**Design Document – Legends of Valor**

**System Architecture**

**Core Components**

1. Board System

* Abstract Board class with specialized implementations (BoardLoV, BoardMaH)
* Tile-based movement with special terrain effects
* Lane-based design (3 lanes separated by inaccessible walls)
* 8x8 grid implementation with specialized spaces

1. Character System

* GameEntity as base abstract class
* Hero hierarchy (Warrior, Sorcerer, Paladin)
* Monster implementation with different types
* Team management (Teamhero, Teammonster)

1. Item System

* Abstract Item base class
* Equipment types: Armor, Weapon, Spell, Potion
* Inventory management
* Market system for trading

**1. Factory Pattern**

**Implementation**: factory/Factory.java, FactoryHero.java, FactoryItem.java, FactoryMonster.java

* **Rationale**: Chosen for its flexibility in object creation and easy extensibility
* **Evaluation**: Successfully implemented, allowing easy addition of new heroes, monsters, and items
* **Achievement**: Met expectations by centralizing object creation and maintaining consistency

**2. Strategy Pattern**

**Implementation**: domain/characters/Heros/

* **Rationale**: Selected to handle different hero types and their unique behaviors
* **Evaluation**: Effectively separated different hero strategies and behaviors
* **Achievement**: Successfully implemented different leveling mechanics and combat strategies

**3. Observer Pattern**

**Implementation**: domain/game/GameLOV.java

* **Rationale**: Chosen to monitor game state changes and terrain effects
* **Evaluation**: Successfully tracked victory conditions and state changes
* **Achievement**: Effectively managed game flow and state transitions

**4. Iterator Pattern**

**Implementation**: domain/characters/Teammonster.java, TeamSetup.java

* **Rationale**: Selected for efficient collection management
* **Evaluation**: Successfully implemented for team management and monster removal
* **Achievement**: Provided clean traversal of collections without exposing implementation

**5. Singleton Pattern**

**Implementation**: util/Inputhandler.java, Outputhandler.java

* **Rationale**: Chosen for consistent I/O operations
* **Evaluation**: Successfully centralized I/O operations
* **Achievement**: Maintained consistent game interface and user interaction

**Key Design Decisions**

**Board System Implementation**

* **Choice**: Abstract Board class with specialized implementations
* **Rationale**: Allows code reuse while maintaining game-specific rules
* **Evaluation**: Successfully supported both game variants
* **Achievement**: Effectively handled different terrain types and their effects

**Character System Design**

* **Choice**: Hierarchical structure with GameEntity as base
* **Rationale**: Provides common attributes while allowing specialized behaviors
* **Evaluation**: Successfully implemented different character types
* **Achievement**: Effectively managed different character behaviors and attributes

**Terrain System Implementation**

* **Choice**: Composition-based approach for terrain effects
* **Rationale**: Provides flexibility in adding new terrain types
* **Evaluation**: Successfully implemented terrain bonuses
* **Achievement**: Met requirements for terrain effects and bonuses

**Post-Implementation Analysis**

**Successful Aspects**

1. Factory pattern implementation for object creation
2. Strategy pattern for hero behaviors
3. Iterator pattern for team management
4. Terrain bonus system implementation

**Areas for Improvement**

1. Monster AI could be more sophisticated
2. Combat system could be more extensible
3. Terrain bonus system could be more modular

**Data Flow**

1. Game Initialization

* Player selects difficulty
* Hero team creation
* Board generation
* Initial monster spawning

1. Game Loop

* Hero turn processing
* Monster turn processing
* Terrain bonus application
* Victory condition checking

1. Combat Resolution

* Attack range validation
* Damage calculation
* Experience and gold distribution
* Character state updates

**Technical Implementation**

1. Input/Output

* ConsoleColor for visual feedback
* Inputhandler for user interaction
* Outputhandler for formatted display
* Loader for resource management

1. Resource Management

* File-based data loading
* Character statistics
* Item properties
* Game configuration

1. State Management

* Round tracking
* Character position management
* Inventory system
* Market interactions

**Conclusion**

The chosen design patterns and implementations successfully met the game requirements while maintaining code modularity and extensibility. The factory and strategy patterns proved particularly effective for character and item management, while the observer pattern effectively handled game state management. This design emphasizes modularity, extensibility, and clear separation of concerns while maintaining compliance with game rules and mechanics.