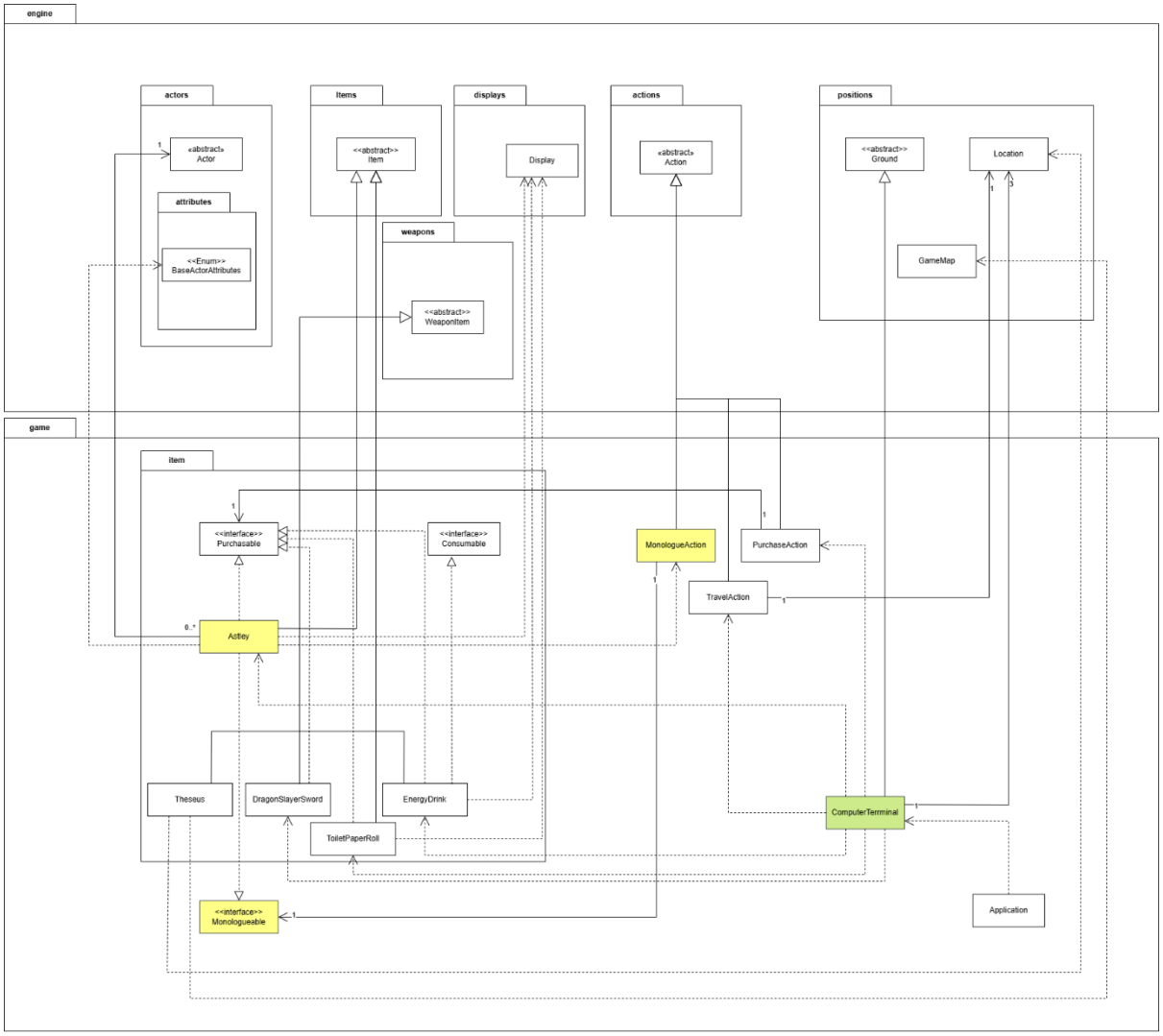

Figure 5 Class Diagram of REQ 3.

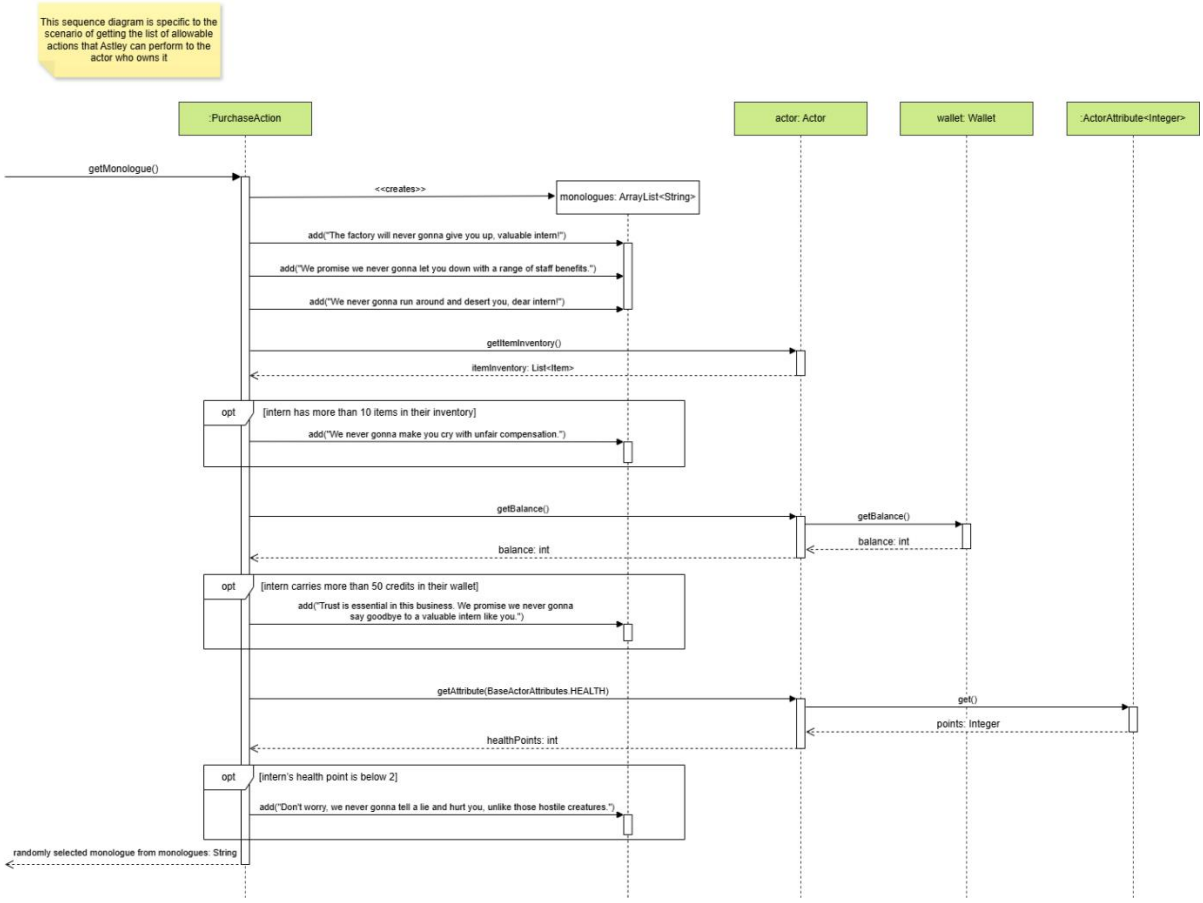

Figure 6 Sequence Diagram of REQ 3, getting allowable actions that Astley can perform to the actor who owns it.

Design Goal:

The design goal for this assignment is to meet the user requirements while adhering closely to relevant design principles and best practices of object-oriented software designs.

Design Decision:

In implementing REQ3, the decision was made to create two new classes, Astley and MonologueAction, where Astley extends the Item class, and MonologueAction extends the Action class. Astley implements the Purchasable interface to add the capability to be purchased. Astley class handles the purchase of Astley like other Purchasable in Assignment 2.

Astley also implements a new interface, Monologueable. Monololoqueable is an interface that represents the capability of an object to provide a monologue. The Monologueable interface has a method getMonologue that returns a string representing the monologue of the object. Astley implements getMonologue from Monologueable interface, randomly select and return a monologue string from the options available. Astley class also handles the deduction of subscription fees every five ticks through the tick method.

The MonologueAction class provides the functionality to perform the action of listening to the monologue. The execute method returns the monologue string when the action is performed. The allowable actions of ComputerTerminal are updated to include providing the action to purchase Astley.

With the adherence to the SOLID principles, the implementation of REQ4 is designed to be efficient, scalable, and maintainable. The use of abstraction and inheritance reduces redundancy and facilitates reusability and extensibility of the code. This design decision allows the code is easy to understand, maintain and extend in the future.

Alternative Design:

One alternative design could be handling the subscription of Astley and interaction with Astley in the Player class without creating new classes since it is owned by the Intern (the Player). However, this violates SOLID principles.

Analysis of Alternative Design:

The alternative design is not ideal because it violates various Design and the SRP in SOLID principles:

1. **Single Responsibility Principle (SRP):**
    - The alternative design would violate the SRP by combining the responsibility of handling the subscription and interaction with Astley with original responsiblity of the Player class. This leads to a violation of the SRP as the Player class would have multiple responsibilities, making it harder to maintain and extend in the future.

In terms of code smell, this alternative design will lead to a large Player class, which a indicates the likeliness of violating the Single Responsibility Principle (SRP).

Final Design:

In our design, we closely adhere to SOLID and DRY.

1. **Single Responsibility Principle (SRP):**
    - **Application:**
        - Astley class is responsible for handling the purchase of Astley, deducting subscription fees, and providing a monologue message.
        - MonologueAction class is responsible for providing the action to listen to the monologue.

    - `ComputerTerminal` class is responsible for providing the actions to purchase `Astley`.
    - As we can see, each class has a clear and distinct responsibility, adhering to the SRP.
- **Why:**
    - The adherence to the SRP makes the code easier to understand, maintain, and extend. By only taking care of a single concern, every class has a clear and distinct responsibility.
- **Pros/Cons:**
    - Improved readability, maintainability and extensibility of the code.
    - The downside could be the need of multiple new classes to handle different responsibilities, which may increase the number of classes, and lead to more complex interactions between classes.
    - In this case, the pros outweigh the cons as the increase in classes is manageable and provides clear separation of concerns.

2. **Open-closed Principle (OCP):**
    - **Application:**
        - Astley, MonologueAction, and ComputerTerminal classes extend their superclasses respectively, adding extra functionalities, without modifying the existing Item class.
        - Tick method in Astley class added the subscription fee deduction functionality without modifying the existing Item class.
        - We determine whether Astley can be picked up or dropped using a portable attribute in the Item class, without explicitly checking the class using `instanceOf` in the parent class.
    - **Why:**
        - Adhering to the OCP leads to a more maintainable and extendable code. It allows for the code to be extended without modifying existing classes by using abstraction.
    - **Pros/Cons:**
        - The advantages of adhering to the OCP include avoiding the need to modify existing classes when adding new functionality, which makes it easier to extend our code.
        - The downside could be the increase in the number of abstract classes or interfaces as an abstraction, which might make our codebase seems more complex.
        - In this case, the pros outweigh the cons in the long run as using abstractions allows for easier extension and maintenance of the codebase, without having to modify existing classes and potentially introducing bugs.
3. **Liskov Substitution Principle (LSP):**

- **Application:**
    - Astley and MonologueAction classes override the necessary methods from their superclasses without changing the meaning of the methods.
    - For instance, Astley overrides the tick method from Item, it still has the same meaning as in the superclass where it informs the carried Item of the passage of time.
    - MonologueAction overrides the execute method from Action, it still has the same meaning as in the superclass where it executes the action and returns a string as a result.
    - The inherited methods remain consistent with the super class.
- **Why:**
    - Adhering to the LSP ensures that the inherited methods maintain their meaning in the subclasses. This ensures consistent behaviour across different classes and following the contracts, facilitating future extensions and maintenance.
- **Cons:**
    - The benefits of adhering to the LSP include ensuring that the inherited methods are consistent and maintain their meaning in the subclasses.
    - There are no apparent drawbacks to following the LSP, as it ensures the robustness and consistency of polymorphism.
    - In this case, the pros outweigh the cons since using abstractions make it easier to extend and maintain the codebase with a lower risk of introducing bugs.

4. **Interface Segregation Principle (ISP):**
    - **Application:**
        - Astley implements the Purchasable and Monologueable interfaces, which are specific to the functionality required by Astley.
        - The new Monologueable interface is segregated from other interfaces, including only the methods required for providing a monologue.
        - Astley only implements the methods it needs, adhering to the ISP.
    - **Why:**
        - Adhering to the ISP ensures that classes only implement the methods they need, eliminating the need for classes to implement methods they do not care about or need.
    - **Pros/Cons:**
        - One of the benefits of adhering to the ISP is ensuring that classes only implement the methods they need.
        - The downside could be the increase of the number of interfaces. However, splitting larger interfaces into smaller ones lead to a code that is cleaner and more maintainable.

5. **Dependency Inversion Principle (DIP)**
    - **Application:**
        - MonologueAction class depends on the Monologueable interface, which is an abstraction, rather than concrete classes. This allows the MonologueAction class to not be dependent on the concrete classes that implement the Monologueable interface.
    - **Why:**
        - Adhering to the DIP ensures that high-level modules depend on abstractions rather than the details of low-level concrete implementations. By depending on abstractions, which can be implemented by different concrete classes with different implementation details, the code can be extended and maintained more easily.
    - **Pros/Cons:**
        - The benefits of adhering to the DIP include reducing dependencies on concrete classes by depending on abstractions, making the code more extensible and easier to maintain.
        - There are no significant downsides to adhering to the DIP, as it follows better design practices and leads to a more extensible and maintainable code.
6. **Don't Repeat Yourself (DRY)**
    - **Application:**
        - The new classes inherit methods from their super classes and overrides only the methods that are specific to them to extend the functionality, without repeating the code that is already implemented in the super classes, adhering to the DRY principle.
    - **Why:**
        - This reduces redundancy in the codebase. Reusing existing code and avoiding duplication results in cleaner code.
    - **Pros/Cons:**
        - The benefits of adhering to the DRY principle include reducing redundancy and improving maintainability.
        - There are no significant downsides to adhering to the DRY principle.

Handling connascense in our code:
By using interfaces to define the behavior of the classes, making our code easier to extend and reducing the number of possible bugs.

- **The connascence of name** is handled by using interfaces to define the capability of the classes. The classes implementing the interfaces must implement the methods with the same name as defined in the interface. The classes that use the interfaces must call the methods with the same name as

defined in the interface. This provides a contract that the classes must adhere to, making our code less error-prone and easier to maintain.

- The contract provided by interfaces also mitigates the possible downsides that comes with **connascence of type and position** by providing clear method signatures. This reduces the likelihood of errors due to incorrect types and positions provided to a method.
- We controlled the dependency, by adhering to the Dependency Inversion Principle. This prevents bugs from occurring due to changes in the concrete classes that implement the interfaces.

Code Smell:

- By keeping the SOLID principles in mind, we have avoided code smells.
- By creating classes that adhere to the SRP, we reduce the code smell of large classes that violate the SRP.
- By creating classes that adhere to the OCP, we reduce the code smell of classes requiring frequent modification when new functionality is added.

What Has Been Done and Why:

Added Astley class, extending Item, implementing Purchasable and Monologueable interfaces.

The Astley class inherits the actions to be picked up (getPickUpAction) and dropped (getDropAction) from Item. It implements the Purchasable interface to add the capability to be purchased. The Astley class implements the Monologue interface to add the capability to get a monologue from available options. The Astley class handles the deduction of subscription fees every five ticks through the tick method from the Item class.

Added MonologueAction class, extending Action.

The MonologueAction class extends the Action class and provides the functionality to perform the action of listening to the monologue. The MonologueAction class takes a Monologueable as a parameter and returns the monologue string when the action is performed through the execute method.

Updated ComputerTerminal class.

The allowable actions of ComputerTerminal are updated to include the action to purchase Astley.

Overall Pros and Cons of the Design

Pros

- The implementation follows SOLID principles, which facilitates the development of the game, including adding new features and debugging.
- By using abstraction and inheritance, it reduces redundancy and facilitates reusability and extensibility of the code.

Cons

- The Monologuable interface is currently only used by Astley, the getMonologue method is now randomly selecting a monologue from a list of monologues. The MonologueAction is using this getMonologue method to get a monologue. If more classes are added that implement Monologueable, the getMonologue method may need to be refactored to allow for different implementations of the method base on the new requirements.

How it can be easily extended

- To add more entities that can provide a monologue, only need to implement the Monologueable interface by implementing the required methods in the concrete class, and make it provide the action to listen to the monologue using the MonologueAction class.

Conclusion:

Overall, our chosen design provides a robust framework for implementing the Requirement 3: dQw4w9WgXcQ in the game. By carefully considering the requirement and adherence to design principles like SOLID and DRY, we have developed a solution that is efficient, scalable, maintainable, paving the way for future extensions and maintenance.

Requirement 4: [survival mode] Refactorio, Connascence's largest moon

UML class diagram

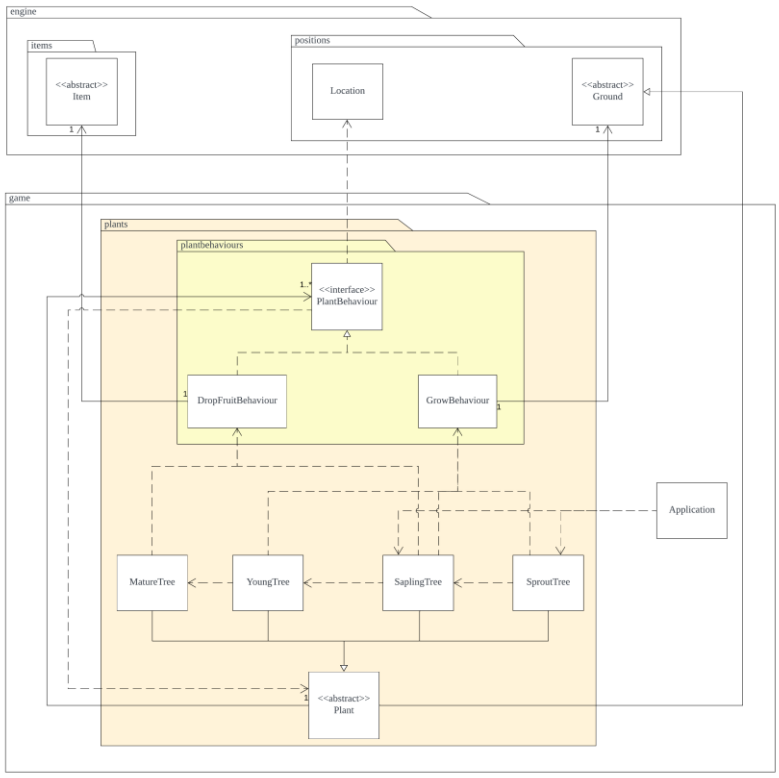

Figure 7 Class Diagram of REQ 4.

Sequence Diagram

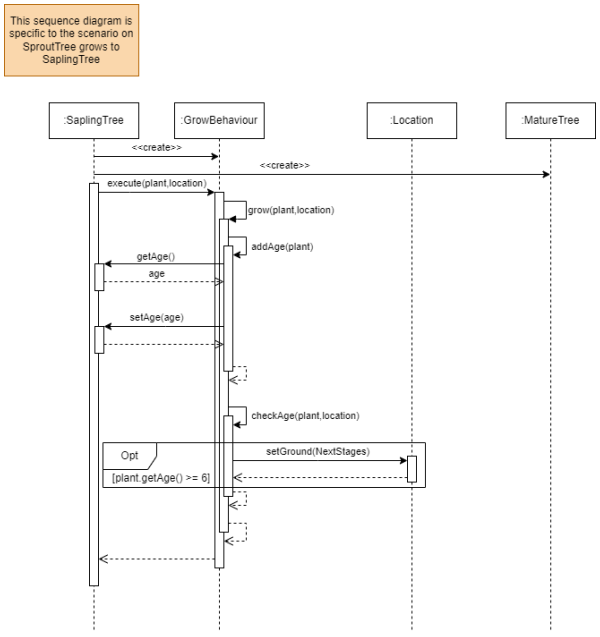

Figure 8 sequence diagram of REQ 4.

Design goal

For requirement 4 with given 2 new tree stages with different features i.e. able to drop fruit and grow. We aim to implement a design that able to provide flexibility on adding the features to all the tree stages without having redundancy and follows the SOLID principles.

Design Decision

I decide to make 2 classes i.e. one to handle the drop fruit feature and one to handle grow feature which then can be implemented to all tree stages rather than having them sitting in one class. Which then leads to 3 designs one proposed design and 2 alternative designs.

Proposed design

PlantBehaviour

So here similar with Enemy class we introduced a new type of Behaviour class called PlantBehaviour for plant since Behaviour is only used for actor. Here we have a PlantBehaviour interface class. Then we have 2 plant behaviours which is DropFruitBehaviour and GrowBehaviour which these 2 behaviours are implemented from the PlantBehaviour interface. Now the Plant base abstract class will have a treemap of PlantBehaviour which is similar to Enemy class. So, whenever we want a specific tree to have the plant behaviours, we can just put it in the PlantBehaviour treemap. Then we let the plant behaviour's constructor to handle the attributes i.e. plant's next stages and turns for next stage for GrowBehaviour; fruit and spawnFruitRate for DropFruitBehaviour.

Solid Principles

Single responsibility principle

Application

As previously the SaplingTree will have grow method and dropFruit method and for MatureTree it will have dropFruit method. Now we let the GrowBehaviour to handle grow method and DropFruitBehaviour to handle addFruit, dropFruit method.

Why

Because we don't want those tree class to handle the feature methods which makes them a god class.

Open-close principle

Application

We can add new tree with different behaviour to the Plantbehaviour treemap and add new behaviour like GrowFlowerBehaviour, WitherBehaviour by just implement PlantBehaviour and parse the attributes to the constructor.

Why

With the implementation above we can easily add new PlantBehaviour or tree without changing the base code.

Liskov Substitution Principle

Application

we move the Addfruit and Dropfruit methods to the DropFruitBehaviour which now fixed the Liskov Substitution principle. And of course, same goes with the grow method which moved to GrowBehaviour.

Why

Previously we have Addfruit and Dropfruit method in the Plant base class but since YoungTree and SproutTree didn't drop fruit, this will violate Liskov Substitution Principle because these tree didn't have the characteristics of Plant parent class as we have the Addfruit and dropfruit method in the Plant parent class, this means that the Plant parent class can drop fruit but in reality YoungTree and SproutTree which inherit from the Plant parent class can't drop fruit. But now after the move of the methods above to the corresponding plantBehaviour classes we resolve the issue.

Interface segregation principle

Application

We introduce PlantBehaviour interface which only have a execute method.

why

The interface PlantBehaviour will have no unnecessary method for both GrowBehaviour and DropFruitBehaviour.

Dependency inversion principle

Application

We depend on PlantBehaviour for the plantBehaviour treemap rather than the child class GrowBehaviour and DropFruitBehaviour.

Why

High-level modules should not depend on low-level modules. Both should depend on abstractions.

Don’t repeat yourself (DRY)

Application

Fruit, spawnFruitRate, next stages, turns to next stages attributes; dropFruit, grow method will only be in GrowBehaviour and DropFruitBehaviour.

Why

To avoid redundancy of the repeating attributes and methods.

Connascence

Name

SproutTree needs to call the correct name of SaplingTree when parsing it to the GrowBehaviour’s constructor parameter since SaplingTree is its next stage.

Type

The fruit needs to be type Item and fruitSpawnRate needs to be type double in the field of the parameter of DropFruitBehaviour’s constructor.

Meaning

For SaplingTree we parse the DropFruitBehaviour with SaplingFruit and 0.3. 0.3 is the probability which range from 0 to 1 of SaplingTree of droppingFruit on surrounding. Here 0.3 means that SaplingTree will have 30% chance of dropping SaplingFruit on surrounding.

Position

DropFruitBehaviour need to have the correct order of parameter i.e. Item fruit, double fruitSpawnRate when calling its constructor.

Algorithm

The checkFruitSpawnRate method must agreed on the generateRandomNumber method which will return a random double number between 0.0 to 1.0. as the fruitSpawnRate double attribute which will be used in the checkFruitSpawnRate method is a probability value between 0.0 and 1.0.

Execution

The grow method in the GrowBehaviour for SaplingTree has to execute setAge(getAge +1) i.e addAge method just can run the if (getAge() >= 6) condition i.e. checkAge method If the condition has met, we turn the tree to next stage. If we run the if condition first, the SaplingTree will grow in 5 turns which violates the specification in requirement 4 (SaplingTree grows in 6 turns).

Reducing connasence

Here we try to reduce the connasence of execution on GrowBehaviour and DropFruitBehaviour classes.

Encapsulate into smaller methods:

By doing so we encapsulate the grow method and dropFruit method into small methods. For GrowBehaviour's grow method we make it into 2 small methods which is addAge which is adding the plant's age by one and a checkAge method which has an if condition to check the current plant's age whether is it on age that it will turn into the corresponding next stage. So, the grow method will call the addAge first and then call the checkAge. Then the execute method in the GrowBehaviour class will call the grow method.

Private access modifier:

We make sure the grow, addAge, checkAge method are private so that the client will not be worrying about the procedure of the method execution i.e. execute which method first since it is private and unable to be accessed by the client. Same goes to DropFruitBehaviour, we have 3 small private methods here which is generateRandomNumber, getRandomLocation and checkFruitSpawnRate. The dropFruit method which is also private will call generateRandomNumber first then getRandomLocation and then checkFruitSpawnRate. After that the execute method will call the dropFruit method.

Pros and cons

Pros: Here the pros are it is easy to add the plant behaviour as we can just put those behaviour to the treemap plant behaviour. Also, it is easy to add new features like GrowFlowerBehaviour or WitherBehaviour as we can just implement those with the PlantBehaviour class.

Cons: The cons here is that we loose locality on the attributes i.e. fruit, fruitSpawnRate, next stage tree to the tree class. This means that these attributes are not in the tree class but in the plant behaviour's class. This impacts the code maintainability which the code will hard to understand and maintain because the related attributes are scattered throughout the codebase.

Alternative design 1

Abstract class – FruitPlant

We will have a parent abstract class called FruitPlant which extends Plant base class and implements spawnItem interface so here the FruitPlant will have fruit Item,

fruitSpawnRate double attributes and SpawnItem method which is a drop fruit method. For those inheritree that can drop fruit can extend this FruitPlant abstract class like Mature tree and Sapling tree. Then those tree that didn't have fruit can just extend Plant base class and they can implement GrowableGround interface if we wish the tree to grow to next stage.

UML class diagram (brief)

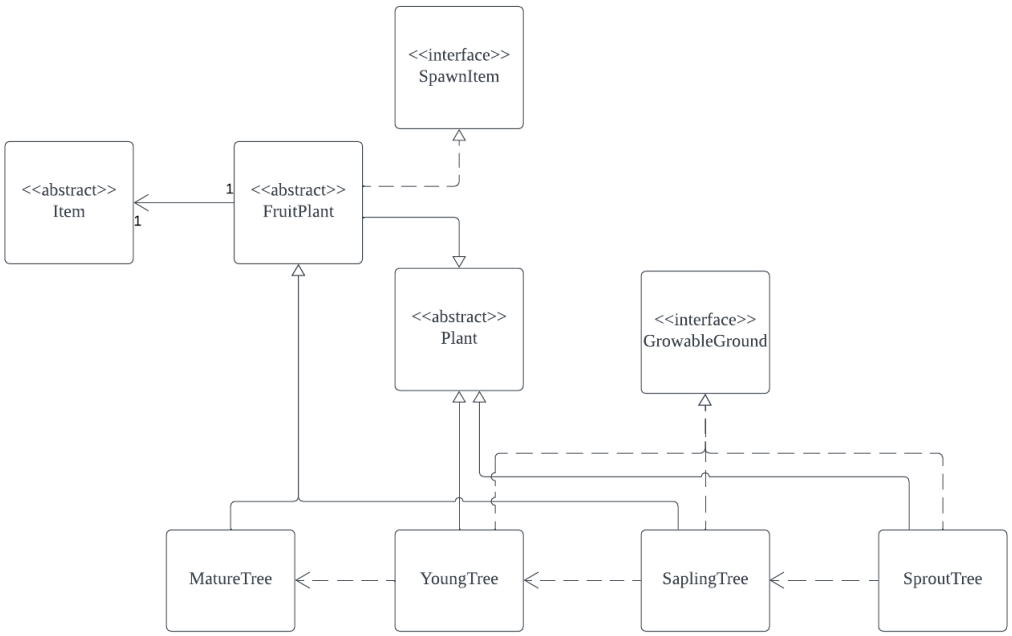

Figure 8 Class Diagram of alternative design 1.

Solid principles

Liskov substitution principle

Application

We introduce two new class which is Young Tree and SproutTree which both does not drop fruit and does not relate to fruit, fruitSpawnRate attribute and getFruit, addFruit, consumableToSpawn method, we move the fruit attribute to both Young Tree and SproutTree and the getFruit,addFruit, getfruitSpawnRate method to the FruitPlant abstract class.

Why

Previously we had fruit attribute and getfruit method in inheritree as both MatureTree and SaplingTree drops fruit. So after the changes, Plant base abstract class will not relate to drop fruit and we obey LSP as now the tree stages will behave as their corresponding parent class i.e. FruitPlant and Plant class.

Open close principle

Application

We add a new FruitPlant abstract base class so that it can be extended by trees that drops fruits.

Why

With the new FruitPlant abstract base class and the previously added GrowableGround interface, we allow new tree that drops fruit or grow to be added to the system by implement with the combination of both abstract class and interface without modified the codes.

Don't Repeat Yourself (DRY)

Application

The addFruit method and spawnItem i.e. drop fruit method is implemented in the FruitPlant abstract base class which then will be inherited by those tree that drop fruit. The tree doesn't need to implement the spawnItem method as it has inherited the method from FruitPlant base class.

why

Since the spawnItem method is big, we need to avoid the redundancy of the method so that our code can looks more cleaner.

Pros and cons

Pros: The pros are that previously we will have a drop fruit method, fruit Item and fruitSpawnRate double attributes in the plant base class so since we move the method and attributes above to FruitPlant abstract class we removed the redundancy and the unnecessary of the method and attributes on this tree that doesn't have drop fruit method. Also, it is easy to implement Growable tree as we can just implement the GrowableGround interface.

Cons: The cons are that if we wanted to add a new feature like it will grow flowers then we must make it interface. The problem here is that if the flower method is big, we have to include it in those flowers tree class which is sometimes will have duplicated code. Second, the features implementations aren't the same which drop fruit feature has to extends from FruitPlant class and Growable tree has to implement interface. As you can see here, we have different implementations here one is extend and one is implement. This will cause the code to be quite messy. We can fix it by simply turn the FruitPlant abstract class to an interface but that will cause significant of duplicated code for drop fruit method.

To resolve the cons above, we proceed to the second alternative design which we use interfaces.

Alternative design 2

Interface - DroppableFruit

Continue from the cons of the design above, so here we have 2 interfaces which are FruitDroppable which extends SpawnItem interface and GrowableGround. The FruitDroppable interface will have a spawnItem method which is a drop fruit method. But we have mentioned above is the redundancy codes from drop fruit method, so we introduce a utility class called TreeUtils which have a DropFruit static method with the Location, Fruit Item and fruitSpawnRate double parameters, so by implement FruitDroppable, the dropFruit method in the interfaces we just parse in the TreeUtils static method with the corresponding parameters. But here we keep the fruit Item and fruitSpawnRate double attributes in the Plant base class.

Uml class diagram (brief)

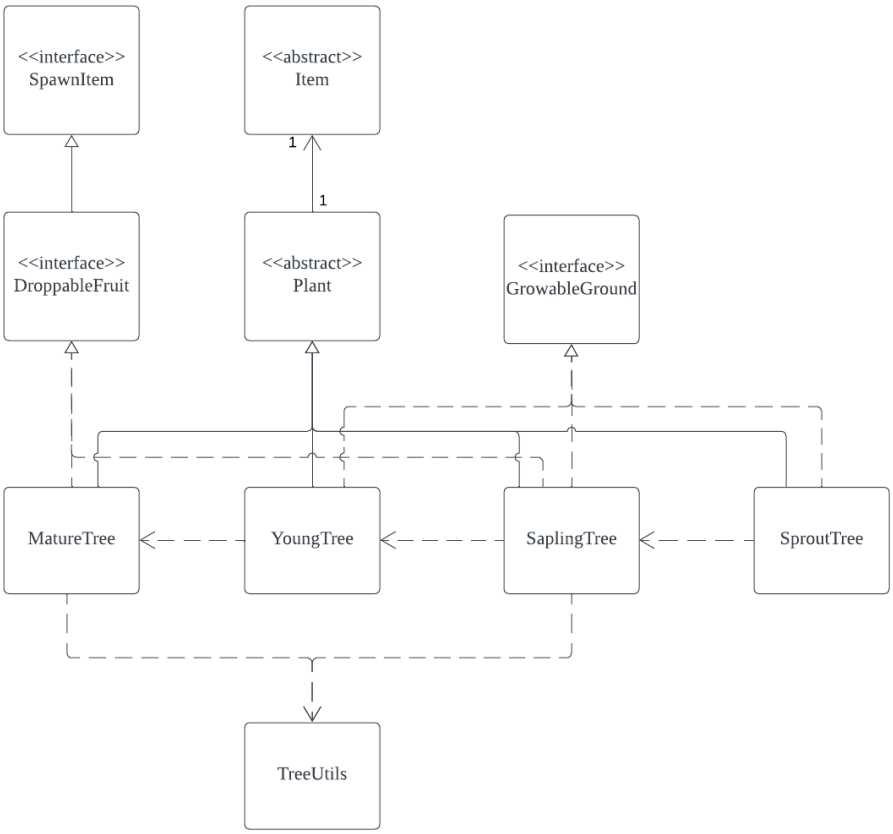

Figure 9 Class Diagram of alternative design 2.

Solid principles

Interface segregation principle

Application

We introduce a FruitPlant interface which allows a tree to drop fruit.

Why

This interface also ensures that those tree stages that can drop fruits i.e. Sapling Tree and Mature Tree to use all method without return null.

Liskov substitution principle

Application

We introduce two new class which is Young Tree and SproutTree which both does not drop fruit and does not relate to fruit, fruitSpawnRate attribute and getFruit, addFruit, consumableToSpawn method, we move the fruit attribute to both Young Tree and SproutTree and the getFruit,addFruit, getfruitSpawnRate method to the FruitPlant interface.

Why

Previously we had fruit attribute and getfruit method in inheritree as both MatureTree and SaplingTree drops fruit. So after the changes, Plant base abstract class will not relate to drop fruit and we obey LSP as both YoungTree and SproutTree behave as Plant parent class.

Open-close Principle

Application

We add a new DroppableFruit interface so that it can be implemented to the tree that drops fruits.

Why

With the new DroppableFruit interface and the previously added GrowableGround interface, we allow new tree that drops fruit or grow to be added to the system by implement with the combination of both interfaces without modified the codes.

Don’t Repeat Yourself (DRY)

Application

we move the dropFruit i.e. spawnItem method from MatureTree and SaplingTree to a utility class TreeUtils as a static Method which accepts the tree’s location,fruit spawn rate, and fruit as parameter. So, now both MatureTree and SaplingTree can just call TreeUtils’ spawnItem static method in their implemented spawnItem method and parse the required parameters to the static method to drop fruits.

Why

As the dropFruit i.e. SpawnItem method is quite big and had redundancy in both MatureTree and SaplingTree.

Pros and Cons

Pros: Flexible, able to easily add GrowableGround and DroppableFruit features by implementation one of the interfaces. Able to handle tree stages by just implementing nextStages method with next stages tree object.

Cons: Unnecessary Fruit Item and fruitSpawnRate double attributes in YoungTree and SproutTree class. Need to always call consumeToSpawn in the constructor to add the corresponding fruit and call spawnItem method in tick method.

Having that said, we need to find a better design which allow the tree class to avoid having the unnecessary attributes which leads to our proposed design that resolve this problem.

Final Design

What has been done and why

Here we choose to implement the system with the proposed design.

Create an interface PlantBehaviour: we introduced a new type of Behaviour class called PlantBehaviour for plant since Behaviour is only used for actor. Here we have a PlantBehaviour interface class.

DropFruitBehaviour and GrowBehaviour: Then we have 2 plant behaviours which is DropFruitBehaviour and GrowBehaviour which these 2 behaviours are implemented from the PlantBehaviour interface.

Added treemap PlantBehaviour on Plant base class: Now the Plant base abstract class will have a treemap of PlantBehaviour which is similar to Enemy class. So, whenever we want a specific tree to have the plant behaviours, we can just put it in the PlantBehaviour treemap.

Let Plant behaviour's constructor to handle the attributes: Then we let the plant behaviour's constructor to handle the attributes which is parse in by the constructor's parameters i.e. plant's next stages and turns for next stage for GrowBehaviour; fruit and spawnFruitRate for DropFruitBehaviour.

Pros and Cons

(Just for recall here, same as the one in proposed design)

Pros: Here the pros are it is easy to add the plant behaviour as we can just put those behaviour to the treemap plant behaviour. Also, it is easy to add new features like GrowFlowerBehaviour or WitherBehaviour as we can just implement those with the PlantBehaviour class.

Cons: The cons here is that we lose locality on the attributes i.e. fruit, fruitSpawnRate, next stage tree to the tree class. This means that these attributes are not in the tree class but in the plant behaviour's class. This impacts the code maintainability which the code will hard to understand and maintain because the related attributes are scattered throughout the codebase.

Ways in which it can be easily extended

Adding new plant: To add new type of plant that have the features i.e. plant behaviours, we just need to add the plant behaviour to the plantbehaviour treemap attributes on that corresponding tree class.

Adding new plant behaviour: To add new plant behaviour, you just need to implement the PlantBehaviour interface. Then you can add the new plant behaviour to the corresponding plant class by just put it on the plant class' plantbehaviour treemap attributes.

Conclusion

We choose the proposed design which is having 2 behaviours i.e. DropFruitBehaviour and GrowBehaviour which is implement from the interface PlantBehaviour. It is because it resolves both issues i.e the cons in alternative design 1 and 2. For alternative design 1 i.e. FruitPlant abstract class, we are forced to add an interface if we want to add new features. This can lead significant of duplicate codes since you have to reimplement the interface method. Also, alternative design 1 also have messy implementations i.e. have to extend FruitPlant for dropping fruit and implement GrowableGround for growing tree (not unite). Then for alternative design 2 we will have unnecessary Fruit Item and fruitSpawnRate double attributes for those non droppable fruit tree. So, with this proposed design we eliminate the cons of alternative design 1 and alternative design 2 ,provide a flexibility on adding the features to all the tree stages and adding new features like grow flower, wither feature etc.