SOLID Refactoring Report

1. Introduction

The MonolithicAdventureGame was initially designed as a single, tightly coupled class that handled player management, combat, item interactions, and level progression within a single module. This approach led to low maintainability, poor scalability, and code that was difficult to extend. To improve the design, we refactored the game following the **SOLID** principles, which allowed us to separate concerns, improve modularity, and create a more maintainable architecture.

2. Refactoring Summary

The original codebase contained a single large class responsible for multiple functionalities. We refactored it into smaller, dedicated classes, each with a distinct responsibility. The major refactoring steps included:

- Separating **Player Management** into a dedicated `Player` class.
- Extracting **Combat Logic** into `CombatManager`.
- Creating a separate `Enemy` hierarchy for enemy types.
- Introducing `ItemManager` and subclasses for handling items.
- Managing **Level Progression** with `LevelManager`.
- Handling **Scoring System** separately via `ScoreManager`.
- Keeping `MainGame` as the entry point to coordinate interactions.

3. Application of SOLID Principles

✅ Single Responsibility Principle (SRP)

Each class now has a single, well-defined responsibility:

- **Player** handles only player attributes (health, experience, inventory).
- **CombatManager** processes fights between players and enemies.
- **ItemManager** manages inventory and item interactions.
- **LevelManager** spawns and manages enemy waves.
- **ScoreManager** tracks and updates scores.

✅ Open/Closed Principle (OCP)

- The system allows new enemy or item types to be added **without modifying existing code**.
- Enemy types (`Skeleton`, `Zombie`, `Vampire`) inherit from a common `Enemy` base class, making it easy to extend.
- Items (`GoldCoin`, `HealthElixir`, `MagicScroll`) inherit from `Item`, supporting new items with minimal changes.

✅ Liskov Substitution Principle (LSP)

- Any derived class (`Skeleton`, `Zombie`, `Vampire`) can replace `Enemy` without breaking `CombatManager`.
- New items can be added without changing existing inventory mechanics.

✅ Interface Segregation Principle (ISP)

- Instead of a single **GameEntity** interface, multiple specific interfaces are used:
  - `IAttackable` for entities that can engage in combat.
  - `IDamageable` for entities that can take damage.
  - `IItemInteractable` for objects that can interact with items.

✅ Dependency Inversion Principle (DIP)

- `CombatManager` depends on the abstract `Enemy` class instead of concrete implementations.
- High-level modules (`LevelManager`, `ItemManager`) do not directly depend on low-level modules but interact through abstraction.

4. Why This Architecture?

💠 Maintainability

- Smaller classes make debugging and updating easier.
- Clear separation of concerns allows independent modifications.

💠 Extensibility

- Adding new features (new enemies, items) does not require changing core classes.
- Game behavior can be expanded without modifying the existing codebase.

💠 Testability

- Individual components can be unit-tested independently.
- Mock implementations can replace real classes in testing environments.

5. Conclusion

The refactoring process successfully transformed a tightly coupled monolithic structure into a modular, maintainable system. The new design adheres to the SOLID principles, improving code quality, scalability, and extensibility. Future enhancements, such as additional enemy types, new combat mechanics, and improved inventory management, can be seamlessly integrated into this structure without major code rewrites.

Submitted by: Oralbek Urusbekov
Date: 10.02.2025