IZMIR KÂTİP ÇELEBİ UNIVERSITY
Faculty of Engineering and Architecture
Department of Computer Engineering

Design Patterns Project
Online Multiplayer Bomberman Game

Prepared by:
Enes Türkmenoğlu
220401078

Supervisor:
Prof. Dr. Doğan AYDIN

December 2025

Contents

1 Project Overview

This project presents a multiplayer Bomberman game developed by effectively applying Object-Oriented Programming (OOP) principles and software Design Patterns. Built using the Unity game engine and C# programming language, the project reimagines classic arcade mechanics through a modern, scalable, and maintainable software architecture.

The primary technical objective is to demonstrate how abstract design concepts can be practically implemented in a real-time simulation environment to solve common software engineering problems.

The core gameplay revolves around strategic combat where players place bombs to destroy obstacles and eliminate opponents. The dynamic map system features three distinct wall types: Breakable walls that may yield items, Hard walls requiring multiple explosions, and Unbreakable blocks.

To enhance the competitive experience, players can collect various power-ups—such as speed boosts, explosion range extensions, and extra bomb capacity—spawned from destroyed crates. To provide visual and structural variety, the game offers three unique themes: Desert, Forest, and City, each dynamically generated using specific factory implementations.

The multiplayer infrastructure is built upon Unity Netcode for GameObjects, utilizing a Client-Server architecture to ensure synchronized gameplay across different clients. For data persistence and management, a local SQLite database system is integrated, enabling features such as secure user registration, authentication, and a competitive Leaderboard that tracks win/loss statistics.

The game challenges players not only against online human opponents but also against AI-controlled bots managed by distinct algorithms, including Static and Chasing behaviors. Following a "Last Man Standing" rule set, the final surviving player is declared the winner.

To ensure high code quality, modularity, and extensibility, various design patterns from Creational, Structural, and Behavioral categories have been rigorously implemented throughout the development process.

2 Software Architecture

During the development process, the MVP (Model-View-Presenter) architectural pattern was adopted to ensure code maintainability, manage complexity, and strictly avoid "Spaghetti Code" structures.

To solve the "God Class" problem frequently encountered in Unity projects—where a single script handles logic, visuals, and data—game logic, visual interface, and data layers have been completely isolated from one another. This architectural approach was rigorously applied to both the Player and Enemy entities, ensuring a consistent and modular codebase.

2.1 What is MVP Architecture?

Model-View-Presenter (MVP) is a derivative of the MVC (Model-View-Controller) pattern that separates the user interface (UI) from the business logic. The primary goal of this architecture is to apply the "Separation of Concerns" principle:

- **Model:** Represents the data and business rules. It is completely unaware of the View or the Unity Scene.
- **View:** The interface seen by the user. It is a Passive View; it makes no logical decisions, merely executing commands from the Presenter and forwarding user inputs or events to the Presenter.
- **Presenter:** Acts as the orchestrator (bridge) between the Model and the View. It receives input from the View, processes data using the Model, and reflects the results back to the View.

2.2 Implementation within the Project (Player and Enemy Modules)

In this project, the MVP pattern is integrated with the multiplayer network structure (Unity Netcode) and is structured as follows for both Player and Enemy characters:

1. **Model (PlayerModel.cs, EnemyModel.cs) - The Data Layer:** Definition: These are POCO (Plain Old C# Object) classes representing the runtime state of the entity. They do not inherit from Unity's MonoBehaviour, ensuring they do not carry unnecessary engine overhead. Function: They hold statistical data such as moveSpeed, activeBombs, and isDead for players, or moveDuration and collisionTolerance for enemies. They serve as the source of truth for network synchronization.
2. **View (PlayerView.cs, EnemyView.cs) - The Visual Layer:** Definition: These classes inherit from MonoBehaviour and are responsible solely for scene visualization. Function (Passive Structure): They do not make decisions regarding game logic. Input/Event Listening: For the Player, the Update loop listens for Input.GetAxis but does not move the character; it forwards this input to Presenter.OnMoveInput(). For the Enemy, it handles animation state changes sent by the server. Visualization: They execute commands received from the Presenter, such as updating Rigidbody2D velocity or triggering Animator parameters (e.g., PlayDeathAnimClientRpc).
3. **Presenter (PlayerPresenter.cs, EnemyPresenter.cs) - The Orchestrator:** Definition: The "brain" of the entity. It has access to both the Model data and the View component. It also inherits from NetworkBehaviour to manage networking operations. Function:
   - **Logic Processing:** For Players: It takes input from the View, calculates the final movement vector based on the Model's speed (netMoveSpeed), and commands the View to move (view.MovePlayer). For Enemies: It utilizes the State Pattern (IEnemyState) to calculate the next AI move and updates the View accordingly.
   - **Network Synchronization:** Since the game is multiplayer, the Presenter

synchronizes Model data across the server and clients using NetworkVariable. For example, when a Player picks up a ”Speed Power-up,” the Presenter processes this logic on the server and synchronizes the new speed to all client Views.

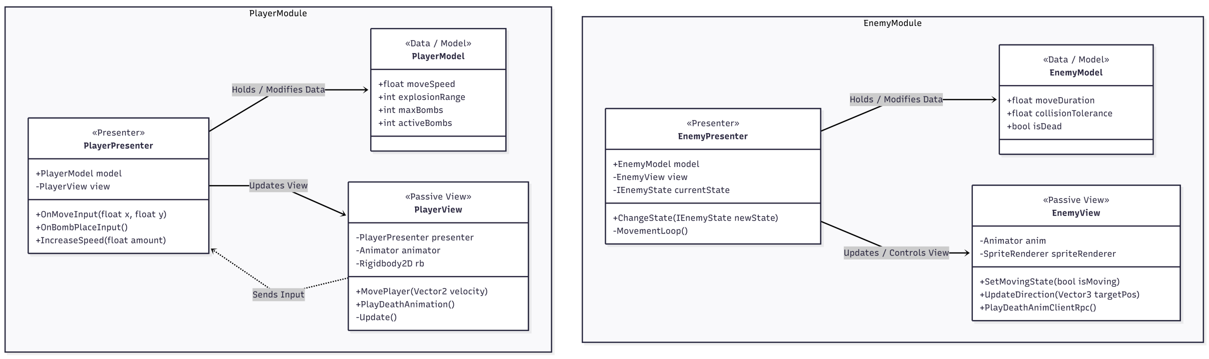

Figure 1: MVP Architecture Diagram: Player and Enemy Modules

3 Design Pattern Implementation

3.1 Creational Design Patterns

3.1.1 Abstract Factory Pattern

Definition: The Abstract Factory is a Creational design pattern that lets you produce families of related or dependent objects without specifying their concrete classes. This pattern enables client code to create objects through an interface, eliminating the need to know the specific types being instantiated.

Problem: Our project features distinct atmospheric themes: ”Desert”, ”Forest”, and ”City”. Each theme possesses a specific ”family of objects,” including unique ground tiles, breakable walls, hard blocks, and theme-specific enemy models. Using traditional approaches, such as `if (selectedTheme == "Desert")` or switch-case structures within the MapGenerator or EnemySpawner classes, led to several critical issues:

- **Tight Coupling:** The generator and spawner classes had to explicitly reference every single wall and enemy prefab in the game.
- **Violation of the Open/Closed Principle:** Adding a new theme (e.g., ”Winter Theme”) required modifying the existing, functional code to add new conditional blocks. This made the system fragile and prone to errors.
- **Risk of Inconsistency:** Complex conditional logic increased the chance of visual inconsistencies, such as accidentally placing a City wall on a Desert ground.

Solution and Implementation: To resolve these issues, the Abstract Factory pattern was implemented in conjunction with Unity’s ScriptableObject architecture:

- **Abstract Factory (ThemeFactory):** An abstract class named ThemeFactory was established to define the common interface for all themes. It declares abstract methods such as GetGround(), GetBreakableWall(), GetHardWall(), GetUnbreakableWall(), and GetEnemyPrefab(), without specifying how they are created.

- **Concrete Factories (DesertThemeFactory, ForestThemeFactory, CityThemeFactory):** Specific factory classes were derived from this abstract base for each theme. For instance, DesertThemeFactory returns cactus and sand objects, while CityThemeFactory returns concrete and brick objects.
- **Client Integration (MapGenerator & EnemySpawner):** The pattern is consumed by two primary clients:
  - **MapGenerator:** Acts as a client to generate the static environment. It requests assets using the abstract reference (e.g., theme.GetGround()) to build the level.
  - **EnemySpawner:** Acts as a client to populate the dynamic actors. It calls theme.GetEnemyPrefab() to instantiate the correct enemy type for the active level.

Through polymorphism, both clients automatically receive the correct visual assets based on the active factory configuration, operating independently of the concrete theme data. This structure allows new themes to be added to the game without changing a single line of existing code in the generator or spawner classes; creating a new factory class is sufficient.

Relevant Classes: ThemeFactory.cs (Abstract Factory), DesertThemeFactory.cs, CityThemeFactory.cs, ForestThemeFactory.cs (Concrete Factories), MapGenerator.cs (Client), EnemySpawner.cs (Client).

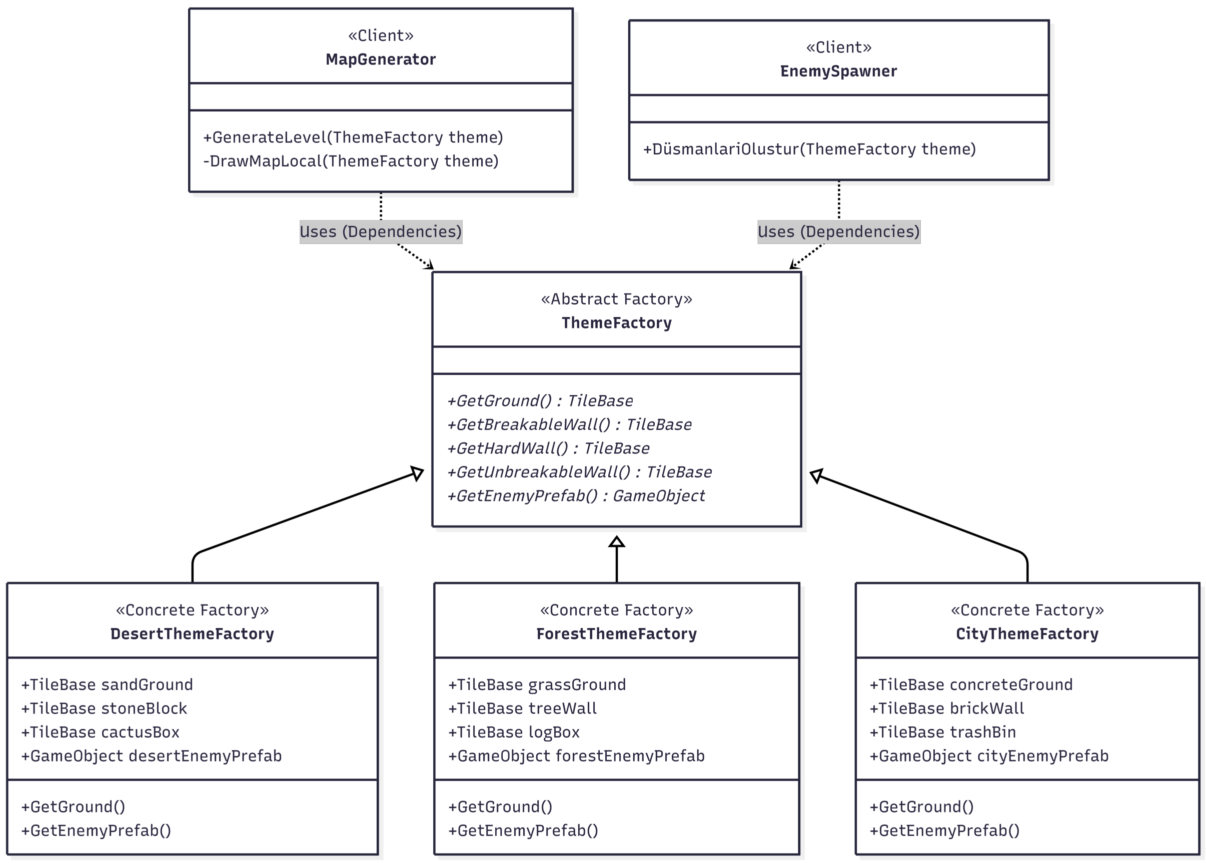

Figure 2: Abstract Factory Pattern Implementation

3.1.2 Singleton Pattern

Definition: The Singleton Pattern is a Creational design pattern that guarantees a class has only one instance at runtime and provides a global point of access to that instance. In game development, it is commonly used for central systems such as game managers, audio systems, or network connection modules.

Problem: The project requires a centralized mechanism to manage the overall game state (e.g., tracking the count of living enemies, win/loss conditions), network connections, and database operations. Implementing this mechanism presented specific challenges:

- **Data Persistence:** In Unity, objects are destroyed by default during scene transitions (e.g., moving from the Menu to the Game scene). Critical game data, such as the active user's credentials (currentUser) and scores, must be preserved across these transitions.
- **Access Complexity:** Various objects scattered across the map (such as an EnemyPresenter reporting a death) or UI panels (GameUIManager) need to communicate with the game manager. Manually dragging and dropping references to the manager for every object leads to "Dependency Hell" and unmanageable code as the project scales.
- **Risk of Multiple Instances:** Accidentally creating multiple game managers could lead to logic conflicts, such as maintaining two different scoreboards or game states simultaneously.

Solution and Implementation: To address these issues, the GameManager class was designed according to Singleton principles:

- **Global Access (static Instance):** A public `static GameManager Instance` { `get; private set;` } property is defined within the class. This allows any script in the project to access the manager (e.g., calling `GameManager.Instance.RegisterEnemy()`) without requiring a direct reference link.
- **Uniqueness Control (Awake):** Within the Awake method, the system checks if the Instance variable is already assigned. If a manager instance already exists, the newly created object self-destructs using `Destroy(this.gameObject)`. This guarantees that only one manager exists at any given time.
- **Persistence (DontDestroyOnLoad):** The `DontDestroyOnLoad(this.gameObject)` command is used to prevent the GameManager object from being destroyed when loading new scenes. This ensures that user data (currentUser) and database connections (userRepo) initialized in the login screen remain available during gameplay.

Relevant Classes: GameManager.cs.

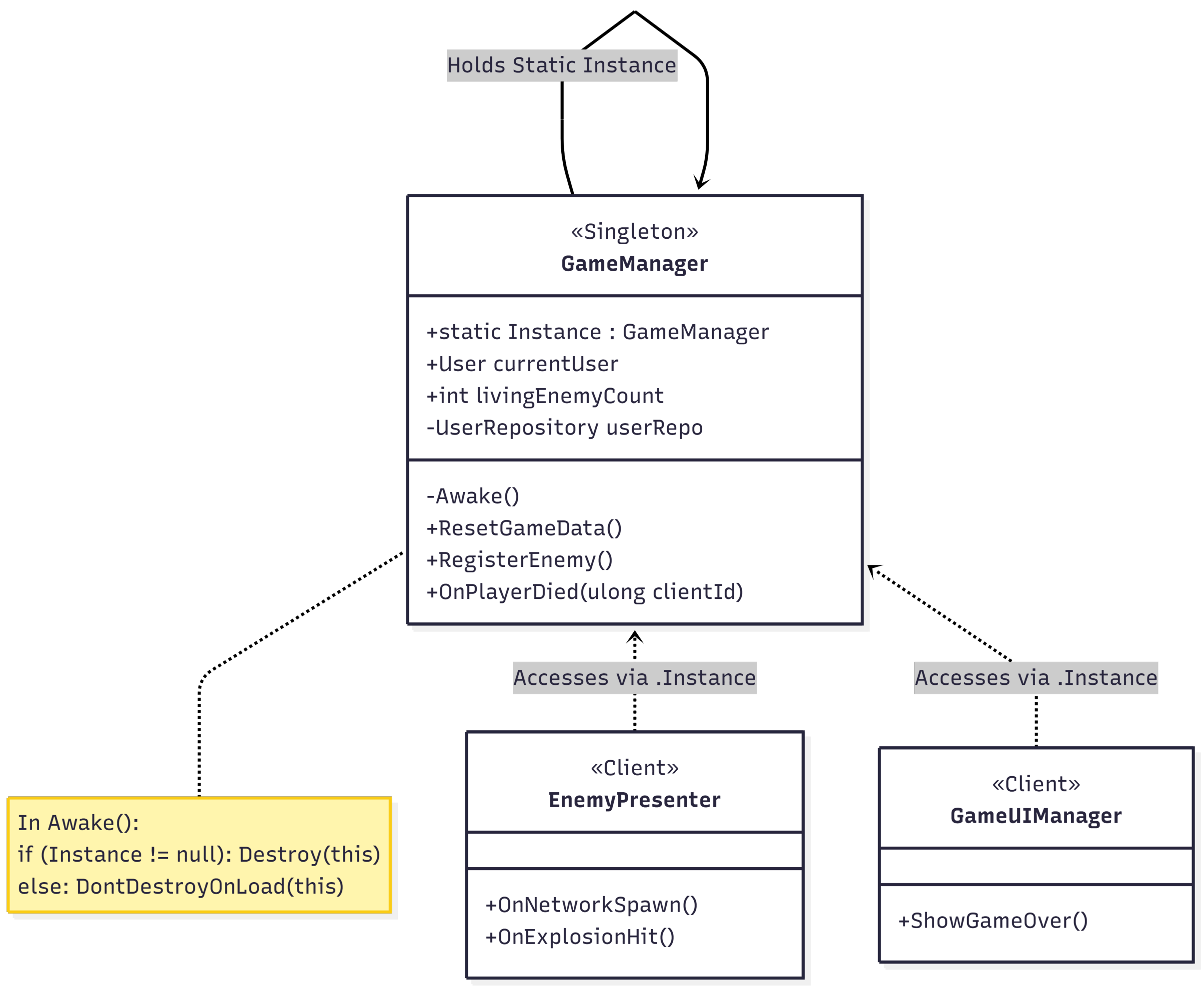

Figure 3: Singleton Pattern Implementation: GameManager

3.2 Structural Patterns

3.2.1 Facade Pattern

Definition: The Facade Pattern is a Structural design pattern that provides a simplified interface to a complex set of subsystems. It prevents client classes from getting lost in the intricate details of the system and reduces the coupling between the subsystems and the client.

Problem: Starting a multiplayer game session involves much more than simply loading a scene; it requires a sequence of interdependent operations to be executed in a specific order without errors. In our project, pressing the "Start Game" button triggers the following complex workflow:

- **Network Initialization:** Establishing a Host or Client connection via NetworkManager.
- **Data Reset:** Clearing old game states (e.g., enemy counts) within the GameManager.
- **Theme Configuration:** Synchronizing the selected visual theme across the network using ThemeNetworkManager.

- **Map and Entity Generation:** Ensuring that enemies (EnemySpawner) are instantiated only after the map (MapGenerator) has been fully constructed to avoid synchronization issues.
- **Character Positioning:** Moving the local player to the correct spawn point immediately after their network object is created.
- **UI Management:** Hiding the menu panel to reveal the gameplay view.

If all this logic were implemented directly within the UI class (ThemeUIManager), the UI layer would become tightly coupled to every core system (Network, Spawner, Map, GameManager). This would lead to "Spaghetti Code," significantly reducing readability and maintainability.

Solution and Implementation: To manage this complexity, the GameStarterFacade class was developed:

- **Single Point of Access:** The ThemeUIManager (Client) is unaware of the complex background operations. It simply calls `facade.StartGame(theme)` or `facade.JoinGame()`.
- **Process Orchestration:** GameStarterFacade coordinates the subsystems (NetworkManager, EnemySpawner, MapGenerator) in the correct order. For example, it utilizes the SpawnEnemiesWithDelay coroutine to introduce a slight delay, ensuring the map is fully loaded before enemies are spawned, thus resolving race conditions.
- **Decoupling:** The UI layer now only recognizes the Facade class. Changes made to the subsystems (e.g., modifying how enemies spawn) do not affect the UI code.

Relevant Classes: GameStarterFacade.cs (Facade), ThemeUIManager.cs (Client), EnemySpawner.cs, MapGenerator.cs (Subsystems).

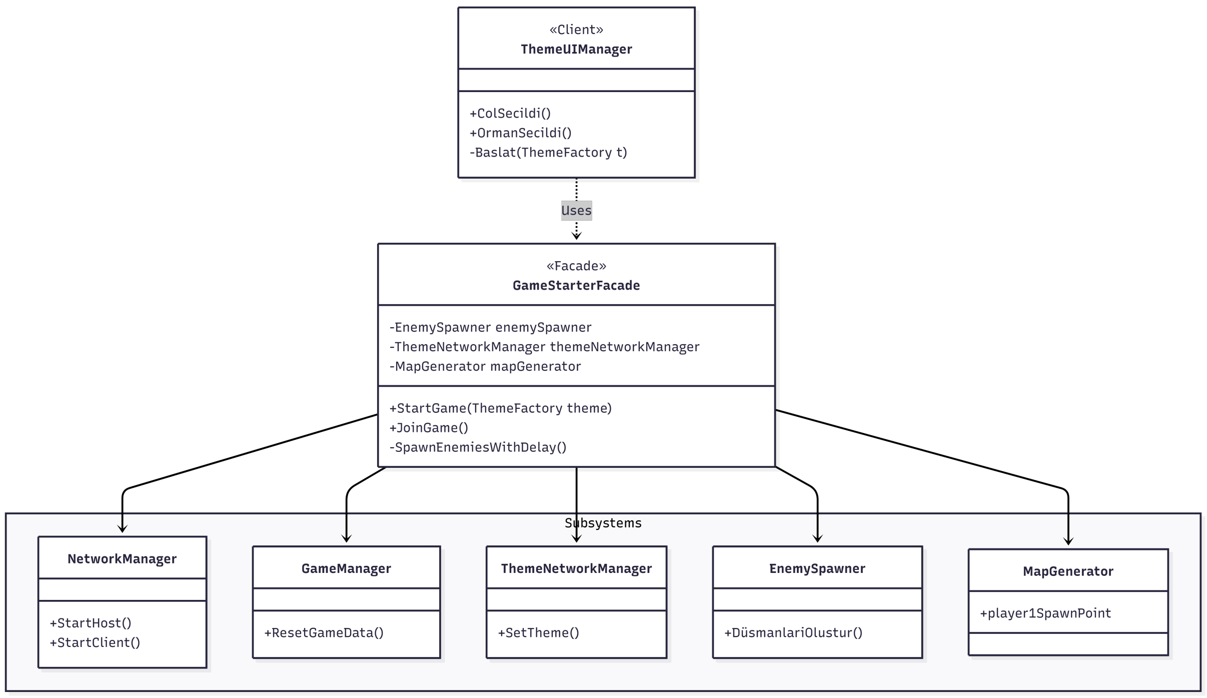

Figure 4: Facade Pattern Implementation

3.3 Behavioral Patterns

3.3.1 State Pattern

Definition: The State Pattern is a Behavioral design pattern that allows an object to alter its behavior when its internal state changes at runtime. By encapsulating state-specific behaviors into separate classes, this pattern reduces the complexity of the main context class and supports the Open/Closed Principle—making the system open for extension but closed for modification.

Problem: The project requires the Enemy AI to make dynamic decisions based on the environment. By default, enemies should roam the map randomly (Wander). However, if they detect a player within a certain range (5 units), they must switch behavior to pursue the player (Chase). If the player moves out of range, they should revert to wandering.

Managing this logic using traditional methods would result in complex, nested if-else or switch-case structures within the EnemyPresenter class's movement loop. Example of problematic logic: `if (distance < 5) { Chase(); } else { Wander(); }`. This approach makes the code brittle and difficult to maintain, as adding a new behavior (e.g., "Attack" or "Flee") would require modifying the core logic of the enemy class every time.

Solution and Implementation: To address this, the State Pattern was implemented to manage enemy behaviors:

- **Abstraction (IEnemyState):** The actions an enemy can perform—specifically Enter, Exit, and CalculateNextMove—are defined in a common interface.
- **Concrete States:** Each behavior is encapsulated in its own class:
  - **WanderState:** Finds a random valid position to move to. If a player is detected within 5.0 units, it triggers a state transition by calling `enemy.ChangeState(new ChaseState())`.
  - **ChaseState:** Calculates the path toward the nearest player. If the player exceeds the distance of 7.0 units, it transitions back to WanderState.
- **Context (EnemyPresenter):** The EnemyPresenter class maintains a reference to the currentState. When it needs to move, it simply calls `currentState.CalculateNextMove(this` delegating the decision-making process to the state object. The Presenter knows that it should move, but it does not know how or where—that logic is fully decoupled.

This structure allows for high extensibility; adding a new "Attack Mode" would only require creating a new AttackState class, without needing to modify the existing EnemyPresenter code.

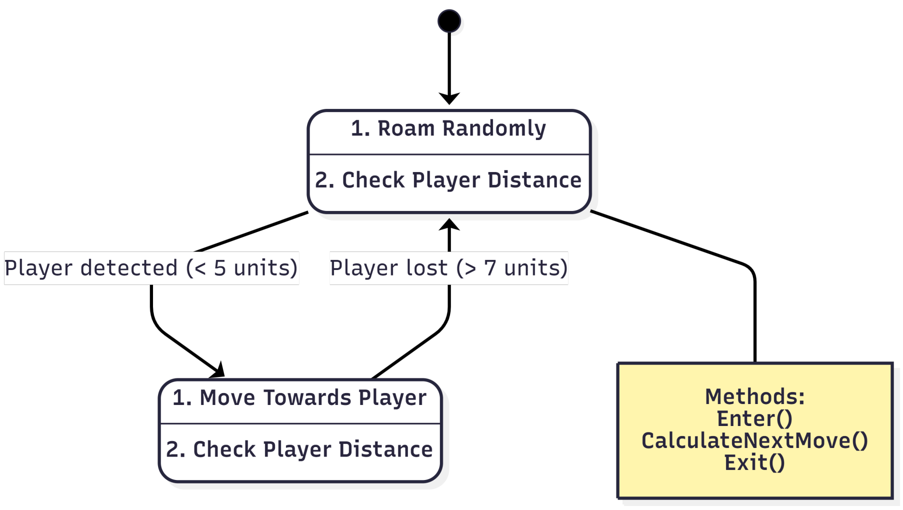

Figure 5: Detailed State Transition Logic

Relevant Classes: IEnemyState.cs (State Interface), EnemyStates.cs (Concrete States), EnemyPresenter.cs (Context).

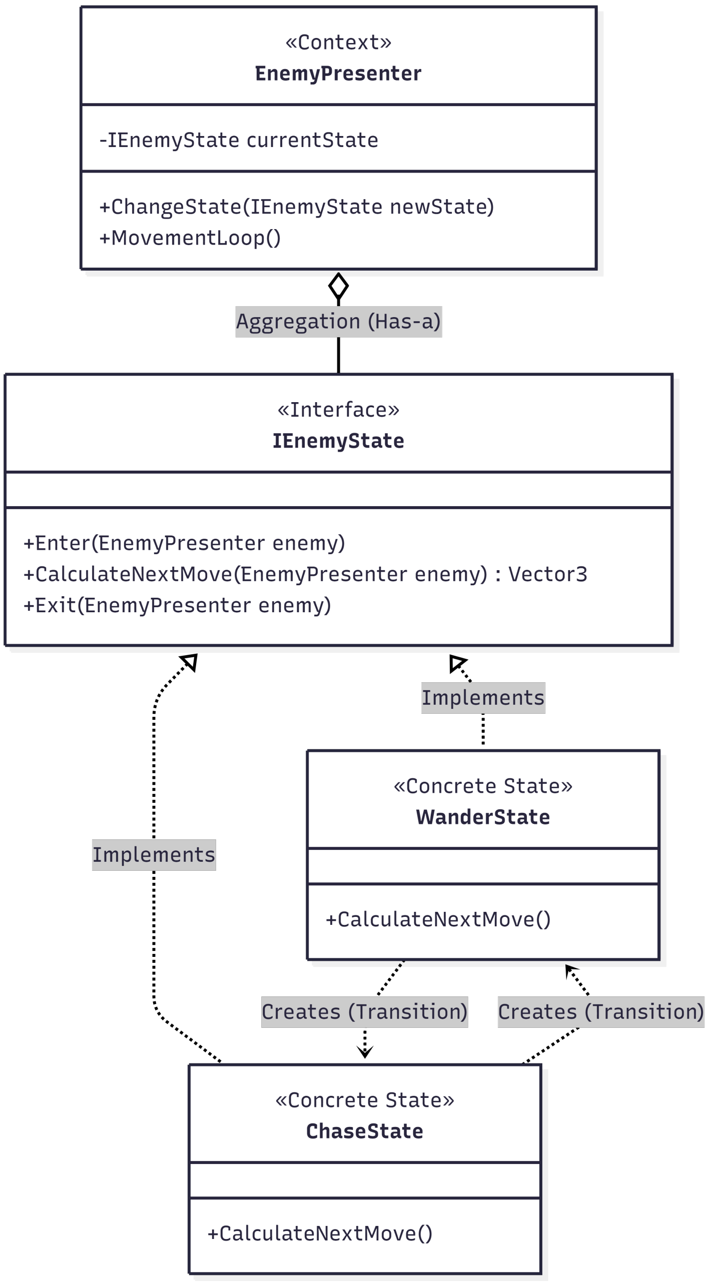

Figure 6: State Pattern Implementation: Enemy AI

3.3.2 Observer Pattern

Definition: The Observer Pattern is a Behavioral design pattern that defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. In game development, it is frequently used for event handling and decoupled interaction systems.

Problem: The core mechanic of the game involving explosions requires the Bomb logic to interact with various distinct objects on the map. When a bomb explodes, specific consequences must occur for different entities:

- Players (PlayerPresenter) must take damage and die.

- Enemies (EnemyPresenter) must be destroyed.
- Walls (MapDestructionSystem) must be broken or degraded.

Without the Observer Pattern, the classes managing the explosion (specifically Bomb and ExplosionArea) would need to explicitly reference every single interactable class in the game. This would result in a structure using hard-coded type checks: `if (obj == Player) KillPlayer(); else if (obj == Enemy) KillEnemy();`. This creates Tight Coupling. Adding a new destructible object (e.g., a "Wooden Crate") would require modifying the core explosion logic, violating the Open/Closed Principle.

Solution and Implementation: To eliminate this dependency, an Interface-based Observer approach was implemented using IExplosionObserver. The implementation consists of three parts:

- **The Contract (Interface):** A common interface named IExplosionObserver was defined, declaring the `OnExplosionHit(Vector3Int gridPosition)` method. This serves as a contract for any object that claims, "I can be affected by an explosion."
- **The Subscribers (Concrete Observers):** Classes that need to react to explosions—PlayerPresenter, EnemyPresenter, and MapDestructionSystem—implement this interface. Each class defines its own unique response logic (e.g., the Player triggers a death animation, the Map removes a tile).
- **The Notification (Evolution of Logic):** Initially, the notification logic was centralized solely within Bomb.cs. However, to resolve gameplay flaws where actors entering the explosion late were not damaged, the notification responsibility was split:
  - **Bomb.cs (Instant/Static Notification):** Handles immediate grid-based calculations at t=0 to destroy static walls.
  - **ExplosionArea.cs (Continuous/Dynamic Notification):** Utilizes Unity's physics engine (OnTriggerEnter2D) to detect collisions throughout the duration. This ensures that any dynamic actor entering the fire zone is correctly notified via the observer interface.

This hybrid approach ensures that the explosion system can interact with any object implementing the interface without knowing its concrete type, ensuring the system remains modular.

Relevant Classes: IExplosionObserver.cs (Interface), Bomb.cs, ExplosionArea.cs (Notifiers), PlayerPresenter.cs, MapDestructionSystem, EnemyPresenter.cs (Observers).

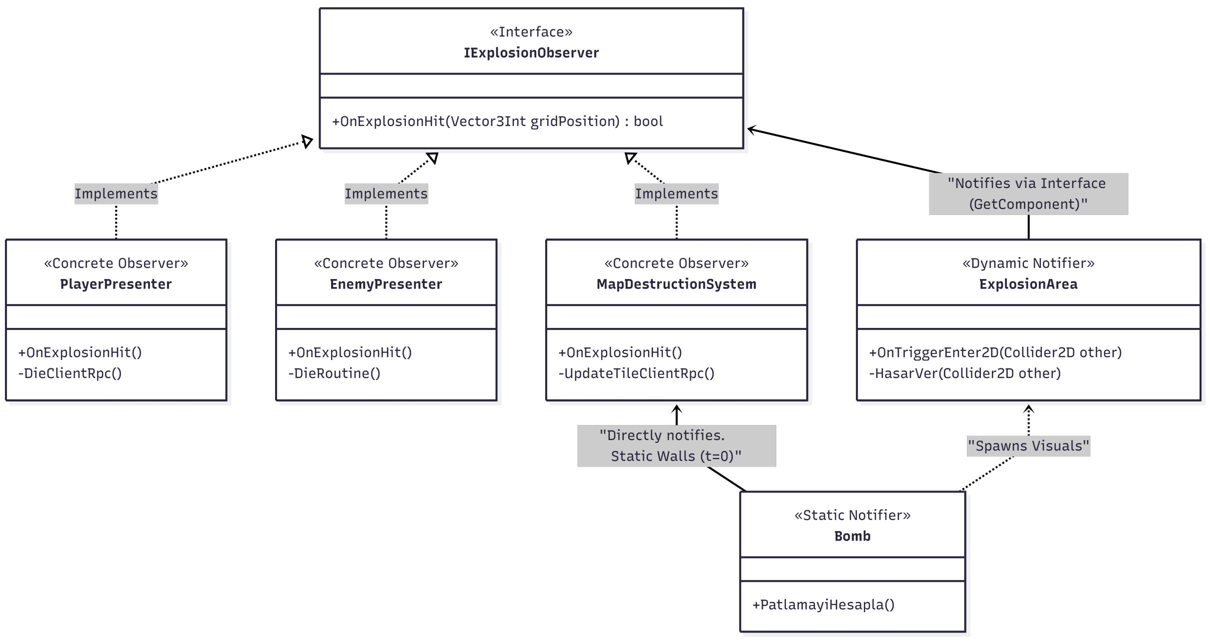

Figure 7: Observer Pattern Implementation

3.3.3 Strategy Pattern

Definition: The Strategy Pattern is a Behavioral design pattern that defines a family of algorithms, encapsulates each one into a separate class, and makes them interchangeable. This pattern allows the algorithm (business logic) to vary independently from the client classes that use it.

Problem: The game features various power-ups that provide players with advantages, such as ”Speed Boost,” ”Explosion Range Increase,” and ”Extra Bomb Capacity.” Using a traditional approach, this logic would be managed within the PowerUpItem class using large if-else or switch-case blocks. This approach violates the Open/Closed Principle. Adding a new power-up type (e.g., ”Invisibility”) would require modifying the already functional and tested PowerUpItem class, introducing the risk of potential bugs.

Solution and Implementation: To resolve this issue, each power-up effect has been encapsulated as a separate ”Strategy” class:

- **Strategy Interface (IPowerUpStrategy):** A common interface was defined containing a single method: `Apply(PlayerPresenter player)`.
- **Concrete Strategies:**
  - **SpeedBoostStrategy:** Temporarily increases the player’s movement speed.
  - **RangeBoostStrategy:** Permanently increases the explosion range of the player’s bombs.
  - **BombCountStrategy:** Increases the maximum number of bombs a player can place simultaneously.
- **Context (PowerUpItem):** The power-up object itself does not know the specifics of the strategy it carries. It simply holds a reference of type IPowerUpStrat-

egy. When a player collides with the item, it calls the `_strategy.Apply(player)` method.

Through this structure, the logic for each power-up is completely abstracted from the PowerUpItem class. Adding a new power-up does not require touching the PowerUpItem class; creating a new strategy class is sufficient.

Relevant Classes: IPowerUpStrategy.cs (Interface), SpeedBoostStrategy.cs, RangeBoost-Strategy.cs, BombCountStrategy.cs, PowerUpItem.cs (Context).

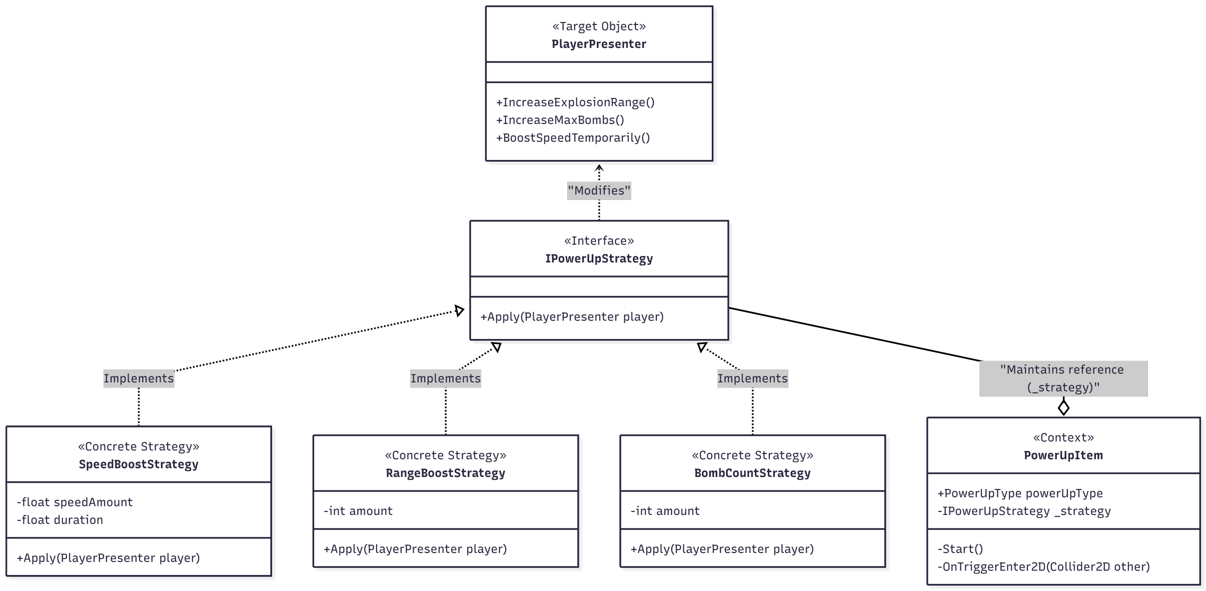

Figure 8: Strategy Pattern Implementation: Power-Ups

4 Data Management and Repository Pattern

To ensure the persistence of game data and decouple data access logic from the core game mechanics, the Repository Design Pattern was implemented utilizing SQLite as the local database solution.

4.1 Data Persistence and SQLite

Instead of a heavy server-based relational database system, a file-based SQLite solution was chosen due to its portability and lightweight nature. The database file (game_data.db) is generated dynamically at runtime, with the file path adjusted based on the platform (Application.dataPath) to ensure compatibility between the Unity Editor and standalone builds.

The data structure is modeled using the User class. This class acts as a Data Transfer Object (DTO), serving as the C# representation of the Users table in the database. It encapsulates properties such as Username, Password, Wins, and Losses.

4.2 Repository Pattern Implementation

Definition: The Repository Pattern is a design pattern that isolates data access logic (SQL queries, connection management) from the business logic, allowing database operations to be managed through a central interface class.

Problem: During development, various data operations were required, such as user registration (Register), authentication (Login), score updates (AddWin), and leaderboard retrieval (GetTopPlayers). If raw SQL queries were embedded directly into game classes like GameManager or ThemeUIManager, it would lead to:

- **Code Duplication:** Boilerplate code for opening and closing database connections would be repeated throughout the project.
- **Security Risks:** Concatenating strings for queries would create vulnerabilities to SQL Injection attacks.
- **Maintenance Difficulty:** Any change in the database schema would require scanning and refactoring the entire codebase.

Solution and Implementation: To address these issues, all data access logic was encapsulated within the UserRepository class:

- **Abstraction:** Higher-level layers (e.g., ThemeUIManager) are agnostic to the underlying SQL implementation. They simply invoke methods like `userRepo.Login(username, password)`.
- **Security (SQL Injection Protection):** User inputs are never inserted directly into query strings. Instead, parameterized queries are used via `cmd.Parameters.AddWithValue ("@u", username)`.
- **Resource Management:** Database connections are wrapped in using blocks. This ensures that connections are automatically closed and disposed of immediately after an operation is completed.
- **Object Mapping:** Raw data rows retrieved from the database are converted into meaningful User objects using the ParseUser helper method before being passed to the game layer.

Relevant Classes: UserRepository.cs (Repository), User.cs (Model/DTO).

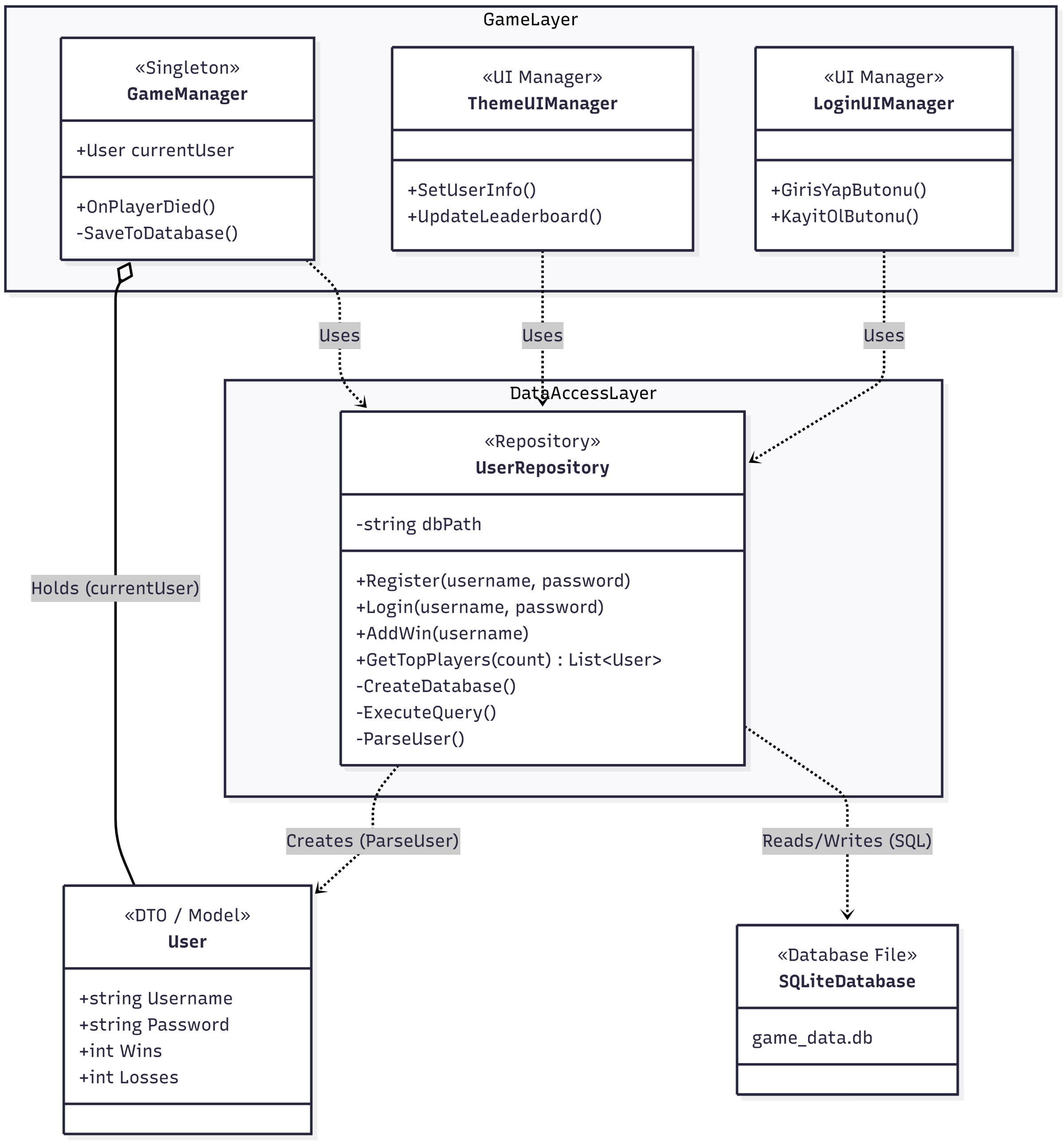

Figure 9: Repository Pattern Implementation

5 Conclusion

This project successfully demonstrates the development of a real-time multiplayer Bomberman game by rigorously applying advanced software engineering principles. By moving away from the ”Spaghetti Code” structures often encountered in game development, a clean, scalable, and testable architecture was achieved through the MVP (Model-View-Presenter) pattern.

Throughout the development process, the implementation of Design Patterns proved essential in solving specific architectural challenges:

- **Scalability:** The Abstract Factory pattern enabled the seamless addition of new visual themes (Desert, Forest, City) without modifying core generation logic.

- **Manageability:** The Facade and Singleton patterns centralized complex system operations, such as network initialization and game state management, reducing dependency chaos.
- **Dynamic Behavior:** The State, Observer, and Strategy patterns allowed for sophisticated AI behaviors, decoupled event handling for explosions, and a modular power-up system that adheres to the Open/Closed Principle.
- **Data Integrity:** The Repository Pattern ensured secure and efficient data management using SQLite, effectively separating database concerns from the game logic.

In conclusion, this project fulfills its primary objective by proving that abstract design concepts and strict architectural patterns can be practically implemented in a dynamic simulation environment (Unity), resulting in a codebase that is not only functional but also maintainable and ready for future expansion.