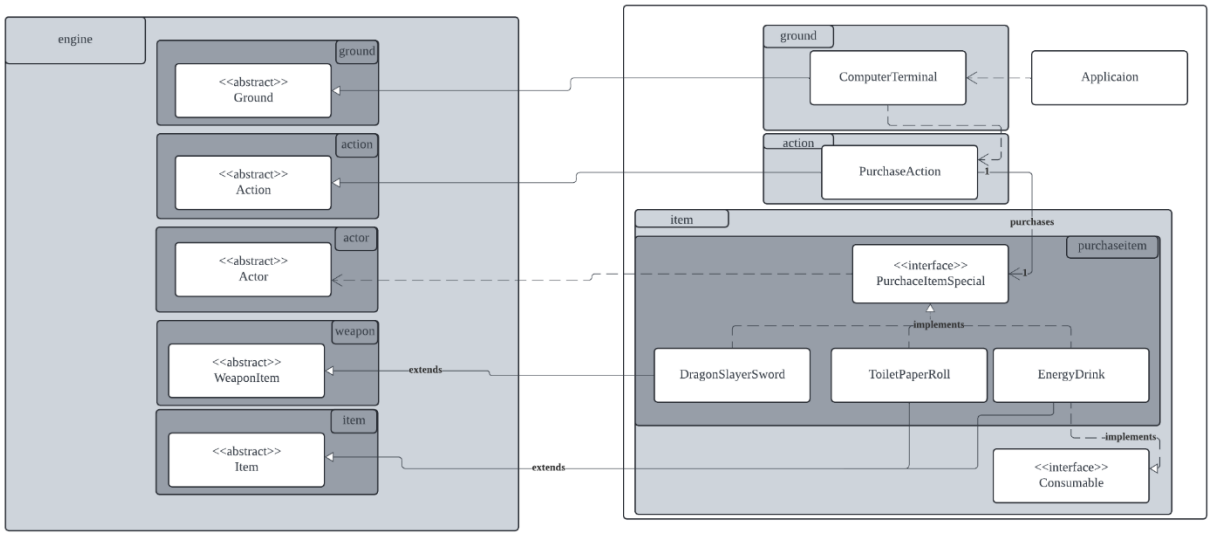
Assignment 2 Design Rationale

Requirement 4: Static factory staff benefits

UML Diagram:



Design Goal:

The design goal for requirement 4 was to create a computer terminal object that the player can interact with and purchase items from. The items available included an energy drink, with 1 health point healing properties when consumed and a price of 10 credits, with a 20% chance of being charged double. A dragon slayer sword which is a weapon that can be used to inflict 50 damage with a 75% hit chance, the weapon has a price of 100 credits with a 50% chance of the computer terminal malfunctioning and not giving the player the item, but still charging them. The final item is a toilet paper roll, it has no function and a price of 5 credits with a 75% chance of being charged only 1 credit. The computer terminal will handle the players balance as in deducting balance on successful purchases or preventing the user from purchasing if their balance is insufficient.

Design Decision:

A priority for requirement 4 was ensuring that the code strictly follows the solid principles, this is evident in the code with the abundant use of interfaces and action classes allowing for the easy implementation of new features into the code. As well as ensuring each class has only a single responsibility, keeping the maintainability of the program high

To maintain Solid principles, such as the single responsibility principle, the code was split into individual classes and interfaces, for example the use of the PurchaseItemSpecial interface easily allows for the creation of many new special items, it removes the necessity for repetition of code in the classes and aids in the adherence of the SRP as well as DRY principle Don’t Repeat Yourself.

In order to implement the purchasing feature, the computer terminal employs a purchase action, easily allowing the implementation of new purchasable items rather than adding that implementation into each class thus aiding in the maintainability and organisation of the code, this was a high priority in our design to ensure that the code is well organized within the repositories and each class is well laid out to ensure high readability.

Alternative Design:

One alternative approach would involve placing the majority of the code into one single god class, this would involve replacing all the interfaces and their implementations with classes that inherit from item. This would remove the need for interfaces but it would involve repeating code in all the classes

Analysis of Alternative Design:

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

1. **[Single Responsibility Principle]:**
    - By removing PurchaseItemSpecial interface, the purchase action would need to be updated to handle all the special features for the items that can be purchased, directly violating the SRP as these functions can be handled easily in their own separate classes
2. **[DRY]:**
    - This would also violate the don't repeat yourself principle as it would involve duplicating many sections of code to allow for the function of all purchasable items, this is bad design as it increases development time for no extra gain

Final Design:

1. The pros of the design are that it is highly readable and extensible and reduces the severity of failure points by having each class only conducting one main action, we remove the reliance on god classes which have only a single point of failure which is a bad design, in the event of an error to these god classes the entire system may malfunction however with the design approach we used for this project we reduce the likelihood of the whole system being affected by any logical errors. The cons of this design is that the program must be split into many different classes and interfaces which can create problems with dependencies in between scripts as well as the organisation of code.

2. **[Single Responsibility Principle]:**

- **Application:** The program adheres to the SRP by using abstract classes and interfaces to reduce the need for god classes, as each method is implemented in its own class thus if one method fails it will not affect the other related methods.
- **Why:** As mentioned above, it is important to follow this principle to reduce points of failure, should one method contain a logical error in a god class it can greatly effect the application as a whole, so by following the principle we can reduce points of failure allowing for the other methods to run as intended in the event of a mistake

[Open-Closed principle]:

- **Application:** By utilising interfaces it makes it incredibly easy to create new items in the game without modifying any existing code, simply create a new implementation of the interface and the item will be created, allowing for a high level of extensibility for the program.
- **Why:** It was of great importance to allow for extensibility of the project as we move into the third iteration of the game and add more features we will need to build off the current design, keeping the design extensible will make this much easier in the future

3. Dependency inversion principle

- **Application:** The design uses inheritance in a carefully designed manner to ensure that each class only inherits from super-classes and there is no dependency on any child classes, as this can cause errors with down-casting which we need to avoid
- **Why:** By avoiding down-casting we can eliminate any errors caused by calling upon child classes that may not contain the needed method

Conclusion:

Overall, our chosen design provides a robust framework for creating the computer terminal ground object which the player can interact with and purchase items from. By carefully considering design factors such as the solid principles and DRY principles, we have developed a solution that is efficient and highly extensible, paving the way for future versions of the program to easily add new content in updates along the way.