

Design Document for Monsters and Heroes

Jerin Joseph - U11191999

1. Class Architecture Overview

The game is structured using **object-oriented design principles** with a strong emphasis on **inheritance, abstraction, and polymorphism**. To support multiple types of characters, items, tiles, and maps, several abstract and base classes were created. These enable code reuse, scalability for future game expansions, and separation of responsibilities.

Core Abstract Classes

1. **Entity** (*Common Parent for Hero & Monster*)
Created to unify shared attributes such as name, level, health, and core stats.
This allows the combat system to reference common attributes regardless of whether the object is a hero or a monster.
2. **Tile**
Represents a single location in the world map. Different tile types (e.g., MarketTile, CommonTile) inherit from Tile.
3. **Item**
Abstract parent for all purchasable/usable objects (Potions, Spells, Weapons, Armor).
4. **Spell**
An abstract subclass of Item to represent magic-based attacks. IceSpell, FireSpell, and LightningSpell extend this.
5. **GameMap**
Base class for map generation, movement logic, and rendering.

Each subclass specializes in behavior based on the type of entity, item, or tile.

2. Character System

Hero Class

The **Hero** class extends *Entity* and contains additional attributes:

- Strength
- Dexterity
- Agility
- Mana
- Money
- Experience
- Inventory (Weapons, Armor, Potions, Spells)

Heroes can:

- Attack
- Cast spells
- Use potions
- Equip weapons and armor
- Gain experience and level up

Different hero types (Warrior, Paladin, Sorcerer) override stat scaling to emphasize specific strengths.

Monster Class

The **Monster** class extends *Entity* and contains:

- Base damage
- Defense

- Dodge chance

Monster subclasses (Dragon, Exoskeleton, Spirit) modify these stats based on their type.

3. Inventory & Item System

Item Hierarchy

All items inherit from **Item**.

This allows uniform handling of price, required level, and name.

Weapon

- Damage value
- Required hands (1H or 2H)
- Equip/unequip logic

Armor

- Damage reduction value
- Equip/unequip behavior

Potion

- Affects multiple attributes
- Stored as a string "Strength_Dexterity_Agility" which is parsed during use
- The updated version supports applying effects to **multiple attributes simultaneously**

Spells

Abstract Spell → **FireSpell**, **IceSpell**, **LightningSpell**

Each spell has:

- Base damage
 - Mana cost
 - Secondary effect (e.g., reduce defense, reduce dodge, reduce damage)
-

4. Map & Tile System

Tile Classes

- **CommonTile**: For exploration and random monster encounters
- **MarketTile**: Player can buy or sell items
- **InaccessibleTile**: Movement blocked

Tile classes encapsulate rendering logic, accessibility, and interactions.

Map Generation

GameMap generates:

- Randomized accessible tiles
- Markets placed at random
- Inaccessible tiles to create obstacles

Movement validation ensures:

- Player cannot move to invalid or blocked tiles
 - Random encounters occur only on CommonTiles
-

5. Combat System

The combat system is turn-based and supports:

- Basic attacks
- Spell casting
- Using potions
- Equipping gear mid-battle

Combat flow:

1. Player selects action
2. Enemy responds
3. Stats update dynamically based on spells, potions, or gear changes

Damage calculation considers:

- Weapon damage
 - Monster defense
 - Hero dexterity scaling (for spells)
 - Armor reduction
 - Dodge probability
-

6. Market System

The MarketTile contains:

- A list of sellable items

- Logic for purchase (checking money + level requirement)
- Logic for selling items from inventory

The hero inventory is updated immediately and rendered back to the player.

7. Input Handling & Validation

- All player input is validated for correctness
 - Menu choices, movement keys, and combat choices default safely when incorrect input is detected
 - For name input, a default player name is assigned if user enters an empty string
-

8. Scalability

The system is designed to be **easily extendable**:

Shared Abstract Classes Enable Modular Growth

- Adding new hero types requires only a new subclass of Hero
- Adding new monsters or spells only requires extending Monster or Spell
- New tile types can be added without affecting map logic
- Inventory system easily supports new Item subclasses

Decoupled Logic Improves Maintainability

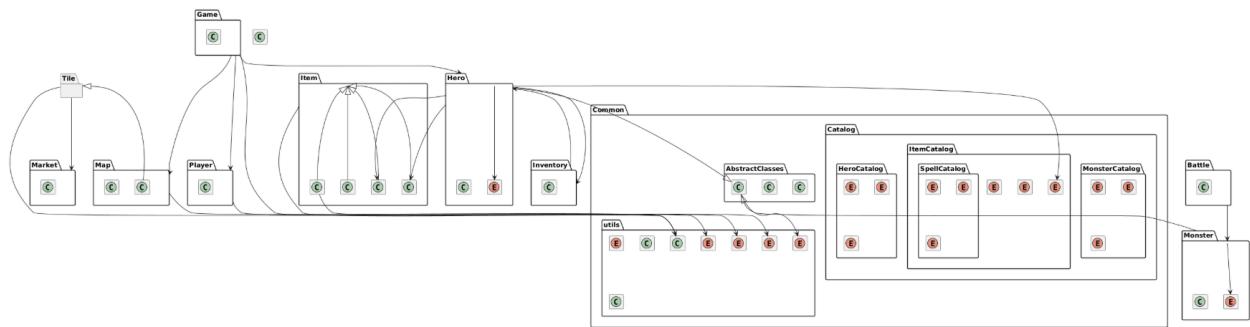
- Combat logic does not depend on hero or monster concrete classes
- Map generation is independent of item or combat classes
- Tile behavior is independent of the entities walking over them

Why the Design Scales Well

- Clear inheritance structure
- Single Responsibility Principle
- Reusable interfaces and methods
- Minimal hard-coding

The project can support new mechanics (quests, bosses, parties, crafting) without refactoring core code.

9. UML Diagram



```
abstract class Entity {  
    - name : String  
    - level : int  
    - hp : int  
    + takeDamage()  
    + isAlive()  
}
```

```
class Hero extends Entity {  
    - strength : int  
    - dexterity : int  
    - agility : int
```

```
- mana : int
- money : int
- experience : int
- inventory : Inventory
+ attack()
+ castSpell()
+ usePotion()
+ equipWeapon()
+ equipArmor()
}

class Warrior extends Hero
class Paladin extends Hero
class Sorcerer extends Hero

class Monster extends Entity {
    - damage : int
    - defense : int
    - dodge : double
}

class Dragon extends Monster
class Exoskeleton extends Monster
class Spirit extends Monster

abstract class Item {
    - name : String
    - price : int
    - levelReq : int
}

class Weapon extends Item {
    - damage : int
    - hands : int
}

class Armor extends Item {
    - reduction : int
```

```
}

class Potion extends Item {
    - attributes : String
    - effectValue : int
}

abstract class Spell extends Item {
    - baseDamage : int
    - manaCost : int
}

class FireSpell extends Spell
class IceSpell extends Spell
class LightningSpell extends Spell

class Inventory {
    - weapons : List<Weapon>
    - armors : List<Armor>
    - potions : List<Potion>
    - spells : List<Spell>
}

abstract class Tile {
    + display()
}

class CommonTile extends Tile
class MarketTile extends Tile
class InaccessibleTile extends Tile

class GameMap {
    - tiles : Tile[][][]
    + movePlayer()
    + printMap()
}
```