**SPRING 2025**

**COMP 302**

***KU Tower Defense***

**R2M2D2**

TeamD

Ahmet Taha Ekim - 79234

Muhammed Talha Sarıtepe - 79264

Mert Tuna – 80363

Doğu Arslan – 76649

Emir Göcen – 79634

**Table of Contents**

**New Use Cases****3**

* New Use Case 1 3
* New Use Case 25

**Refined Package Diagrams****7**

* Old Package Diagram7
* New Package Diagram8

**System Sequence Diagrams****12**

* Upgrade Tower SSD12
* Game Over SSD13

**Operation Contracts****14**

* New Contract 114
* New Contract 215

**Interaction Diagrams** **16**

* Upgrade Tower Sequence Diagram16
* Upgrade Tower Communication Diagram17
* Behavior Changes Sequence Diagrams18
* Behavior Changes Communication Diagrams19

**Class Diagrams****21**

* Old Class Diagram21
* New Class Diagram22

**Design Patterns****23**

* Decorator Pattern23
* Singleton Factory Pattern26
* Strategy Pattern29

# 

# **New Use Cases**

### **New Use Case-1 (Use Case 16: Upgrade Tower):**

**Scope:** KU Tower Defense  
 **Level:** User goal  
 **Primary Actor:** Player

**Stakeholders and Interests:**

* **Player:** Wants to strengthen towers to handle tougher enemies.
* **Game System:** Must validate upgrade eligibility, update tower stats, and maintain game balance.

**Preconditions:**

* Player has at least one placed tower.
* Player has sufficient gold to upgrade the selected tower.

**Postconditions:**

* Selected tower is upgraded with enhanced attributes.
* Player’s gold is reduced accordingly.

**Main Success Scenario:**

1. Player selects a tower already placed on the map.
2. System displays tower info panel including upgrade cost and effects.
3. Player clicks the "Upgrade" button.
4. System checks if player has enough gold.
5. If sufficient, system deducts gold and applies the upgrade: increased damage, range, or fire rate based on the tower type, if it is archer tower it will be upgraded to have 50% wider attack range and 2x higher rate of fire, if it is artillery tower, 20% larger attack range and the artillery shells cause 20% more AOE (area of effect), if it is mage tower, it will have a slow down effect on the enemies by 20% for the following 4 seconds.
6. System visually indicates the upgraded status (e.g., animation, level badge).

**Extensions:** 1a. **Insufficient gold:**  – System shows warning: “Not enough gold.”  
  – Upgrade is blocked.

2a. **Tower already at max level:**  – Upgrade button is disabled.  
  – Tooltip shows: “Maximum upgrade reached.”

**Special Requirements:**

* Upgrade must trigger real-time changes in tower behavior.
* Upgraded stats must be reflected in UI immediately.
* Animation or sound effect should signal successful upgrade.

### **New Use Case-2 (Use Case 17: Display Game Statistics on Game Over Screen):**

**Scope:** KU Tower Defense  
 **Level:** User goal  
 **Primary Actor:** Game System

**Stakeholders and Interests:**

* **Player:** Wants to view a performance summary after the game ends.
* **Game System:** Provides feedback and supports post-game options (replay, exit, submit score).

**Preconditions:**

* Game has ended (either by defeat or completion of final wave).

**Postconditions:**

* Game Over screen is displayed with comprehensive statistics.
* Player can proceed to main menu or replay.

**Main Success Scenario:**

1. Game ends due to player defeat or wave completion.
2. System transitions to Game Over screen.
3. System displays the following statistics:  
   * Total Gold Earned
   * Enemies Spawned
   * Enemies Reached End
   * Enemies Defeated
   * Total Damage Done
   * Time Played
4. Player sees "Main Menu" and "Replay Game" buttons.
5. Player selects an option and proceeds.

**Extensions:** 1a. **Victory:**  – Special animation or music plays for victory.  
 1b. **Defeat:**  – Different animation or music and message shown for losing.

2a. **Optional: Submit score to leaderboard (if implemented):**  – Player can click “Submit Score” to record performance.  
  – Score is validated and sent to backend.

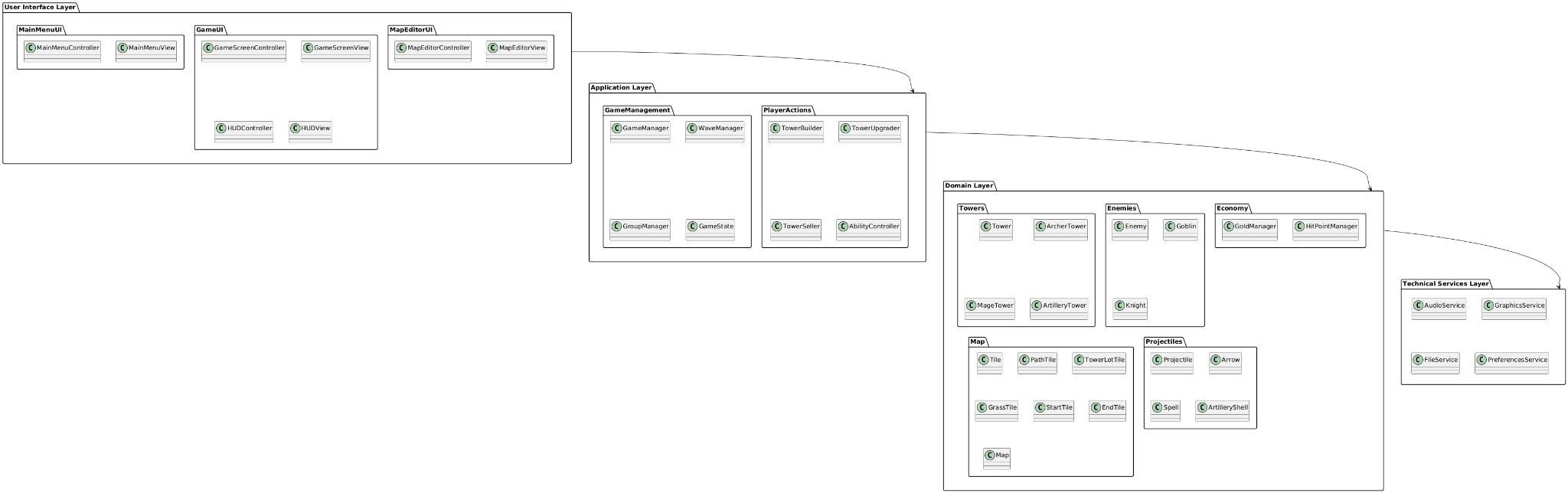
**Special Requirements:**

* Display must be responsive and accurate.
* All values should reflect real gameplay data.

Background music and UI must match the outcome (win/lose).

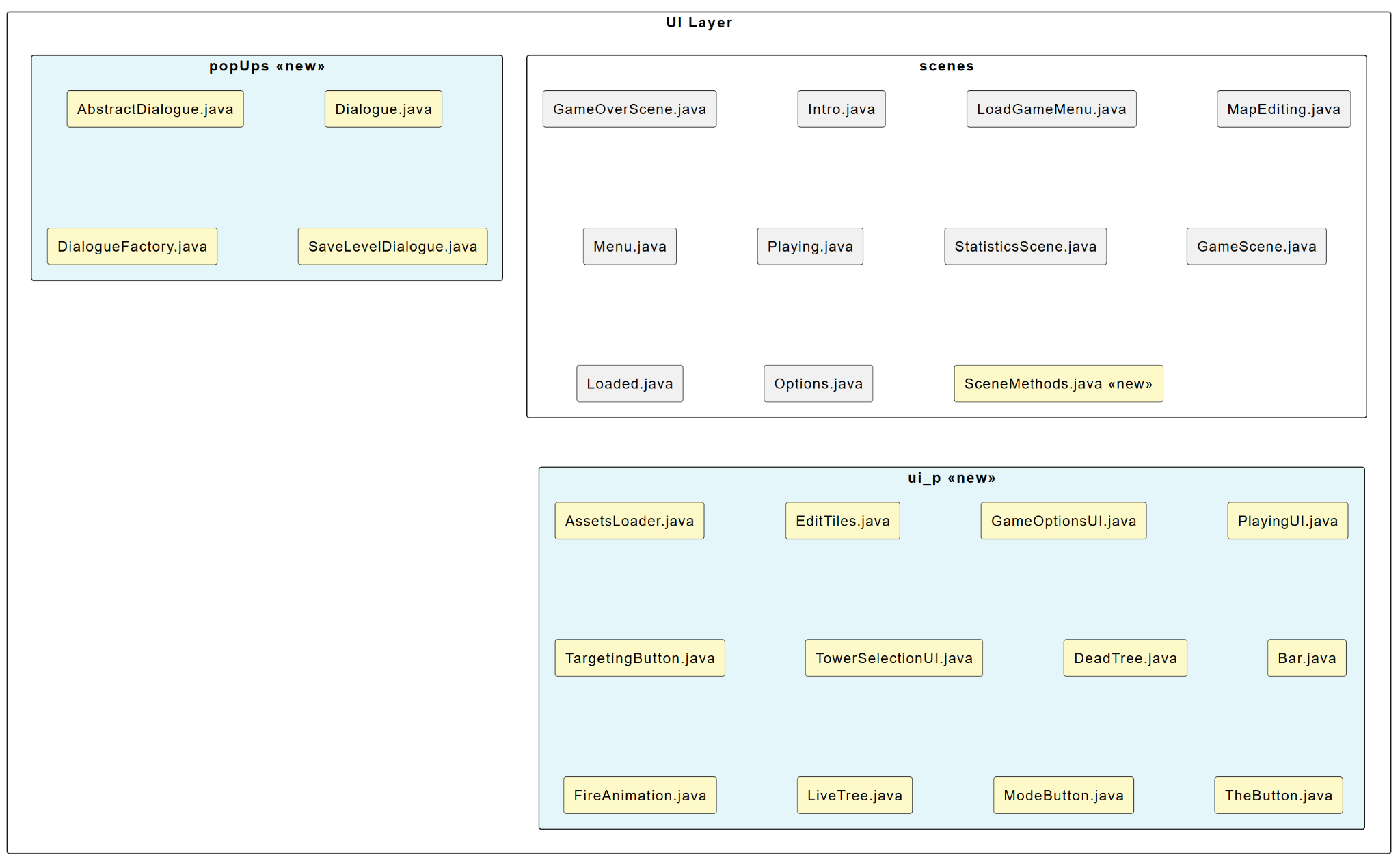
# **Refined Package Architecture**

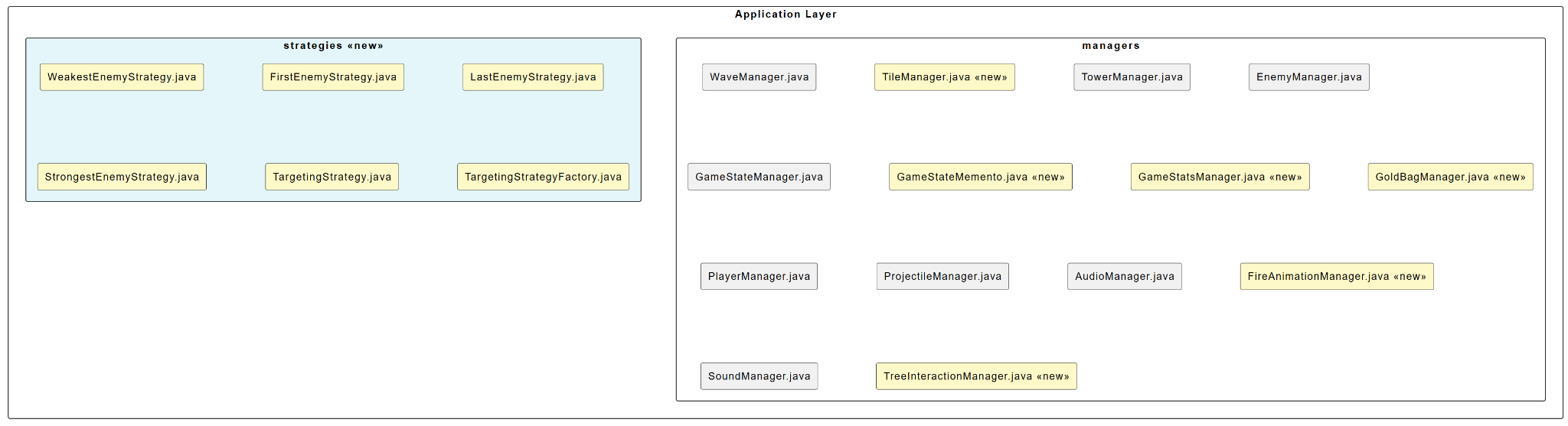
**Old Package Architecture (P1):**

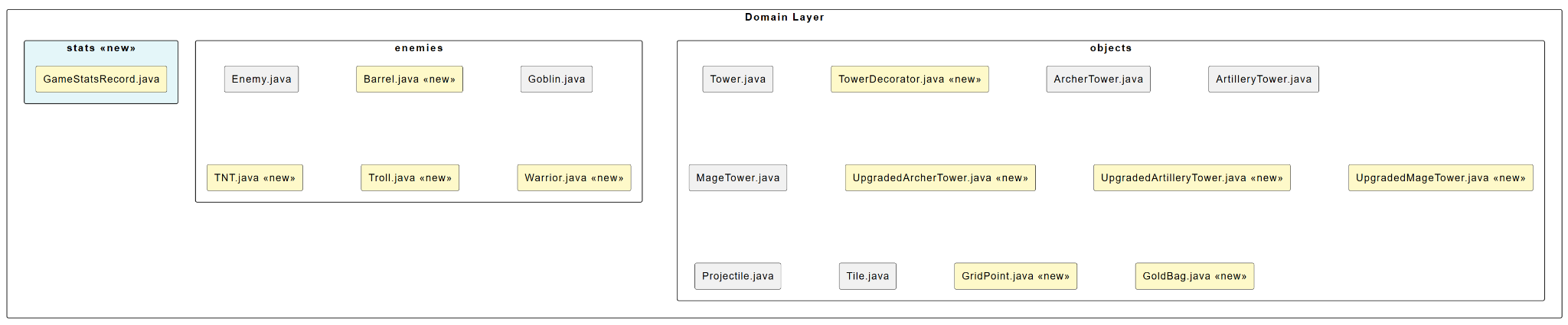


***Figure 1: Old Package Architecture***

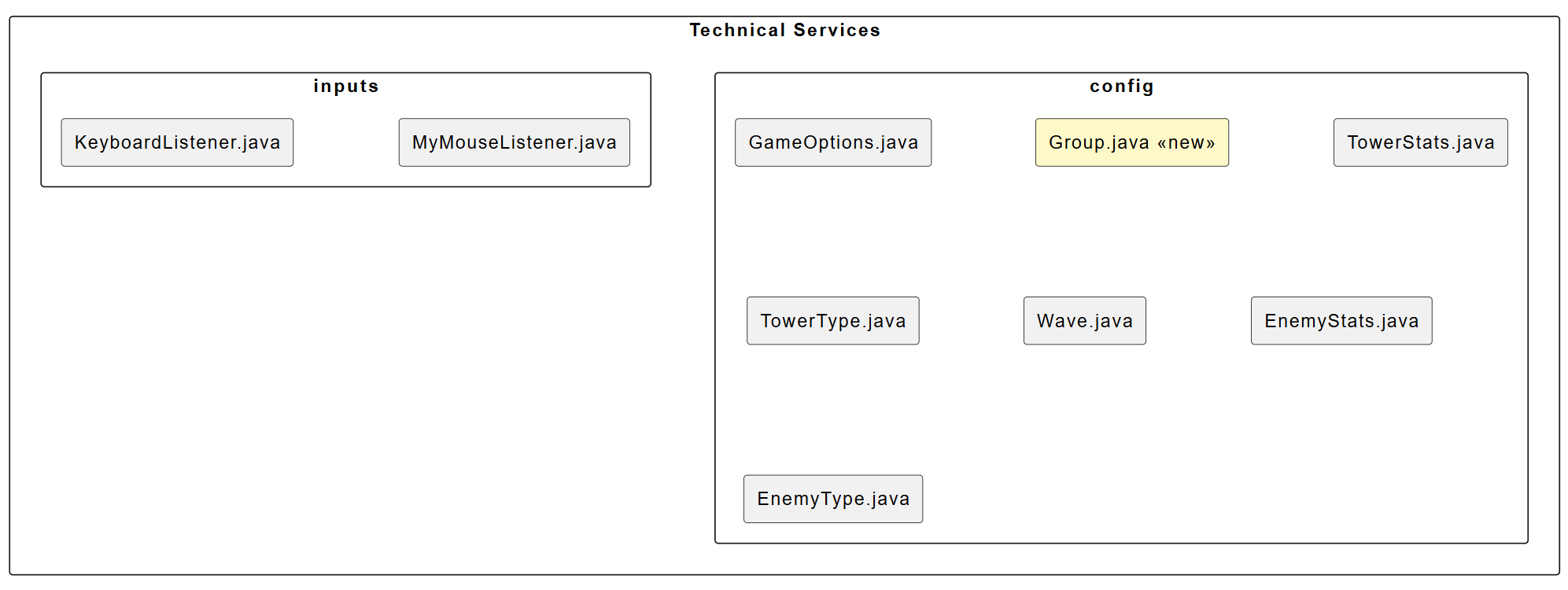
**New Package Architecture (P2):**

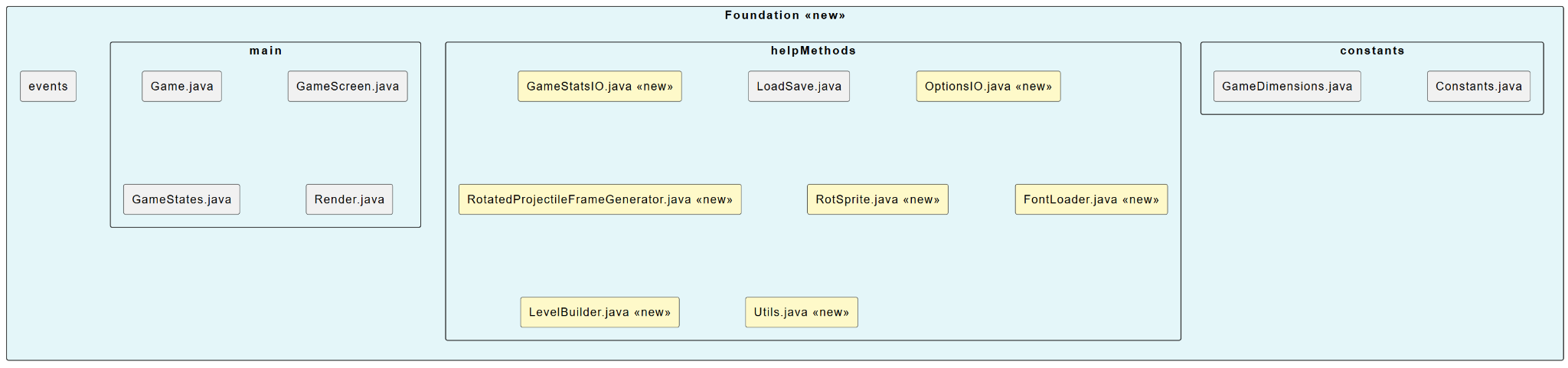






***Figure 2: New Package Architecture-1***





***Figure 2: New Package Architecture-2***

New packages are highlighted with blue and new classes are highlighted as yellow as seen from Figure 2.

### **1. Layering and Structure**

| **Aspect** | **Phase 1** | **Phase 2** |
| --- | --- | --- |
| **Layers** | Clear separation: UI, Application, Domain, Technical Services | Expanded: UI, Application, Domain, Technical Services, Foundation |
| **Dependency Flow** | UI → Application → Domain → Tech Services | Same as Phase 1, but extended to include Foundation layer |
| **Organization** | High-level conceptual components grouped logically | Real file structure with Java classes grouped in packages |

### **2. UI Layer**

| **Phase 1** | **Phase 2** |
| --- | --- |
| Subdivided into: MainMenuUI, GameUI, MapEditorUI | Divided into: scenes, popUps, ui\_p |
| Conceptual classes like MainMenuController, HUDView | Actual implementation: PlayingUI.java, MapEditing.java, Dialogue.java, etc. |
| Emphasis on MVC-style Controller & View | Implementation shows full UI behavior logic, animations, and UI assets |

### **3. Application Layer**

| **Phase 1** | **Phase 2** |
| --- | --- |
| Packages: GameManagement, PlayerActions | Packages: managers, strategies |
| Classes like GameManager, TowerBuilder, WaveManager | More detailed classes like EnemyManager.java, FireAnimationManager.java, and targeting strategies |

**Evolution:** Phase 2 introduces sophisticated behaviors via **Strategy pattern** for targeting enemies, which is only implied in Phase 1.

### **4. Domain Layer**

| **Phase 1** | **Phase 2** |
| --- | --- |
| Clear domain segmentation: Towers, Enemies, Projectiles, Map | Matches Phase 1 closely with real class files (e.g., Tower.java, Enemy.java, Tile.java) |
| Decorators and Upgrades are **not shown** | Implemented using Decorator pattern (e.g., TowerDecorator.java, UpgradedMageTower.java) |

**Insight:** Design patterns are explicitly used in Phase 2 that were only conceptually represented in Phase 1.

### 

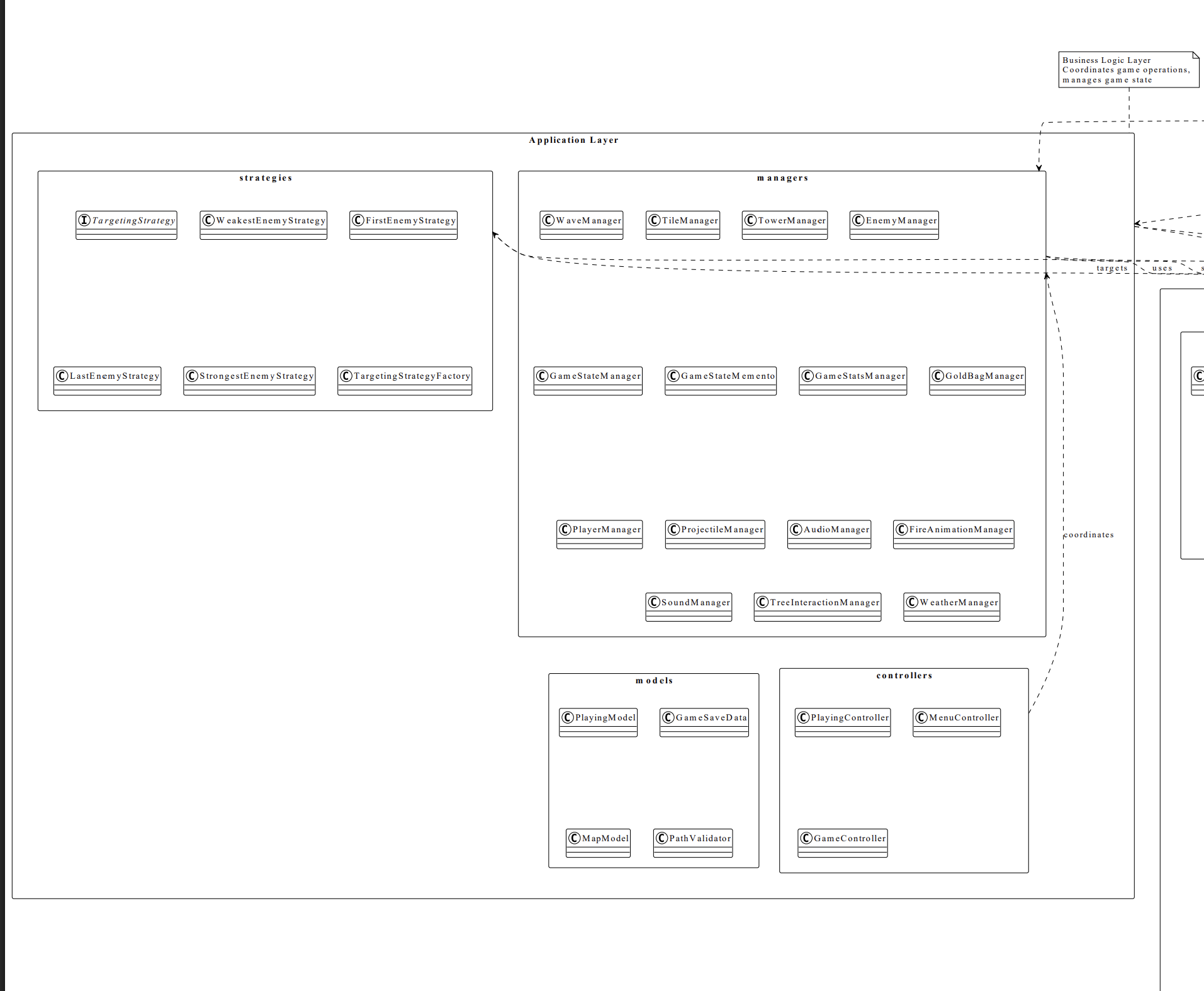
### **5. Technical Services**

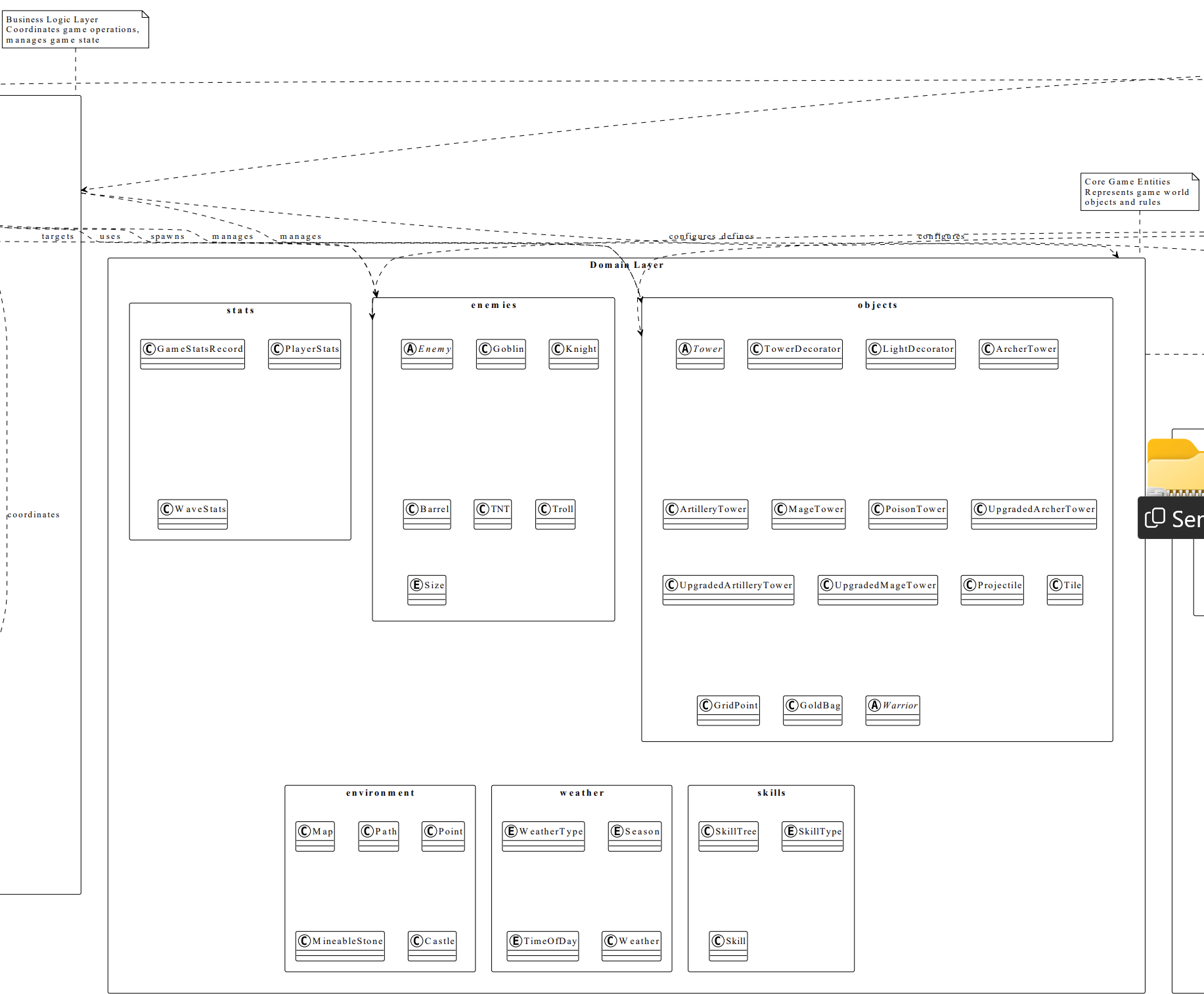
| **Phase 1** | **Phase 2** |
| --- | --- |
| Abstract services: AudioService, GraphicsService, PreferencesService | Implemented managers and listeners: AudioManager.java, KeyboardListener.java, GameOptions.java |
| Simple functional services in theory | Richer system integration (e.g., SoundManager, Input Listeners) |

### **6. Additional Elements in Phase 2**

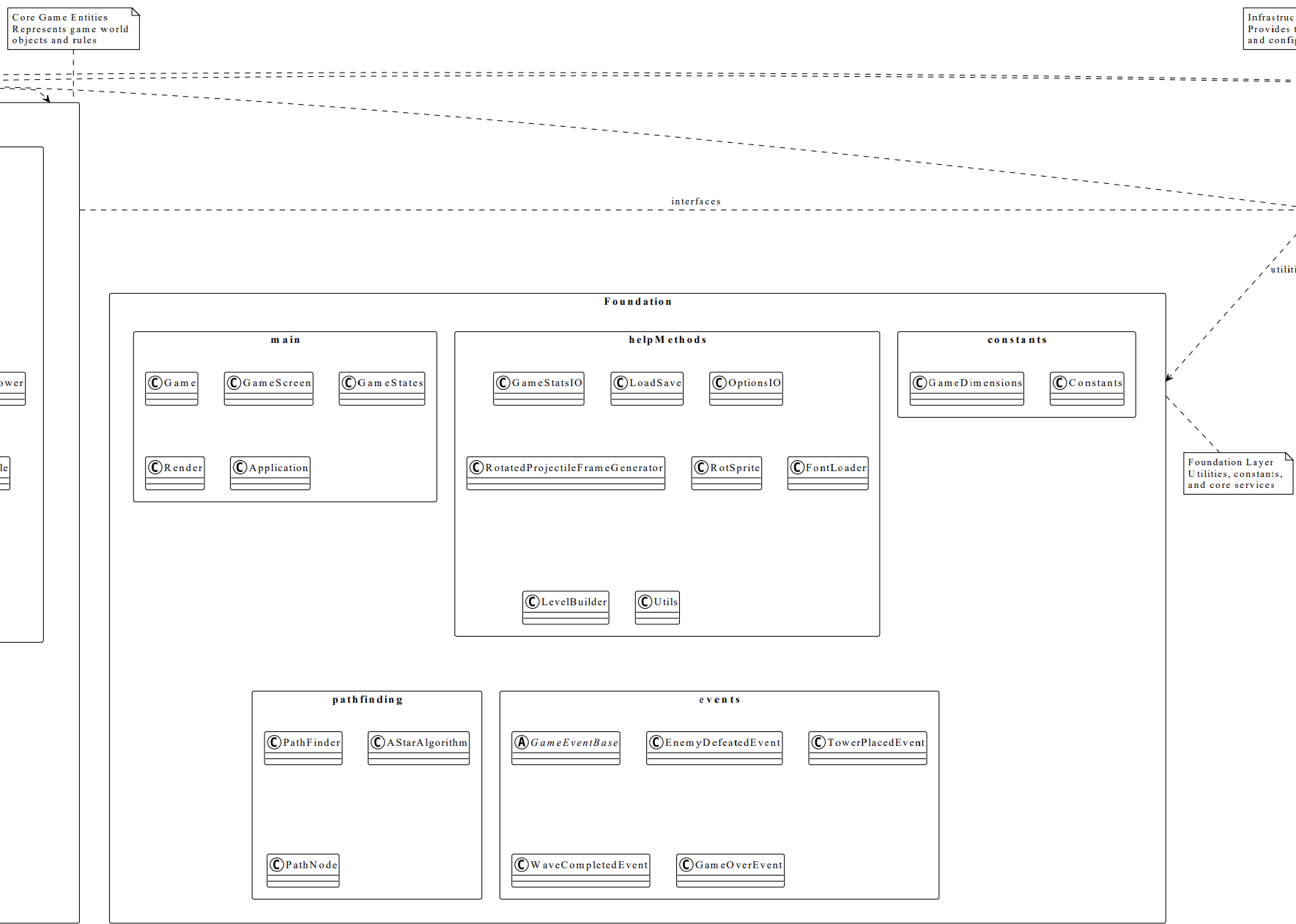
| **New in Phase 2** | **Description** |
| --- | --- |
| **Foundation Layer** | Utilities, constants, file I/O, and main entry points (e.g., Game.java) |
| **Help Methods & Events** | LoadSave.java, GameStatsIO.java handle persistence, a concern absent in Phase 1 |
| **GameStats Management** | Added in GameStatsManager.java, GameStatsRecord.java—useful for score/history tracking |

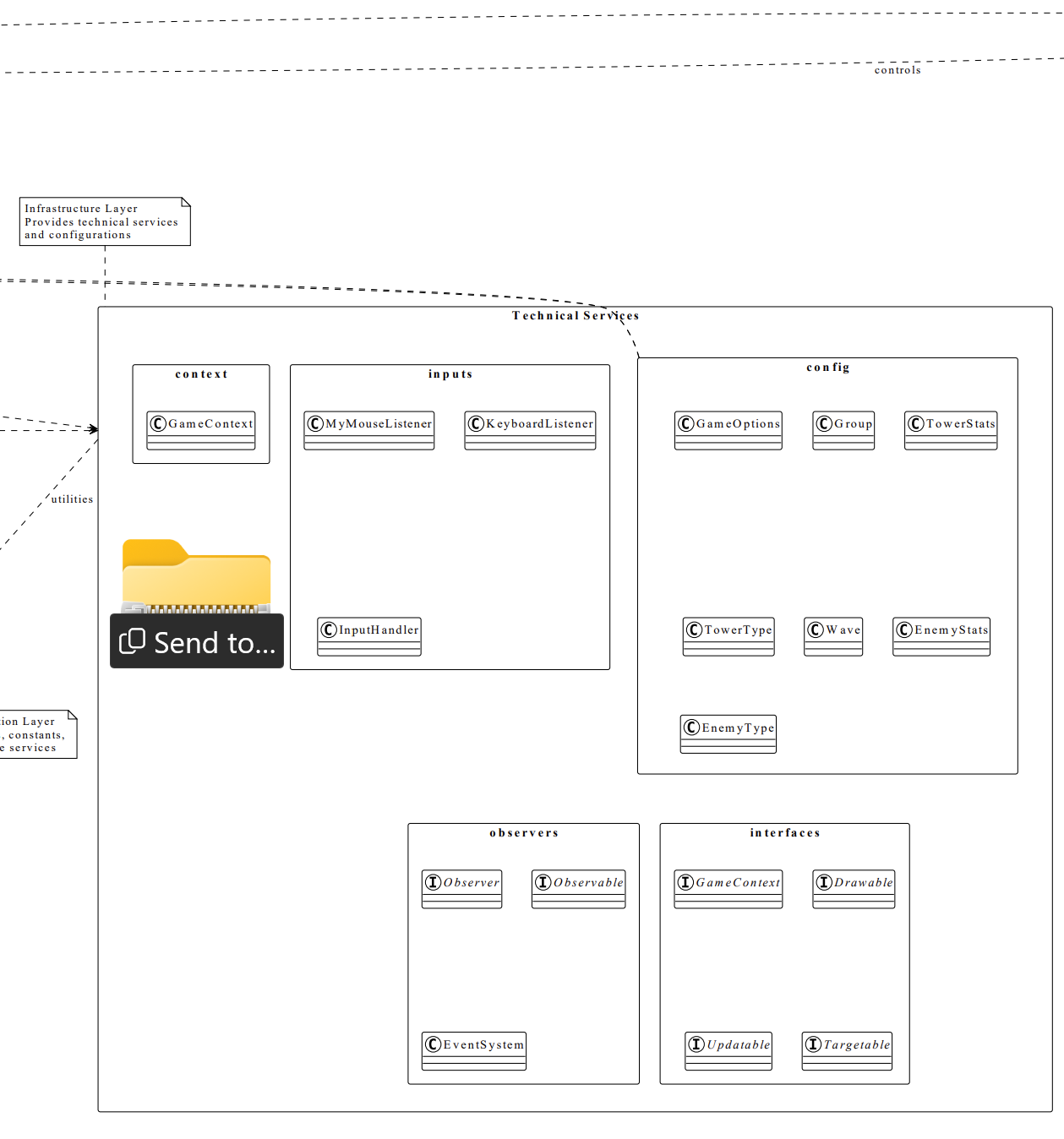
***\***

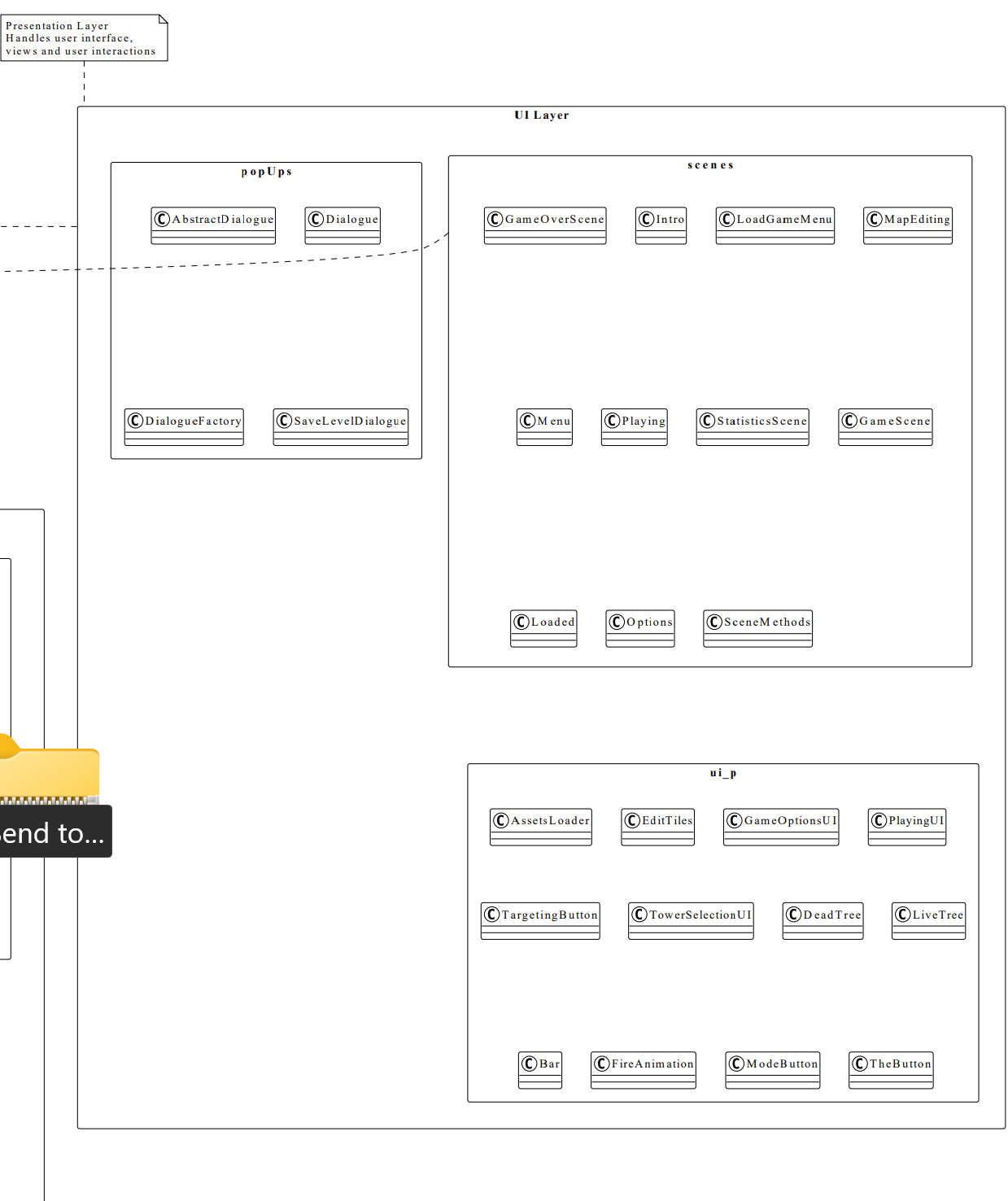
***Final Logical Architecture and UML Package Diagram:***



***\***

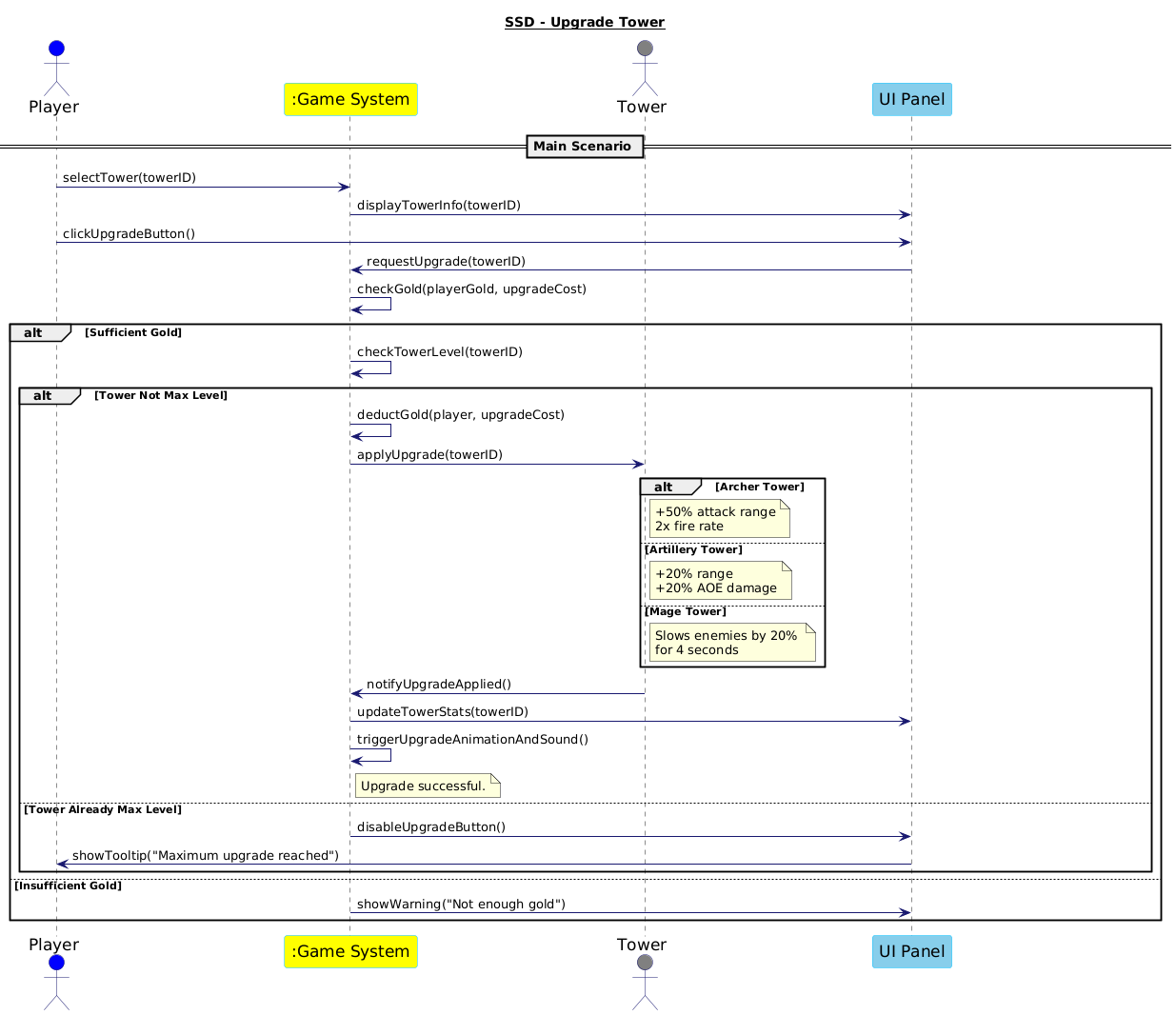






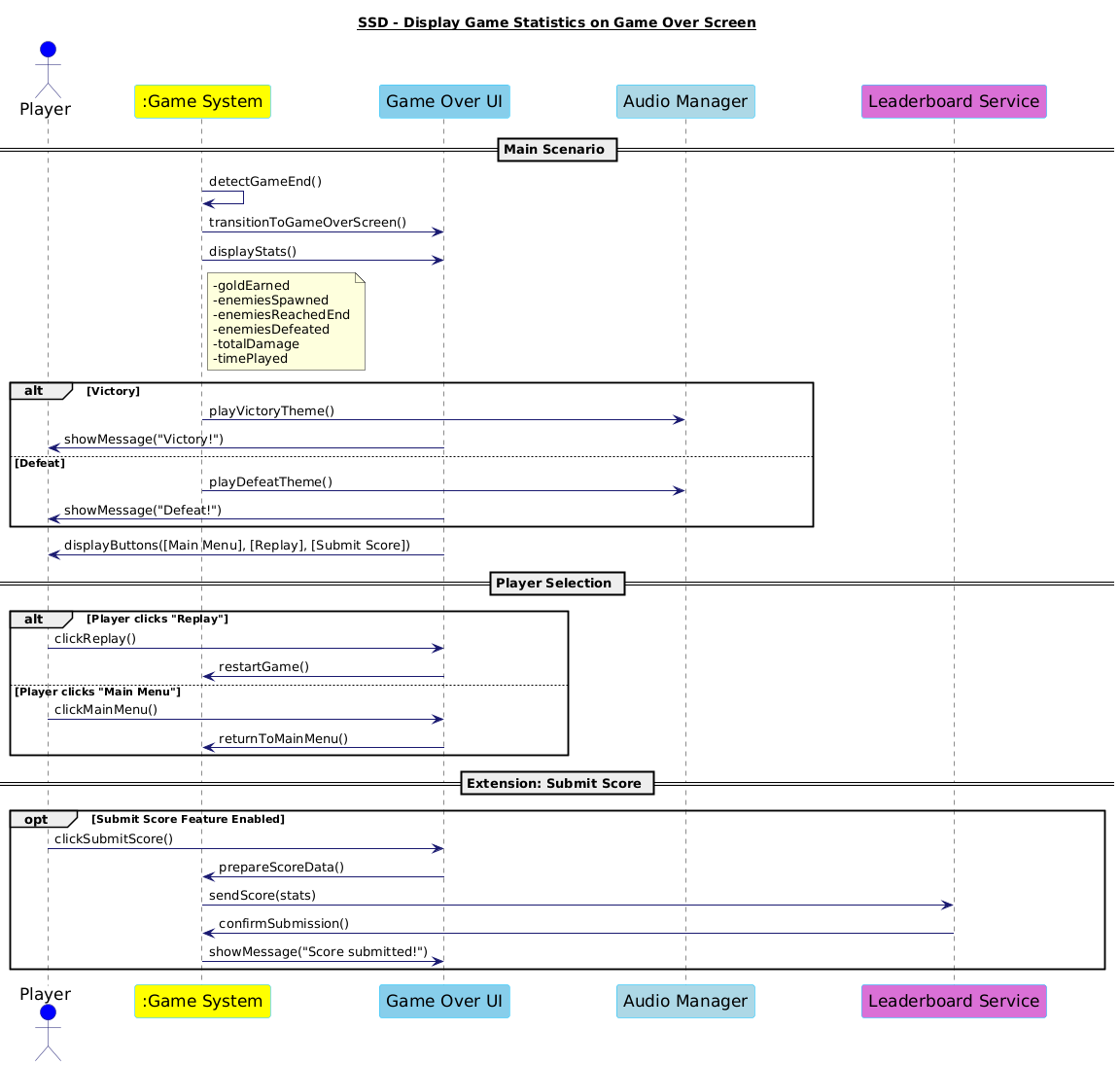
# **System Sequence Diagrams**

**Upgrade Tower SSD:**

****

***Figure 3: Upgrade Tower SSD***

**Game Over SSD:**

****

***Figure 4: Game Over SSD***

# **Operation Contracts**

# **New Operation Contract -1 (Contract CO16: upgradeTower):**

**Operation Name:** upgradeTower(Tower selectedTower)

**Cross-References:** Use Case 16: Upgrade Tower

**Preconditions:**

selectedTower is not null and is already placed on the map.

Player has enough gold to afford the upgrade.

selectedTower is not at its maximum upgrade level.

**Postconditions:**

Player's gold is reduced by the cost of the upgrade.

The tower's level increases.

The tower's attributes are updated as follows based on its type:

Archer Tower: +50% attack range 2× rate of fire

Artillery Tower: +20% attack range +20% AOE damage

Mage Tower: Adds a slowing effect (20% speed reduction for 4 seconds)

Tower upgrade animation/sound effect is triggered.

The upgrade is shown on the tower UI (e.g., level badge, upgraded stats).

**Constrains:**  
 Gold deduction and tower upgrade must occur atomically—if one fails, both are rolled back.  
 All UI elements (range display, level badge, stats panel) must update within 100ms after upgrade completion.  
 Upgrade logic must strictly adhere to tower type—no mixing of attribute changes between tower classes.  
 Upgrade must be disabled if the tower is already at maximum level; this must be enforced both in UI and backend logic.  
 Upgrade can only be triggered if the tower is not currently reloading or under attack animation (if such a mechanic exists).  
 Animation/sound for upgrade must not delay or block game logic; must run in parallel without stalling the game loop.

# **New Operation Contract -2 (Contract CO17: displayGameOverScreen):**

**Operation Name:** displayGameOverScreen(GameResult result)

**Cross-References:** Use Case 17: Display Game Statistics on Game Over Screen **Preconditions:**

The game session has ended, either by:

All waves successfully completed (victory), or

Player defeated (base health reached 0 or similar condition).

**Postconditions:**

Game Over screen is fully visible.

All game statistics are displayed correctly.

The player can now:

Return to the Main Menu

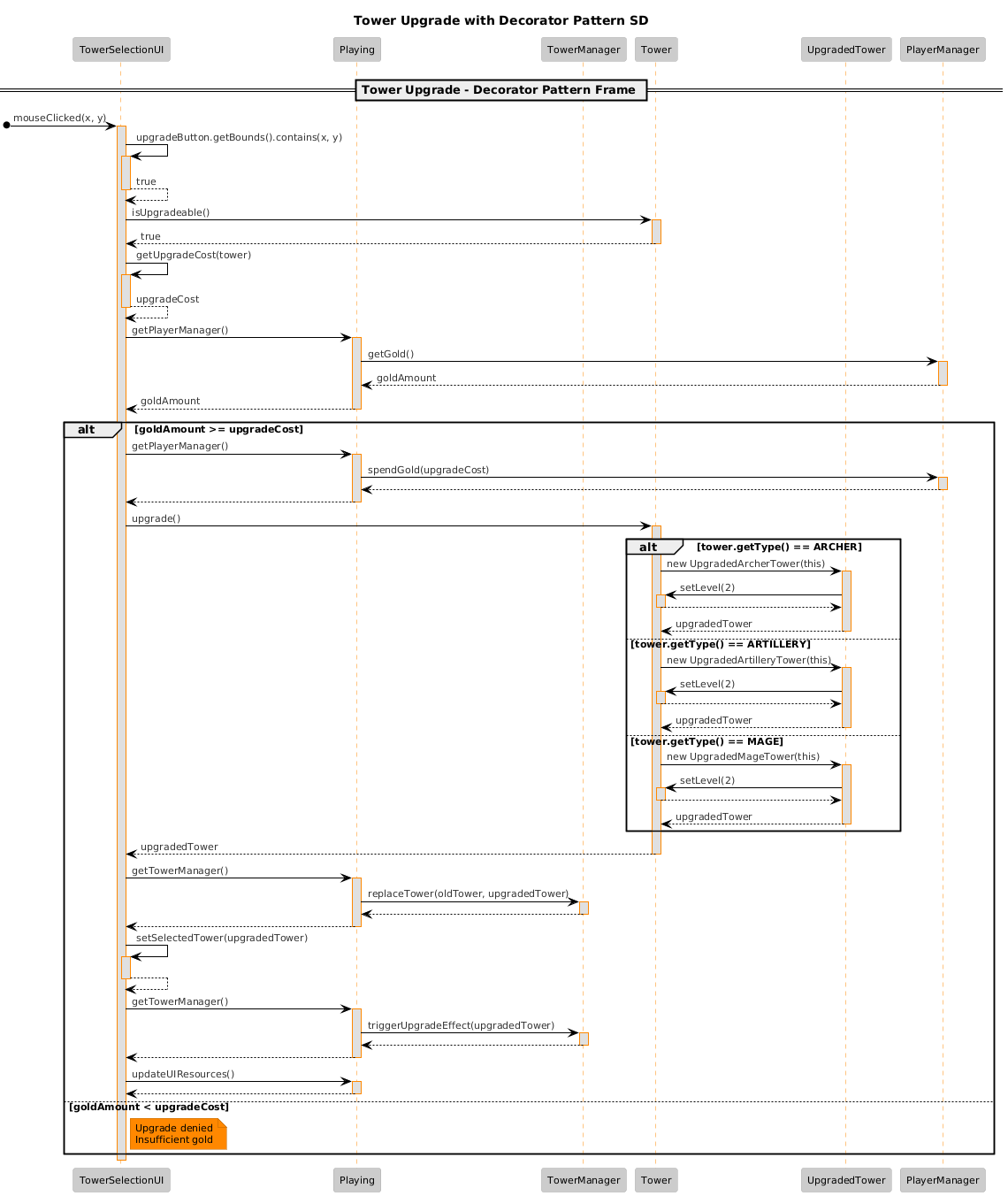
Replay the game

Submit score (optional, if feature exists)

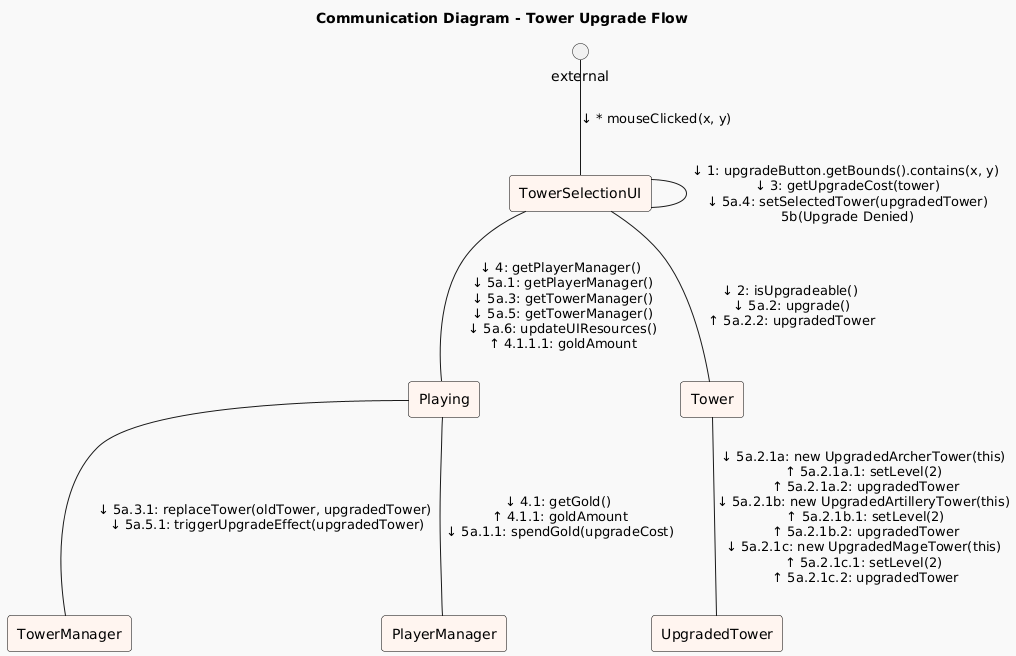
**Constraints:**  
 All statistics must be finalized and frozen at the moment the game ends—no background updates allowed after transition.  
 Game Over screen must load within 1 second after the game ends to ensure smooth UX.  
 Displayed statistics must match the internal counters exactly—no rounding or approximation unless explicitly designed.  
 The score can be submitted only once per session to avoid duplicate leaderboard entries.  
 Only one set of music/animation (win or lose) can be played—must not overlap or be played simultaneously.

# **Interaction Diagrams**

Upgrade Tower Sequence Diagram

****

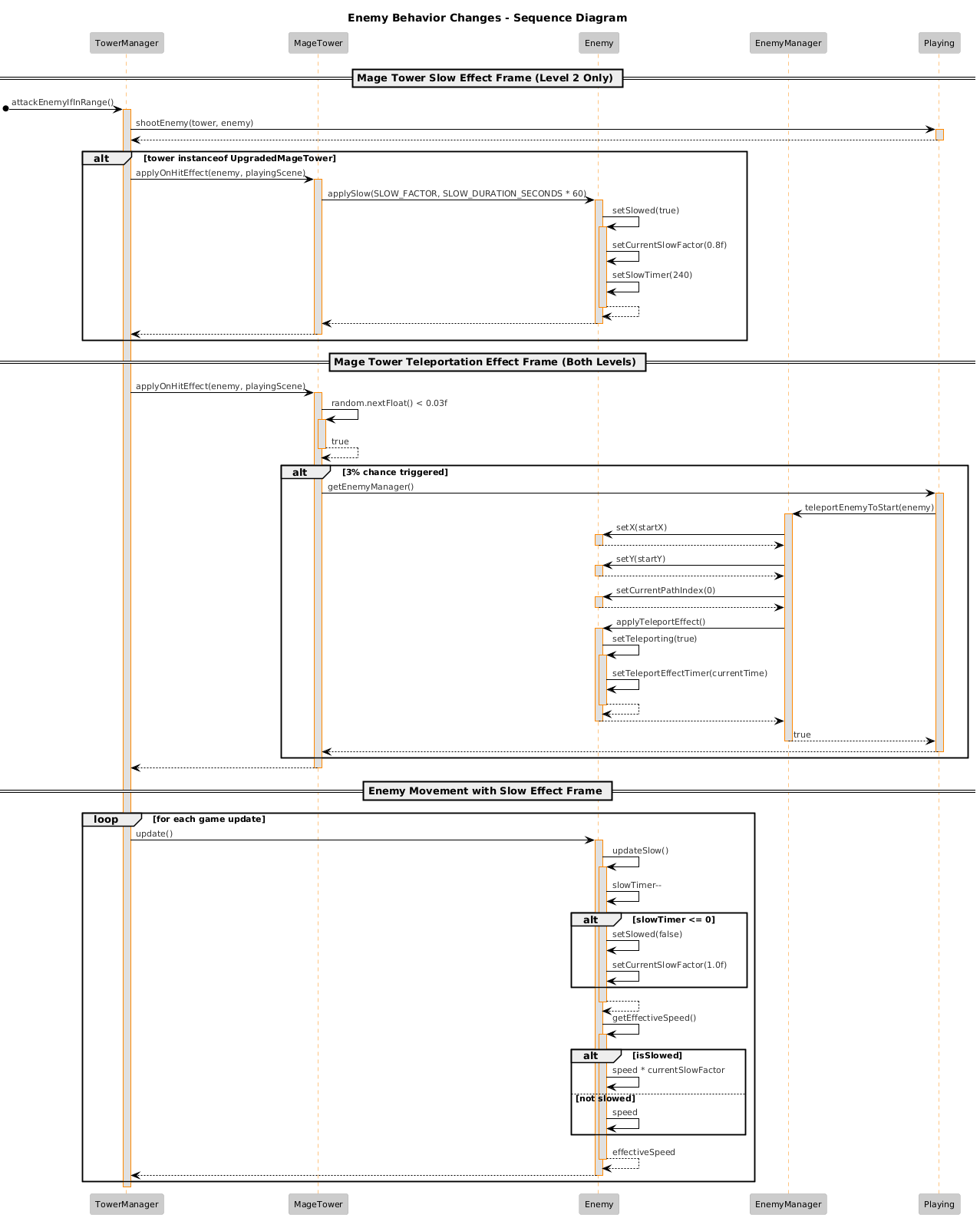
Upgrade Tower Communication Diagram

****

Behavior Changes Sequence Diagrams

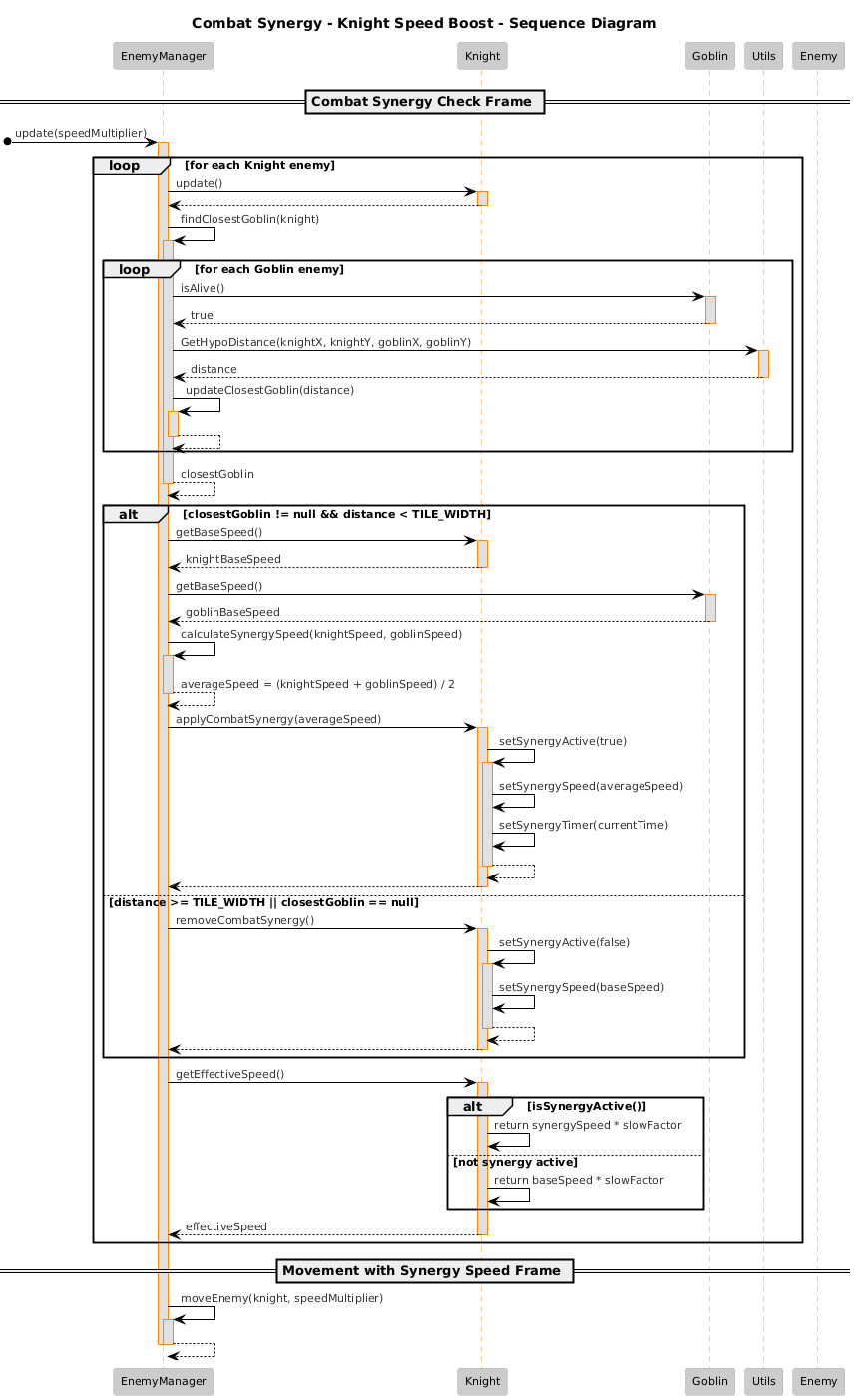
**Back to Step 1 - Behavior Change and Upgraded Mage Tower**

**- Sequence Diagram -**

****

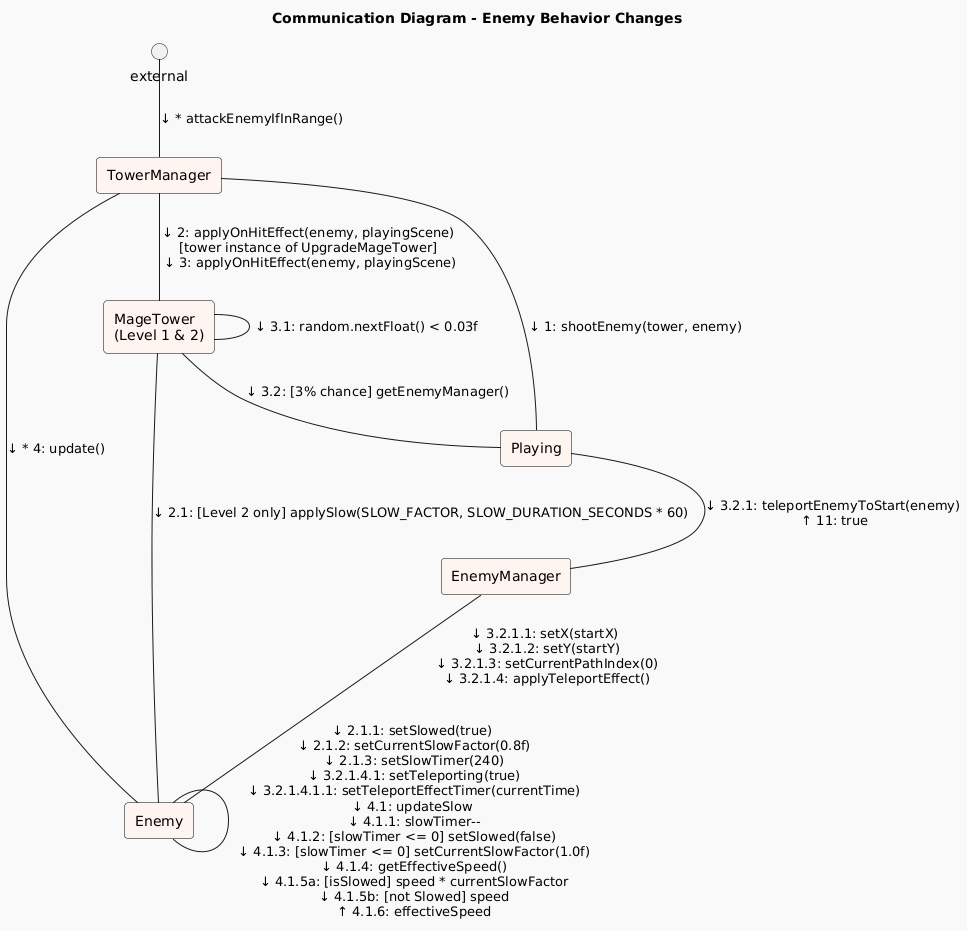
**Combat Synergy - Behavior Change**

* **Sequence Diagram -**

****

Behavior Changes Communication Diagrams

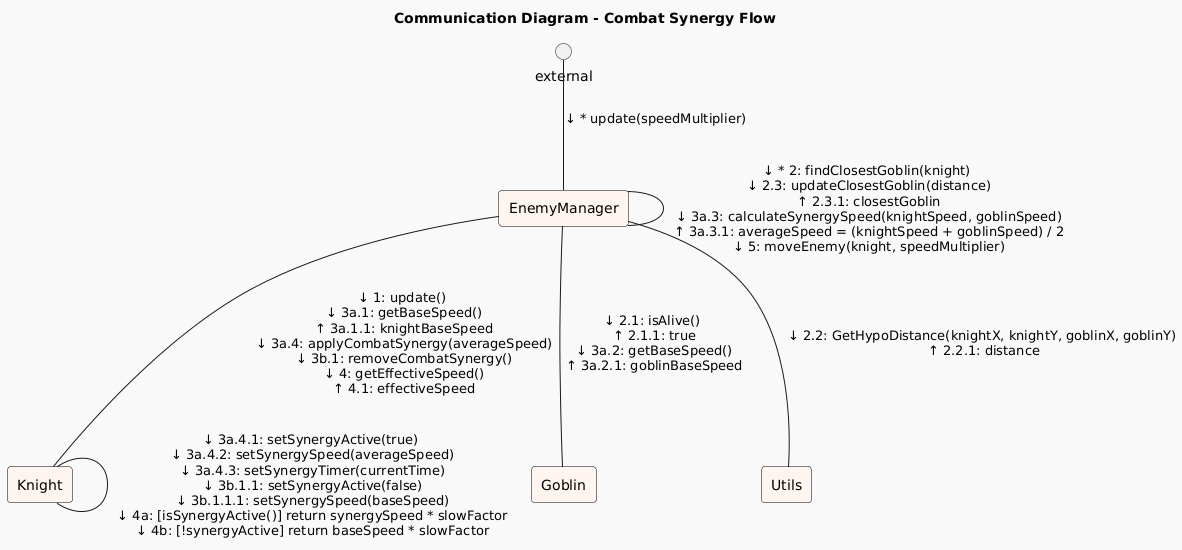
**Back to Step 1 - Behavior Change and Upgraded Mage Tower**

**- Communication Diagram -   
  
**

# 

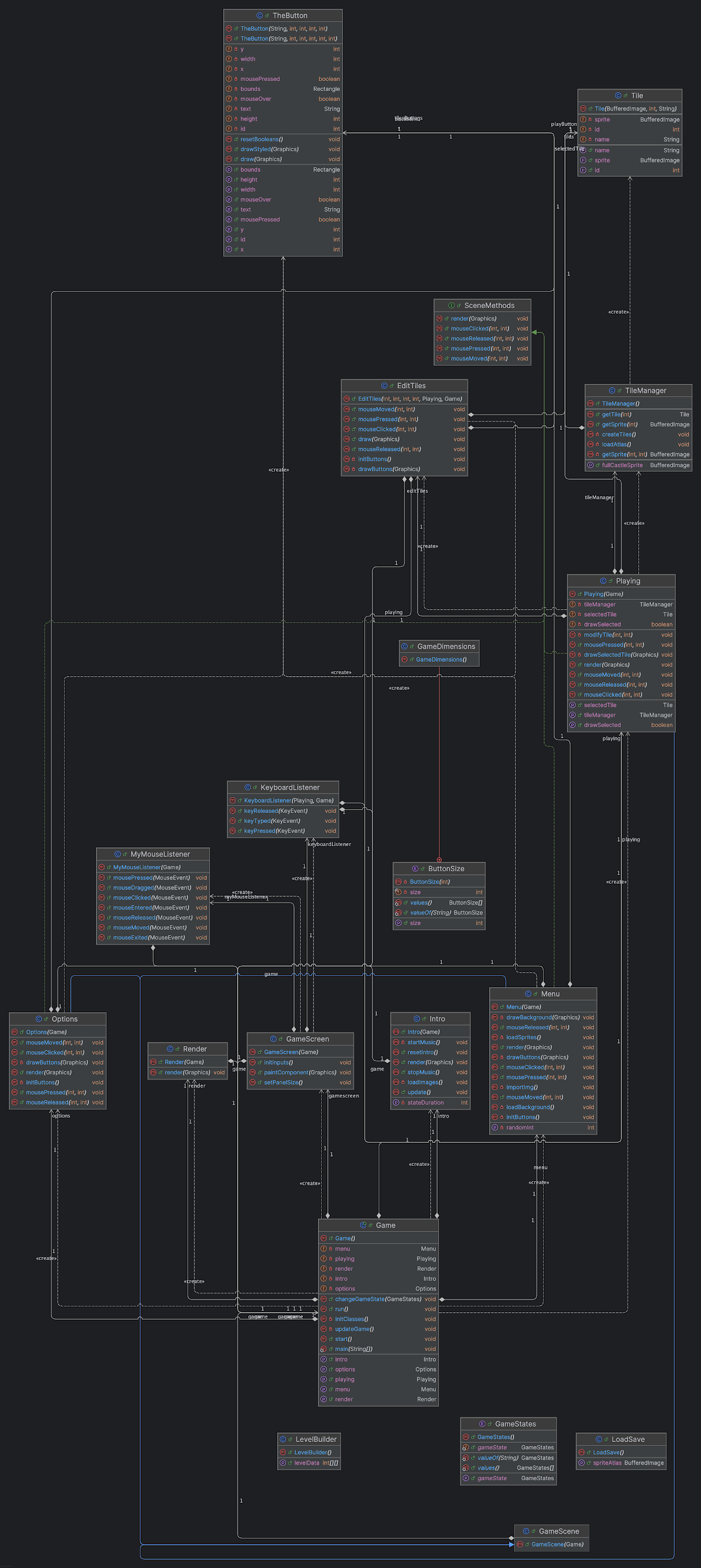
**Combat Synergy - Behavior Change**

* **Communication Diagram -**



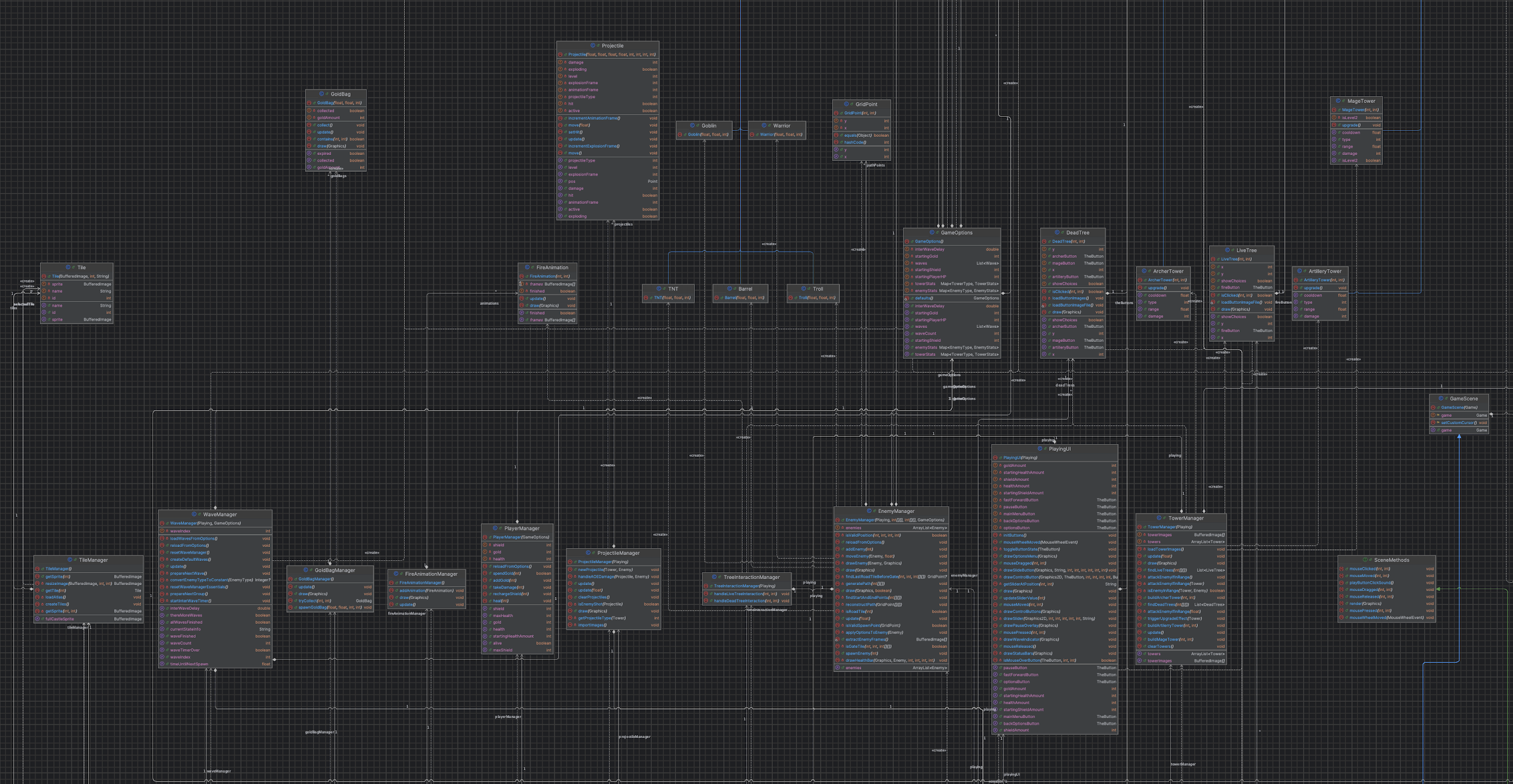
# **Class Diagrams**

# **Old Class Diagram:**

******

***Figure 5: Old Class Diagram***

# **New Class Diagram:**

******

***Figure 6: New Class Diagram***

***(Note: Since our new class diagram is too big we only included the screen shot of the part where the new classes are shown)***

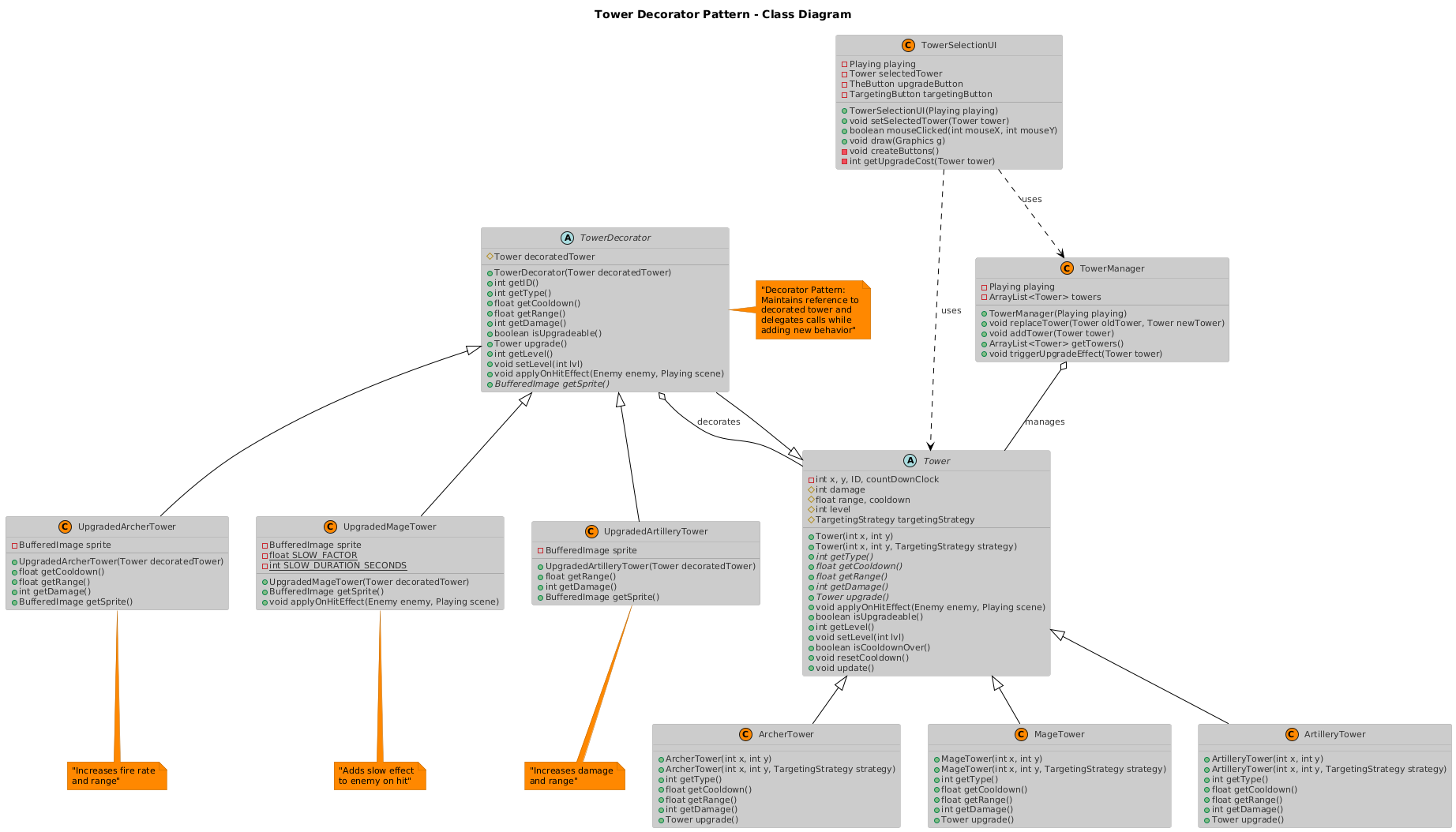
# **Design Patterns**

# **Decorator Pattern:**

* **Intent & Motivation:**

The Decorator Pattern allows behavior to be added to objects dynamically without altering their structure. In our tower defense game, we use this pattern to upgrade towers by wrapping base tower objects with enhanced functionality while maintaining the same interface. This approach avoids the need for complex inheritance hierarchies and allows for flexible combinations of tower abilities.

* **UML class diagram:**



* **Focused code snippet with in-line comments:**

*// Base Tower class - defines the interface that decorators will enhance*

public abstract class Tower {

protected int damage;

protected float range, cooldown;

protected int level = 1;

public abstract int getType();

public abstract float getCooldown();

public abstract float getRange();

public abstract int getDamage();

public abstract Tower upgrade(); *// Returns decorated version*

*// Default implementation for special effects*

public void applyOnHitEffect(Enemy *enemy*, Playing *playingScene*) {

*// Base towers have no special effects*

}

}

*// Decorator base class - maintains reference to wrapped tower*

public abstract class TowerDecorator extends Tower {

protected Tower decoratedTower; *// The tower being decorated*

public TowerDecorator(Tower *decoratedTower*) {

super(decoratedTower.getX(), decoratedTower.getY());

this.decoratedTower = decoratedTower;

}

*// Delegate most behavior to the wrapped tower*

@Override

public int getType() { return decoratedTower.getType(); }

@Override

public int getID() { return decoratedTower.getID(); }

*// Abstract method for decorators to provide enhanced sprites*

public abstract BufferedImage getSprite();

}

*// Concrete decorator - adds slow effect to Mage Tower*

public class UpgradedMageTower extends TowerDecorator {

private BufferedImage sprite;

private static final float SLOW\_FACTOR = 0.8f; *// 20% slow*

private static final int SLOW\_DURATION\_SECONDS = 4;

public UpgradedMageTower(Tower *decoratedTower*) {

super(decoratedTower);

decoratedTower.setLevel(2); *// Mark as upgraded*

*// Load enhanced sprite for visual upgrade*

this.sprite = LoadSave.getImageFromPath("/TowerAssets/mage\_up.png");

}

@Override

public BufferedImage getSprite() {

return sprite; *// Return upgraded visual*

}

*// Enhanced behavior - adds slow effect on hit*

@Override

public void applyOnHitEffect(Enemy *enemy*, Playing *playingScene*) {

super.applyOnHitEffect(enemy, playingScene); *// Call base behavior first*

*// Add new slow effect functionality*

enemy.applySlow(SLOW\_FACTOR, SLOW\_DURATION\_SECONDS \* 60);

}

}

*// Usage in upgrade system*

public boolean mouseClicked(int mouseX, int mouseY) {

if (upgradeButton != null && upgradeButton.getBounds().contains(mouseX, mouseY)) {

if (selectedTower.isUpgradeable()) {

*// Decorator pattern in action - wrap existing tower with enhanced version*

Tower upgradedTower = selectedTower.upgrade(); *// Returns decorated tower*

playing.getTowerManager().replaceTower(selectedTower, upgradedTower);

setSelectedTower(upgradedTower); *// Update UI to show decorated tower*

}

}

}

* **Explanation / role mapping:**
  + Component (Tower): Defines the interface for objects that can have responsibilities added dynamically
  + ConcreteComponent (ArcherTower, MageTower, ArtilleryTower): Base tower implementations that can be decorated
  + Decorator (TowerDecorator): Abstract base class that maintains a reference to a Component object and delegates operations to it
  + ConcreteDecorator (UpgradedMageTower, UpgradedArcherTower, UpgradedArtilleryTower): Add specific enhanced behaviors like slow effects, increased damage, or improved range

The pattern allows us to "wrap" a basic tower with enhanced capabilities while preserving the original tower's core functionality and maintaining the same interface for the game system.

* **Benefits:**
  + Flexibility: Can add new behaviors to towers without modifying existing code
  + Runtime Composition: Upgrades are applied dynamically during gameplay rather than at compile time
  + Single Responsibility: Each decorator focuses on one specific enhancement (slow effect, damage boost, etc.)
  + Interface Preservation: Upgraded towers work seamlessly with existing game systems (TowerManager, rendering, etc.)
  + Extensibility: Easy to add new upgrade types by creating new decorator classes
  + Avoids Class Explosion: No need for separate classes like "UpgradedSlowMageTower" - decorators can be combined if needed
  + Maintains Original Object: The base tower's state and behavior remain intact, just enhanced

# **Singleton Factory Pattern:**

* **Intent & Motivation:**

To ensure that only one instance of resource managers (for audio and assets) exists throughout the game, providing a global point of access and efficient resource management. The classes also act as factories by encapsulating the creation/loading of audio clips and images, abstracting the complexity from the rest of the codebase.

1. Prevent redundant loading of resources (audio, images).
2. Centralize resource management for consistency and memory efficiency.
3. Provide easy, global access to resources without passing references everywhere

* **Focused code snippet with in-line comments:**

**public class AudioManager {**

**// Singleton instance**

**private static AudioManager instance;**

**// Private constructor prevents instantiation from outside**

**private AudioManager() {**

**loadAudio(); // Factory: loads all audio resources**

**}**

**// Global access point**

**public static AudioManager getInstance() {**

**if (instance == null) {**

**instance = new AudioManager();**

**}**

**return instance;**

**}**

**// Factory methods for loading resources**

**private void loadAudio() { /\* ... \*/ }**

**private void loadMusic(String name, String filename) { /\* ... \*/ }**

**private void loadSound(String name, String filename) { /\* ... \*/ }**

**// ... other methods ...**

**}**

**public class AssetsLoader {**

**// Singleton instance**

**private static AssetsLoader instance;**

**// Private constructor prevents instantiation from outside**

**private AssetsLoader() {**

**loadAll(); // Factory: loads all image resources**

**}**

**// Global access point**

**public static AssetsLoader getInstance() {**

**if (instance == null) {**

**instance = new AssetsLoader();**

**}**

**return instance;**

**}**

**// Factory methods for loading resources**

**private void loadAll() { /\* ... \*/ }**

**private BufferedImage loadImage(String path) { /\* ... \*/ }**

**// ... other methods ...**

**}**

* **Explanation / role mapping:**
  + Singleton Pattern:
* Both classes ensure only one instance exists (**private static instance, private constructor, public static getInstance()).**
* **This instance is globally accessible, ensuring all parts of the game use the same resource manager.**
* **Factory Pattern:**
  + **Both classes encapsulate the creation/loading of resources (audio clips, images).**
  + **The loading logic is hidden inside the manager, so other classes don’t need to know how to load or parse resources—they just request them.**
* **Role Mapping:**
  + **Singleton: Controls instantiation and provides a single access point.**
  + **Factory: Handles the creation/loading of complex objects (audio, images).**
* **Benefits:**
  + **Resource Efficiency:** Prevents duplicate loading of the same resource, saving memory and load time.
  + **Centralized Management:** All resource loading and access is managed in one place, making maintenance and debugging easier.
  + **Global Access:** Any part of the code can access resources without passing references.
  + **Encapsulation:** The complexity of loading resources is hidden from the rest of the codebase.
  + **Extensibility:** New resource types or loading strategies can be added in one place without affecting the rest of the code.

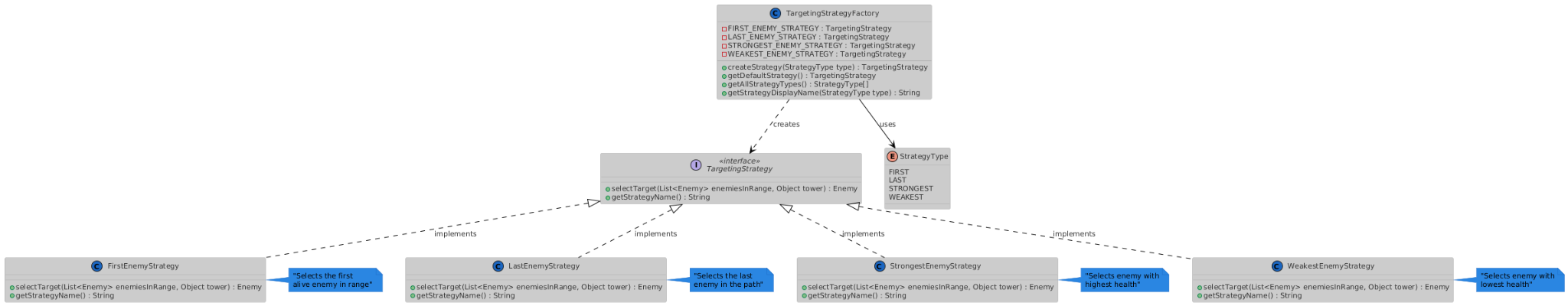
# **Strategy Pattern:**

* **Intent & Motivation:**

The Strategy Pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

In our tower defense game, towers can use different targeting strategies (e.g., attack the first, last, strongest, or weakest enemy in range). The Strategy Pattern allows us to easily switch or add new targeting behaviors without modifying the tower or enemy code.

* **UML class diagram:**

****

* **Focused code snippet with in-line comments:**
* **// Strategy interface for tower targeting behavior**
* **public interface TargetingStrategy {**
* **Enemy selectTarget(List<Enemy> enemiesInRange, Object tower);**
* **String getStrategyName();**
* **}**
* **// Concrete strategy: Selects the first alive enemy in range**
* **public class FirstEnemyStrategy implements TargetingStrategy {**
* **@Override**
* **public Enemy selectTarget(List<Enemy> enemiesInRange, Object tower) {**
* **for (Enemy enemy : enemiesInRange) {**
* **if (enemy.isAlive()) {**
* **return enemy;**
* **}**
* **}**
* **return null;**
* **}**
* **@Override**
* **public String getStrategyName() { return "First"; }**
* **}**
* **// Factory for creating and managing strategies**
* **public class TargetingStrategyFactory {**
* **public enum StrategyType { FIRST, LAST, STRONGEST, WEAKEST }**
* **private static final TargetingStrategy FIRST\_ENEMY\_STRATEGY = new FirstEnemyStrategy();**
* **// ... other strategies**
* **public static TargetingStrategy createStrategy(StrategyType type) {**
* **switch (type) {**
* **case FIRST: return FIRST\_ENEMY\_STRATEGY;**
* **// ... other cases**
* **default: return FIRST\_ENEMY\_STRATEGY;**
* **}**
* **}**
* **}**

* **Explanation / role mapping:**
  + TargetingStrategy (interface):

Defines the contract for all targeting strategies. Any new targeting logic must implement this interface.

* FirstEnemyStrategy, StrongestEnemyStrategy, etc. (concrete strategies):

Each class implements a different targeting algorithm (e.g., first enemy, strongest enemy).

* TargetingStrategyFactory:

Centralizes the creation and management of strategy instances. Provides a way to get a strategy by type and ensures strategies are reused (singleton pattern for stateless strategies).

* Client (e.g., Tower):

Uses a TargetingStrategy to select which enemy to attack. The strategy can be swapped at runtime.

* **Benefits:**
  + Open/Closed Principle:

You can add new targeting strategies without modifying existing code.

* Flexibility:

Towers can change their targeting behavior at runtime (e.g., via player selection or upgrades).

* Encapsulation:

Each targeting algorithm is encapsulated in its own class, making the code easier to maintain and test.

* Reusability:

Strategies can be reused across different tower types or even other game entities.

* Separation of Concerns:

The logic for selecting a target is separated from the tower and enemy logic.

# 