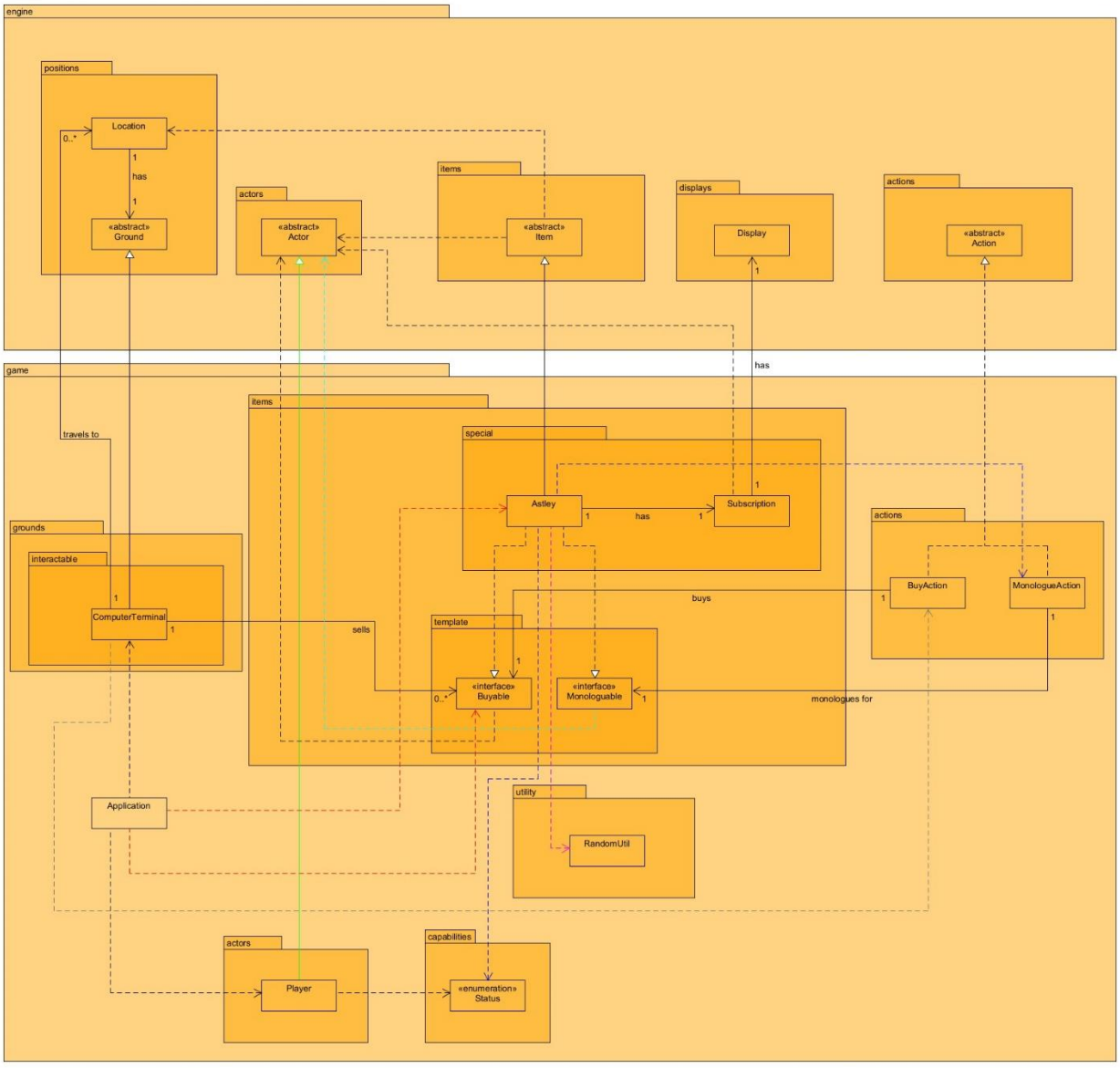
Assignment 3 Design Rationale

Requirement 3: dQw4w9WgXcQ

UML Diagram:



Design Goal:
The design goal for this assignment is to add the new class Astley which represents an AI device that can be purchased from the computer terminal for 50 credits while adhering closely to relevant design principles and best practices. If Astley is in the player's inventory, the player must pay a subscription fee of 1 credit every 5 ticks to continue to use Astley. With an active subscription, the player is able to listen to Astley monologue. If the player fails to pay the subscription fee, their subscription is paused and they will not be able to listen to Astley until they pay the subscription fee.

Design Decision:
In implementing Astley and its behaviour, the decision was made to create a Monologuable interface. The Monologuable interface contains the method generateMonologue, which returns a String monologue from the class that is monologuing. This interface is implemented by classes that can monologue such as Astley.
As the player needs to have the option to listen to Astley monologue, a MonologueAction class that extends the Action class from the game engine was created. MonologueAction has an instance of Monologuable as an attribute, and calls the Monologuable's generateMonologue method in MonologueAction's execute method.

This design makes it easy to add new classes that can monologue to the player in the future when the game is extended. Any new classes that monologue would simply need to implement the Monologuable interface, and fulfil the contract of implementing the generateMonologue method, and contain MonologueAction in the new class's allowableActions method.

The pros of this design choice are that new classes that monologue can be added without the need to modify the Monologuable interface. Our design is therefore open to extension and closed for modification, and adheres to the Open-Closed Principle. The Monologuable interface only contains the single method generateMonologue, no class implementing it will be forced to have any additional methods that it does not use, our design hence adheres to the Interface Segregation Principle. Having Monologuable as an interface also allows the classes implementing it to provide their own implementation of generateMonlogue, with different monologue options and conditions.

The cons of this design choice are that if future Monologuable items had similar or identical logic in their generateMonologue, there would be code duplication and a violation of the DRY principle. The design also has Connascence of Name, as both the Monologuable interface and MonologueAction must agree on the name for generating monologues.

An alternative design would be to have Monologuable as an abstract class that extends the Item abstract class. The problem with this alternative design is that future classes that can monologue will not necessarily be items like Astley. If Monologuable was an abstract class that extends Item, classes would be forced to have the functionality of item that they do not need. This would be a violation of the Interface Segregation Principle. If subclasses of Monologuable do not use the functionality of Item, this would result in the code smell of Refused Bequest. An advantage of our chosen design is that by having Monologuable as an interface, our design is more flexible in the sense that we can add not just new items that can monologue, but actors and grounds too.

In order for Astley to be bought from the ComputerTerminal, Astley implements the Buyable interface created in Assignment 2. The ComputerTerminal was refactored from its implementation from Assignment 2. The ComputerTerminal now has a non-default constructor that takes in the list of Buyable items that can be bought from the ComputerTerminal. This design gives us the flexibility to have ComputerTerminals that sell different Buyables. This design also adheres to the Open Closed Principle as ComputerTerminal is open for extension and closed for modification in the sense that different ComputerTerminals selling different Buyables can be created without modifying the ComputerTerminal class.

To handle Astley's subscription logic, a Subscription class was created which Astley has as an attribute. The Subscription class contains a non-default constructor that takes in the subscriptionFee, the number of credits that a subscribing actor is charged by, and the billingCycle, which represents how many ticks before the subscribing actor is charged again. In the case of Astley, Astley has a subscriptionFee of 1 and a billingCycle of 5 ticks. The Subscription class contains a tickCounter attribute which is incremented by 1 each time its tick method is called.

A new capability in the Status enum class was added called FACTORY_EMPLOYEE. In Astley's tick method, the Subscription instance's tick method is called only if the actor carrying Astley has the Status FACTORY_EMPLOYEE. This FACTORY_EMPLOYEE capability was added to the Player class and prevents actors that can pick up items such as the AlienBug from using Astley's subscription. It was initially considered to use the HOSTILE_TO_ENEMY status instead, however actors added in the future that work for the factory and have access to Astley's services might not necessarily be hostile to enemy actors. This FACTORY_EMPLOYEE capability thus gives the design more flexibility. However, a disadvantage to this design choice is that the FACTORY_EMPLOYEE capability is currently only used by the Player class, and this might be an unnecessary abstraction, exhibiting the Speculative Generality code smell. Another disadvantage is that this design leads to a Connascence of Meaning. If the meaning of

FACTORY_EMPLOYEE changes, for example if employees can no longer use services such as listening to Astley, it could affect the Astley class and any other classes dependent on the capability.

The Subscription class's tick method handles the logic of Astley's subscription. If the actor carrying Astley has the status FACTORY_EMPLOYEE, the Astley's Subscription instance's tick method is called, which increments the Subscription's tickCounter by 1. If the tickCounter is a multiple of the Subscription's billingCycle (5 in the case of Astley), the tick method then checks if the actor can afford the Subscription's billingFee. If so, the billingFee is deducted from the actor's balance and Subscription's subscriptionActive boolean attribute is set to true. Else, subscriptionActive is set to false. The Subscription class contains the method isSubscriptionActive, which returns a boolean stating if the subscription is active or not. Astley uses this method in its allowableActions method, where a MonologueAction is only added if the subscription is active.

An alternative approach considered was to have the subscription logic inside the Astley class instead of creating a separate Subscription class. However, with this approach Astley would have to handle its monologuing, buying, and subscription logic. This would be an example of a God class with the Large Class code smell and a violation of the Single Responsibility Principle. If the subscription logic was included in the Astley class, all the attributes related to Astley's subscription would be in the Astley class. This leads to the Data Clumps code smell, where variables such as subscriptionFee, billingCycle and subscriptionActive are always used together and should be moved to a separate class.

With a separate Subscription class, our design avoids the Large Class and Data Clumps code smells, by separating subscription related logic and attributes from the Astley class. Our design also more closely adheres to the Single Responsibility Principle, as Astley is no longer responsible for its subscription logic. The Subscription class also makes it easier to add new classes with subscriptions. These classes could use the Subscription class to handle their subscription logic, instead of rewriting the same logic, preventing code duplication and adhering to the DRY principle.

Conclusion:

Overall, our chosen design provides a robust framework for implementing Astley in its logic. By carefully considering design principles along with the requirements of this assignment, we have developed a solution that is efficient and scalable, paving the way for future extensions.