The refactoring of the MonolithicAdventureGame into separate classes significantly improves the design by adhering to SOLID principles. Below is an explanation of how each principle is applied.

1) Single Responsibility Principle (SRP)

Before refactoring:

The original game class handled multiple responsibilities: player management, combat logic, enemy spawning, item management, and level progression.

This made the class hard to maintain and extend.

After refactoring:

Each responsibility is separated into different classes:

Player → Manages player attributes and actions.

CombatManager → Handles combat mechanics.

EnemyManager → Creates and stores enemies.

ItemManager → Manages item pickups and effects.

LevelManager → Controls level progression.

ScoreManager (to be added) → Tracks player scores.

Now, each class has a single responsibility, making it easier to modify and test.

2) Open/Closed Principle (OCP)

Before refactoring:

Adding a new enemy or item required modifying the main game class.

Violated OCP because modifications were necessary for extensions.

After refactoring:

Encapsulation of behavior:

New enemies can be added by creating a new class implementing IEnemy.

New items can be introduced by extending IItem without modifying existing code.

Now, the system is open for extension but closed for modification.

3) Liskov Substitution Principle (LSP)

Before refactoring:

All game logic was tightly coupled in a single class, limiting flexibility.

After refactoring:

Abstract interfaces (IEnemy and IItem) allow for polymorphism:

A Skeleton, Zombie, or Vampire can be used interchangeably via IEnemy.

GoldCoin, HealthElixir, and MagicScroll implement IItem correctly.

Now, any new enemy or item can replace existing ones without breaking the system.

4) Interface Segregation Principle (ISP)

Before refactoring:

There was no separation of concerns between items and enemies.

After refactoring:

IEnemy and IItem interfaces ensure that:

Enemies implement combat-related methods.

Items only apply effects to the player.

Now, objects only implement the methods they need, avoiding unnecessary dependencies.

5) Dependency Inversion Principle (DIP)

Before refactoring:

The MonolithicAdventureGame directly instantiated and controlled game objects.

This tightly coupled the system, making it hard to modify individual components.

After refactoring:

Dependence on abstractions (IEnemy and IItem interfaces) rather than concrete implementations.

The CombatManager, EnemyManager, and ItemManager interact through interfaces.

Now, high-level modules do not depend on low-level implementations, making the system more modular.

Key Benefits of the Refactoring:

(1) Code is more modular – Each component has a well-defined responsibility.

(2) Easier to extend – Adding new enemies, items, or mechanics does not require modifying core logic.

(3) Improved testability – Each component can be tested independently.

(4) Better maintainability – The game logic is now clearer and more organized.

(5) Adheres to SOLID principles – Making future development and modifications easier.