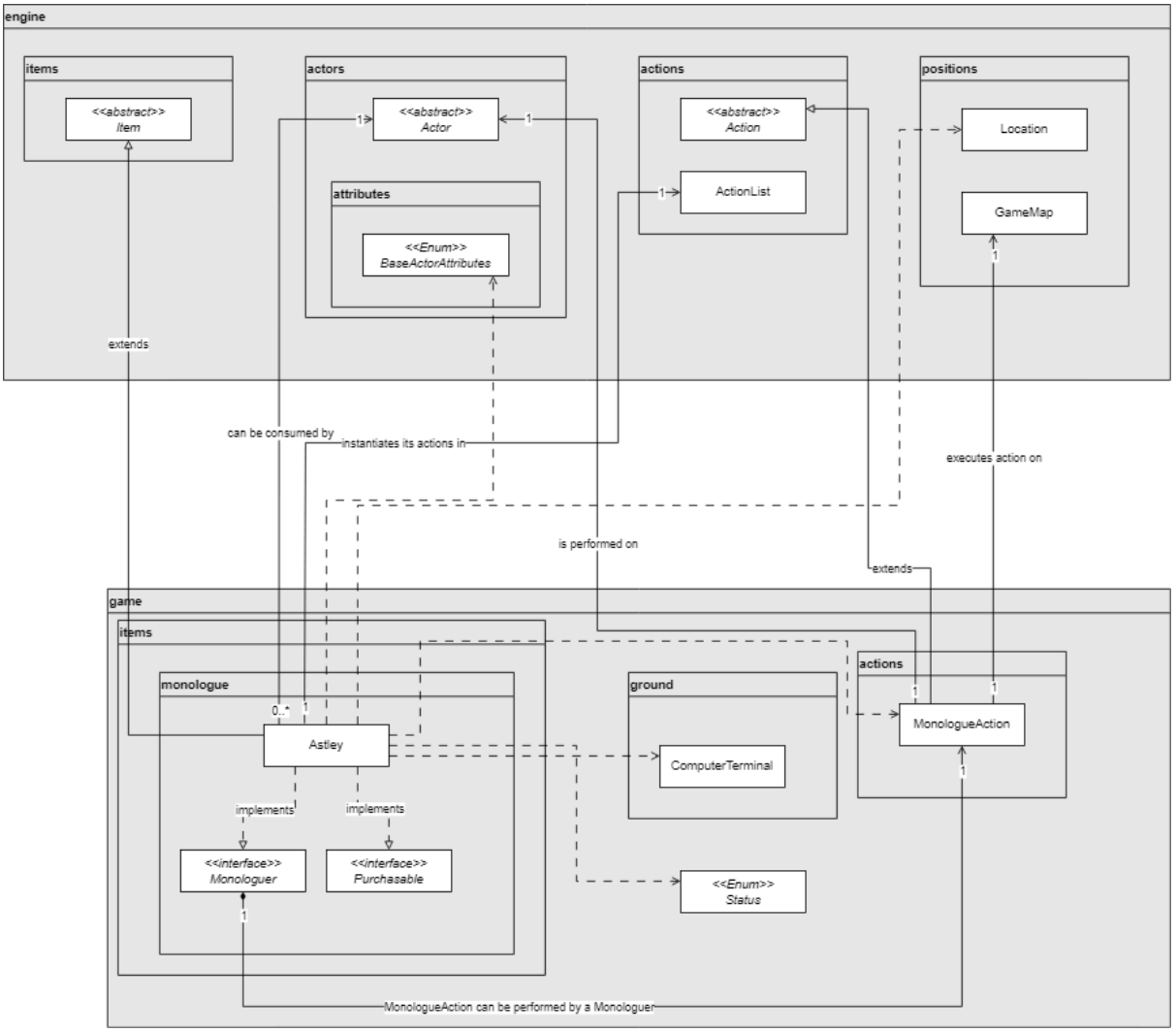
Assignment 3 Design Rationale
REQ3 : dQw4w9WgXcQ

UML



Design Goal

The goal of this requirement is to introduce a new item, AI Device called Astley. The Player should be able to interact with this item through the ComputerTerminal.

- Astley should be available for purchase through the ComputerTerminal
- Once purchased, the program should check if the item is in the Player's inventory :
    - Charge for subscription (1 credit) every 5 ticks that the item is in the Player's inventory
    - WIll present the option to perform monologues if item is in Player's Inventory

The goal is to implement the required specifications while adhering to OOP (SOLID and/or DRY) principles. This approach is considered as good design practice as it highly supports code extendability.

Design Decision

A class was created to represent the Astley device. Initially, the Astley class extends the Item abstract class and implements the Purchasable interface.

Astley extends Item and its attributes such as name, display character and portability are constructed. Astley also implements purchasable as it can be purchased by the Player through the ComputerTerminal. It inherits the purchase method from the interface to handle the effects it gives to the Player upon purchasing it. In this case, it can only deduct the Player's balance by its price (5o credits) and adds itself into the Player's inventory.

Then, after it has been purchased, the Player automatically subscribes to it. This will reduce the Player's credit by 1 for each 5 ticks that the item is in the Player's inventory. This is handled by the inherited tick method where it checks if the Player's subscription is active. If it is, the counter will increment by 1 and again, check if it has been 5 ticks to decide if Player should pay the subscription fee. Once payment is made, the counter is reset to initial value.

The way the implementation checks if subscription is active, is by making another method called isSubscriptionActive. This method checks if the item is in the Player's inventory and if said Player has enough balance to pay subscription fee. If both conditions are satisfied, subscription is made active.

For now, the only action it can imply towards the Player (once purchased) is to monologue with the Player. This is when we realised that introducing a Monologue interface and MonologueAction class would be a better approach. The current specifications only required us to allow Astley to have a monologue with Player so we introduced a getMonologue method to the Monologue-r interface to get the line of monologue the item can "say". So, all the subclasses implementing this interface can return the String of monologue it can say.

Similarly, the MonologueAction extending Action inherits the execute and menuDescription method. It takes in Monologue-r as a parameter, indicating that MonologueAction can only be performed by a Monologue-r item. The execute method executes the getMonologue method from the Monologue-r and menuDescription returns the respective string description for the action.

So, implementation of Astley changes. Now, it extends Item and implements both Purchasable and Monologuer. With this new implementation of the Monologuer interface, Astley can generate a monologue, according to the specifications provided. With that, once an actor possesses this item, it (the Astley AIDevice) is allowed to perform the MonologueAction.

Alternative Design

The alternative to the approach is to not make a Monologue interface, and only let Astley extend Item and implement Purchasable. The code implementation would look similar, just that there won't be "@Override" statements over the getMonologue method, as it will be a concrete method in the Astley class.

Additionally, one way we could tackle this requirement is to make a Subscribable interface and SubscribeAction class. This is to prepare if in the future, there are items that Actors can subscribe to. But we feel that, implementing this at this stage would be considered as over-abstraction. So, we decided not to go with this idea.

This way, Astley will still be able to perform as it should, but we'll explain why thai implementation might not be the best approach.

Analysis of Alternative Design

The alternative design is not ideal as it violates this principle

Don't repeat Yourself (DRY)

If in the future, the requirement decides to introduce other entities (actors, items, or ground) that are able to perform monologues, they would need the getMonologue method to return the string of words they will be "saying". This would serve the same functionality as the other getMonologue methods in other classes. Hence, this approach would break encapsulation and technically be redundant.

Final Design

The implementation follows the following principles

Single Responsibility Principle (SRP)

Each class and method has a single responsibility.

- Astley Class
  The Astley class constructs the Astley item. It handles the purchase, subscribing, and performing monologues functionality. Each of the functions under the class handles one functionality, as well. For instance, the isSubscriptionActive class is in charge of checking the validity of the subscription. Likewise, tick handles the subscription (deducting Actor's balance).
- Monologuer Interface
  An interface which encapsulates the behaviour of a Monologue-r. In this case, it defines that every monologue-r should be able to return a monologue in which it will "say".
- Monologue Action Class
  This action class handles the execution of the behaviour of the Monologue-r.

Hence, adhering to SRP.

Open/Closed Principle (OCP)

The design allows for easy extension – new entities that are able to perform monologues can easily be added in without needing to change any of the existing code. For example, if a new type of actor which can perform monologues (YapperAstro) were to be introduced to the game, all we have to do is create a class for it and have it implement Monologuer. All its attributes can be constructed and what monologues it "says" can be settled within the inherited getMonologue method. This can be done without needing to change the existing code. With that, this approach follows OCP.

Liskov Substitution Principle (LSP)

The LSP deduces that objects of the subclasses can be accepted in place of objects of the superclass, when expected. The implementation adheres to this as Astley implements the Monologuer. And when MonologueAction expects a Monologuer, it can and will accept Astley as its instance. Therefore, adhering to LSP.

Dependency Inversion Principle (DIP)

The current implementation depends on abstractions, instead of concrete classes. Astley implements (depends) on the Monologuer interface. This allows the Player to interact with a Monologuer through the Astley item. This makes the Player depend on abstractions, rather than concrete implementations.

Don't repeat Yourself (DRY)

The code follows the DRY by using interfaces. Interfaces encapsulates common behaviours of said entities. In this case, the Monologue-r. By doing this, all entities implementing this interface have to return a monologue, by default. Thus, each entity that is able to perform monologues reuses the common behaviour defined in the interface class.

Connascence of Name

The implementation minimises connascence of name by properly giving each variable and function appropriate and consistent names. Each method and attribute in the implementation has been given an appropriate and descriptive title, to ensure no confusion and clear transparency of understanding.

Connascence of Type

The implementation minimises connascence of type by ensuring that type parameters are clear. This helps in reducing future type errors. One example of this is in the MonologueAction, where a Monologuer type object is expected.

Connascence of Identity

The implementation minimises connascence of identity by maximising the use of private attributes. This protects the integrity of data within the class. This can be seen in the private attributes under Astley.

Pros & Cons

Pros

The approach supports extendability. Like mentioned, new monologue entities can easily be introduced without having to change any of the existing code. Furthermore, it is relatively easy to maintain. This is due to the clear separation of functionality throughout the implementation. The approach focuses on giving each class its own responsibility, which allows it to be very maintainable and

Cons

However, this implementation might be quite complex as the Astley AI Device itself can have the PurchaseAction acted upon it, and also act upon Actors. Having these requirements, implementing different abstract classes and interfaces were necessary but as expected, contributes to the complexity of the system.

Conclusion

The approach adheres to OOP principles, namely the SRP, OCP, LSP, and DIP. It follows the DRY principles, too. Overall, the design ensures that the code is extendible while minimising the potential risks of errors when adding on functionalities.