**Refactoring Documentation**

### Introduction

This document explains how the monolithic MonolithicAdventureGame was refactored using SOLID principles. The original version of the game contained all functionality in a single class, making it difficult to maintain, extend, and debug. The refactored version follows a structured and modular approach, making it easier to add new enemies, items, and mechanics in the future. The goal of this refactoring was to achieve better code organization, improve readability, and enhance maintainability.

### Issues in the Original Code

The monolithic MonolithicAdventureGame had several issues:

**Violation of the Single Responsibility Principle (SRP):** The class handled player attributes, combat, inventory, game progression, and more.

**Violation of the Open/Closed Principle (OCP):** Adding new enemy types or items required modifying existing code.

**Violation of the Dependency Inversion Principle (DIP):** The high-level game logic was tightly coupled with specific implementations.

**Poor Maintainability:** Any changes in one part of the game could lead to unintended side effects elsewhere.

### Changes Made

To address these issues, we separated the responsibilities into different classes:

1. **Created Separate Classes**
   * **Player Class:** Manages the player's health, experience, and inventory.
   * **Enemy Interface and Implementations:** Defines a general enemy type and allows for easy addition of new enemies (Skeleton, Zombie, Vampire).
   * **CombatManager:** Handles battles between the player and enemies.
   * **LevelManager:** Manages game levels, enemy spawning, and item distribution.
   * **ItemManager:** Deals with item collection and effects on the player.
2. **Applied SOLID Principles**
   * **Single Responsibility Principle (SRP):** Each class is responsible for only one functionality, making the code easier to manage.
   * **Open/Closed Principle (OCP):** New enemies, items, and mechanics can be added without modifying existing code.
   * **Liskov Substitution Principle (LSP):** The Enemy interface ensures that different enemy types can be used interchangeably.
   * **Interface Segregation Principle (ISP):** The Enemy interface defines only the essential methods needed for all enemies.
   * **Dependency Inversion Principle (DIP):** High-level classes rely on abstractions (Enemy interface) instead of specific implementations.

### Conclusion

The refactored game is now modular, flexible, and scalable. By following SOLID principles, we ensured that the code is maintainable and future-proof. This structure allows for easy addition of new features, such as different types of enemies, more complex level progression, and advanced inventory management. The game is now well-structured, making it easier for developers to work with in the future.