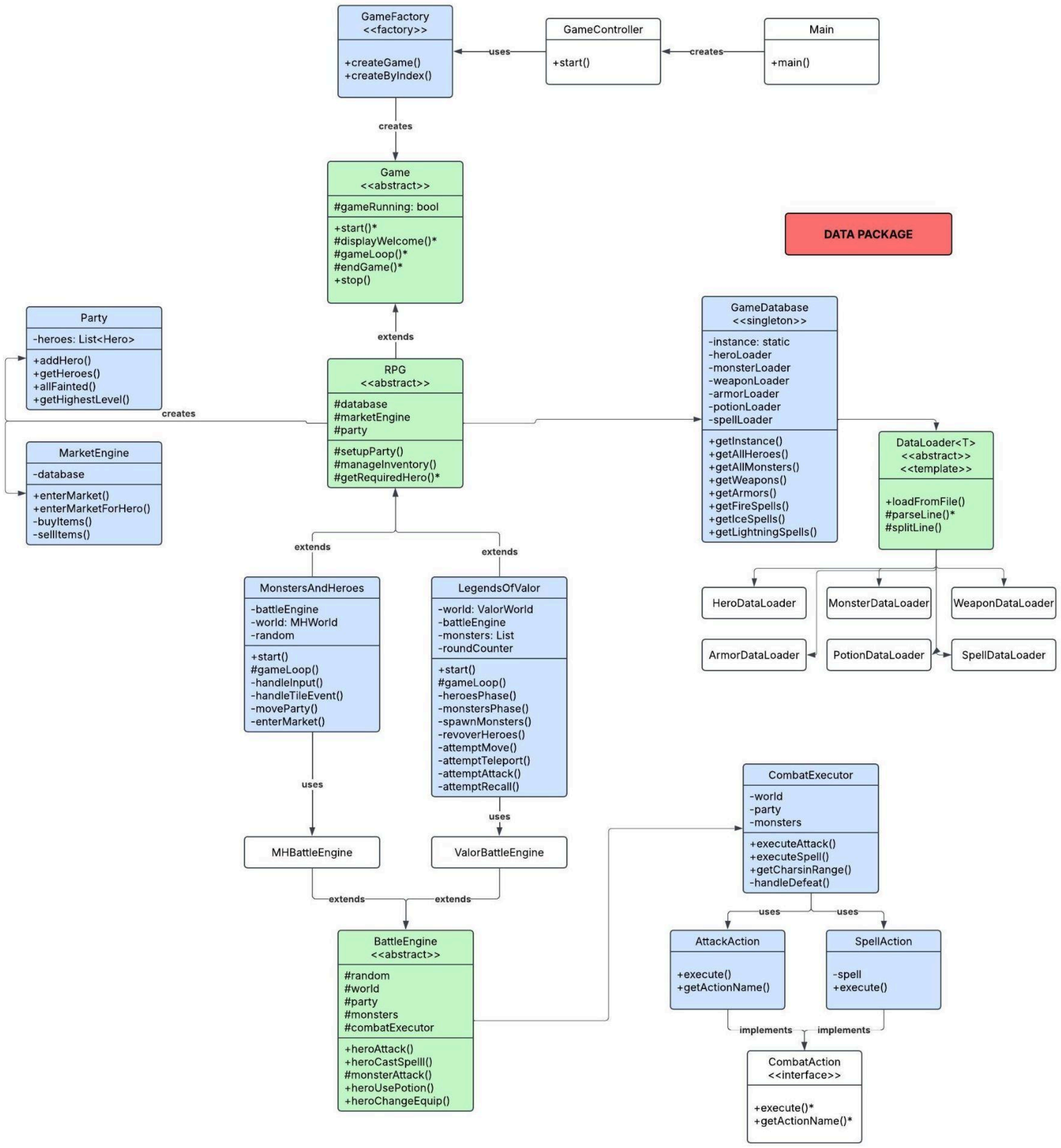
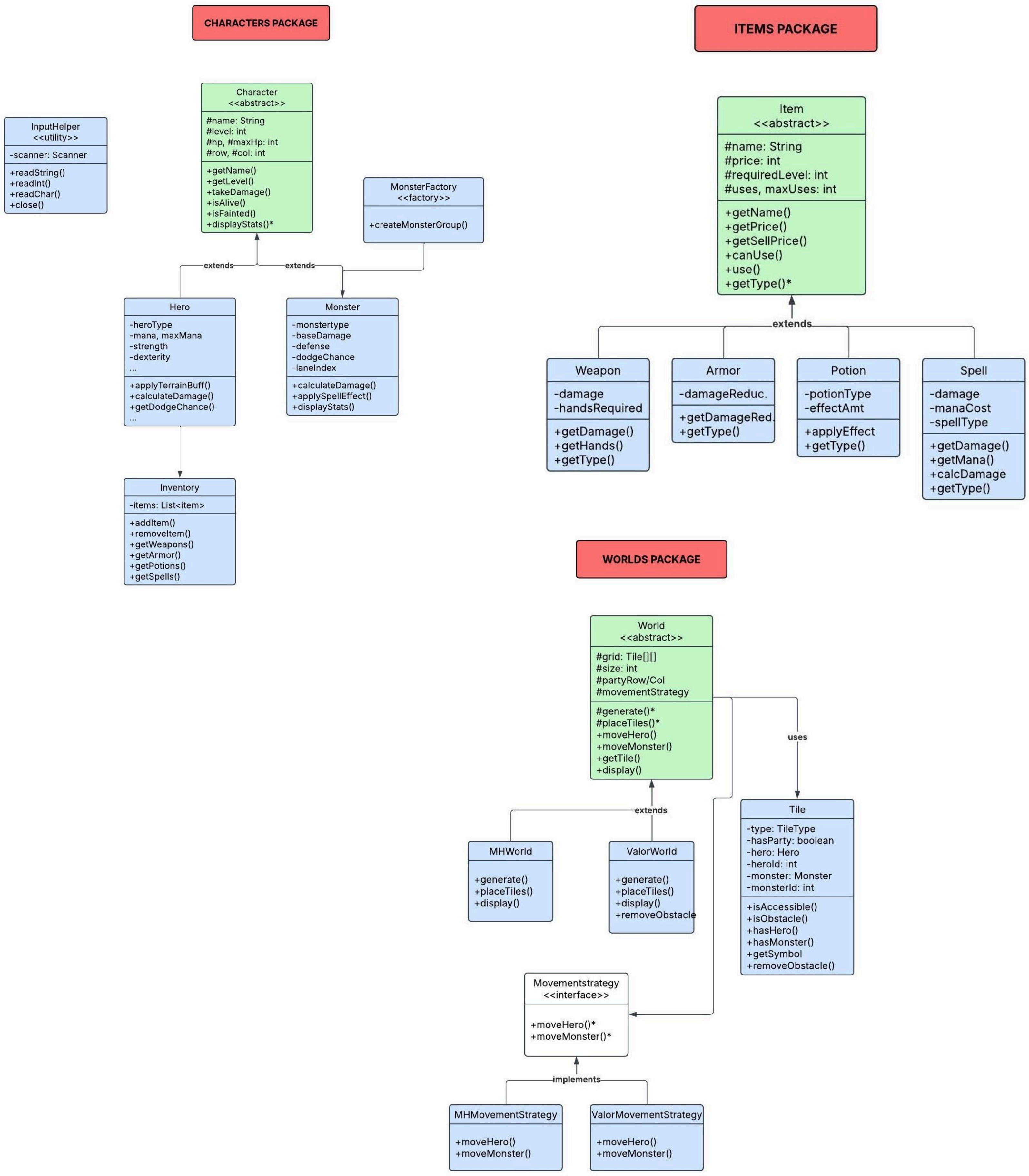
UML


GameFactory <<factory>>
+createGame()
+createByIndex()
uses
GameController
+start()
creates
Main
+main()
creates
Game <<abstract>>
#gameRunning: bool
+start()*
#displayWelcome()*
#gameLoop()*
#endGame()*
+stop()
DATA PACKAGE
Party
-heroes: List<Hero>
+addHero()
+getHeroes()
+allFainted()
+getHighestLevel()
creates
extends
RPG <<abstract>>
#database
#marketEngine
#party
#setupParty()
#manageInventory()
#getRequiredHero()*
MarketEngine
-database
+enterMarket()
+enterMarketForHero()
-buyItems()
-sellItems()
GameDatabase <<singleton>>
-instance: static
-heroLoader
-monsterLoader
-weaponLoader
-armorLoader
-potionLoader
-spellLoader
+getInstance()
+getAllHeroes()
+getAllMonsters()
+getWeapons()
+getArmors()
+getFireSpells()
+getIceSpells()
+getLightningSpells()
DataLoader<T> <<abstract>> <<template>>
+loadFromFile()
#parseLine()*
#splitLine()
extends
extends
MonstersAndHeroes
-battleEngine
-world: MHWorld
-random
+start()
#gameLoop()
-handleInput()
-handleTileEvent()
-moveParty()
-enterMarket()
LegendsOfValor
-world: ValorWorld
-battleEngine
-monsters: List
-roundCounter
+start()
#gameLoop()
-heroesPhase()
-monstersPhase()
-spawnMonsters()
-revoverHeroes()
-attemptMove()
-attemptTeleport()
-attemptAttack()
-attemptRecall()
HeroDataLoader
MonsterDataLoader
WeaponDataLoader
ArmorDataLoader
PotionDataLoader
SpellDataLoader
uses
uses
MHBattleEngine
ValorBattleEngine
CombatExecutor
-world
-party
-monsters
+executeAttack()
+executeSpell()
+getCharsinRange()
-handleDefeat()
extends
extends
uses
uses
BattleEngine <<abstract>>
#random
#world
#party
#monsters
#combatExecutor
+heroAttack()
+heroCastSpelll()
#monsterAttack()
+heroUsePotion()
+heroChangeEquip()
AttackAction
+execute()
+getActionName()
SpellAction
-spell
+execute()
implements
implements
CombatAction <<interface>>
+execute()*
+getActionName()*


CHARACTERS PACKAGE
InputHelper <<utility>>
-scanner: Scanner
+readString()
+readInt()
+readChar()
+close()
Character <<abstract>>
#name: String
#level: int
#hp, #maxHp: int
#row, #col: int
+getName()
+getLevel()
+takeDamage()
+isAlive()
+isFainted()
+displayStats()*
MonsterFactory <<factory>>
+createMonsterGroup()
extends
extends
Hero
-heroType
-mana, maxMana
-strength
-dexterity
...
+applyTerrainBuff()
+calculateDamage()
+getDodgeChance()
...
Monster
-monstertype
-baseDamage
-defense
-dodgeChance
-laneIndex
+calculateDamage()
+applySpellEffect()
+displayStats()
Inventory
-items: List<item>
+addItem()
+removeItem()
+getWeapons()
+getArmor()
+getPotions()
+getSpells()
ITEMS PACKAGE
Item <<abstract>>
#name: String
#price: int
#requiredLevel: int
#uses, maxUses: int
+getName()
+getPrice()
+getSellPrice()
+canUse()
+use()
+getType()*
extends
Weapon
-damage
-handsRequired
+getDamage()
+getHands()
+getType()
Armor
-damageReduc.
+getDamageRed.
+getType()
Potion
-potionType
-effectAmt
+applyEffect
+getType()
Spell
-damage
-manaCost
-spellType
+getDamage()
+getMana()
+calcDamage
+getType()
WORLDS PACKAGE
World <<abstract>>
#grid: Tile[][]
#size: int
#partyRow/Col
#movementStrategy
#generate()*
#placeTiles()*
+moveHero()
+moveMonster()
+getTile()
+display()
uses
extends
MHWorld
+generate()
+placeTiles()
+display()
ValorWorld
+generate()
+placeTiles()
+display()
+removeObstacle
Tile
-type: TileType
-hasParty: boolean
-hero: Hero
-heroId: int
-monster: Monster
-monsterId: int
+isAccessible()
+isObstacle()
+hasHero()
+hasMonster()
+getSymbol
+removeObstacle()
Movementstrategy <<interface>>
+moveHero()*
+moveMonster()*
implements
MHMovementStrategy
+moveHero()
+moveMonster()
ValorMovementStrategy
+moveHero()
+moveMonster()

Design Choices

1. Singleton Pattern (GameDatabase)

How it's used: The GameDatabase class uses a private static instance variable and a private constructor. The getInstance() method provides global access to the single instance that loads and caches all game data (heroes, monsters, items, spells).

Why it was used: The game data from text files (Armory.txt, Dragons.txt, etc.) only needs to be loaded once and should be accessible from anywhere in the application, market, battle engine, game setup. Having multiple instances would waste memory and cause redundant file I/O.

Benefits:

- Ensures consistent data access across all game components
- Eliminates redundant file loading operations
- Provides a centralized point for data management and potential caching
- Lazy initialization delays loading until first needed

2. Factory Pattern (GameFactory and MonsterFactory)

How it's used: GameFactory creates instances of MonstersAndHeroes or LegendsOfValor based on user selection through createGame(GameType). MonsterFactory generates groups of monsters scaled to party size and level through createMonsterGroup().

Why it was used: The creation logic for games involves complex initialization (world generation, engine setup), and monster creation requires level scaling and randomization. Encapsulating this in factories separates "what to create" from "how to create it."

Benefits:

- Decouples client code from concrete class instantiation
- Centralizes object creation logic for easy modification
- Makes adding new game modes straightforward (just add to enum and factory switch)
- Enables consistent monster scaling across the application

3. Template Method Pattern (DataLoader<T>)

How it's used: The abstract DataLoader<T> class defines a loadFromFile() method that handles file opening, line iteration, and error handling. Subclasses (HeroDataLoader, WeaponDataLoader, etc.) only implement the abstract parseLine() method to define how each specific line format is parsed.

Why it was used: All data files follow a similar structure (header + data rows), but each file type has different column formats and object construction. The template method allows sharing the common loading algorithm while varying the parsing logic.

Benefits:

- Eliminates duplicate file-reading boilerplate across 7 data loaders
- Enforces consistent error handling and file processing
- New data types only require implementing one method
- Follows the "Don't Repeat Yourself" (DRY) principle

4. Strategy Pattern (MovementStrategy)

How it's used: The MovementStrategy interface declares moveHero() and moveMonster() methods. MHMovementStrategy implements simple 4-directional party movement, while ValorMovementStrategy implements lane-based movement with monster-blocking rules, terrain buffs, and individual hero/monster positioning.

Why it was used: Movement rules differ drastically between the two games where Monsters & Heroes moves a party token on a simple grid, while Legends of Valor has individual heroes, lane restrictions, and combat-blocking mechanics. Rather than polluting World with conditional logic, the strategy pattern allows swapping movement behavior.

Benefits:

- Each game mode has clean, isolated movement logic
- World class remains focused on grid management, not game-specific rules
- Easy to add new movement variants (e.g., flying heroes, teleportation)
- Promotes Open/Closed Principle, open for extension, closed for modification

5. Facade Pattern (GameDatabase)

How it's used: GameDatabase presents a simple API (getAllHeroes(), getWeapons(), getFireSpells()) while internally managing 7 different DataLoader instances. Clients never interact with loaders directly.

Why it was used: The data loading subsystem is complex: different files, different parsers, different return types. Exposing this complexity would couple game logic to data implementation details.

Benefits:

- Simplifies client code, one import, one class to call
- Hides complexity of multiple specialized loaders
- Easy to change data sources (e.g., from files to database) without affecting clients
- Provides type-safe convenience methods for each data category

Scalability

Polymorphic collections: Characters and items are stored as base types (List<Monster>, List<Item>), enabling the system to handle any number of entities uniformly regardless of specific subclass.

Singleton data caching: GameDatabase loads all game data once and provides centralized access, preventing redundant file operations as the game scales in complexity.
Parameterized world generation: Board size and terrain ratios are configurable constants, allowing worlds of any dimension while maintaining proportional distribution.

Extendability

Inheritance hierarchies: Abstract base classes (Game → RPG, Character, Item, World) provide shared functionality while allowing subclasses to specialize behavior. New game modes or entity types simply extend the appropriate base class.

Strategy pattern for behavior: MovementStrategy interface decouples movement rules from world structure, enabling new movement systems (e.g., flying, teleporting) without modifying existing classes (Open/Closed Principle).

Encapsulated subsystems: Packages (characters, items, combat, world, data) enforce separation of concerns, combat logic doesn't depend on world rendering, and data loading is hidden behind a facade, making each module independently modifiable.

Enum-based type safety: HeroType, MonsterType, SpellType, and TileType enums allow adding new variants by simply extending the enum and handling new cases in switch statements.

Our Initial Implementation Choice

Our team went with Edaad’s implementation as it had a very strong foundational system, with the proper tweaks to the combat mechanics and character attributes we thought made the monster and heroes game really fun to play, for example things like dodge chance starts off at 0 in lower level monsters and then increases as you play to keep the game more interesting. He also had a very strong use of two design patterns, Factory for creating groups of monsters, as well as Singleton for the Game Database. Rest of the team’s implementation were very strong too as well with additional extra credit features too which we tried to bring into Edaad’s M&H implementation. At the end we mostly decided based on very small factors like gameplay mechanics and code readability/organization and were very easily able to adapt to that implementation. After getting feedback for the assignment, we noticed there were a few things that all of us could’ve improved on like using an RPG Game class and the games extend from that class, which we were able to change and improve on in our new implementation. So overall we thought that the choice was right and we were able to adapt and work off of it really well.

How the Group Split up the Work:

The group was able to split up the work pretty evenly by separating and tackling each aspect of the code where Keenan worked on the Game, Danny worked on the World, and Edaad worked on Combat. While working, we were able to help each other out and had overlap in some of our contributions. Keenan and Danny were also able to add extra credit features like Difficulty Selections, ASCII Art/Colors and things like using command h to display a how to play screen. Meanwhile, Edaad worked to build out the extensive UML Diagram and design documentation as well as organize the README.