SOLID Refactoring Report

1. Overview of the Original Code

The original implementation of MonolithicAdventureGame violates multiple SOLID principles by centralizing all game logic into a single class. This results in high coupling, low maintainability, and difficulty in extending game features. The key issues include:

Single Responsibility Principle (SRP) Violation: The class manages player attributes, combat, inventory, game levels, and progression, making it difficult to modify any aspect without affecting others.

Open/Closed Principle (OCP) Violation: Adding new enemies, items, or behaviors requires modifying this class, instead of extending the functionality via new classes.

Liskov Substitution Principle (LSP) Violation: Since all enemy types and items are handled via conditional statements, we cannot easily replace or extend them with new subclasses.

Interface Segregation Principle (ISP) Violation: The class lacks interfaces, forcing any potential subclasses to inherit all methods, even those they may not need.

Dependency Inversion Principle (DIP) Violation: High-level game logic is tightly coupled with specific implementations of items and enemies, making it difficult to substitute different behaviors dynamically.

2. Refactored Design

To address these issues, i refactored the monolithic structure into separate classes, applying SOLID principles:

2.1. Class Structure After Refactoring

- Player – Manages player attributes like health, experience, and inventory.
- Enemy (Interface) – Defines the common behavior of all enemies.

  - Skeleton, Zombie implements from Enemy, each defining unique combat behavior.
  - Item (Interface) – Defines items with specific effects on the player.
  - GoldCoin, HealthElixir, and MagicScroll implement Item.
  - CombatManager – Controls damage and fights.
  - AdventureGame – Ties together player, enemies, items.

3. Key Changes & Applied SOLID Principles

3.1. Single Responsibility Principle (SRP) Compliance

Split responsibilities into multiple classes:

Player handles only player-related attributes.

CombatManager - Responsible for damage processing and fights.

3.2. Open/Closed Principle (OCP) Compliance

New enemy types can be introduced by creating new subclasses of Enemy without modifying existing logic.

New item types can be added by implementing Item.

3.3. Liskov Substitution Principle (LSP) Compliance

Enemy is an abstract class that all enemies inherit from, ensuring polymorphic behavior without breaking existing code.

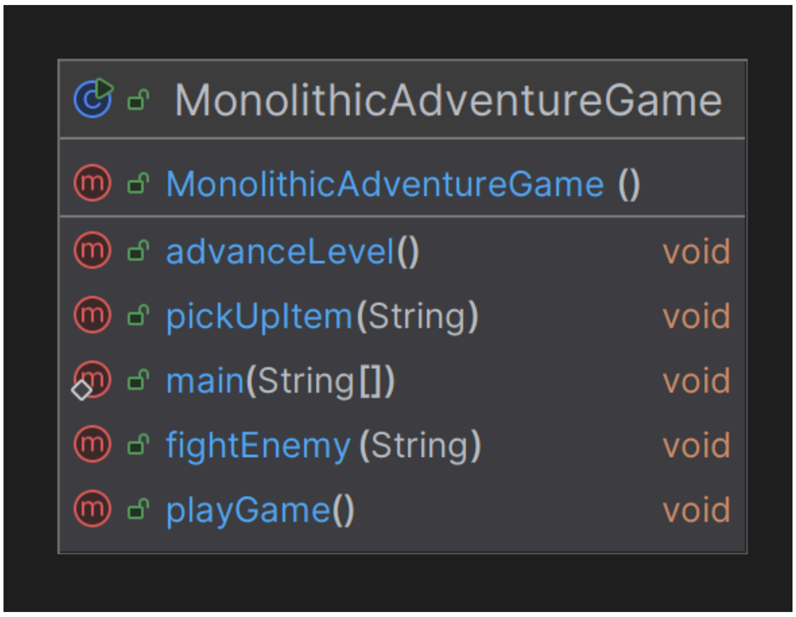
3.4. Interface Segregation Principle (ISP) Compliance

Introduced Item interface to define item behavior, ensuring that each item class implements only the methods relevant to it.

3.5. Dependency Inversion Principle (DIP) Compliance

Game depends on high-level abstractions (Enemy, Item), allowing new implementations to be injected dynamically.

4. UML Diagram (Refactored Structure)



5. Conclusion

Refactoring the monolithic MonolithicAdventureGame into separate classes improves maintainability, readability, and extensibility. Following SOLID principles ensures that new game mechanics can be introduced without modifying core logic, making future expansions easier and reducing technical debt.