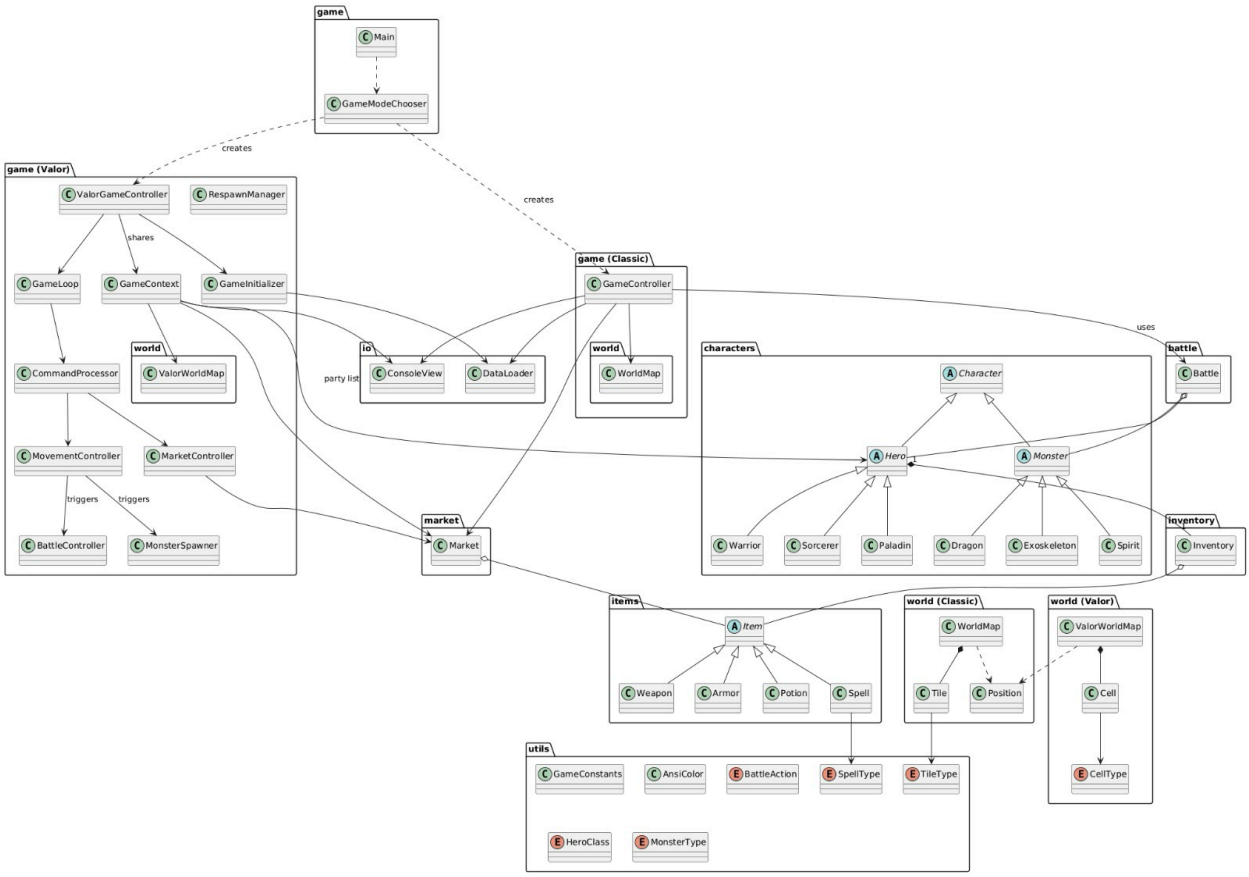
UML



Design Patterns

Model-View-Controller (MVC): Complete separation between game state (Hero, Monster, ValorWorldMap), presentation (ConsoleView), and control logic (specialized controllers). Could swap to GUI without changing game logic.
Strategy Pattern: Hero subclasses (Warrior, Sorcerer, Paladin) implement different favored stat strategies. Monster subclasses apply different combat bonuses. Easy to add new types without modifying base classes.
Factory Method: DataLoader creates game objects from text files. MonsterSpawner.createMonsterOfLevel() instantiates appropriate monster types matching hero levels.
State Pattern: RespawnManager tracks hero respawn states via countdown timers. Cell manages occupancy states (empty/hero/monster/both).
Facade Pattern: ValorGameController provides simple interface (initialize(), run()) hiding complex subsystem interactions. CommandProcessor facades all player actions.
Composite Pattern: Inventory aggregates Items, Hero aggregates Inventory. Treat individual items and collections uniformly.

Scalability

The design enables scalability through controller decomposition and shared abstractions. The modular controller architecture (MovementController, BattleController, MarketController, MonsterSpawner, RespawnManager) allows each component to be developed and optimized independently. GameContext acts as a shared state container, enabling controllers to scale without tight coupling. The Cell-based architecture supports co-occupancy and scales efficiently with board size through localized state management. Abstract base classes (Hero, Monster, Item) use polymorphism so adding new character types or items requires only extending one class without modifying existing game logic. The separation of concerns ensures clear boundaries: ValorGameController handles orchestration, specialized controllers handle specific mechanics, ConsoleView handles all I/O, and DataLoader manages content loading. This means performance optimizations (like spatial partitioning for proximity detection) or UI changes (switching to GUI) only require modifying single components. Finally, the game mode system demonstrates architectural scalability—Classic and Valor modes share core systems (Battle, Hero, Monster, Item) while implementing different controllers and map types, proving the framework accommodates diverse gameplay without code duplication.

Extendability

The design enables extendability through flexible inheritance hierarchies and composition patterns. New game modes can be added by implementing a controller and map type (ValorGameController + ValorWorldMap demonstrate this pattern separate from Classic's GameController + WorldMap). The Hero and Monster abstract classes allow new character types through simple extension—Warrior, Sorcerer, and Paladin show how favored stat bonuses are implemented by overriding template methods. The Cell and CellType system demonstrates extensible terrain—adding new terrain types requires only extending the enum and adding bonus logic to MovementController's switch statement. The Battle class operates on Hero and Monster abstractions, automatically supporting any new character types. Controller composition through GameContext shows how new mechanics (like RespawnManager) can be added without modifying existing controllers. The Strategy pattern in hero classes and Factory pattern in DataLoader and MonsterSpawner enable content scaling—adding heroes/monsters/items only requires data files and extending base classes. The GameModeChooser switch-based menu easily accommodates new game variants. The separation between game state (GameContext), game logic (controllers), and presentation (ConsoleView) allows independent extension of each layer while maintaining framework integrity.

Design Implementation Choices

Controller Decomposition (Valor) vs Monolithic Design (Classic)
Decision: For Legends of Valor, we decomposed game logic into specialized controllers (MovementController, BattleController, MarketController, MonsterSpawner, RespawnManager) coordinated through GameContext, rather than extending the monolithic GameController (825 lines) used in Classic mode.
Reasoning: Classic mode's GameController handles initialization, game loop, movement, markets, and battles in a single class. Adding Valor's mechanics (terrain bonuses, teleportation, recall, proximity battles, periodic spawning, respawn timers) would have pushed it past 1500+ lines. We considered inheritance but Valor doesn't need Classic's random battle system or simple tile-based movement. Decomposition gave each controller a single responsibility and kept them under 300 lines.
Result: Bug isolation improved (terrain bonus fixes only touched MovementController), parallel development enabled, and adding the respawn system required zero changes to existing controllers.

Cell Co-Occupancy vs Battle Removal
Decision: Cell objects allow both a hero and monster to occupy the same space simultaneously, unlike Classic mode which removes monsters from the map during battles.
Reasoning: Valor's proximity-based combat requires both units visible on the board. Removing monsters (like Classic) lost spatial context—which lane was the monster in? Could the hero push forward or was another monster blocking? Co-occupancy enables blocking mechanics (heroes can't move past enemy lines) and lets heroes defend strategic positions (standing on nexus/markets while fighting).
Result: Battles feel spatially grounded. Rendering complexity increased (displaying "H1/M2" for co-occupied cells) but the strategic depth justified it.

Delayed Respawn System
Decision: Fainted heroes respawn after 3 rounds at their nexus with full HP/MP, rather than Classic's immediate post-battle revival at 50%.
Reasoning: Immediate revival made death meaningless in Classic mode—heroes would eventually win through attrition. We needed a penalty without permanent death (too harsh given endless monster waves). The 3-round delay forces remaining heroes to hold positions shorthanded and creates dramatic moments when reinforcements arrive. Implemented via separate RespawnManager to avoid cluttering battle logic.
Result: Death has meaningful consequences. Teams that lose all heroes simultaneously face 3 rounds of vulnerability, encouraging smart positioning. Separate manager made the system easy to tune without touching core controllers.

Group Split

Adithya implemented the 8x8 lane-based board with 3 lanes separated by walls, cell types (Nexus, Obstacle, Plain, Bush, Cave, Koulou), movement validation (blocking, co-occupancy), teleport/recall mechanics, and board rendering.
Saksham extended hero/monster systems from Classic mode, built the terrain buff system (Bush +2 DEX, Cave +2 AGI, Koulou +2 STR), implemented the respawn manager (3-round delay), monster AI (forward movement with lateral bypassing), periodic spawner (difficulty-based intervals), and win/loss detection.
Nanyu designed the hero selection flow (3 heroes assigned to lanes), turn-based UI with action prompts, market system at Nexus/Market tiles, difficulty selection, battle UI integration, and comprehensive board display with ANSI colors.