Table of content

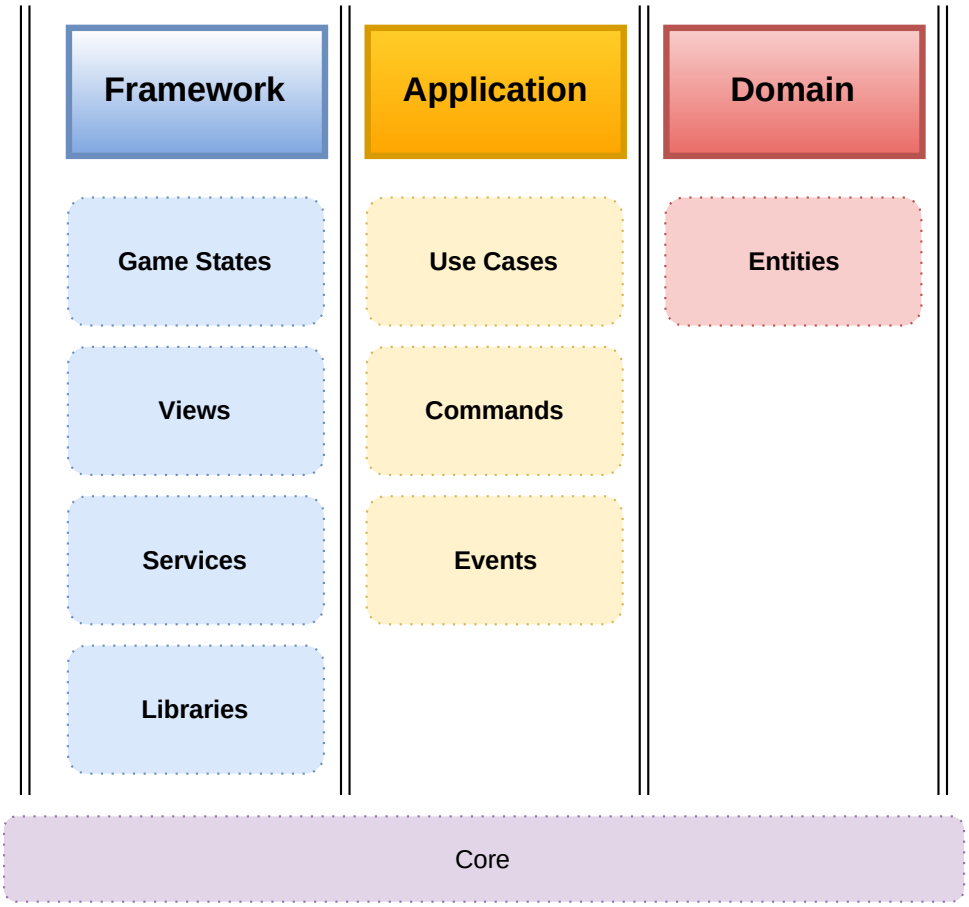
Approach

The solution was structured around a three-layer architecture (Domain / Application / Framework), with each layer owning a clear responsibility and interacting through explicit boundaries. Features were implemented iteratively while keeping dependencies explicit and minimizing coupling between systems.

To support this separation and keep the gameplay flow readable as the project grew, a small set of infrastructure building blocks was introduced upfront (message bus, state machine, and a composition root).

Architecture

- **Three-layer separation (Framework / Application / Domain):** responsibilities were split so Unity-facing concerns stayed in **Framework**, use cases lived in **Application**, and core game rules/invariants lived in **Domain**. Dependencies were kept **inward-only** (Framework → Application → Domain), which prevents Unity types and scene concerns from leaking into the core, limits "ripple effects" when presentation/runtime details change, and enables fast unit testing of gameplay rules without requiring Unity/Play Mode.



Architecture overview

Infrastructure

- **Message bus (commands/events):** enabled fast communication between systems without direct references, keeping the gameplay flow readable and reducing coupling.
- **State machine:** provided a clear high-level game flow (menu → playing → game over) and a single place to manage enter/exit responsibilities like suscribing and unsuscribing to events specific to game states.
- **Composition root:** centralized wiring and dependency injection, making dependencies explicit, controlling initialization order, and keeping runtime setup deterministic.

Gameplay actions modeled as Use Cases

- **Commands** were used to represent player/system intents and **use cases** to implement application behavior in a discoverable way (it makes easy to locate “what the app does”). Use cases published **events** to represent outcomes and trigger reactions elsewhere.

Data-driven content

- **ScriptableObjects** were used for enemies, waves, and towers to support easy extension (new content was largely created as new assets rather than new code).

Target selection without physics queries

- Physics queries were avoided for tower targeting; t**owers selected enemies through EnemiesService**, which maintained a list of alive enemies. This kept the targeting logic simple and deterministic, and also avoided relying on physics overlap scans (helpful for WebGL constraints).

Lifecycle and cleanup

- Services subscribed/unsubscribed via IStartable, and states performed explicit cleanup on exit (despawning enemies/towers). This prevented duplicated subscriptions and “ghost objects” across runs.

Challenges / Trade offs

- Under the time-box, **some Application messages carried runtime handles** (Unity/Framework objects). In a larger project, this would be replaced with ID-only messages resolved through registries/repositories to keep Application/Domain fully Unity-agnostic.
- **Object pooling was intentionally omitted** due to time constraints. Instantiation is currently localized (only in EnemySpawner and TowerService), so introducing

pooling later would be straightforward and would not require widespread changes.

- Defining a correct **“victory” condition** required coordinating two signals: “all waves spawned” and “no alive enemies”. This avoided false positives between waves.
- Deterministic **cleanup across state transitions** required explicit lifecycle management (subscribe/unsubscribe via IStartable and cleanup on state exit) to prevent duplicated subscriptions and lingering runtime objects across runs.

Sessions breakdown

S1 – Project Setup & Level Infrastructure

Estimated focused development time: ~2 hours

Done

- Created Unity 6000.0.26f1 URP project.
- Defined project folder structure separating Core, Domain, Application and Framework layers.
- Created Main scene.
- Implemented path abstraction via `IPathProvider`.
- Implemented `IPathProvider` as `WaypointPath` with serialized waypoints.
- Implemented editor-only gizmo drawer using `[DrawGizmo]`.
- Designed and placed a 12-waypoint path.
- Created 9 tower slot prefabs placed intentionally across early/mid/late zones.
- Implemented `TowerSlot` runtime component with mandatory anchor.

Notes

- TowerSlot designed minimal; will evolve when tower runtime exists.

Next

- Implement application infrastructure:
  - EventBus
  - Game state machine (State pattern)
  - GameCompositionRoot

S2 – Application Infrastructure & UI State Flow

Estimated focused development time: ~2.5 hours

Done

- Implemented `IGameState` and `GameStateMachine` in Core (no UnityEngine dependency).
- Implemented EventBus abstraction `IEventBus` and `DictionaryEventBus` (Subscribe/Unsubscribe/Publish).
- Implemented framework `GameStateBase` with `SetNextState(IGameState)` and `GoToNextState()`.
- Created a single Canvas with three panels: MainMenu / Gameplay / GameOver.
- Implemented panel Views and wired them to raise UI events.
- Implemented `GameCompositionRoot` to instantiate and wire `GameStateMachine`,

EventBus, and all states.

- Implemented `MainMenuState` to show panel, subscribe to Start, and transition to Gameplay.

- Implemented `LevelGameplayState` to show panel, subscribe to EndGame, and transition to GameOver.

- Implemented `GameOverState` to show panel, subscribe to Restart, and transition back to Gameplay.

Next

- Implement enemies: prefab + basic runtime + domain, waypoint movement, spawner, and minimal waves.

S3 – Enemies, Waves & Fortress Attack Loop

Estimated focused development time: 4 hours

Done

- Added enemy domain model with validated constructor invariants (speed, hp, reward, strength) and damage handling.

- Created enemy ScriptableObject definitions and an EnemiesLibrary to support multiple enemy types.

- Implemented runtime enemy spawning via EnemySpawner using EnemyDefinition prefabs.

- Implemented path following movement for enemies using IPathProvider waypoints.

- Added framework “sensors” to publish AttackFortressCommand when an enemy reaches the goal.

- Implemented AttackFortress use case and Fortress domain model; publishes FortressDestroyedEvent when lives reach zero.

- Wired PlayingGameState to react to FortressDestroyedEvent and transition to GameOverState.

- Implemented wave spawning via WaveDefinition ScriptableObject (bursts) and WaveSpawner; starts/stops with PlayingGameState.

- Added EnemiesService + EnemyLifecycle to manage enemy lifetime and cleanup (goal reached and state exit).

Notes

- Enemy pooling intentionally omitted due to time constraints.

- Commands vs events clarified; marker interfaces ICommand/IEvent added for navigation (bus enforcement noted as future improvement).

Next

Towers (build on slots, targeting, damage, enemy death + reward economy) and basic

HUD feedback.

S4 – Towers: Targeting, Damage & Shot Feecback

Estimated focused development time: 5 hours

Done

-Implemented tower firing driven by TowerDefinition stats (range, fire rate, damage).

-Added closest-enemy-in-range targeting in EnemiesService (no physics queries).

-Wired damage pipeline: TowerView publishes DamageEnemyCommand, ApplyDamageToEnemyUseCase applies damage to Enemy and updates Wallet/Score, then publishes EnemyKilledEvent.

-Extended IEnemyLifecycle to expose EnemyEntity to keep the targeting/damage flow simple and avoid extra lookups.

-Added wave completion + “all enemies cleared” end condition: after AllWavesSpawned, publish AllEnemiesKilled when the alive enemy count reaches zero.

-Added minimal shot feedback using a LineRenderer view component triggered via a UnityEvent fired from TowerView.

Notes

Pragmatic time-boxed trade-off: some Application commands/events carry runtime handles (Unity/Framework objects). Examples: TowerBuiltEvent includes TowerSlot and TowerDefinition, and several messages carry IEnemyLifecycle. In a larger project, this would be replaced by ID-only messages resolved through registries/repositories to keep Application/Domain fully Unity-agnostic.

Future extensions

Towers configurable targeting

-Add configurable targeting strategies per tower (e.g. Closest/First/Last/Strongest) as a pluggable strategy or TowerDefinition setting.

-Set by a new use case SetTargetingStrategy

-Current targeting is centralized in EnemiesService, so introducing a strategy parameter is localized and avoids touching the damage/economy pipeline.

-Keeps TowerView simple (request target → shoot) while enabling richer gameplay variety.

Additional tower types

-New TowerDefinition assets (different range/fireRate/damage, cost balancing).

-This fits naturally because tower behavior is already driven by data (TowerDefinition) and instantiation is handled by the existing build pipeline (BuildTowerOnSlot → TowerBuiltEvent → TowerService).

-Allows scaling content without changing code, and demonstrates the intended data-driven approach.

Refactor to keep Unity dependencies inside Framework

-Enforce asmdef boundaries, replace runtime handles in Application messages with IDs + registries/mapping).

-Convenient to improve portability/testability: Application/Domain become plain C# and can be unit-tested without Unity, while Framework becomes the only Unity integration layer.

-Reduces coupling, make dependencies more explicit and visible and makes features like replay, or networking easier (messages carry IDs instead of live scene references).