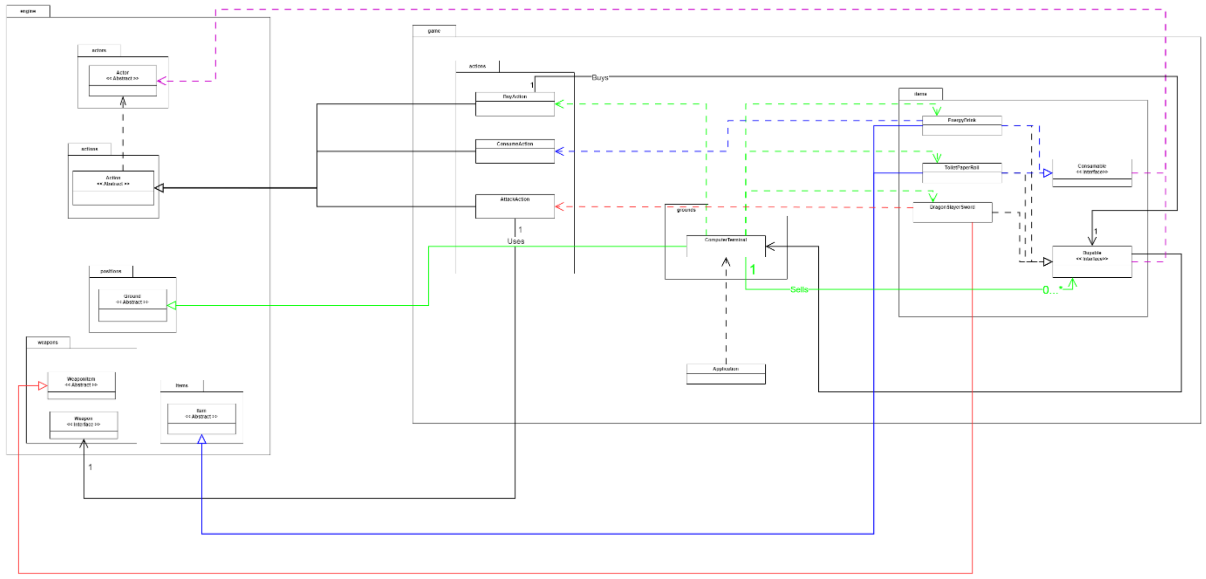
Assignment 2 Design Rationale

Requirement 4: Static factory's staff benefits

UML Diagram:



Design Goal:

The goal for this requirement is to implement a child class of Ground called "Computer Terminal" that will act as a shop for the players to purchase new items. Currently the design does not need the player to sell items to the shop, only that the player can buy from it.

Other than that, the goal should also to add 3 new items that can only be obtained from the shop, which is:

- "Energy Drink" that costs 10 credits and heals the user by 1 point. When buying there's a 20% chance for the shop to ask the user to pay double the original price.
- "Dragon Slayer Sword" that costs 100 credits and deals 50 damage and has 75% accuracy. When buying there's a 50% chance for the shop to just steal the credits without giving the weapon
- "Toilet Paper Roll" that costs 5 credits and does nothing else. When buying there's a 75% chance that the price will be reduced to 1 credit.

Overarching goal is to reuse the game engine's existing code as much as possible and adhere to the SOLID principles.

Design Decision:
In implementing the Computer Terminal, we chose it to extend from the Ground abstract class as this follows the Don't Repeat Yourself (DRY) principle. This was similarly done for Energy Drink, Dragon Slayer Sword and Toilet Paper Roll. The Dragon Slayer Sword extends from WeaponItem, and the Energy Drink and Toilet Paper Roll extends from Item. Additionally the Energy Drink implements from Consumable interface (a modified version of Consumable abstract class from previous assignment requirement to meet current requirement and extendibility)

We also introduced 2 new classes, which is the Buyable interface and the BuyAction class. With these classes we hope to achieve extendibility and adhere to the Single Responsibility Principle. An alternative approach would have been to code it directly inside the Computer Terminal class but that would violate the Single Responsibility principle as well as Inversion Dependency principle as we would need to keep creating items inside the Computer Terminal and thus have too many dependencies.

However with these 2 implementation, it may have violated Liskov Substitution principle as the BuyAction relies on the classes that implement the Buyable interface to have their procedures be correct and not return any values outside of our expectations or else the parent class Action will cause issues in the program. But this does to allow the allowableActions method to be used while not affecting other classes that extends from Action.

It also means that it may violate the Single Responsibility principle, because each classes that implement Buyable interface now has to implement the code to be able to interact with the Player to deduct their balance. So like the EnergyDrink class should only be concerned with healing the player but now they also have attributes for their prices and how they should be sold. But we nonetheless we chose these implementations as it allows the Computer Terminal to not hold too much responsibility, as the code for deducting balance and alternative methods will not be solely dependent on Computer Terminal and thus not creating a God class.

To store each of these items to sell, the Computer Terminal will use an ArrayList containing Buyable objects. This should follow the Open Closed principle and makes it scalable to store objects on a larger scale without having to rewrite or read through thousands of lines of code.

Alternative Design:
Instead of implementing the Buyable interface, it could have been implemented as an abstract class.
The code will need to have implemented the buyItem(Actor actor), setPrice(int itemPrice), and getPrice() method. The buyItem method will have to be an abstract method as any items that extends from it will possibly have their own methods and implementation for buying the items. This approach has a few disadvantages that we cannot ignore, for one thing we can only extend from the Item abstract class once. Meaning we will need a roundabout way of extending classes like EnergyDrink twice. This makes the code too rigid and if any of the code from Buyable abstract class gives an error, it will affect the code down the line.

Not to mention this also violated the Open Closed Principle. An example will Toilet Paper Roll and Energy Drink class. While although both methods will allow the player to "buy" the items, the Toilet Paper Roll needs to contain the discounted price while the Energy Drink class contains a multiplier to double the costs of it's original price. Having an interface would have given it loose coupling but letting it be an abstract class would make it hard for extendibility. An example would be polymorphism for the buyItem method. If we were to have different input arguments for each buyItem method, this would make scalability hard as all classes that extends from the abstract class needs to reimplement every method. And the child classes might even not need those methods leaving most of them redundant. An interface solves this problem as we can use the Interface Segregation Principle by having multiple interface, such that only the classes that needs those methods will implement that specific interface.

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

1. **Open Closed Principle:**
   - If we implement Buyable as an abstract class, EnergyDrink, ToiletPaperRoll, and DragonSlayerSword will have a very tight coupling with the Buyable abstract class as future features might need to force the abstract class to be modified. Classes should be open for extensions, not modifications. Thus this abstract class violated this principle.

Final Design:

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

1. **Open Closed Principle/Inversion Dependency Principle:**
    - **Application:** The interface implementation for Buyable allows the extending of features of EnergyDrink without having to affect other classes like ToiletPaperRoll, because it doesn't contain any concrete methods like from an abstract class.
    - **Why:** This is important as different items have different effects and implementation, while we can create a whole new class instead of implementing the interface this would have violated the DRY principle.
    - **Pros/Cons:** Other than the justifications given above, one downside is that if there are many methods needed to be implemented, if we also adhere to Inversion Dependency Principle, there would be far too many interfaces that we need to search through and implement. So for scalability, there is a limit if we over do the interfaces.
2. **DRY Principle:**
    - **Application:** By having the new class BuyAction extend from the Action abstract class, other Action classes would not have a conflict with this class.
    - **Why:** As the game grows larger and larger, we should not have the code to be unnecessarily repeated or else they will be hard to read and make it hard to refactor codes when new methods are implemented or old methods are changed.
    - **Pros/Cons:** This limits our ways to implement methods such as buying something, we need to rely on the methods outside of the class to not fail or else the input given to the BuyAction class like the execute() method will fail. This gives it a bit more coupling, but allows the whole programme to be more polymorphic.

Conclusion:
Overall, our chosen design provides a robust framework for implementing the shop items and still allow methods with very different requirements to be implemented without having much modifications. By carefully considering relevant factors, such as design principles, requirements, constraints, we have developed a solution that is scalable and maintainable, paving the way for future enhancements, extensions and optimizations.