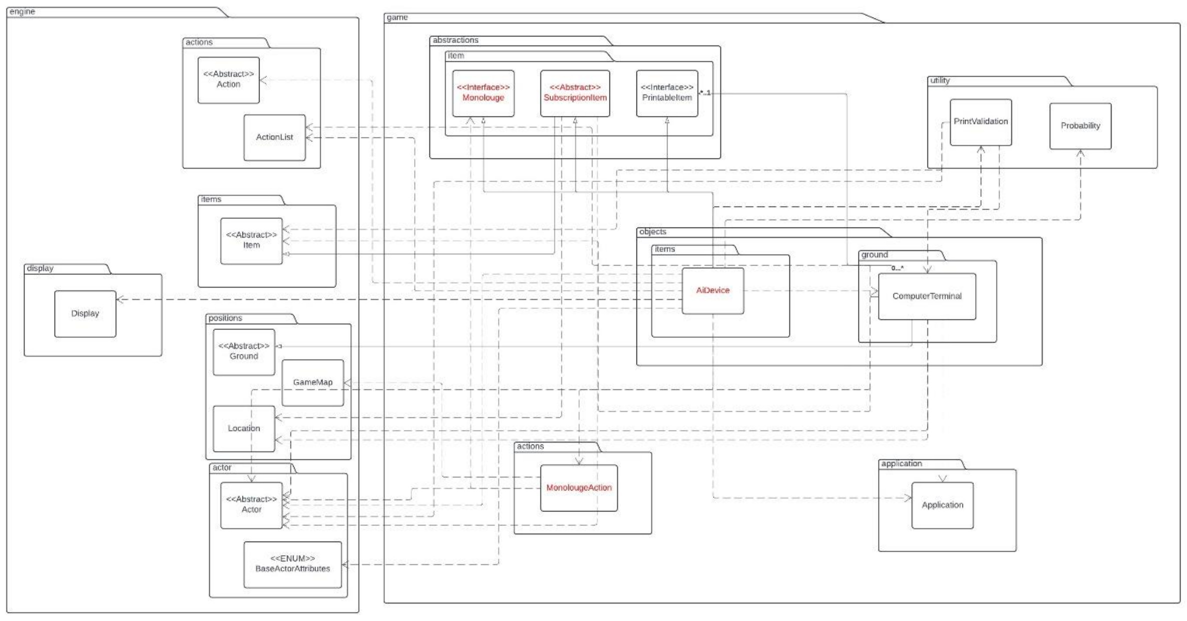
Assignment 3 Requirement 3 Design Rationale

UML



Design Goal

The goal of this requirement is to create a purchasable item called the AI device that costs 50 credits which is subscription-based meaning for it to perform any actions it charges a fixed number of credits every predefined period. The action that this device can perform is to monologue which means to display some predefined text on the screen if the user wishes to. Ideally this implementation will have close adherence to SOLID and DRY principles, as well as considerations to future extensibilities.

Design Decision

In implementing the requirement for the AI Device the decision was made to have an abstract class called SubscriptionItem as there is a lot of repeated functionality for the subscription which can be handled by having an abstract class. The other advantage of having an abstract class is that it is a base class for any items that can be subscribed too by only providing the extra functionality of handling subscriptions to the Item abstract class. We then chose to have a monologue interface as every class that can monologue will have a different implementation of it, and then we had a monologueAction which just calls the monologue method on a class that extends the monologue interface. The AI Device then inherits from the SubscriptionItem class, Monologue interface, PrintableItem interface as any item that can be bought from the computer terminal must implement the PrintableItem interface. The AI Device then declares any variables that are unique to it such as the subscription length, cost, occurrence, cost to buy, name and display character and is able to fall back on the SubscriptionItem abstract class to manage the subscription.

Alternative Design:

An alternative would be to not have the SubscriptionItem as an abstract class but as an interface instead. This would mean that there would be no shared functionality between the classes and each class would have to define its own method of doing a subscription. This was not done due to the fact that a subscription will be the handled the same way for all items that can be subscribed to all that needs to change are the parameters. Another way we could have implemented the monologue action was to not have a monologue interface but instead have the monologue action take in a list of options and randomly pick one of those options.

This design was not ideal due to the following reasons below:

1. Single Responsibility Principal (SRP)
    a. SRP was violated as the AI Device is not handling its own responsibility of being monolgued instead that responsibility is handed off to another class, the MonolgueAction class. Therefore, this principal was violated with this approach.
2. Open-Close Principal (OCP)
    a. OCP was also violated as if in the future another class that was able to monologue but did not just randomly pick from one of the options but it instead depended on attributes of the actor either another method would have to be created in the class that monologued that would change these options or the MonologueAction would have to be changed. This breaches OCP.
3. Liskov Substitution Principal (LSP)
    a. This method did adhered to this principle as either of the classes could be substituted where its parent class was required and the game would still be functional.
4. Do Not Repeat Yourself (DRY)
    a. Not having an abstract class for the SubscriptionItem results in repeated code for every class that could be subscribed too and the whole purpose of an abstract class is to provide some base functionality which is common therefore it made more sense to have an abstract class instead of an interface.

    For the violations mentioned above this design was not ideal and instead the decision was made to have an abstract class for the SubscriptionItem and a Monologue interface. Some downsides to having an abstract class instead of an interface is that we cannot take advantage of multiple inheritance if there are other abstract classes that the SubscriptionItem could be like, in our current design this is not an issue but if later on another abstract class was made and a concreate class needed to be able to be subscribed to but also inherit from the other abstract class then it could be an issue.

Our Design

Below is the following discussion of how our design adheres to the SOLID and DRY principles.

1. Single Responsibility Principle (SRP):

    - **Application**: Each class is responsible for handling a single responsibility: its own Monologue ability and what happens to it when the subscription is triggered and is open for modification (existing classes remain unchanged).
    - **Why**: This approach allows different classes to have different behaviours when getting a class monologues. Shared responsibility like managing the subscription will be common for all subscription-based classes and that can be shared by having a parent class. The parent class also outlines what attributes a class that has a subscription must have, something that cannot be done with interfaces.
    - **Pros**: Very flexible in what happens when an object is can monologue, saves rewriting code for managing the subscription and can rely on the parent class
    - **Cons**: The actually isn't any to have monologue as an interface as every monologue would be different. Cannot take advantage of multiple inheritance with the SubscriptionItem abstract class.

2. Open-Close Principle (OCP):
    - **Application**: The system is open for extension (new classes that can monologue can be added or existing classes can be changed to be monologue) but closed for modification (existing classes remain unchanged). Also classes that have a subscription can easily be added as they can just inherit from SubscriptionItem.
    - **Why**: This approach allows any class to become a class that can monologue with minimal impact on existing code, promoting extensibility. This is done by just having the class implement the interface monologue and have the monologue added to its allowable actions. Also, if a new subscription item needs to be created it will just inherit from the subscription item abstract class and be provided with base functionality.
    - **Pros**: Easy to make classes monologue and have a subscription
    - **Cons**: Reduces flexibility as subscriptionItems cannot inherit from other abstract class
3. Liskov Substitution Principle (LSP):
    - **Application**: The AI Device all maintain the functionality of their respective parent classes and can be interchanged without causing issues
    - **Why**: This is crucial for maintaining integrity of the game as all child classes should have the expected behaviour of their parent classes
    - **Pros**: Easy to introduce new item types by extending the SubscriptionItem class and maintain their functionality.
    - **Cons**: May require additional design considerations if new items substantially differ from the parent class
4. Interface Segregation Principle (ISP):
    - **Application**: The monologue interface has minimal responsibility and only handles making an object monologue by having just the monologue method in it.
    - **Why**: With this approach an item can do multiple things alongside monologue which increases flexibility and extensibility.

- **Pros**: Classes can be spawnable, monologue among other things and extend a base class such as item, ground or weapon.
- **Cons**: no cons to having monologue an interface as every monologue will be unique anyway.

5. Don’t Repeat Yourself (DRY) Principle:
    - **Application**: Having the subscriptionItem be an abstract class saves repeating code for managing the subscription, the subscriptionAction and monologue will be unique for every class therefore does isn’t repeated.
    - **Why**: The abstract class saves repeating managing subscription code
    - **Pros**: Not having repeated code is good because functionality is not repeated and if one change is made somewhere you may not have to change it in many places also each class that has a subscription will have to have the same attributes which leads to consistency
    - **Cons**: reduces flexibility

The chosen design adheres well to the SOLID and DRY principles, providing a robust framework for managing classes that are able to be consumed. Each class is designed with a single responsibility, extending from a common base and implementing an interface to implement the consumable functionality, which allows for clear and maintainable code structure. This approach not only ensures that enhancements can be made with minimal impact on existing functionalities but also sets a strong foundation for future requirements and extensions.