Bomberman

Ceren Tolunay-220401077, Halil Abacı-220401025

December 2025

1 Introduction

1.1 Project Motivation

Modern software systems require scalable and maintainable architectures, especially in game development where gameplay mechanics, AI behaviors, and user interactions frequently evolve. This project is motivated by the need to demonstrate how design patterns can be effectively applied in a real-world software system to manage complexity and reduce tight coupling. A Bomberman-style game was chosen as it naturally supports modular components such as map generation, enemy behaviors, power-ups, and game states.

1.2 Objectives of the Study

The objective of this study is to design and implement an online multiplayer Bomberman game using a clean and extensible architecture based on object-oriented design principles and design patterns. The project aims to improve code maintainability, flexibility, and reusability while demonstrating the practical use of multiple creational, structural, and behavioral patterns.

1.3 Scope of the Project

This project covers the development of a two-player online Bomberman game implemented in Unity using C#. It includes core gameplay mechanics, multiplayer synchronization, and database-backed user management. The focus is on architectural design and pattern usage rather than creating a full commercial game.

2 Project Overview

2.1 Game Concept

The project is based on a Bomberman-style game that follows the core principles of the classic Bomberman gameplay. Players navigate a grid-based map, place bombs to destroy obstacles, defeat enemies, and eliminate opponents. The game environment consists of different wall types, including unbreakable, breakable, and hard walls, as well as collectible power-ups that enhance player abilities. The overall design aims to preserve the original Bomberman experience while providing a flexible and extensible software architecture.

2.2 Core Gameplay Mechanics

The core gameplay mechanics include player movement on a tile-based map, bomb placement with a fixed explosion delay, and explosion propagation in four directions. Explosions interact with the environment by destroying breakable walls, damaging hard walls, and eliminating players or enemies caught in their range. Power-ups may appear after walls are destroyed and can increase attributes such as bomb count, bomb power, or player speed. Enemy characters exhibit different movement behaviors, contributing to the overall challenge of the game.

2.3 Single-Player and Multiplayer Modes

The game supports both single-player and online multiplayer modes. In single-player mode, the player competes against computer-controlled enemies with varying behaviors. In multiplayer mode, two players connect using a host–client architecture and compete on the same map in real time. Player actions such as movement, bomb placement, explosions, and game state transitions are synchronized across the network to ensure a consistent gameplay experience for all participants.

3 Technology Stack

3.1 Game Engine

The game is developed using the Unity game engine, which provides a robust framework for 2D game development, physics handling, scene management, and cross-platform support. Unity's component-based architecture enables modular design and integrates naturally with object-oriented programming principles.

3.2 Programming Language

The project is implemented in C#, the primary scripting language supported by Unity. C# was chosen due to its strong object-oriented features, such as inheritance, interfaces, and polymorphism, which are essential for implementing design patterns in a clear and structured manner.

3.3 Networking Technology

Online multiplayer functionality is implemented using a host–client networking model. The networking layer is responsible for synchronizing player movements, bomb placement, explosions, and game state transitions across connected clients. This approach ensures consistent gameplay behavior while maintaining authoritative control on the host side.

3.4 Database Technology

The project uses SQLite as the database management system for storing user information and game statistics. SQLite was selected due to its lightweight nature, ease of integration, and suitability for small to medium-scale applications. Database access is abstracted using the Repository pattern to maintain separation between game logic and data persistence.

Table 1: Technology Stack Overview

Component	**Technology Used**
Game Engine	Unity
Programming Language	C#
Networking Model	Host–Client Architecture
Database System	SQLite

4 Software Architecture

4.1 Overall System Architecture

The project is designed as a modular game system where core gameplay logic, user interaction, networking, and data persistence are separated into distinct responsibilities. At runtime, the game loop and scene management are coordinated by central manager components, while gameplay entities such as players, bombs, walls, enemies, and power-ups are implemented as independent components. Multiplayer communication is handled by the networking layer, which synchronizes critical game actions and state transitions between the host and the client. User accounts and match statistics are stored using a lightweight local database, accessed through an abstraction layer to prevent direct coupling between gameplay code and persistence logic.

4.2 Layered / MVC-Inspired Architecture

Although Unity uses a component-based paradigm, the project follows an MVC-inspired layered architecture to improve maintainability and support design pattern integration. The system is structured around three conceptual layers: *Model*, *Controller*, and *View*. This separation helps keep the game rules and data structures independent from input handling and presentation logic.

4.2.1 Model Layer

The Model layer represents the game data and rules that define the domain. It includes data structures and abstractions such as player statistics, wall types, power-up definitions, match results, and database entities. The Model layer is designed to be as independent as possible from Unity-specific UI or input concerns, enabling easier extension and testing of the game rules.

4.2.2 Controller Layer

The Controller layer contains the runtime behavior and decision-making components. It is responsible for interpreting player inputs, executing commands, controlling enemy behaviors, managing bomb placement and explosion logic, and coordinating state transitions during gameplay. This layer is also where most behavioral patterns are applied (e.g., Command for input handling, Strategy for enemy AI, and State for game flow), allowing the system to evolve without introducing large conditional structures.

4.2.3 View Layer

The View layer is responsible for presentation and user interaction. It includes Unity scenes, UI panels, visual prefabs, animations, and feedback elements such as menus, HUD components, and game-over screens. The View components display the current game state and provide the interface through which players interact with the system, while the underlying game logic remains within the Model and Controller layers.

4.3 Architectural Decisions

Several architectural decisions were made to reduce complexity and improve extensibility: (i) responsibilities were separated into layers to avoid tight coupling between gameplay logic and UI, (ii) game features that change frequently (enemy behavior, power-ups, map generation, and input handling) were designed around interchangeable abstractions, (iii) persistence and authentication were isolated behind repository-style interfaces, and (iv) multiplayer synchronization was structured to ensure consistent outcomes across networked instances by synchronizing key gameplay actions and state transitions. These decisions collectively support clean integration of multiple design patterns and facilitate future improvements without major refactoring.

5 Design Patterns Used

5.1 Factory Pattern (Abstract Factory / Factory Method)

5.1.1 Problem Definition

The game includes multiple themes (e.g., Desert, Forest, City) and at least three wall types (unbreakable, breakable, and hard). A direct implementation that creates walls using concrete prefabs or hard-coded references causes tight coupling between the map generation logic and theme-specific assets. As the number of themes and wall variations increases, the code becomes harder to maintain and extending the system requires modifications across multiple files.

5.1.2 Solution Approach

To decouple theme-specific wall creation from the map generation logic, the Factory approach is applied. An abstract factory interface defines a common set of creation methods for wall types. Concrete factories implement these methods for each theme and return the appropriate prefabs or wall objects. As a result, the map generation layer depends only on abstractions (interfaces), not on concrete theme implementations, enabling easy theme switching and future extension.

5.1.3 Factory Structure for Themes and Walls

The solution is structured around an `IWallFactory` abstraction that provides factory methods such as `CreateUnbreakableWall()`, `CreateBreakableWall()`, and `CreateHardWall()`. Theme-specific factories (e.g., `DesertWallFactory`, `ForestWallFactory`, `CityWallFactory`) implement this interface and encapsulate all theme-dependent creation details.

The map generation component selects a factory based on the current theme and uses it to build the map. This design allows adding a new theme by introducing a new factory class without changing existing map generation logic (Open/Closed Principle).

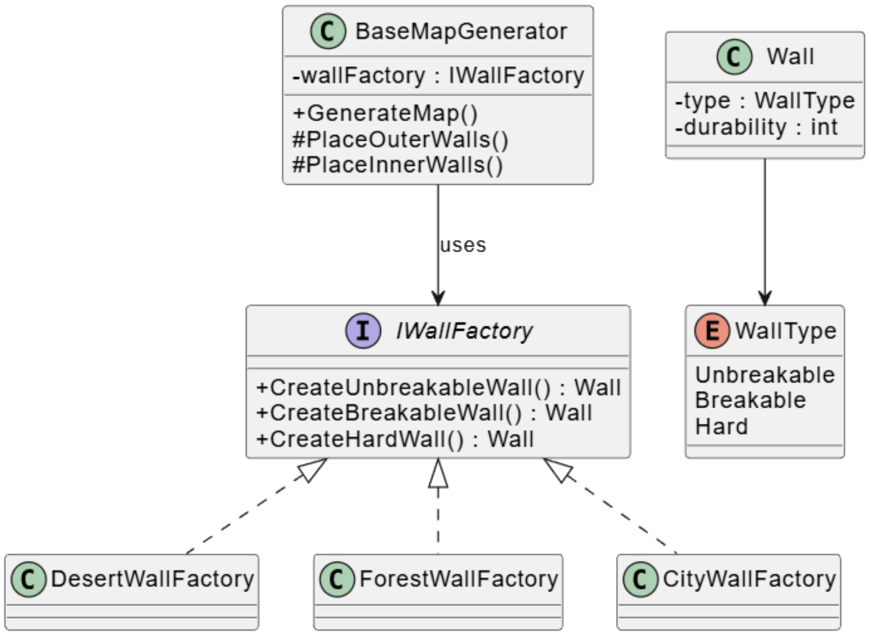

Figure 1: Abstract Factory / Factory Method structure for theme-based wall creation.

5.2 Decorator Pattern

5.2.1 Problem Definition

In the game, players can collect multiple power-ups such as speed boost, increased bomb power, and increased bomb count. A straightforward implementation that directly modifies player variables (e.g., `speed += 1`, `bombCount++`) quickly leads to scattered logic and conditional complexity. Moreover, stacking multiple power-ups over time becomes harder to manage, and adding a new power-up requires changing existing player code, which increases coupling and reduces maintainability.

5.2.2 Player Stats Extension with Power-Ups

To support flexible and stackable upgrades, the player attributes are represented through a statistics abstraction (e.g., speed, bomb power, bomb count). When a power-up is collected, the current statistics object is wrapped by a new decorator instance that modifies only the relevant attribute. This approach allows combining multiple power-ups dynamically during gameplay without altering the base player implementation.

5.2.3 Decorator Structure

The solution is built on an `IPlayerStats` interface that defines the available player attributes. A `BasePlayerStats` class provides the default values. A shared `PlayerStatsDecorator` class implements `IPlayerStats` and forwards calls to a wrapped `IPlayerStats` instance. Concrete decorators such as `SpeedBoostDecorator`, `BombPowerDecorator`, and `BombCountDecorator` override only the relevant attribute calculations.

A holder component (e.g., `PlayerStatsHolder`) maintains the current `IPlayerStats` reference and applies decorators when power-ups are collected.

5.2.4 UML Class Diagram

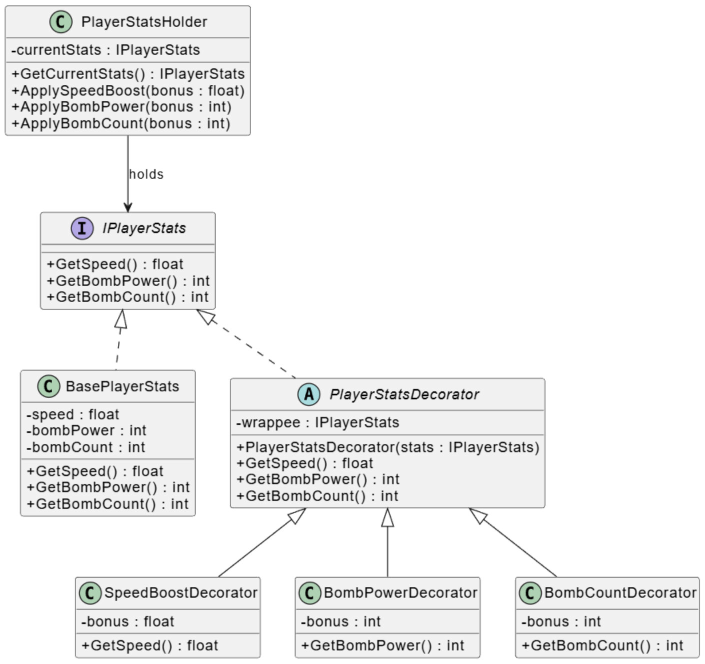

Figure 2: Decorator Pattern structure for extending player statistics with stackable power-ups.

5.3 Strategy Pattern

5.3.1 Problem Definition

In the game, enemies are required to exhibit different movement and decision-making behaviors depending on the difficulty level and game mode. A naive implementation that embeds enemy behavior directly into the enemy controller using conditional statements (e.g., `if` or `switch`) leads to rigid and hard-to-maintain code. Adding a new enemy behavior would require modifying existing logic, increasing coupling and violating the Open/Closed Principle.

5.3.2 Enemy AI Behaviors

The game includes multiple enemy behavior types, such as randomly moving enemies and enemies that actively chase the player. These behaviors differ only in their decision-making logic, while sharing the same movement execution structure. Therefore, enemy behavior is treated as a replaceable algorithm that can be selected dynamically based on the game state or difficulty level.

5.3.3 Strategy Implementations

To address this problem, the Strategy Pattern is applied by defining a common interface for enemy movement behavior. Each concrete strategy encapsulates a specific AI behavior and implements the same interface. The enemy controller holds a reference to a strategy object and delegates movement decisions to it. This design allows changing enemy behavior at runtime and adding new behaviors without modifying the enemy controller itself.

5.3.4 UML Class Diagram

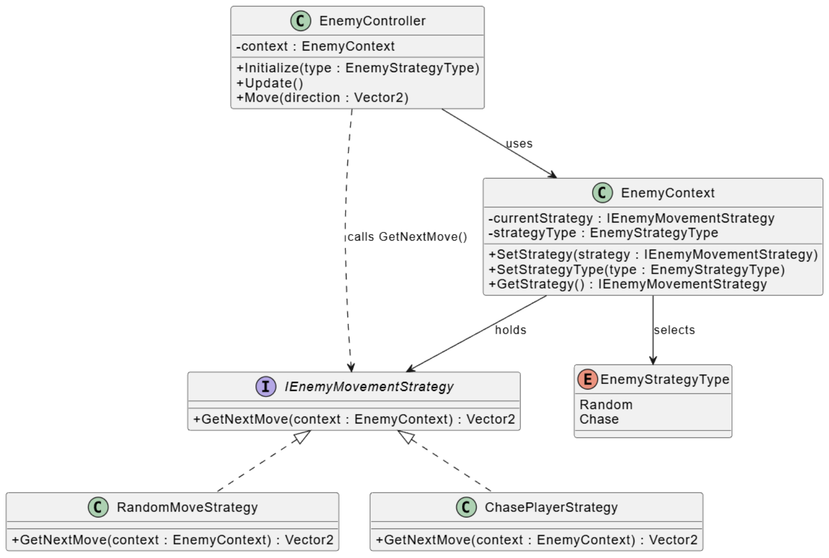
EnemyController
-context : EnemyContext
+Initialize(type : EnemyStrategyType)
+Update()
+Move(direction : Vector2)
uses
calls GetNextMove()
EnemyContext
-currentStrategy : IEnemyMovementStrategy
-strategyType : EnemyStrategyType
+SetStrategy(strategy : IEnemyMovementStrategy)
+SetStrategyType(type : EnemyStrategyType)
+GetStrategy() : IEnemyMovementStrategy
holds
selects
IEnemyMovementStrategy
+GetNextMove(context : EnemyContext) : Vector2
EnemyStrategyType
Random
Chase
RandomMoveStrategy
+GetNextMove(context : EnemyContext) : Vector2
ChasePlayerStrategy
+GetNextMove(context : EnemyContext) : Vector2

Figure 3: Strategy Pattern structure for modular enemy movement behaviors.

5.4 State Pattern

5.4.1 Problem Definition

Game flow in a Bomberman-style game involves multiple distinct phases such as the main menu, active gameplay, pause, and game over. A naive implementation typically manages these phases using large conditional blocks (e.g., nested `if/else` or `switch` statements) inside a central manager. This approach quickly becomes difficult to maintain as new states and transitions are added, and it tightly couples state-specific logic to a single class.

5.4.2 Game Flow Management

To organize the game flow in a modular and extensible way, the State Pattern is applied. Each game phase is represented as a separate state class implementing a common interface. A context component (`GameStateMachine`) holds the active state and delegates runtime behavior to it. State-specific logic such as UI visibility, input handling, and time control is isolated within each concrete state, improving maintainability and readability.

5.4.3 State Transitions

State transitions are performed through a single mechanism (e.g., `ChangeState`) in the state machine. Typical transitions include: (i) `MainMenuState` $\rightarrow$ `PlayingState` when the game starts, (ii) `PlayingState` $\rightarrow$ `PausedState` when the player pauses, (iii) `PausedState` $\rightarrow$ `PlayingState` when resuming, (iv) `PlayingState` $\rightarrow$ `GameOverState` when a win/lose condition occurs, and (v) `GameOverState` $\rightarrow$ `MainMenuState` when returning to the menu. This design avoids complex conditionals and enables adding new states with minimal changes to existing code.

5.4.4 UML Diagrams

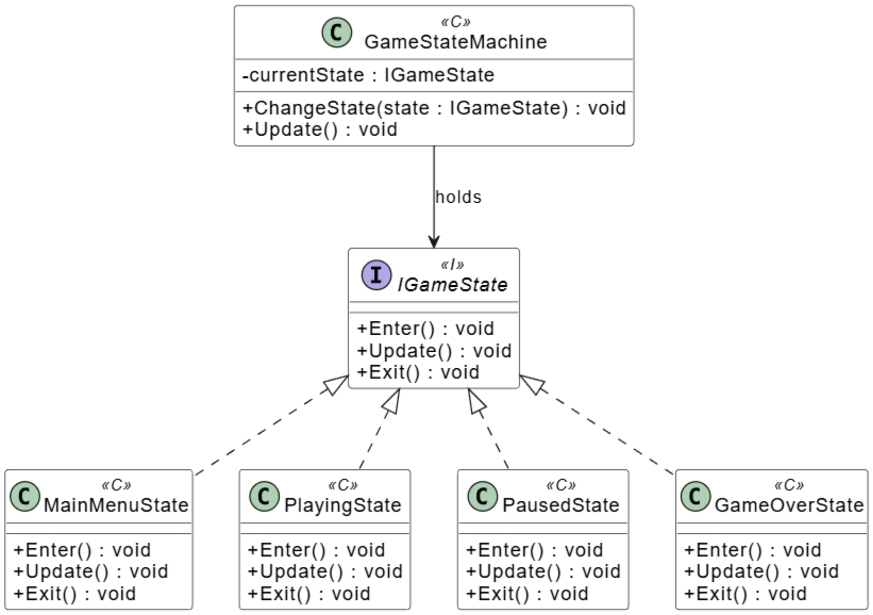

Figure 4: State Pattern class structure for managing game flow.

5.5 Command Pattern

5.5.1 Problem Definition

In a Bomberman-style game, player actions such as movement and bomb placement are triggered frequently and must be handled in a clean and maintainable way. A direct implementation that maps input keys to gameplay logic inside the player controller (e.g., `if` statements for each key) leads to tightly coupled code. Over time, adding new actions, supporting different control schemes, or introducing features such as action buffering/recording becomes difficult because input handling and gameplay logic are mixed.

5.5.2 Input Handling Abstraction

To decouple input detection from action execution, the Command Pattern is applied. Each player action is represented as a command object that implements a common command interface. The input layer becomes responsible only for detecting inputs and selecting the appropriate command, while the actual gameplay logic is encapsulated in concrete command classes. This separation improves readability, reduces conditional complexity, and allows extending the input system without modifying core gameplay components.

5.5.3 Command Execution and History

When an input event occurs, the system creates or selects a corresponding command (e.g., `MoveUpCommand`, `MoveLeftCommand`, `PlaceBombCommand`) and executes it on the target receiver (typically the player controller). Optionally, executed commands can be stored in a history list or stack. Maintaining a command history enables additional capabilities such as debugging player actions, replaying sequences, and implementing undo-like behavior if needed. In this project, command history provides a structured way to track actions and supports future extensions without significant refactoring.

5.5.4 UML Class Diagram

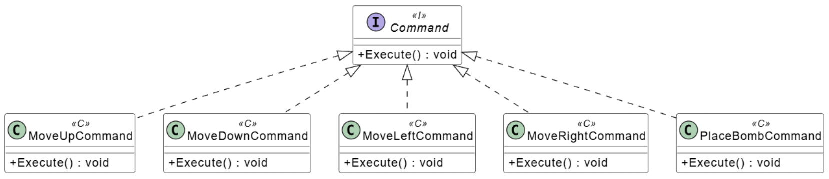

Figure 5: UML class diagram of the Command Pattern used for input handling.

5.6 Template Method Pattern

5.6.1 Problem Definition

Map generation requires a fixed workflow (clearing the map, placing boundaries, generating inner walls, and spawning entities). Implementing this workflow separately for each theme or level leads to code duplication and inconsistent generation steps. Additionally, changing the generation order becomes error-prone when the logic is scattered across multiple scripts.

5.6.2 Map Generation Workflow

To keep the generation order consistent, the map generation process is expressed as a single template algorithm. The workflow is executed in a fixed sequence: clearing the map, placing outer walls, placing inner walls, then optionally placing players, enemies, and power-ups, followed by a final post-processing step.

5.6.3 Template Structure

The class `BaseMapGeneratorTemplate` defines the template method `GenerateMap`, which enforces the generation order. Core steps (`ClearMap`, `PlaceOuterWalls`, `PlaceInnerWalls`) are defined as abstract operations and must be implemented by concrete generators. Optional steps (`PlacePlayers`, `PlaceEnemies`, `PlacePowerUps`, `AfterGenerate`) are provided as virtual hooks and can be overridden when needed. Theme-specific generators (e.g., `DesertMapGenerator` and `ForestMapGenerator`) customize only selected steps while preserving the overall workflow.

5.6.4 UML Class Diagram

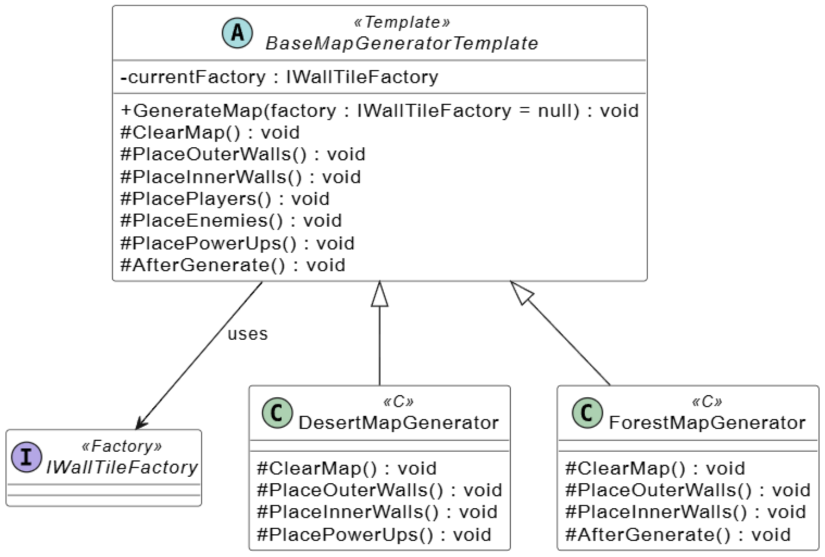

Figure 6: Template Method Pattern structure for map generation.

5.7 Repository Pattern

5.7.1 Problem Definition

The project requires persistent storage for user accounts, game statistics, and player preferences. If database queries and SQL statements are directly embedded into gameplay or UI components, the code becomes tightly coupled to the storage technology and difficult to maintain. Such coupling also makes it harder to test the system and to change the database schema or technology in the future.

5.7.2 Data Access Abstraction

To decouple persistence logic from the rest of the application, the Repository Pattern is applied. Each repository provides a dedicated interface for a specific data domain (e.g., users, game statistics, preferences) and encapsulates all database access operations such as querying, inserting, and updating records. As a result, higher-level modules interact with repositories through well-defined methods rather than raw SQL, improving maintainability and supporting cleaner architectural boundaries.

5.7.3 User and Game Statistics Repositories

In this project, the data layer is organized around separate repositories: (i) `UserRepository` for user-related operations (e.g., registration, login validation, retrieving user information), (ii) `GameStatsRepository` for match statistics and leaderboard-related operations (e.g., wins, losses, total games, high scores), and (iii) `PreferencesRepository` for storing and retrieving player preferences (e.g., selected theme). This separation ensures that each repository has a focused responsibility and database changes remain localized within the data layer.

5.7.4 UML Class Diagram

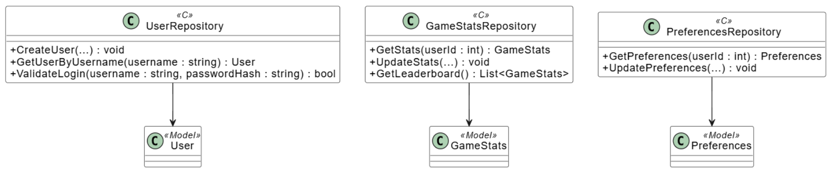
«C»
UserRepository
+CreateUser(...) : void
+GetUserByUsername(username : string) : User
+ValidateLogin(username : string, passwordHash : string) : bool
«C»
GameStatsRepository
+GetStats(userId : int) : GameStats
+UpdateStats(...) : void
+GetLeaderboard() : List<GameStats>
«C»
PreferencesRepository
+GetPreferences(userId : int) : Preferences
+UpdatePreferences(...) : void
«Model»
User
«Model»
GameStats
«Model»
Preferences

Figure 7: Repository Pattern structure for abstracting database access.

6 Supporting Design Patterns

6.0.1 Singleton Pattern

Global Game Services Certain core game services must exist as a single instance throughout the application lifecycle to ensure consistent behavior across scenes. In particular, game flow management and state coordination require a centralized control mechanism.

Usage in the Project In this project, the `GameManager` is implemented following the Singleton pattern. A static `Instance` property ensures global access, while the `Awake` method enforces the single-instance constraint by destroying duplicate objects. The instance is preserved across scene transitions using `DontDestroyOnLoad`. Other components, such as the scene controller and UI systems, interact with the game flow exclusively through `GameManager.Instance`, confirming its role as a centralized global service.

Advantages and Limitations The Singleton Pattern simplifies access to global game services and guarantees a single source of truth for core game state management. However, excessive reliance on global state may reduce testability and increase coupling. Therefore, the pattern is applied selectively and limited to components that inherently require global coordination.

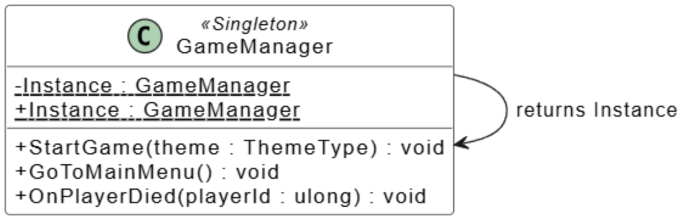

Figure 8: Singleton-style structure used for global access to GameManager.

6.0.2 Facade Pattern

GameDataService as a Unified Interface The project includes multiple repositories for different data domains such as users, match statistics, and preferences. Directly accessing each repository from upper layers (UI or gameplay components) would increase coupling and spread data-related logic across the codebase.

Facade Implementation To simplify access to the persistence layer, `GameDataService` is implemented as a facade. It internally composes three repositories: `UserRepository`, `GameStatsRepository`, and `PreferencesRepository`, and exposes higher-level operations through a single interface. For example, after a successful registration, the facade automatically ensures default rows for statistics and preferences (`EnsureDefaults`), which centralizes and standardizes the initialization workflow.

Interaction with UI and GameManager Upper-level modules such as UI controllers and game flow management components interact only with `GameDataService` (e.g., `Login`, `Register`, `RecordMatch`, `GetLeaderboardTop`, `GetTheme/SetTheme`). This reduces dependency complexity, improves readability, and allows changes in repository implementations without affecting the UI or gameplay logic.

6.0.3 UML Class Diagram

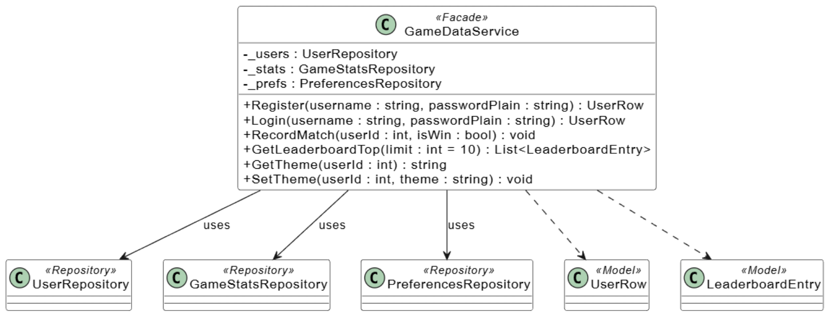
«Facade»
GameDataService
-_users : UserRepository
-_stats : GameStatsRepository
-_prefs : PreferencesRepository
+Register(username : string, passwordPlain : string) : UserRow
+Login(username : string, passwordPlain : string) : UserRow
+RecordMatch(userId : int, isWin : bool) : void
+GetLeaderboardTop(limit : int = 10) : List<LeaderboardEntry>
+GetTheme(userId : int) : string
+SetTheme(userId : int, theme : string) : void
uses
uses
uses
«Repository»
UserRepository
«Repository»
GameStatsRepository
«Repository»
PreferencesRepository
«Model»
UserRow
«Model»
LeaderboardEntry

Figure 9: Facade Pattern: GameDataService provides a unified interface over multiple repositories.

6.0.4 Adapter Pattern

Problem Definition The game uses Unity Tilemaps to represent the visual map layers (solid, breakable, and hard walls). However, core gameplay logic such as rule evaluation, AI reasoning, and grid-based queries require a lightweight and engine-independent representation of the map. Directly coupling gameplay logic to Unity Tilemap APIs would reduce testability and increase dependency on engine-specific details.

Solution Approach To decouple game logic from Unity-specific structures, an adapter component (`MapLogicAdapter`) is used. It reads tile information from multiple Tilemaps and converts them into a logical grid representation (`MapGrid`) using a cell classification (`CellType`). This provides a clean interface for logic systems while preserving Tilemap usage for rendering.

Outcome With this approach, upper-level systems can operate on `MapGrid` without relying on Unity Tilemap operations, improving modularity and simplifying future extensions (e.g., different map sources or alternative rendering layers).

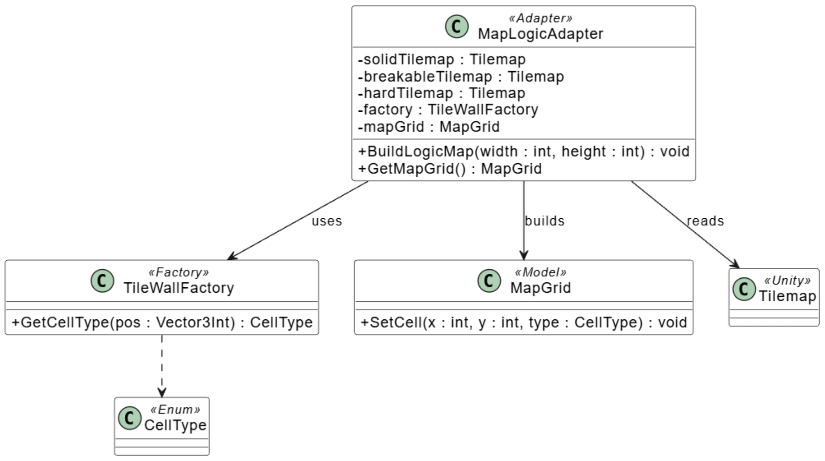
«Adapter»
MapLogicAdapter
-solidTilemap : Tilemap
-breakableTilemap : Tilemap
-hardTilemap : Tilemap
-factory : TileWallFactory
-mapGrid : MapGrid
+BuildLogicMap(width : int, height : int) : void
+GetMapGrid() : MapGrid
uses
builds
reads
«Factory»
TileWallFactory
+GetCellType(pos : Vector3Int) : CellType
«Model»
MapGrid
+SetCell(x : int, y : int, type : CellType) : void
«Unity»
Tilemap
«Enum»
CellType

Figure 10: Adapter Pattern: MapLogicAdapter converts Unity Tilemap data into a logic-level MapGrid.

7 Event-Driven Communication

8 Event-Driven Communication

8.1 Event-Based Decoupling in the Game

To reduce tight coupling between gameplay components, the project follows an event-driven communication approach. Key gameplay occurrences such as player death, game over conditions, and scene transitions trigger specific actions instead of relying on continuous state polling or direct component dependencies.

8.2 Observer-Like Behavior

Although a classical Observer Pattern with explicit event subscription is not implemented, the system exhibits observer-like behavior. Centralized game state management triggers reactions in UI and scene management components when relevant state changes occur, preserving modularity without unnecessary complexity.

8.3 Player Death, Game Over, and State Updates

Player elimination and game over events update the game state through centralized control logic, leading to appropriate transitions such as restarting the level or returning to the main menu. UI components respond to these updates by presenting the corresponding user actions, maintaining a clear separation between gameplay logic and interface behavior.

9 Multiplayer Architecture

9.1 Host–Client Model

The multiplayer system is based on a host–client architecture, where one player acts as the authoritative host while other players connect as clients. The host is responsible for maintaining the authoritative game state, including player positions, bomb placement, and explosion resolution. Clients send input requests to the host and receive synchronized state updates to ensure consistent gameplay across all connected instances.

9.2 Network Synchronization Strategy

To maintain consistency between players, critical gameplay events are synchronized over the network rather than relying on local simulation. Player actions such as movement and bomb placement are validated and executed by the host, then propagated to all clients. This approach prevents state divergence and ensures that all players observe the same game state regardless of network latency.

9.3 Bomb Placement and Explosion Synchronization

Bomb placement and explosion events represent time-critical interactions in the game. When a player places a bomb, the request is sent to the host, which validates the action and spawns the bomb in the authoritative game state. The explosion timing and affected tiles are computed by the host and synchronized to all clients, guaranteeing that damage, wall destruction, and player elimination occur consistently across the network.

10 Database Design

10.1 Database Schema

The project uses a lightweight relational database design to manage user accounts, gameplay statistics, and leaderboard data. SQLite is preferred due to its simplicity, portability, and suitability for small to medium-scale applications. The database schema is designed to separate concerns by storing user credentials, match statistics, and preference data in distinct tables, improving maintainability and clarity.

10.2 User Table

The user table stores authentication-related information for each player. Each user is identified by a unique identifier and associated with a username and a securely stored password representation. This separation allows the authentication logic to remain independent from gameplay and statistical data.

10.3 Game Statistics Table

The game statistics table records match-related data for each user, such as the total number of matches played and the number of wins. These statistics are updated after each completed match and serve as the primary data source for performance tracking and leaderboard generation.

10.4 Leaderboard Structure

The leaderboard is derived from aggregated game statistics rather than being stored as a separate table. By querying and sorting users based on their win count or performance metrics, the system dynamically generates the leaderboard. This approach avoids data duplication and ensures that leaderboard information always reflects the most up-to-date game statistics.

11 User Interface Design

11.1 Main Menu

The main menu serves as the entry point of the game and provides access to core functionalities such as starting a new game, selecting game modes, and navigating to authentication screens. The layout is designed to be simple and intuitive, allowing users to quickly access essential options.

11.2 Login and Register Screens

The login and registration screens enable user authentication and account creation. These interfaces interact with the underlying data layer through centralized services, ensuring that authentication logic remains separated from presentation concerns.

11.3 In-Game UI

The in-game user interface displays real-time gameplay information, including player status and game progression indicators. UI elements are designed to minimize visual clutter and avoid interfering with core gameplay mechanics.

11.4 Game Over and Leaderboard Screens

The game over screen provides options to restart the level or return to the main menu upon match completion. The leaderboard screen displays player rankings based on aggregated game statistics, offering feedback on player performance and encouraging replayability.

12 Conclusion and Future Work

12.1 Conclusion

This project demonstrated the practical application of software design patterns within a Bomberman-style game developed using Unity and C#. Core gameplay systems such as map generation, player mechanics, enemy behavior, game flow, and data management were designed using well-known creational, structural, and behavioral design patterns. By applying patterns such as Factory, Decorator, Strategy, State, Command, Repository, Facade, Adapter, Singleton, and Template Method, the project achieved a modular, extensible, and maintainable architecture.

The use of design patterns reduced tight coupling between components, simplified future extensions, and improved overall code readability. The project successfully shows that design patterns are not only theoretical concepts but also effective tools for managing complexity in real-world game development scenarios.

12.2 Future Work

Several improvements and extensions can be considered for future development. Multiplayer functionality can be further stabilized and expanded with advanced synchronization techniques. The event-driven communication model can be enhanced by introducing a full observer-based event system. Additional enemy AI strategies and map generation variants may be implemented to increase gameplay diversity. Finally, automated testing and performance profiling could be integrated to further improve system reliability and scalability.

13 References

- Unity Technologies. *Unity Documentation.* Available at: https://docs.unity.com/ (Accessed: 2025).
- Course Lecture Notes. *Design Patterns.* Department of Computer Engineering, course material provided by the instructor.
- Kenney. *Free Game Assets.* Available at: https://kenney.nl/ (Accessed: 2025).