# MAC&CHEESE

# PHASE-I-DEMO DOCUMENT

- **Implementation of classes reflecting the class diagrams (all methods not necessarily implemented):**

1. **GameSession Class**

From the class diagram and implementation, we can see matching methods:

- startGame()
- pauseGame()
- resumeGame()
- endGame()
- toggleGameSpeed()
- updateGameState(float deltaTime)
- placeTower(TowerSlot slot, TowerType type)
- removeTower(Tower tower)
- addProjectile(Projectile projectile)
- addEnemy(Enemy enemy)
- removeEnemy(Enemy enemy)
- removeProjectile(Projectile projectile)
- startWave(int waveNumber)
- checkGameOver()
- rewardPlayer(int goldAmount)

2. **Player Class**

Matching methods from diagram to implementation:

- getGold()
- getHitPoints()
- adjustGold(int amount)
- adjustHitPoints(int amount)
- hasEnoughGold(int cost)

3. **Map Class**

Matching methods from diagram to implementation:

- Map(int width, int height)

- Map(String serializedMap)
- validateMap()
- getTileAt(int x, int y)
- setTileAt(int x, int y, TileType type)
- getPath()
- getTowerSlots()
- getStartPoint()
- getEndPoint()

4. **Tile Class**

Matching methods from diagram to implementation:

- Tile(Point position, TileType type)
- getPosition()
- getType()
- isWalkable()
- isPlaceable()

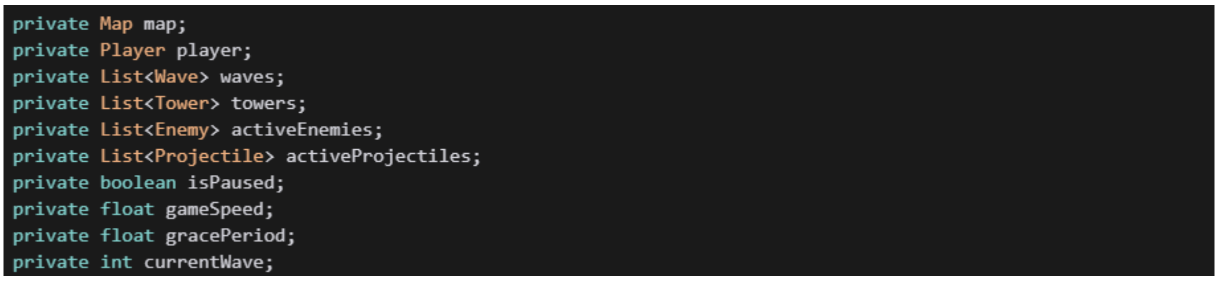
The implementation closely follows the class diagram structure, with all the core methods being implemented as specified. The class relationships and method signatures match what was defined in the class diagrams. For example, the GameSession class maintains the relationships with other classes through its fields (Map, Player, List<Wave>, List<Tower>, etc.) exactly as shown in the class diagram.

## <u>**Detailed Analysis**</u>

### <u>1. GameSession Class</u>

The GameSession class is the central controller of the game, managing all game elements and their interactions.

Attributes Implementation:

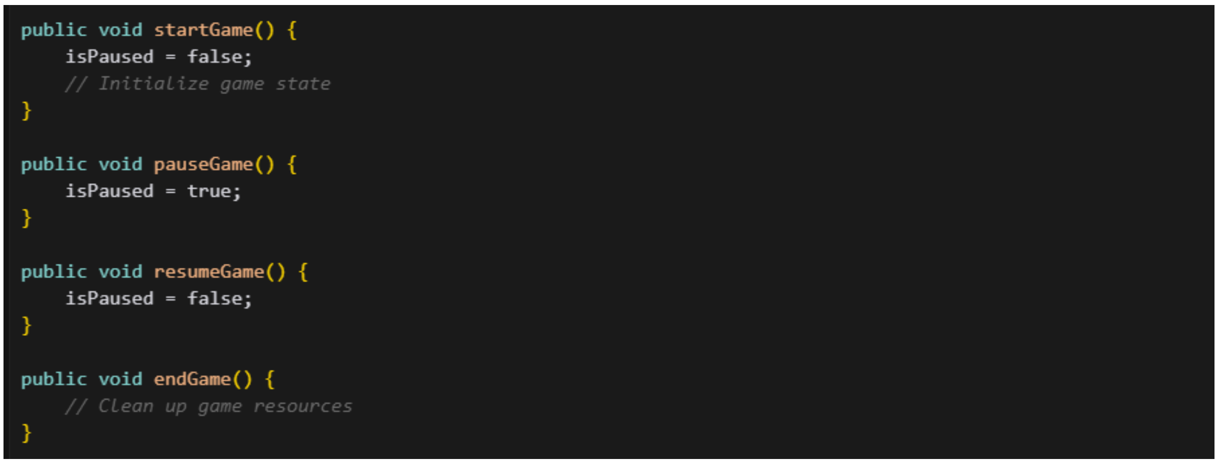


```java
private Map map;
private Player player;
private List<Wave> waves;
private List<Tower> towers;
private List<Enemy> activeEnemies;
private List<Projectile> activeProjectiles;
private boolean isPaused;
private float gameSpeed;
private float gracePeriod;
private int currentWave;
```

These match exactly with the class diagram's specified attributes.

Key Methods Implementation:

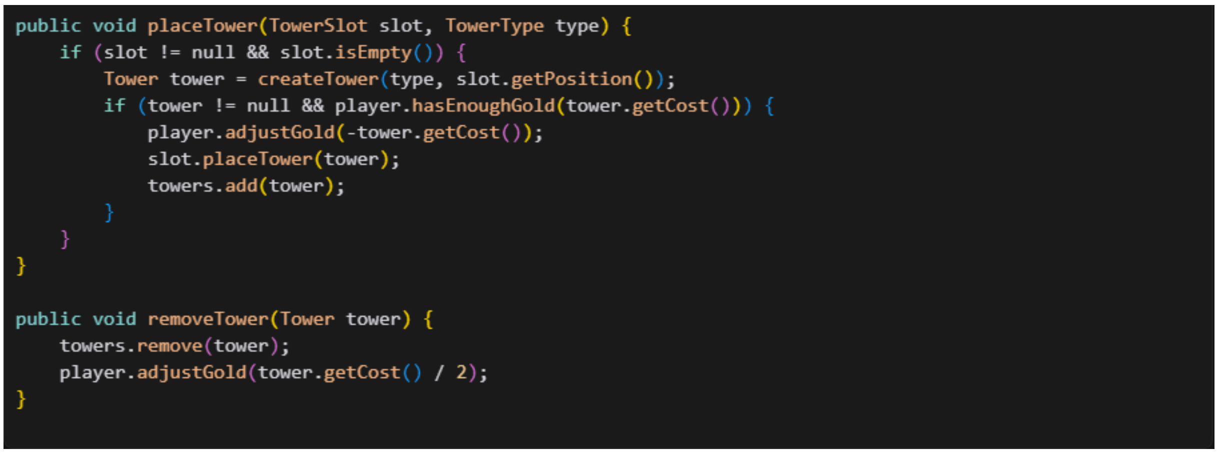
-Game State Management Methods:



```java
public void startGame() {
    isPaused = false;
    // Initialize game state
}

public void pauseGame() {
    isPaused = true;
}

public void resumeGame() {
    isPaused = false;
}

public void endGame() {
    // Clean up game resources
}
```

These methods directly correspond to the class diagram's game control methods.

-Tower Management Methods:

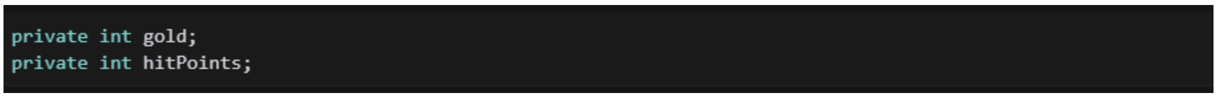


```java
public void placeTower(TowerSlot slot, TowerType type) {
    if (slot != null && slot.isEmpty()) {
        Tower tower = createTower(type, slot.getPosition());
        if (tower != null && player.hasEnoughGold(tower.getCost())) {
            player.adjustGold(-tower.getCost());
            slot.placeTower(tower);
            towers.add(tower);
        }
    }
}

public void removeTower(Tower tower) {
    towers.remove(tower);
    player.adjustGold(tower.getCost() / 2);
}
```

## 2. Player Class

The Player class manages the player's resources and state.

Attributes Implementation:



```java
private int gold;
private int hitPoints;
```

These match the class diagram's specified attributes.

-Key Methods Implementation:

```java
public void adjustGold(int amount) {
    this.gold += amount;
    if (this.gold < 0) {
        this.gold = 0;
    }
}

public void adjustHitPoints(int amount) {
    this.hitPoints += amount;
    if (this.hitPoints < 0) {
        this.hitPoints = 0;
    }
}
```

These methods implement the resource management logic as specified in the class diagram.

**State Checking**:

```java
public boolean hasEnoughGold(int cost) {
    return gold >= cost;
}
```

This method implements the resource validation logic as specified in the class diagram.

## 3. Map Class

The Map class manages the game's map structure and tile layout.

Attributes Implementation:

```java
private String name;
private Tile[][] grid;
private Point startPoint;
private Point endPoint;
private Path path;
private List<TowerSlot> towerSlots;
private int width;
private int height;
```

These match the class diagram's specified attributes.

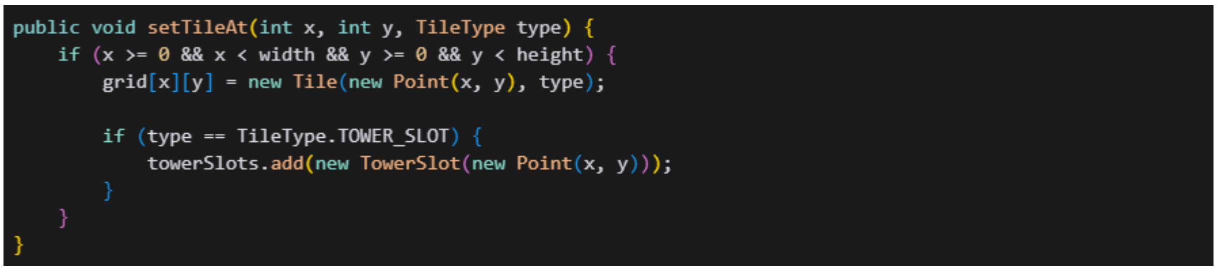
-Key Methods Implementation:

Map Initialization:

```java
public Map(int width, int height) {
    this.width = width;
    this.height = height;
    this.grid = new Tile[width][height];
    this.towerSlots = new ArrayList<>();
    initializeGrid();
}
```

This constructor implements the map creation logic as specified in the class diagram.

Tile Management:



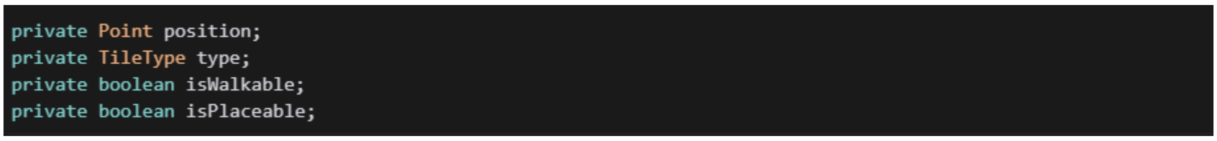
```java
public void setTileAt(int x, int y, TileType type) {
    if (x >= 0 && x < width && y >= 0 && y < height) {
        grid[x][y] = new Tile(new Point(x, y), type);

        if (type == TileType.TOWER_SLOT) {
            towerSlots.add(new TowerSlot(new Point(x, y)));
        }
    }
}
```

This method implements the tile modification logic as specified in the class diagram.

## 4.Tile Class

The Tile class represents individual map tiles.

Attributes Implementation:

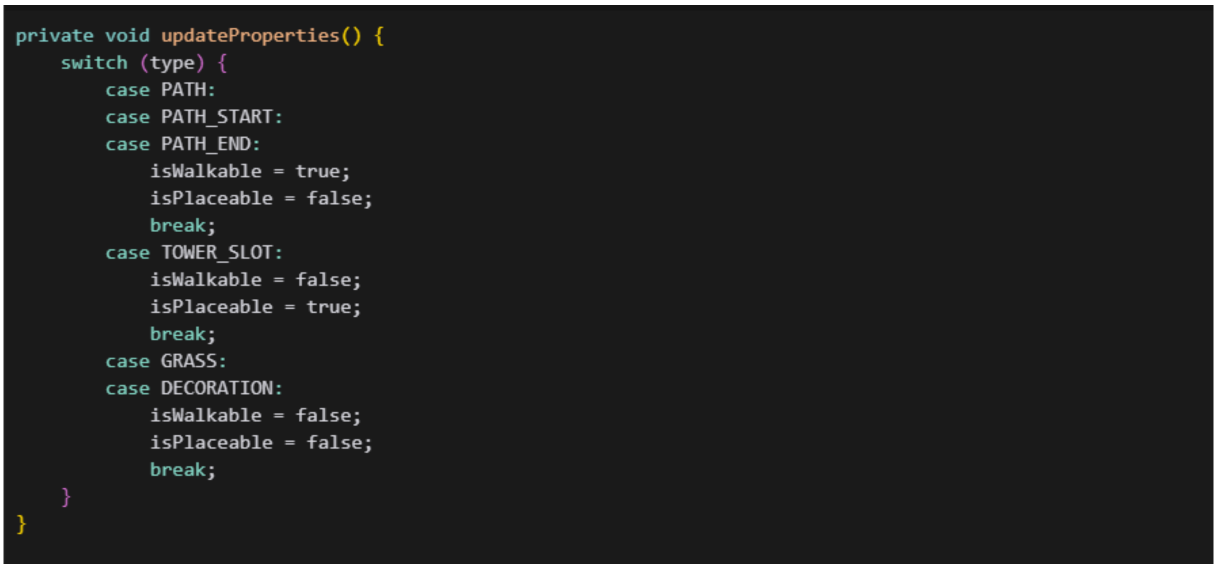


```java
private Point position;
private TileType type;
private boolean isWalkable;
private boolean isPlaceable;
```

These match the class diagram's specified attributes.

-Key Methods Implementation:

Tile Properties



```java
private void updateProperties() {
    switch (type) {
        case PATH:
        case PATH_START:
        case PATH_END:
            isWalkable = true;
            isPlaceable = false;
            break;
        case TOWER_SLOT:
            isWalkable = false;
            isPlaceable = true;
            break;
        case GRASS:
        case DECORATION:
            isWalkable = false;
            isPlaceable = false;
            break;
    }
}
```

This method implements the tile property logic based on tile type as specified in the class diagram.

**Class Relationships**

The implementation maintains the relationships specified in the class diagram:

1. **GameSession Relationships**:

- Has-a relationship with Map
- Has-a relationship with Player
- Has-many relationship with Wave, Tower, Enemy, and Projectile

2. **Map Relationships**:

- Has-many relationship with Tile
- Has-many relationship with TowerSlot
- Has-a relationship with Path

3. **Tile Relationships**:

- Has-a relationship with Point
- Has-a relationship with TileType

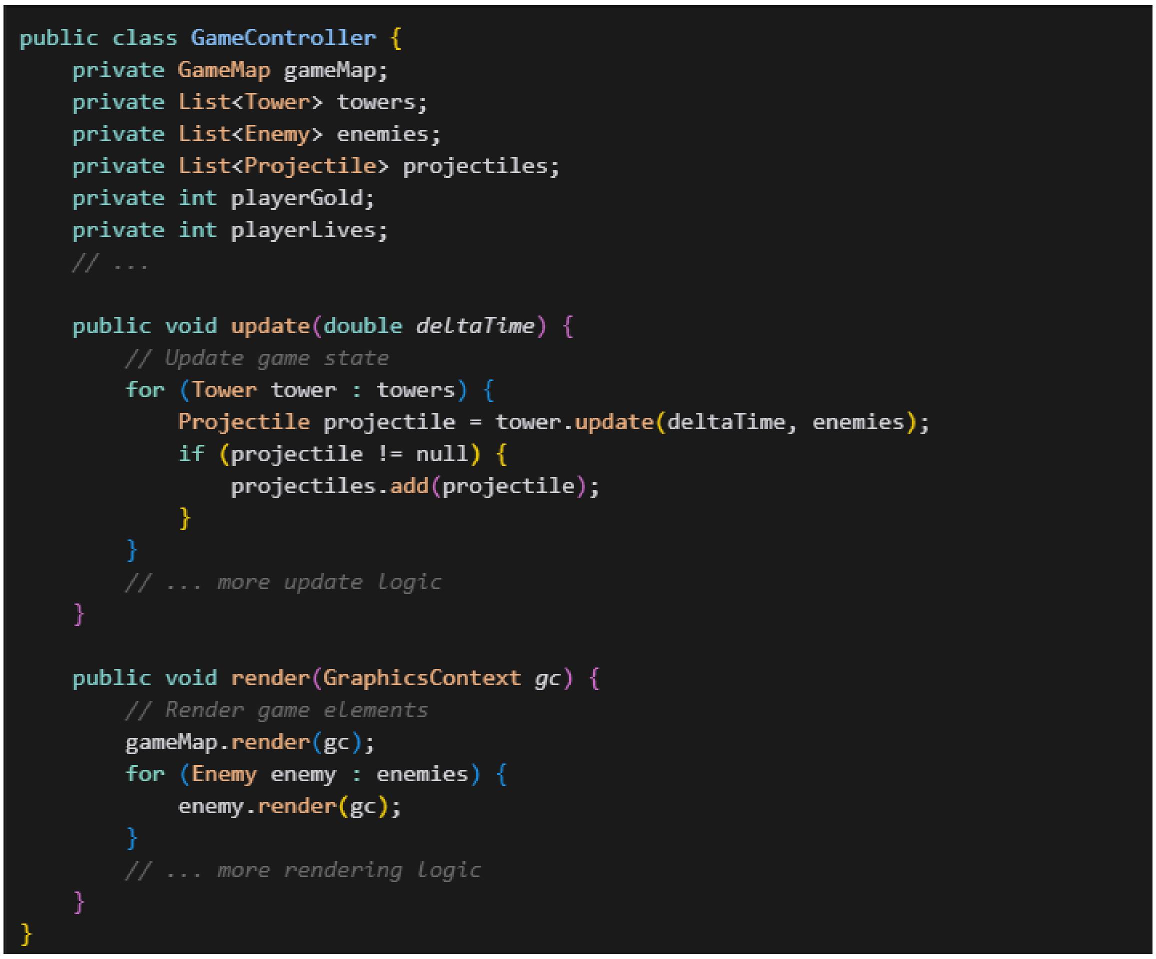
This implementation closely follows the class diagram specifications, maintaining all the required relationships, attributes, and methods while providing the necessary functionality for the tower defense game.

- **Design Patterns Implementation Analysis**

1. <u>Controller Pattern</u>

The GameController class implements the Controller pattern, acting as the central coordinator between the model and view components.

-Key Implementation Details:



```java
public class GameController {
    private GameMap gameMap;
    private List<Tower> towers;
    private List<Enemy> enemies;
    private List<Projectile> projectiles;
    private int playerGold;
    private int playerLives;
    // ...

    public void update(double deltaTime) {
        // Update game state
        for (Tower tower : towers) {
            Projectile projectile = tower.update(deltaTime, enemies);
            if (projectile != null) {
                projectiles.add(projectile);
            }
        }
        // ... more update logic
    }

    public void render(GraphicsContext gc) {
        // Render game elements
        gameMap.render(gc);
        for (Enemy enemy : enemies) {
            enemy.render(gc);
        }
        // ... more rendering logic
    }
}
```

**Controller Pattern Responsibilities:**

1. **Game State Management**:
- Manages game loop
- Controls game speed
- Handles wave progression
- Manages player resources

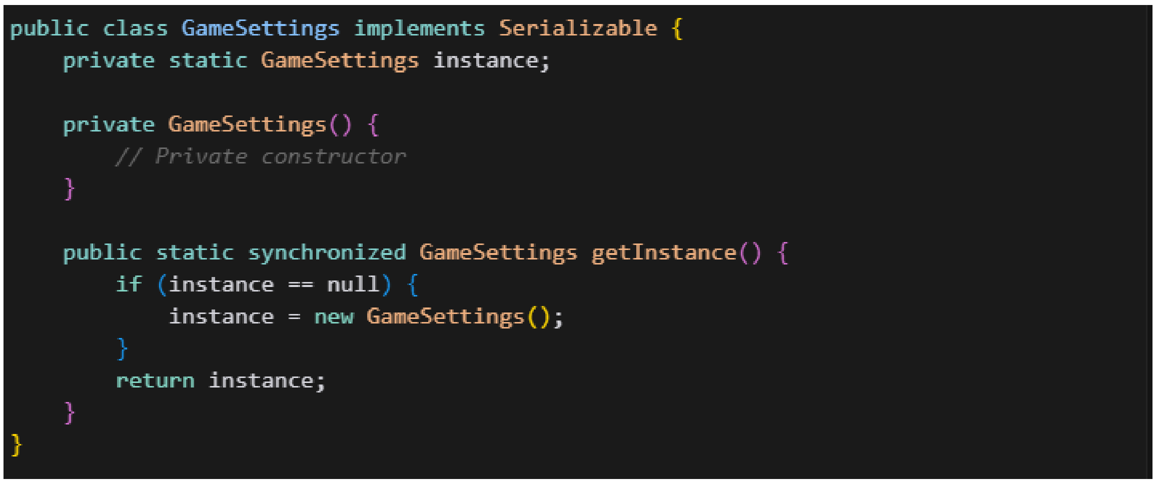
2. **Entity Coordination**:
- Updates all game entities
- Handles entity interactions
- Manages entity lifecycle

3. **Input Handling**:
- Tower placement
- Game speed control
- Wave management

2. Singleton Pattern

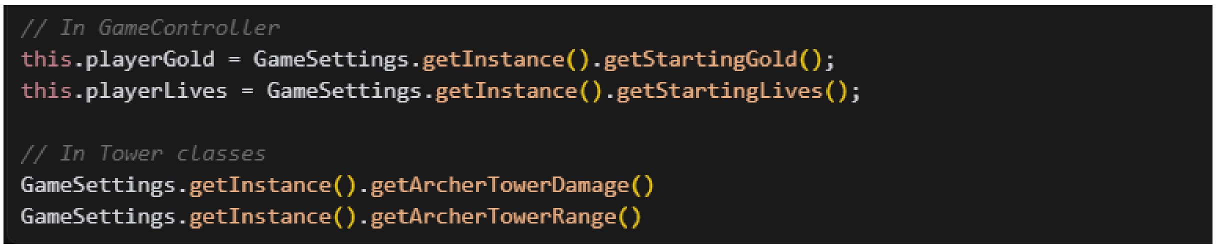
The GameSettings class implements the Singleton pattern to provide global access to game settings.

-Implementation:



```java
public class GameSettings implements Serializable {
    private static GameSettings instance;

    private GameSettings() {
        // Private constructor
    }

    public static synchronized GameSettings getInstance() {
        if (instance == null) {
            instance = new GameSettings();
        }
        return instance;
    }
}
```

-Usage Examples:



```java
// In GameController
this.playerGold = GameSettings.getInstance().getStartingGold();
this.playerLives = GameSettings.getInstance().getStartingLives();

// In Tower classes
GameSettings.getInstance().getArcherTowerDamage()
GameSettings.getInstance().getArcherTowerRange()
```

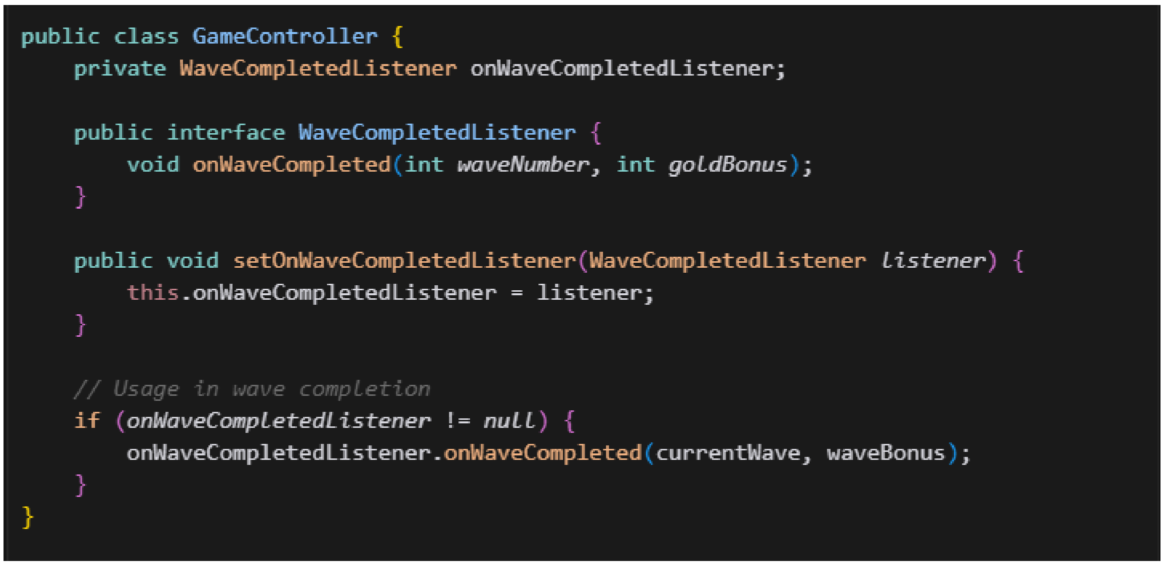
Benefits of Singleton Implementation:

1. Centralized Configuration:
- Single source of truth for game settings
- Consistent values across the application
- Easy to modify game parameters
2. Resource Management:
- Efficient memory usage
- Thread-safe access to settings
- Persistent settings through serialization

3. <u>Observer Pattern</u>

The GameController implements the Observer pattern through the WaveCompletedListener interface.

-Implementation:



```java
public class GameController {
    private WaveCompletedListener onWaveCompletedListener;

    public interface WaveCompletedListener {
        void onWaveCompleted(int waveNumber, int goldBonus);
    }

    public void setOnWaveCompletedListener(WaveCompletedListener listener) {
        this.onWaveCompletedListener = listener;
    }

    // Usage in wave completion
    if (onWaveCompletedListener != null) {
        onWaveCompletedListener.onWaveCompleted(currentWave, waveBonus);
    }
}
```

Observer Pattern Benefits:

1. Loose Coupling:

- UI components can subscribe to game events
- Game logic remains independent of UI
- Easy to add new observers

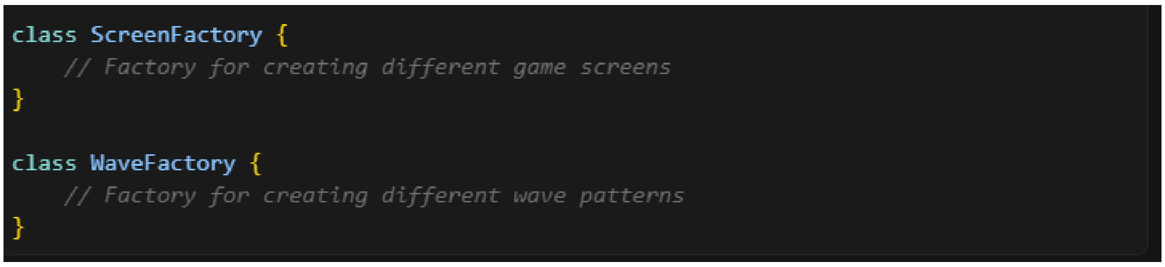
2. Event Handling:

- Decentralized event management
- Clean separation of concerns
- Flexible notification system

4. Factory Pattern (Referenced in Class Diagrams)

While not fully implemented in the current code, the class diagrams show the intention to use the Factory pattern through ScreenFactory and WaveFactory classes.

-Planned Implementation:



```
class ScreenFactory {
    // Factory for creating different game screens
}

class WaveFactory {
    // Factory for creating different wave patterns
}
```

Factory Pattern Benefits:

1. Object Creation:

- Centralized creation logic
- Encapsulated instantiation
- Easy to extend with new types

2. Flexibility:

- Easy to add new screen types
- Simplified wave generation
- Maintainable code structure

***Design Pattern Integration:***

The patterns work together to create a well-structured game:

1. Controller-Singleton Integration:

- Controller uses Singleton for settings
- Consistent game parameters
- Centralized configuration

2. Controller-Observer Integration:

- Controller notifies observers of events
- UI updates based on game state
- Clean separation of concerns

3. Future Factory Integration:

- Will provide structured object creation
- Will enhance code maintainability
- Will support game expansion

This implementation demonstrates a solid understanding of design patterns and their practical application in game development.

- **Correct implementation of the Model-View Separation principle:**

1. Clear Package Structure

The codebase demonstrates clear separation through its package structure:

```
com.ku.towerdefense/
├── model/          # Data and business logic
├── ui/             # User interface components
├── controller/     # Mediators between model and view
└── util/           # Utility classes
```

2. Model Layer Implementation

<u>Core Model Classes:</u>

```java
// GameSession.java - Core game state
public class GameSession {
    private Map map;
    private Player player;
    private List<Wave> waves;
    private List<Tower> towers;
    // ... pure game logic, no UI dependencies
}

// Player.java - Player data model
public class Player {
    private int gold;
    private int hitPoints;
    // ... pure data and business logic
}

// Map.java - Game map model
public class Map {
    private Tile[][] grid;
    private Point startPoint;
    private Point endPoint;
    // ... pure map data and logic
}
```

Model Layer Characteristics:

1. No UI Dependencies:

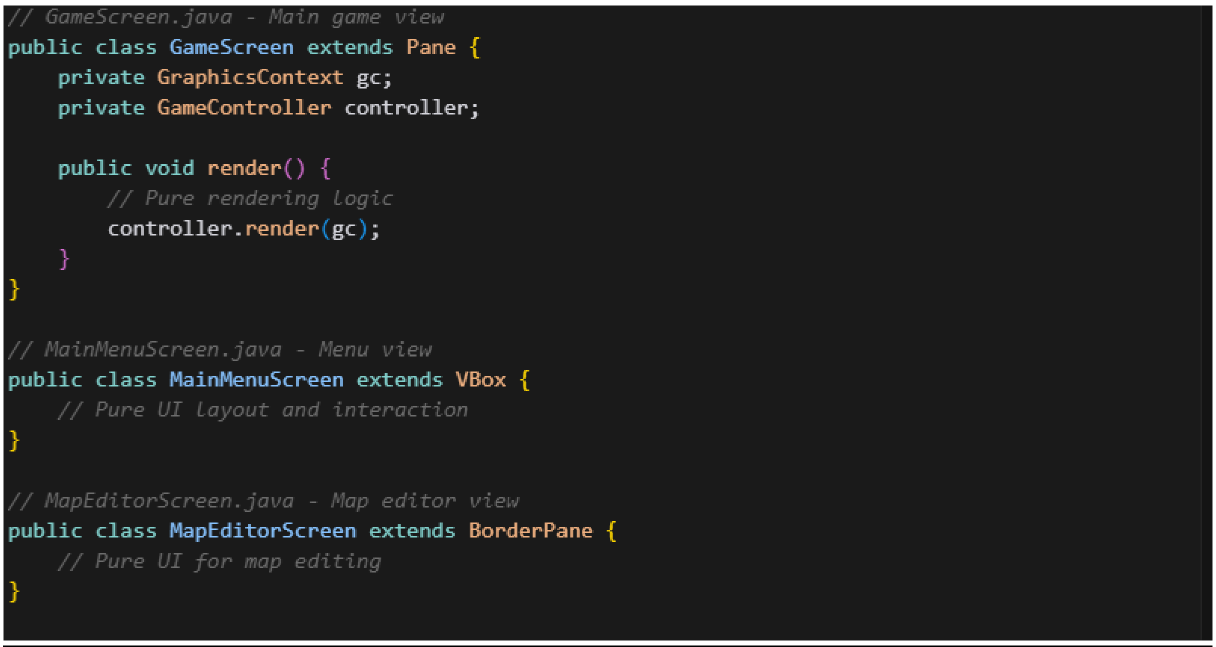
- Models contain no JavaFX or UI-related code
- Pure business logic and data structures
- No direct rendering or user interaction

2. Data-Focused:

- Represents game state
- Handles game rules
- Manages entity relationships

3. View Layer Implementation

UI Components



```java
// GameScreen.java - Main game view
public class GameScreen extends Pane {
    private GraphicsContext gc;
    private GameController controller;

    public void render() {
        // Pure rendering logic
        controller.render(gc);
    }
}

// MainMenuScreen.java - Menu view
public class MainMenuScreen extends VBox {
    // Pure UI layout and interaction
}

// MapEditorScreen.java - Map editor view
public class MapEditorScreen extends BorderPane {
    // Pure UI for map editing
}
```

View Layer Characteristics:

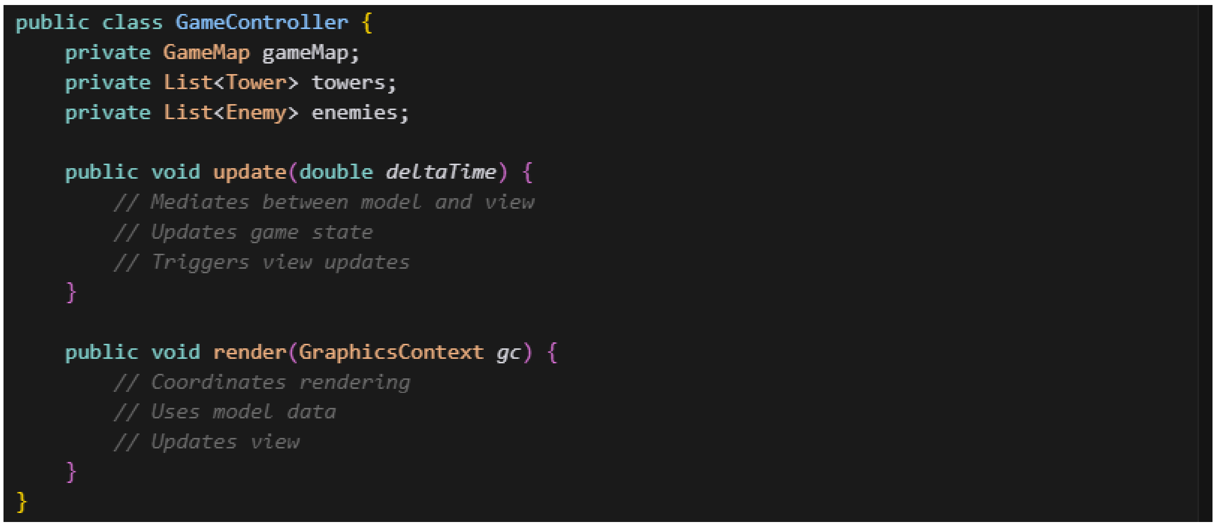
1. UI-Focused:

- Handles rendering
- Manages user input
- Controls visual presentation

2. No Business Logic:

- Delegates game logic to controller
- Focuses on presentation
- Uses controller for state changes

4. Controller Layer Implementation



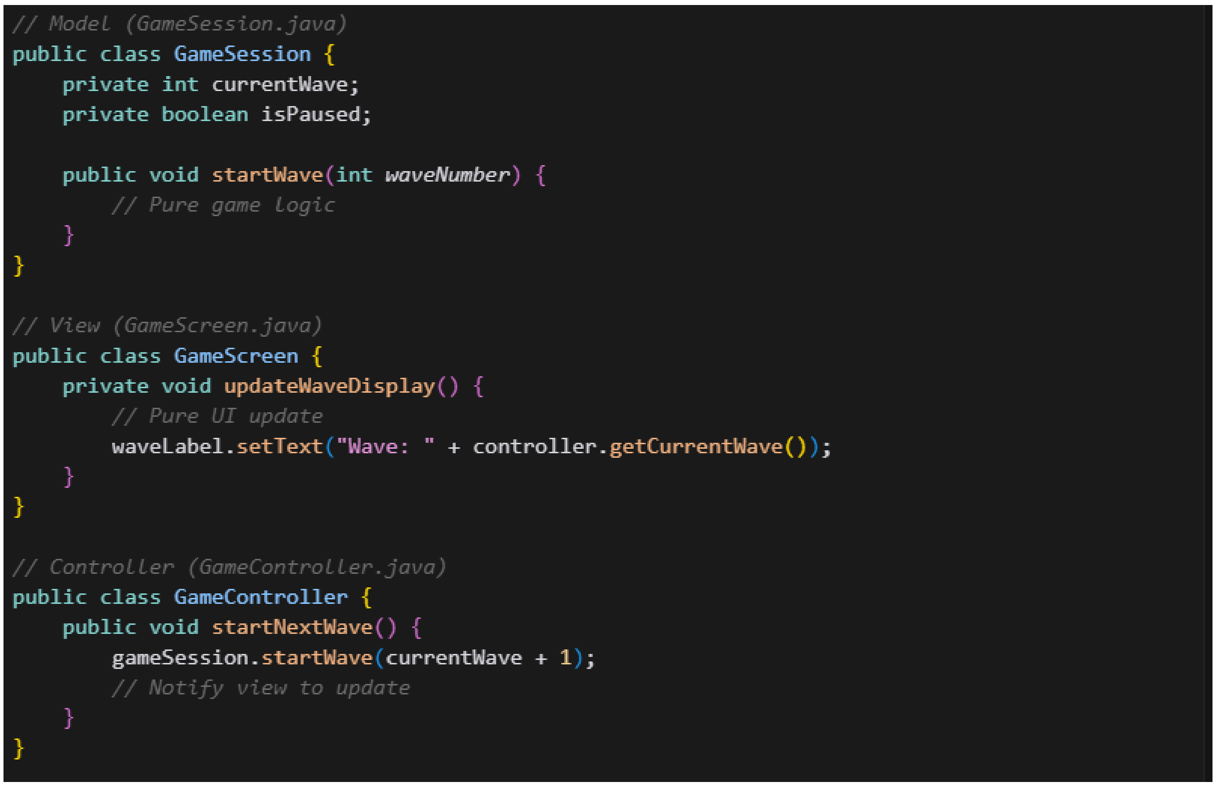
```java
public class GameController {
    private GameMap gameMap;
    private List<Tower> towers;
    private List<Enemy> enemies;

    public void update(double deltaTime) {
        // Mediates between model and view
        // Updates game state
        // Triggers view updates
    }

    public void render(GraphicsContext gc) {
        // Coordinates rendering
        // Uses model data
        // Updates view
    }
}
```

Controller Layer Characteristics:

1. Mediation:
   - Coordinates model and view
   - Handles user input
   - Updates game state

2. Separation Enforcement:
   - Prevents direct model-view communication
   - Manages data flow
   - Controls update cycles

**Separation Example:**



```java
// Model (GameSession.java)
public class GameSession {
    private int currentWave;
    private boolean isPaused;

    public void startWave(int waveNumber) {
        // Pure game logic
    }
}

// View (GameScreen.java)
public class GameScreen {
    private void updateWaveDisplay() {
        // Pure UI update
        waveLabel.setText("Wave: " + controller.getCurrentWave());
    }
}

// Controller (GameController.java)
public class GameController {
    public void startNextWave() {
        gameSession.startWave(currentWave + 1);
        // Notify view to update
    }
}
```

Benefits of the Implementation

1. Maintainability:
- Clear separation of concerns
- Easy to modify UI without affecting game logic
- Simple to update game rules without touching UI

2. Testability:
- Models can be tested independently
- UI can be tested separately
- Controllers can be unit tested

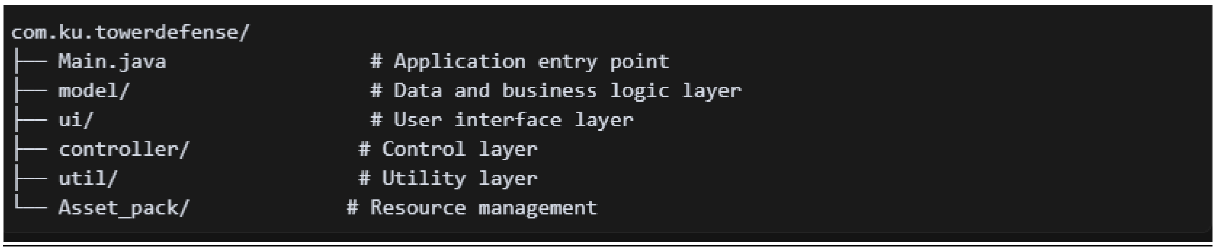
3. Flexibility:
- Easy to change UI framework
- Simple to modify game rules
- Can add new features without breaking existing code

4. Scalability:
- Easy to add new views
- Simple to extend game logic
- Clear structure for new features

This implementation demonstrates a strong adherence to the Model-View separation principle, with clear boundaries between layers and proper use of controllers to mediate between them.

- **Demonstration of logical architecture - layers/packages/subsystems**

1.High-Level Architecture

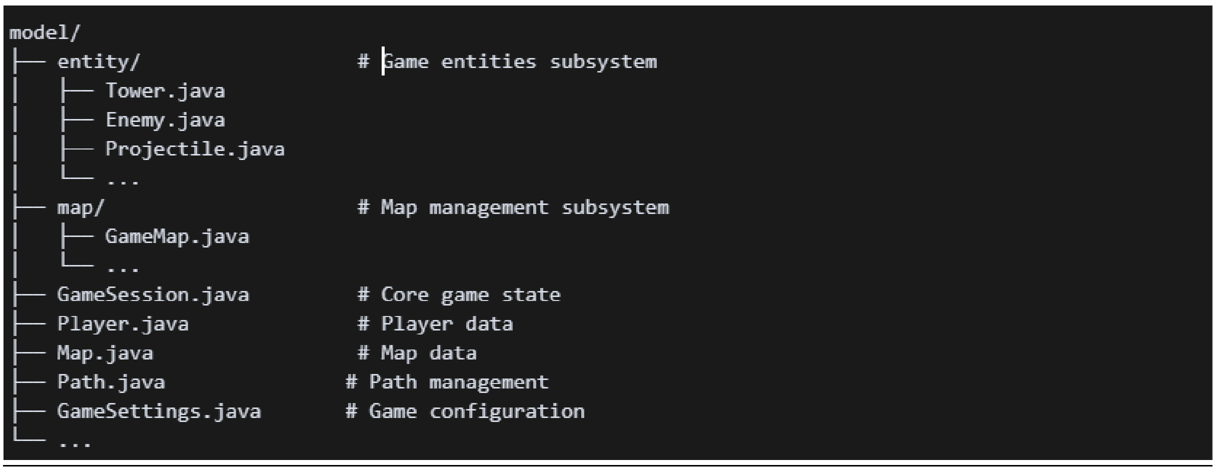


```
com.ku.towerdefense/
├── Main.java                 # Application entry point
├── model/                    # Data and business logic layer
├── ui/                       # User interface layer
├── controller/              # Control layer
├── util/                    # Utility layer
└── Asset_pack/             # Resource management
```

2. Layer-by-Layer Analysis

2.1 Model Layer (model/):

The model layer is further subdivided into logical subsystems:



```
model/
├── entity/                  # Game entities subsystem
│   ├── Tower.java
│   ├── Enemy.java
│   ├── Projectile.java
│   └── ...
├── map/                     # Map management subsystem
│   ├── GameMap.java
│   └── ...
├── GameSession.java         # Core game state
├── Player.java              # Player data
├── Map.java                 # Map data
├── Path.java               # Path management
├── GameSettings.java       # Game configuration
└── ...
```

Model Layer Responsibilities:

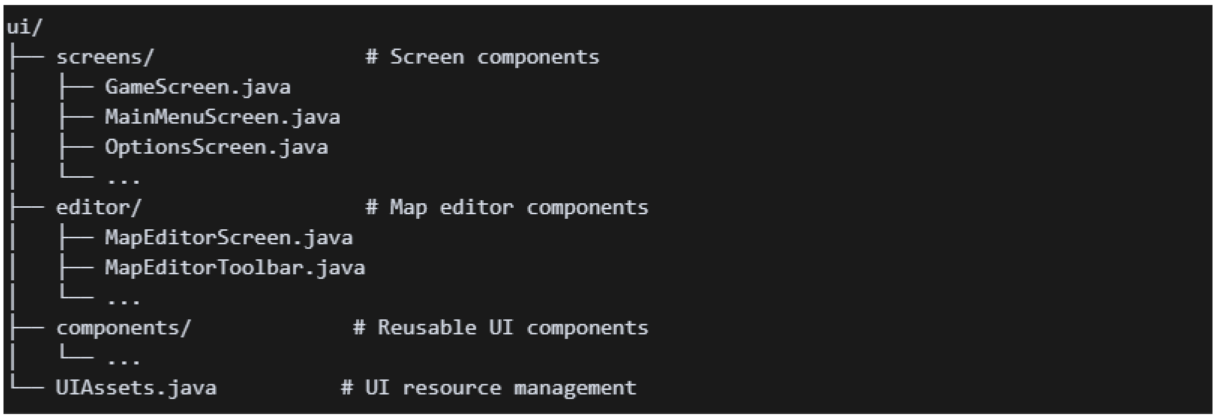
1. Entity Management:
- Game objects (Towers, Enemies, Projectiles)
- Entity behaviors and properties
- Entity relationships

2. Map Management:
- Map structure
- Tile management
- Path finding

3. Game State:
- Player data
- Game session management
- Game settings

### 2.2 UI Layer (ui/):

The UI layer is organized by screen and component:

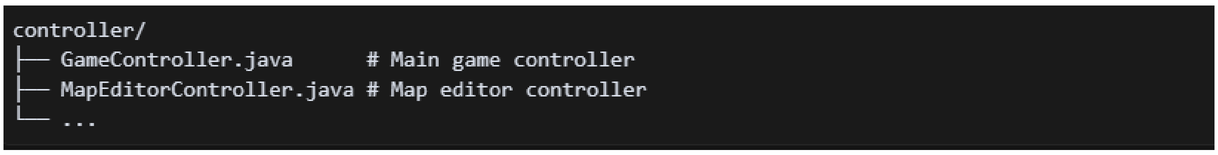


```
ui/
├── screens/                 # Screen components
│   ├── GameScreen.java
│   ├── MainMenuScreen.java
│   ├── OptionsScreen.java
│   └── ...
├── editor/                  # Map editor components
│   ├── MapEditorScreen.java
│   ├── MapEditorToolbar.java
│   └── ...
├── components/             # Reusable UI components
│   └── ...
└── UIAssets.java          # UI resource management
```

UI Layer Responsibilities:

1. Screen Management:
   - Game screen
   - Menu screens
   - Option screens
2. Editor Components:
   - Map editor
   - Tile palette
   - Editor tools
3. UI Resources:
   - Asset management
   - UI styling
   - Resource loading

### 2.3 Controller Layer (controller/):

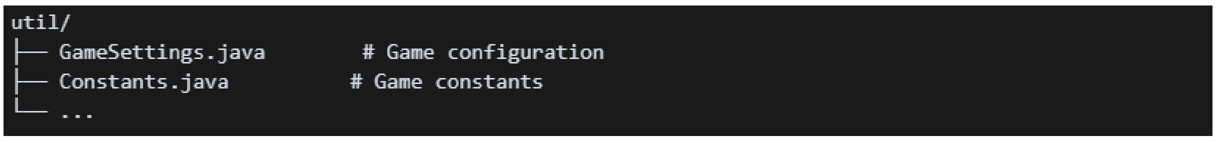


```
controller/
├── GameController.java      # Main game controller
├── MapEditorController.java # Map editor controller
└── ...
```

Controller Layer Responsibilities:

1. Game Control:
   - Game loop management
   - Entity updates
   - State coordination
2. Input Handling:
   - User input processing
   - Command execution
   - Event management

2.4 Utility Layer (util/):



```
util/
├── GameSettings.java        # Game configuration
├── Constants.java          # Game constants
└── ...
```

Utility Layer Responsibilities:

1. Configuration:
- Game settings
- Constants
- Configuration management

2. Helper Functions:
- Utility methods
- Common operations
- Shared functionality

**Design Principles Demonstrated**

1. Separation of Concerns:
- Clear layer separation
- Distinct subsystem responsibilities
- Modular component design

2. Single Responsibility:
- Each class has a specific purpose
- Subsystems handle specific domains
- Clear component boundaries

3. Dependency Management:
- Controlled dependencies between layers
- Clear subsystem interfaces
- Proper encapsulation

4. Modularity:
- Independent subsystems
- Reusable components
- Clear interfaces

**Benefits of the Architecture**

1. Maintainability:
- Clear structure
- Modular design
- Easy to locate and fix issues

2. Scalability:
- Easy to add new features
- Simple to extend subsystems
- Clear extension points

3. Testability:
- Isolated components
- Clear dependencies
- Easy to unit test

4. Flexibility:
- Easy to modify components
- Simple to replace subsystems
- Clear upgrade path