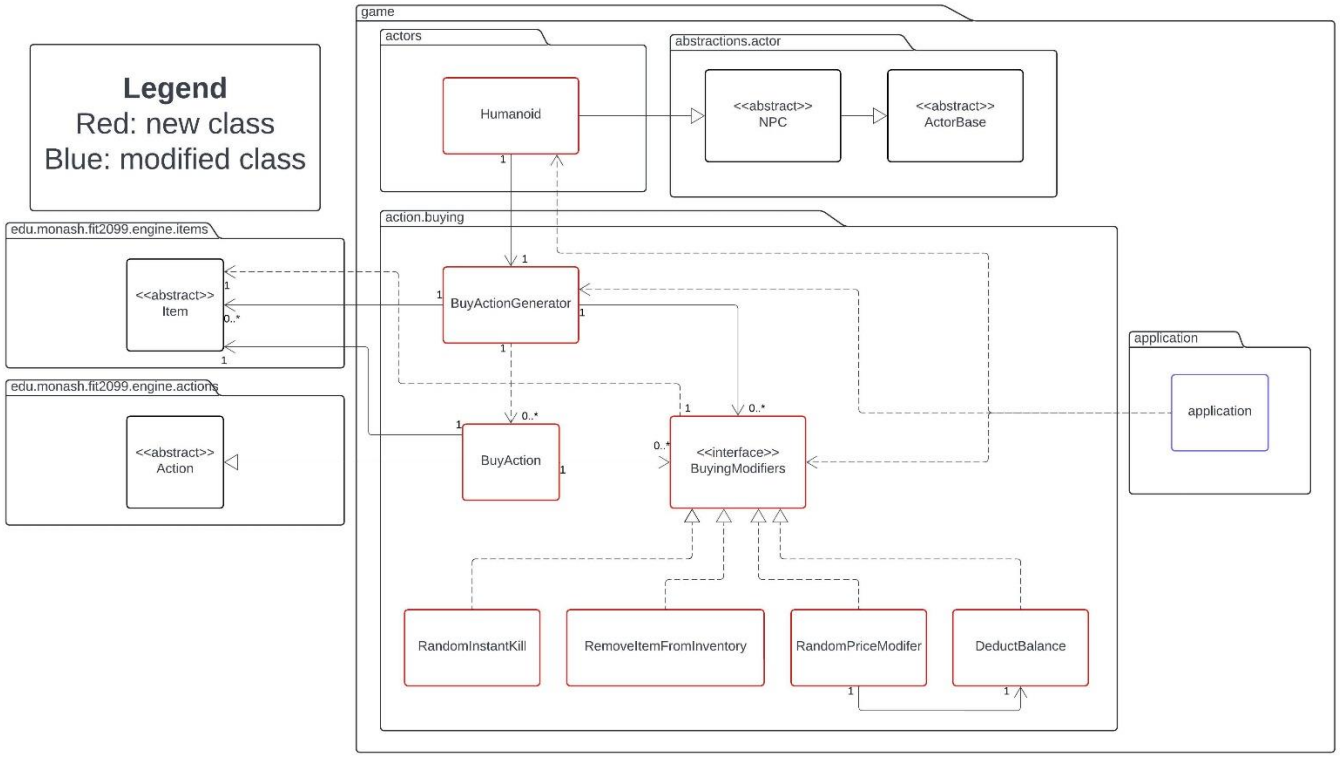
Assignment 3 Req2 Design rationale



The design effectively leverages SOLID and DRY principles to enhance maintainability and extensibility. Each class maintains a single responsibility: 'BuyAction' encapsulates the buying process, 'BuyActionGenerator' creates instances of 'BuyAction', and the modifier classes ('RandomPriceModifier', etc.) encapsulate specific modifications. This adherence to the Single Responsibility Principle (SRP) simplifies understanding and modification.

In adherence to the Liskov Substitution Principle (LSP), the 'BuyActionGenerator' was designed to operate independently of specific actor types. This means it will not have direct dependencies on classes like Humanoid or NPC. Instead, it can be used by any class, that needs to use it.

The design embraces the Open/Closed Principle (OCP) through the 'BuyingModifiers' interface, allowing easy addition of new buying modifiers without altering the core 'BuyAction'. This flexibility is further reinforced by the Dependency Inversion Principle (DIP), where 'BuyAction' depends on the interface rather than specific modifier implementations.

DRY (Don't Repeat Yourself) is achieved by the interface and modifier classes, eliminating duplicate buying modification logic. This improves maintainability and reduces the likelihood of errors during code changes.

There is positional connascence due to the order of modifier application, however, in the given task all modifiers were independent of each other. Moreover, SOLID principles, notably OCP and DIP, mitigate this by localizing changes. DRY further reduces the risk by minimizing code duplication. The use of interfaces like 'BuyingModifiers' helps to mitigate static connascence by providing a level of abstraction. If new modifiers adhere to the interface, changes to their internal implementation won't necessarily ripple through the 'BuyActionGenerator' or 'BuyAction' itself. Moreover, the dynamic connascence between 'BuyAction' and the modifiers is intentional and central to the design's flexibility. This coupling enables the system to adapt to diverse buying scenarios by dynamically applying different modifiers. However, it also underscores the importance of thoroughly testing modifier combinations to ensure predictable and desired outcome.

The code's inherent extensibility is a testament to its adherence to SOLID principles and the decoupling of the 'BuyActionGenerator' from specific actor types. This design choice allows for seamless integration of

new features and scenarios. Not only can different actors, have different market prices and sell different items, but the same class of actors can also sell different items and at different market prices of the same item or different items. This flexibility highlights the system's robustness and adaptability, allowing it to evolve and cater to new requirements while maintaining its core structure and stability.

We leveraged Java's '.class' and '.getClass()' mechanisms within this design, which offers a pragmatic approach to managing item configurations and modifiers. Employing '.class' to construct a price/modifier table provides a centralized, type-based configuration system. It allows for dynamic price and modifier lookup at runtime, enhancing the system's flexibility without scattering configuration details across classes.

Meanwhile, utilizing '.getClass()' on an item instance enables runtime polymorphism when determining and applying modifiers. This ensures that the correct modifiers are associated with the right item types, even as new items are introduced. Encapsulating the configuration and modifier logic not within the dedicated item classes or modules helps uphold the Single Responsibility Principle and minimizes coupling between components, promoting a more maintainable and extensible design.