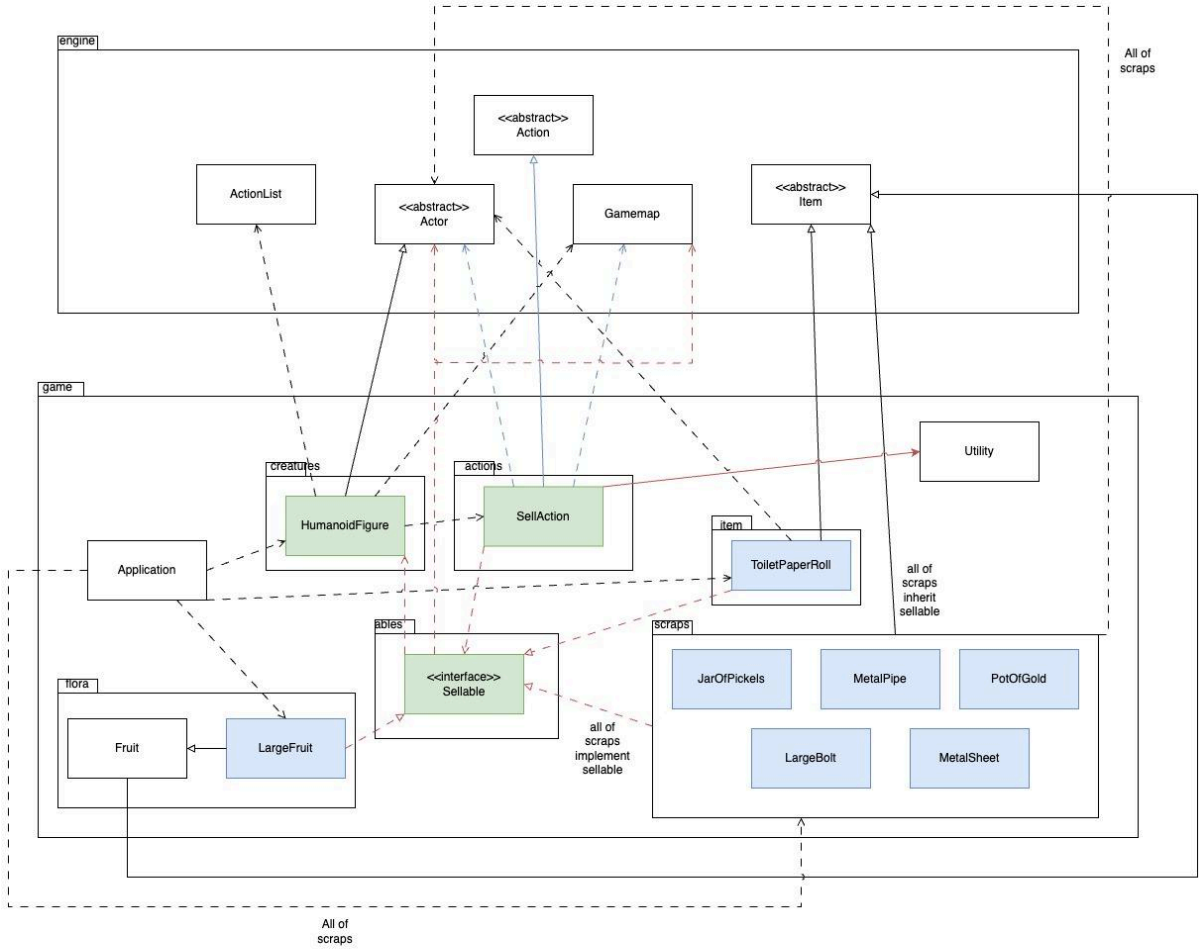
UML Diagram



Green classes are newly created classes while blue classes are existing classes that were modified

Design Goals

The goal for this requirement is to create a humanoid figure actor that the player is able to sell certain items to while adhering closely to SOLID and DRY principles.

Design Decision

For the implementation of the requirement I decided to create a sellable interface to flexibly enable sellable items with methods in an extendable manner, the interface contains methods to return the result of action as a string, sellAction, int selling price and item to be sold. The SellAction class was created to encapsulate the logic of selling items executing the action of selling action by removing the item from players inventory as returning the result of the action as a string. The HumanoidFigure class was created to represent the actor humanoid figure as well as providing allowable actions for the player based on checking if items in player's inventory are sellable, the class directly extends actor. For the item classes they are responsible for encapsulating the logic for selling by implementing selling.

Alternative Design

An alternate approach would be to create a sell action for each class that will be responsible for handing the selling action of different items.

Analysis of Alternate Design

While this design can promote code readability and maintainability but doesn't promote code extensibility violating OCP as existing code may have to be changed to accommodate new sell actions. It also violates the SRP as some actions may have similar or same functions and uses.

Final Design

For the implementation of the requirement we have tried our best to adhere to SOLID and DRY principles.

Single Responsibility Principle (SRP):

Application: The sealable interface defines the contract for sellable items, encapsulating the behaviour related to selling items in a separate interface. This ensures that each class that implements the interface has only one reason to change.

Pros/Cons:

Pros: Simplifies the codebase, making it easier to understand and maintain. Promotes code reusability by allowing different types of sellable items to share common behaviour through the interface.

Cons: May require additional classes/interfaces for each type of sellable item, leading to increased complexity.

Open-Closed Principle (OCP):

Application: The codebase is designed to be open to extension but closed to modification by encapsulating behaviour within classes and relying on polymorphism for extensibility. New types of sellable items can be added without modifying existing code, promoting extensibility and maintainability.

Pros/Cons:

Pros: Increases code flexibility and adaptability to change. Facilitates the addition of new sellable item types without affecting existing code.

Cons: Requires careful design to ensure that classes are properly abstracted and interfaces are well-defined, potentially leading to added complexity.

Liskov Substitution Principle (LSP):

Application: Classes that implement the sellable interface can be substituted for each other without affecting the correctness of the program. This promotes interoperability and allows for polymorphic behaviour when interacting with sellable items.

Pros/Cons:

Pros: Promotes code reuse and simplifies testing and maintenance. Enables developers to create more robust and flexible systems.

Cons: Requires adherence to the interface contract to ensure substitutability, which may introduce constraints on the implementation.

Interface Segregation Principle (ISP):

Application: The sellable interface contains only methods related to selling items, preventing clients from depending on methods they do not use. This promotes code decoupling and flexibility.

Pros/Cons:

Pros: Simplifies the interface for sellable items, making it easier to implement and maintain. Reduces the risk of introducing unintended dependencies between clients and the interface.

Cons: May require additional interfaces or abstractions to accommodate different client needs, potentially increasing the complexity of the design.

Dependency Inversion Principle (DIP):

Application: The design relies on abstractions, such as interfaces, to decouple higher-level modules from lower-level details. This promotes loose coupling and facilitates the replacement of concrete implementations with alternative implementations, promoting flexibility and testability.

Pros/Cons:

Pros: Promotes code reuse, maintainability, and testability. Reduces the impact of changes to lower-level modules on higher-level modules, making the system more resilient to change.

Cons: Requires additional upfront design effort to define abstractions and interfaces, potentially adding complexity to the system architecture.

DRY Principle: The design follows the DRY principle by encapsulating common behaviour related to selling items within the sellable interface. This eliminates code duplication and promotes code reusability by allowing different types of sellable items to share common behaviour through the interface. While the code shares similarities to buyable but sellable provides a completely new and different use.

Conclusion

Overall our design proves to implement sellable items while adhering to SOLID Principles as well as DRY Principles. The code proves to be extensible, maintainable, and promotes separation of concerns, laying the foundation for further enhancements and optimizations.