同濟大學
TONGJI UNIVERSITY

目录

1.Project Description

1.1 Background

This project is a 2D action role-playing game (ARPG) developed with Unity, featuring a fully interactive combat system, enemy artificial intelligence, skill mechanics, inventory management, and player progression. The game showcases modular system design commonly used in professional Unity projects, including component-based entity architecture, finite state machines for both player and enemy behaviors, and event-driven interactions among gameplay components.

The project integrates typical RPG features such as character attributes, elemental effects, equipment with stat modifiers, consumables, boss encounters, crafting mechanics, and persistent save/load functionality. As the game grew in scope, the source code accumulated various design challenges, particularly in extensibility, complexity, and code reuse—making it a suitable candidate for systematic refactoring using software design patterns.

This project was selected for the final coursework because it offers:

- Multiple interacting subsystems suitable for design pattern applications,
- Several areas with evident code smells (e.g., duplicated logic, tightly coupled components, complex update loops, large state switch statements),
- A real-world Unity architecture where design patterns significantly improve maintainability.

1.2 Major Functionalities

Player System

- Full state-machine-driven player controller with states such as idle, move, jump, attack, evade, counterattack, and special skill execution.
- A rich skill system including:
  - Homing sword (aim, throw, recall)
  - Dash strike
  - Dark-hole skill with clone attacks
  - Assassination teleport
  - Lightning and crystal skills
- Inventory and equipment system (weapons, armor, accessories, consumables).
- Elemental effects system supporting burn, frost, and shock debuffs.

Enemy & Boss AI

- Hierarchical finite state machines for all enemies.
- Includes multiple enemy types:

- Skeleton (basic melee AI),
- Slime (multi-size splitting behavior),
- NightBorn Boss—a multi-phase, advanced AI boss featuring:
  - Melee and ranged attacks,
  - Charging, healing, teleportation,
  - Elemental projectile attacks,
  - Magic circle AoE abilities.

Combat System

- Physical and magical damage types.
- Status effects: ignite (DoT), freeze (slow + armor reduction), shock (chain lightning + accuracy penalty).
- Blocking, parrying, knockback, and directional combat mechanics.

UI & Gameplay Systems

- HUD: health, stamina, currency
- Inventory UI with item/equipment slots
- Skill tree system
- Audio settings with separate SFX/BGM controls
- Crafting system and resource management
- Save/load system with player progress, items, skills, and settings persistence

Technical Architecture

- Component-based design (Unity standard architecture)

- Entity core system for shared character logic
- Extensive use of FSM pattern for players and enemies
- Event system for decoupled communication
- Modular managers (SkillManager, AudioManager, SaveManager)

1.3 Motivation for Refactoring

The refactoring work for this Unity 2D ARPG project was driven by concrete structural and maintainability problems discovered in the codebase. The following motivates the refactor: observed code smells, concrete design problems, symptoms of high coupling and low cohesion, and limitations that block scalability, extensibility and maintainability. Each item below references specific classes or subsystems from the codebase and links the problem to the chosen refactoring strategies.

1.3.1 Code Smells

God classes / Overly large scripts

Examples: *Player.cs, Enemy.cs, SkillManager.cs, Inventory.cs.* These files aggregate many responsibilities (input handling, state management, skill logic, UI updates, audio triggers, and persistence). Result: poor readability, difficult unit testing and fragile modification surface.

Repeated logic and duplication

Common code (damage application, elemental effect handling, cooldown updates, state transition checks) appears in multiple enemy and skill classes instead of being centralized. This increases bug-fix surface and causes inconsistent behavior between entities.

Primitive obsession

Use of enums and flags to represent behaviors and states (rather than polymorphic strategy objects) is pervasive in state and skill handling, limiting flexibility for introducing new behaviors.

Tight low-level access

Many high-level scripts directly access low-level managers (e.g., *AudioManager.instance, PlayerManager.instance, DroppedItemManager.instance*) rather than using abstractions or a central access layer.

Scattered event handling

Event bindings and UI updates are implemented ad-hoc across *UI_InGame.cs, Player* states, and skill scripts; subscription/unsubscription occurs in many places, making lifecycle management error prone.

1.3.2 Design Problems

Single Responsibility Principle violations

Core classes mix presentation, business logic and side-effects. For example, *Player.cs* manages state creation, input mapping, skill invocation, and direct UI/audio calls. This makes each change risky and costly.

Inconsistent abstraction boundaries

Many systems use concrete class references instead of interfaces. The *Skill* family and *SkillManager* are tightly coupled; *SkillManager* hard-codes skills rather than exposing a pluggable interface. This prevents adding new skills without altering core managers.

Duplication of state creation

Enemy subclasses repeat nearly identical state-instantiation code. This duplication hurts maintainability and makes consistent bug fixes difficult.

Poor separation of concerns between UI and game logic

UI_InGame.cs directly queries and manipulates game objects and managers, which couples UI changes to gameplay internals and makes UI rework or decoupling harder.

Inefficient lookup and lifecycle patterns

SaveManager relies on searching (*FindAllSaveManagers()*) at runtime, and many systems rely on singleton lookups; both patterns impede predictable initialization order and add runtime cost.

1.3.3 High coupling and low cohesion

High coupling examples

- *Player* ↔ many singletons (*SkillManager.instance, PlayerManager.instance, AudioManager.instance, etc.*) — changes to managers ripple into player code.
- *Inventory* directly manipulates UI slots and is directly referenced by *UI_InGame* and *Player*, causing cross-module dependency cycles.

Low cohesion examples

- *CharacterStats.cs* mixes damage calculation, elemental logic and state effect bookkeeping in a single class (*DoDamage()* contains many if/else branches); the class therefore combines unrelated concerns.

- *GameManager.cs* handles checkpoints, chest logic, pause, and save orchestration — multiple unrelated responsibilities exist in one manager.

These symptoms create code that is hard to test in isolation, hard to reason about, and brittle to change.

1.3.4 Scalability, extensibility and maintainability limitations

Difficulty adding new features

Because skills, enemies and UI are tightly coupled and often hardcoded (e.g., *SkillManager* skill list, manual state creation in enemies), adding a new skill type or enemy variant requires changes across multiple files rather than adding a single new component.

Testing & debugging overhead

Side-effects (audio, UI, save) performed inline in domain code make unit testing impractical without heavy mocking. Debugging requires running larger parts of the game.

Performance and resource management concerns

Managers like *DroppedItemManager* and *AfterImageManager* create/destroy many GameObjects without object pooling, which can cause frame spikes and memory churn at scale.

1.3.5 How the refactor addresses these problems

To directly address the above issues, the refactor introduced explicit patterns and targeted changes that map to concrete problems:

Service Locator & Facade (*Core/ServiceLocator.cs, Core/GameFacade.cs***)**

Rationale: centralize and standardize service access to reduce scattered singleton access and simplify dependency management. This reduces ad-hoc singleton usage and clarifies initialization order.

Factory Method for enemy states (*Enemy/Factories/*****) and PlayerStateFactory**

Rationale: remove duplicated state creation logic, enable consistent state composition and simplify extending enemy types.

Observer / Event Bus (*Core/Events/GameEventBus.cs***)**

Rationale: decouple UI and gameplay logic by replacing direct cross-module calls with publish/subscribe events for skill use, equipment changes and state transitions.

Command Pattern for input (*Skills/Commands/*, Player/PlayerInputHandler.cs***)**

Rationale: encapsulate skill invocation to decouple input handling from skill execution and support extensible input-to-action mapping.

Bridge, Composite, Decorator, Strategy where appropriate

Rationale: modularize equipment effects (*Item/Effect/** uses Bridge), build UI component trees (*UI/Composite/**), and prepare *CharacterStats* and effect systems for pluggable, testable strategies for damage and status effects.

1.3.6 Summary

The refactor was undertaken to transform a codebase that had evolved in an ad-hoc way into a modular, extensible and testable architecture. By identifying concrete code smells and design problems (documented per class in the source description) and

applying targeted design patterns (*Service Locator, Factory Method, Observer, Command, Facade, Bridge, Composite, Strategy*), we improve separation of concerns, reduce coupling, increase cohesion and enable easier, safer extension of gameplay content (new skills, enemies, UI features). These changes directly support the project goals of long-term *maintainability* and scalable content growth.

2.Refactoring Work & Design Pattern Application

2.1 Overview of All Applied Design Patterns

2.1.1 Creational pattern

- **Factory Method**

2.1.2 Structural patterns

- **Bridge**
- **Composite**
- **Facade**

2.1.3 Behavioral patterns

- **Observer**
- **Command**
- **Strategy**

2.1.4 Additional pattern

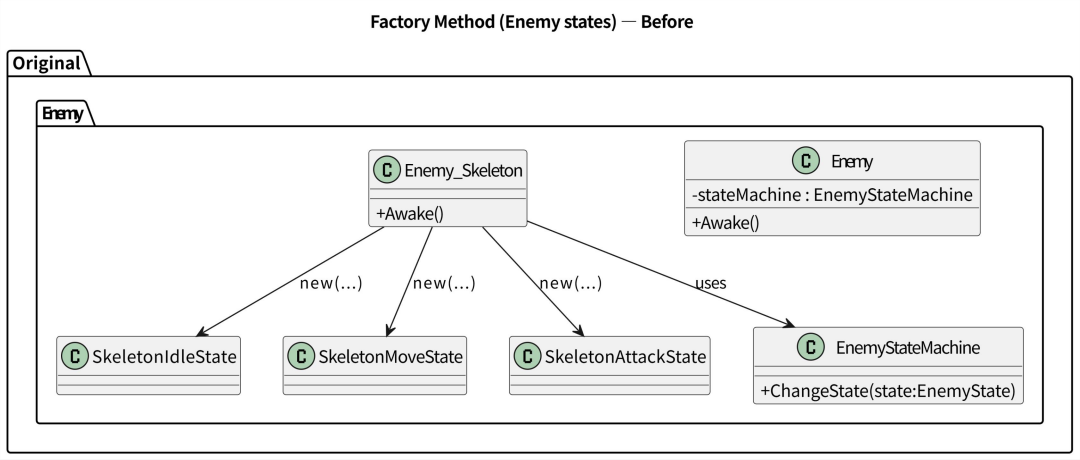
- **Service Locator**

2.2 Detailed Refactoring

2.2.1 Factory Method (Creational Pattern)

Before Refactoring

Orignal UML class diagram



Related source code (only key excerpts)

```
public class Enemy_Skeleton : Enemy
{
    private void Awake()
    {
        // Before: directly new each state (duplicated across enemies)
        idleState   = new SkeletonIdleState(this, stateMachine);
        moveState   = new SkeletonMoveState(this, stateMachine);
        attackState = new SkeletonAttackState(this, stateMachine);
        stateMachine.ChangeState(idleState);
    }
}
```

Problem analysis

Originally, each enemy class (e.g., `*Enemy_Skeleton*`, `*Enemy_Slime*`) directly created and managed its state instances (such as `*IdleState*`, `*MoveState*`, and `*AttackState*`). This led to the following issues:

1. Code Duplication: State initialization logic was repeated in every enemy class, making maintenance difficult and error-prone.

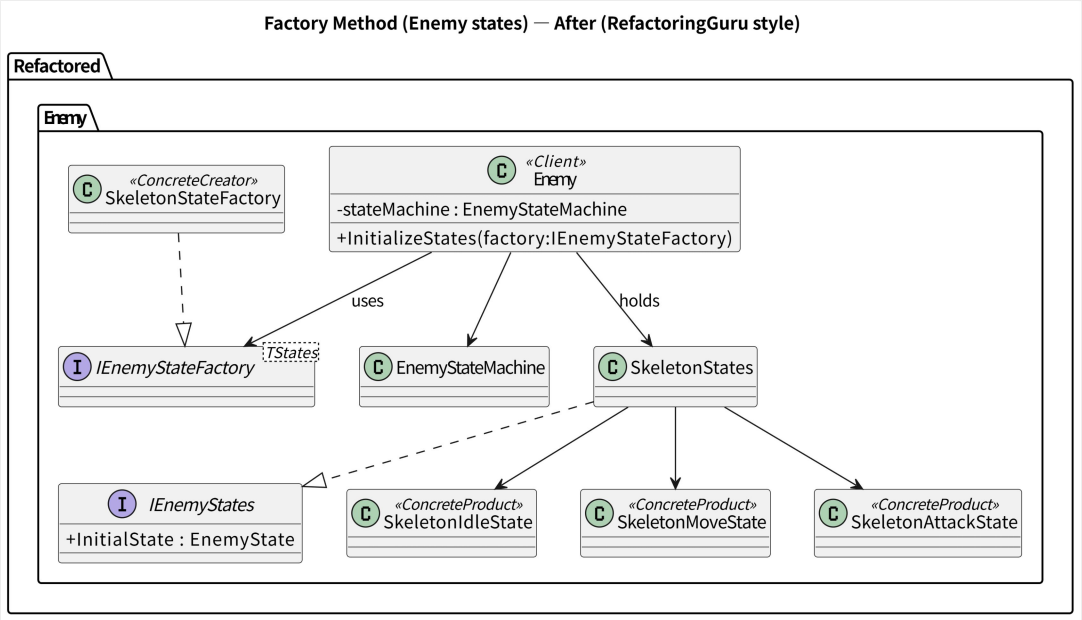
2. Responsibility Coupling: Enemy objects combined both behavior logic and state object creation, violating single responsibility.

3. Limited Extensibility: Adding new enemy types required duplicating state construction logic and synchronizing any changes across all classes.

4. Testing Challenges: Hardcoded state creation prevented easy unit testing or mock injection.

After Refactoring

Updated UML class diagram



Key refactored code excerpts

```
public interface IEnemyStateFactory<out TStates> where TStates : IEnemyStates
{
    TStates CreateStates(Enemy enemy, EnemyStateMachine stateMachine);
}
```

```
public interface IEnemyStates { EnemyState InitialState { get; } }
```

```
public class EnemyStateMachine
{
    public EnemyState currentState { get; private set; }

    public void Initialize(EnemyState _startState) { currentState = _startState; currentState.Enter(); }
    public void ChangeState(EnemyState _newState) { currentState.Exit(); currentState = _newState; currentState.Enter
}
```

```
public class EnemyState
{
    protected EnemyStateMachine stateMachine;
    protected Enemy enemyBase;
    protected Rigidbody2D rb;
    protected bool triggerCalled;
    private string animBoolName;
    protected float stateTimer;
    protected IAudioManager audioManager;
    protected IPlayerManager playerManager;

    public EnemyState(Enemy _enemyBase, EnemyStateMachine _stateMachine, string _animBoolName)
    {
        this.stateMachine = _stateMachine;
        this.enemyBase = _enemyBase;
        this.animBoolName = _animBoolName;
    }

    public virtual void Enter()
    {
        triggerCalled = false;
        rb = enemyBase.rb;
        enemyBase.anim.SetBool(animBoolName, true);
        if (audioManager == null) audioManager = ServiceLocator.Instance.Get<IAudioManager>();
        if (playerManager == null) playerManager = ServiceLocator.Instance.Get<IPlayerManager>();
    }
    public virtual void Exit() { enemyBase.anim.SetBool(animBoolName, false); }
    public virtual void Update() { stateTimer -= Time.deltaTime; }
    public virtual void AnimationFinishTrigger() { triggerCalled = true; }
}
```

```
public class Enemy : Entity
{
    public EnemyStateMachine stateMachine { get; private set; }
    protected IPlayerManager playerManager;
    protected IAudioManager audioManager;

    protected override void Awake()
    {
        base.Awake();
        stateMachine = new EnemyStateMachine();
        playerManager = ServiceLocator.Instance.Get<IPlayerManager>();
        audioManager  = ServiceLocator.Instance.Get<IAudioManager>();
    }

    protected override void Update()
    {
        base.Update();
        if (stateMachine == null || stateMachine.currentState == null) return;
        stateMachine.currentState.Update();
        CheckNormalColor();
    }

    public void AnimationTrigger() { stateMachine.currentState.AnimationFinishTrigger(); }

    public void Die() { /* drop exp/currency etc. */ }
}
```

Explanation of how the pattern was applied

- Each enemy type provides its own StateFactory responsible for creating and managing state objects.
- Enemy class retrieves the state aggregate via the factory and sets its initial state without direct instantiation.Each enemy type provides its own StateFactory

responsible for creating and managing state objects.

- Enemy class retrieves the state aggregate via the factory and sets its initial state without direct instantiation.

Changes in file structure or class responsibilities

Added:

- `refactored_src/Entity/Enemy/Factories/*` — Dedicated StateFactory classes for each enemy type, centralizing state creation logic.
- `refactored_src/Entity/Enemy/States/*` — State aggregate classes (e.g. `SkeletonStates`, `SlimeStates`) encapsulating all states per enemy.

Modified:

- Core enemy classes in `refactored_src/Entity/Enemy/*` now use their corresponding factory and state aggregate for initialization, removing direct state instantiation logic from their own implementation.

Comparation

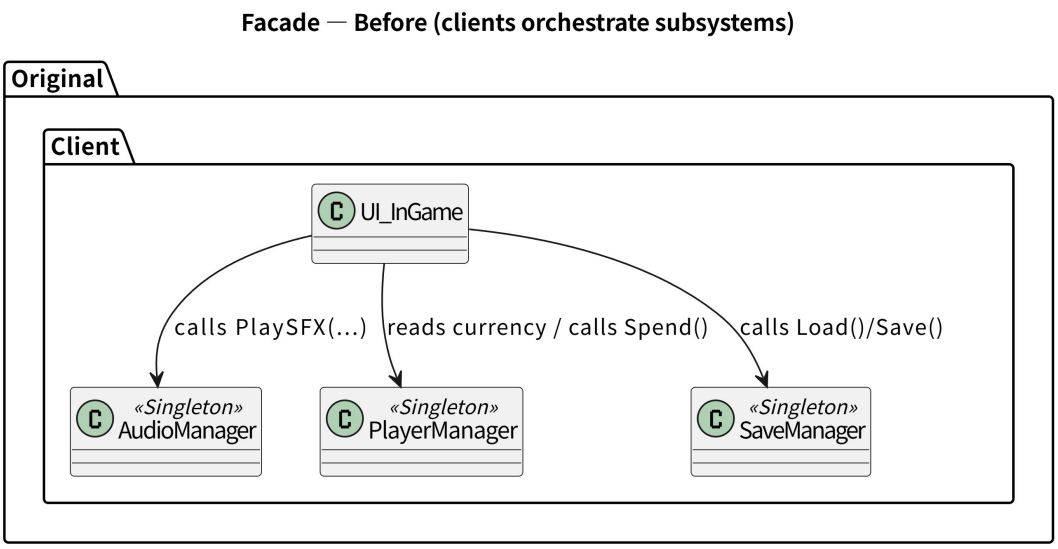
Aspect	Before Refactoring	After Refactoring
State Creation	Scattered & repeated in each enemy class	Centralized in factory, clear responsibility
Extensibility	New enemy type = copy-paste/init logic again	Just add new factory & state aggregate
Maintainability	Updating states = edit many classes	Just update relevant factory
Testability	Hard to mock/init states	Factory logic easily unit

	for testing	tested/mockable

2.2.2 Facade (Structrual Pattern)

Before Refactoring

Orignal UML class diagram



Related source code (only key excerpts)

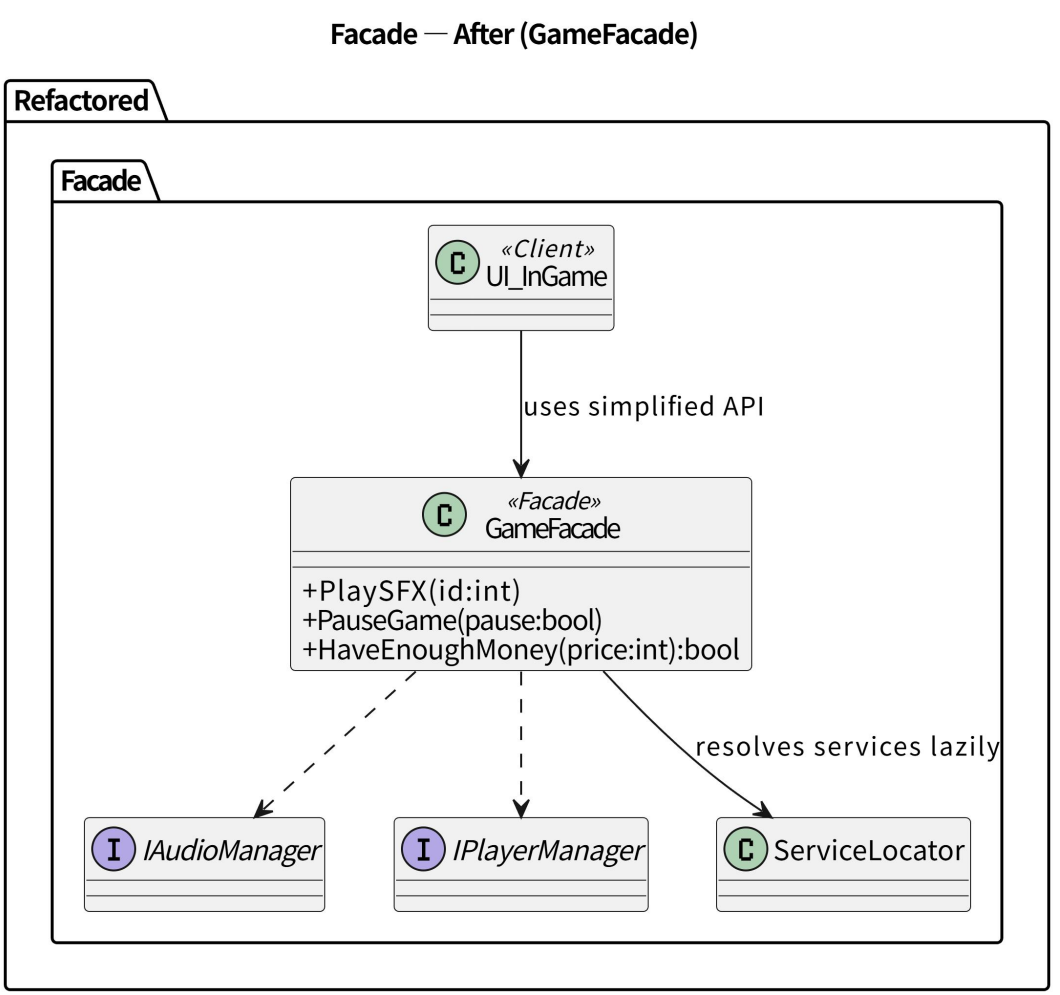
```
private void SomeAction()
{
    // Before: multiple explicit lookups and orchestration in callers
    AudioManager.Instance.PlaySFX(5);
    if (SaveManager.Instance.HasSave()) { SaveManager.Instance.Load(); }
    if (PlayerManager.Instance.Currency >= price) { PlayerManager.Instance.Spend(price); }
}
```

Problem analysis

- Repeated multi-service lookups clutter client code across multiple locations
- Callers orchestrate multiple service calls directly, leading to tight coupling
- Service dependencies are scattered throughout UI and Manager code
- Code duplication in service resolution logic

After Refactoring

Updated UML class diagram



Key refactored code excerpts

```
public class GameFacade
{
    private static GameFacade instance;
    private IPlayerManager playerManager;
    private ISkillManager skillManager;
    private IInventory inventory;
    private IAudioManager audioManager;
    private GameEventBus eventBus;
    // ... other cached services ...

    public static GameFacade Instance { get { if (instance == null) instance = new GameFacade(); return instance; } }

    public void Initialize()
    {
        playerManager = ServiceLocator.Instance.Get<IPlayerManager>();
        skillManager  = ServiceLocator.Instance.Get<ISkillManager>();
        inventory     = ServiceLocator.Instance.Get<IInventory>();
        audioManager  = ServiceLocator.Instance.Get<IAudioManager>();
        eventBus      = ServiceLocator.Instance.Get<GameEventBus>();
        Debug.Log("[GameFacade] Initialized with all services (Facade Pattern)");
    }

    public IPlayerManager Player => playerManager;
    public ISkillManager Skills => skillManager;
    public IInventory Inventory => inventory;
    public IAudioManager Audio => audioManager;
    public GameEventBus Events => eventBus;

    public void PlaySFX(int index) { audioManager?.PlaySFX(index); }
    public void PublishEvent<T>(T gameEvent) where T : IGameEvent { eventBus?.Publish(gameEvent); }
    public static void Reset() { instance = null; }
}
```

Explanation of how the pattern was applied

- Added *refactored_src/Core/GameFacade.cs* to present a small set of commonly used operations (PlaySFX, PauseGame, HaveEnoughMoney, property accessors).
- *GameFacade* lazily resolves underlying services via ServiceLocator and simplifies client code.

Changes in file structure or class responsibilities

- Added: *refactored_src/Core/GameFacade.cs* as the central facade class
- Modified: All callers in *refactored_src/UI/** and *Managers/** now use GameFacade instead of direct service access
- Removed: Direct service resolution code from client classes

Comparison

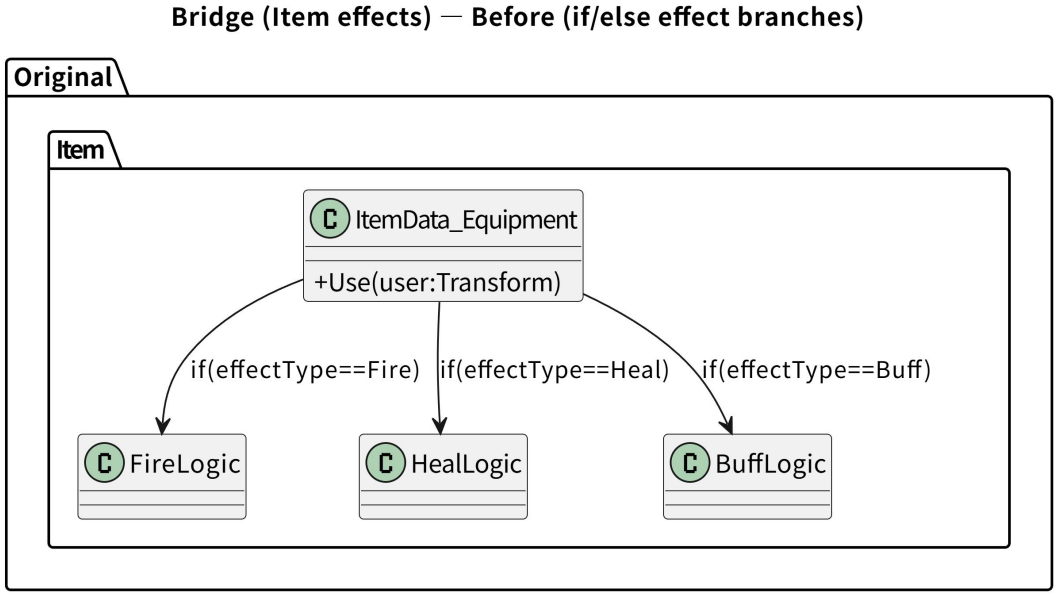
Aspect	Before Refactoring	After Refactoring

Aspect	Before Refactoring	After Refactoring
Client Complexity	High - clients manage multiple service interactions	Low - single facade interface simplifies usage
Coupling	Tight coupling between clients and multiple services	Loose coupling - clients depend only on facade
Code Duplication	Service resolution logic repeated across clients	Centralized service access in facade
Maintainability	Changes require updates in many client files	Changes isolated to facade implementation

2.2.3 Bridge (Structrual Pattern)

Before Refactoring

Orignal UML class diagram



Related source code (only key excerpts)

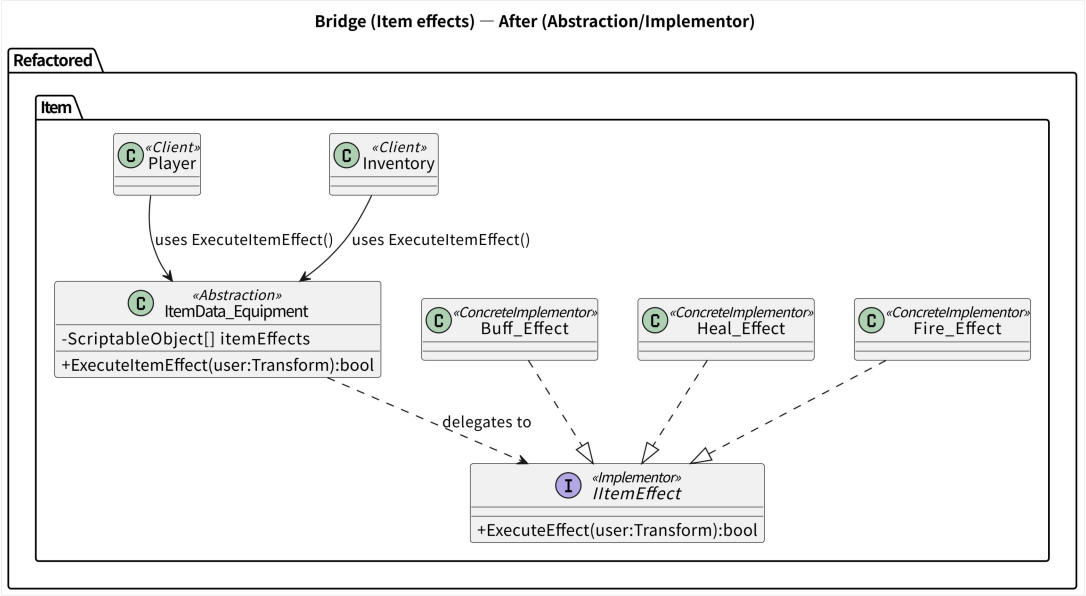
```
public class ItemData_Equipment : ScriptableObject
{
    // Before: equipment handled effects inline or via specific classes (example pseudo)
    public void Use(Transform user)
    {
        // multiple if/else for each effect type
        if (effectType == EffectType.Fire) SpawnFire(user.position);
        else if (effectType == EffectType.Heal) HealUser(user);
        // ... more branches per equipment-effect
    }
}
```

Problem analysis

- Each equipment-effect pairing could cause class explosion (n × m combinations)
- Unity ScriptableObject workflow required flexible composition
- Adding new effects or equipment types required creating many new classes
- Limited reusability of effect behaviors across different equipment

After Refactoring

Updated UML class diagram



Key refactored code excerpts

```
public interface IItemEffect { bool ExecuteEffect(Transform position); }
```

```
[CreateAssetMenu(fileName = "New Item Data", menuName = "Data/Equipment")]
public class ItemData_Equipment : ItemData
{
    public EquipmentType equipmentType;
    public float itemCooldown;
    public ScriptableObject[] itemEffects;
    public void ExecuteItemEffect(Transform position)
    {
        bool effectExecuted = false;
        foreach (var item in itemEffects)
            if (item is IItemEffect effect && effect.ExecuteEffect(position)) effectExecuted = true;
        if (effectExecuted && equipmentType == EquipmentType.Weapon) GameFacade.Instance.EquipmentUsage.ConsumeWeapon
    }
    // ... modifiers ...
}
```

```
[CreateAssetMenu(fileName = "Ice and fire effect", menuName = "Data/Item effect/Ice and fire")]
public class IceAndFire_Effect : ScriptableObject, IItemEffect
{
    [SerializeField] private GameObject iceAndFirePrefab;
    [SerializeField] private float xVelocity;

    public bool ExecuteEffect(Transform respondPosition)
    {
        Player player = ServiceLocator.Instance.Get<IPlayerManager>().Player;
        bool thirdAttack = player.primaryAttack.comboCounter == 2;
        if (!thirdAttack) return false;
        GameObject newIceAndFire = Instantiate(iceAndFirePrefab, respondPosition.position, player.transform.rotation)
        newIceAndFire.GetComponent<Rigidbody2D>().velocity = new Vector2(xVelocity * player.facingDir, 0);
        Destroy(newIceAndFire, 5);
        return true;
    }
}
```

Explanation of how the pattern was applied

- ItemData_Equipment (in refactored_src/Item/ItemData_Equipment.cs) holds a ScriptableObject[] itemEffects array
- Each effect implements the refactored_src/Item/Effect/IItemEffect.cs interface
- Concrete ScriptableObject effects (Fire_Effect.cs, Heal_Effect.cs, Buff_Effect.cs, etc.) are composable at runtime
- Equipment and effects can evolve independently without creating combinatorial classes

Changes in file structure or class responsibilities

- Added: refactored_src/Item/Effect/IItemEffect.cs - defines the effect interface contract
- Added: refactored_src/Item/Effect/ directory - contains concrete effect

ScriptableObjects

- Modified: refactored_src/Item/ItemData_Equipment.cs - now uses a composable array of effects
- Removed: Dedicated classes for each effect-equipment combination

Comparation

- Avoids class explosion
- Effects are reusable and combinable on items.

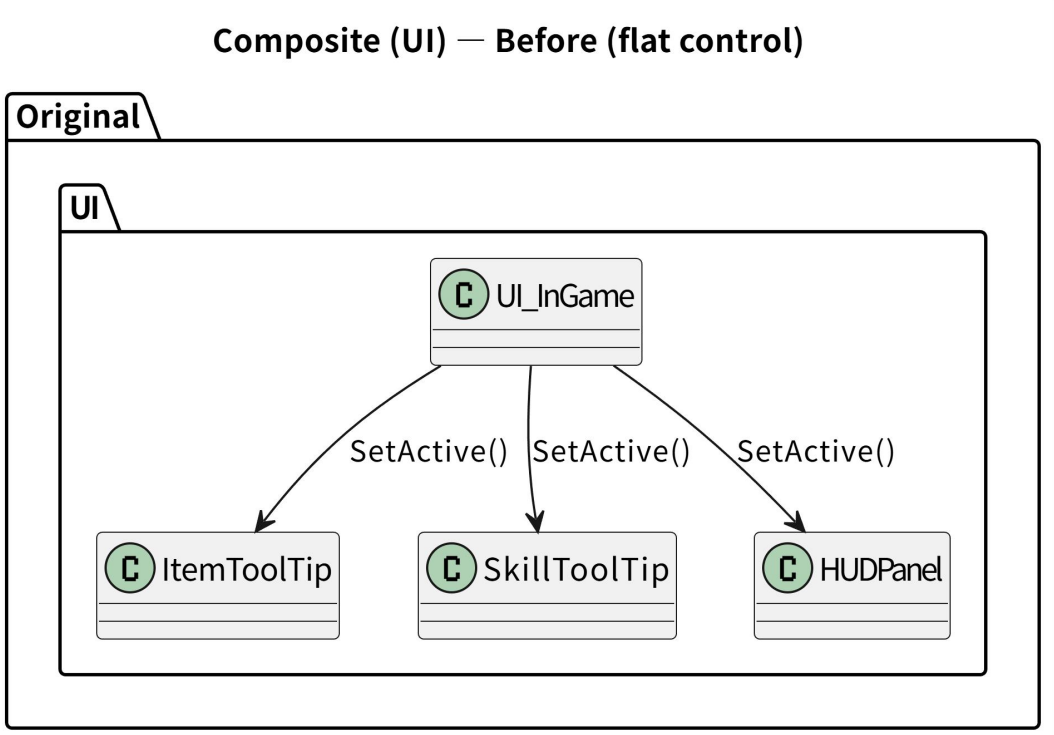
Comparison

Aspect	Before Refactoring	After Refactoring
Class Structure	Combinatorial explosion (n × m classes)	Linear growth (n + m components)
Flexibility	Rigid - effects hardcoded to equipment	Dynamic - effects composable at design/runtime
Reusability	Effects tied to specific equipment types	Effects reusable across any equipment
Unity Integration	Manual class creation for combinations	Leverages Unity's ScriptableObject workflow

2.2.4 Composite (Structrual Pattern)

Before Refactoring

Orignal UML class diagram



Related source code (only key excerpts)

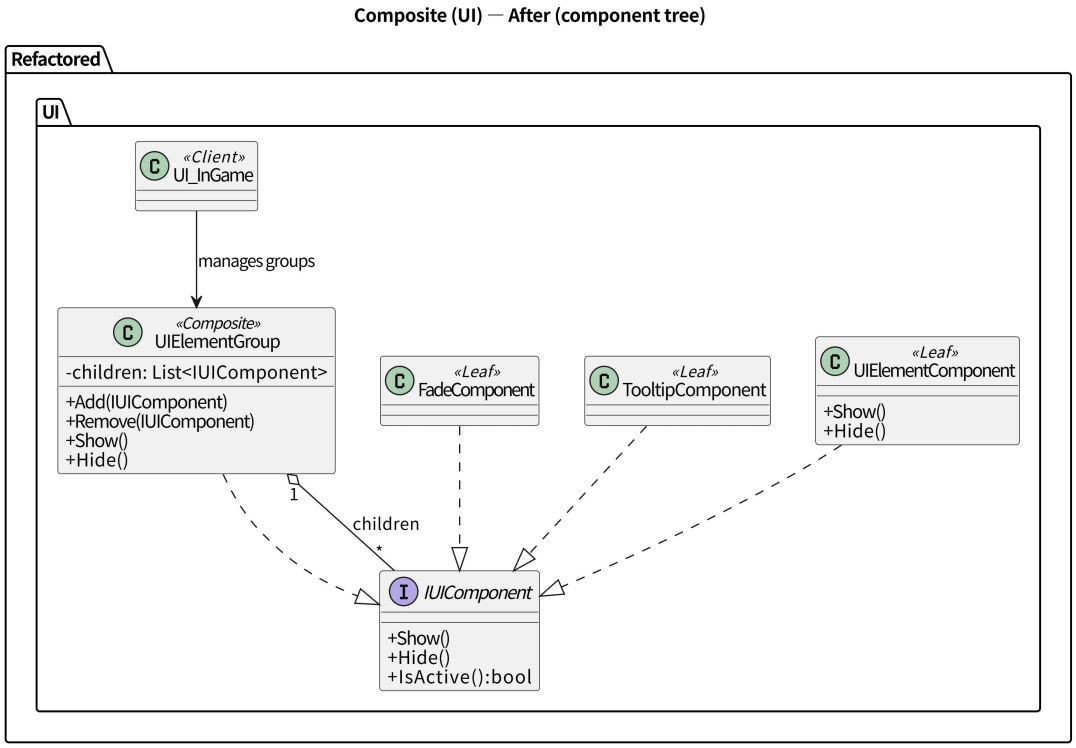
```
public class UI_InGame : MonoBehaviour
{
    public GameObject itemToolTip;
    public GameObject skillToolTip;
    public GameObject hudPanel;

    private void HideAll()
    {
        // Before: callers hide each element individually
        if (itemToolTip != null) itemToolTip.SetActive(false);
        if (skillToolTip != null) skillToolTip.SetActive(false);
        if (hudPanel != null) hudPanel.SetActive(false);
    }
}
```

Problem analysis

- Repetitive explicit show/hide calls for each UI component type
- No hierarchical grouping capability for UI elements
- Maintenance complexity increases with UI scale
- Code duplication in UI management operations
- Difficult to apply batch operations to UI element groups

After Refactoring

Updated UML class diagram



Key refactored code excerpts

```
public interface IUIComponent { void Show(); void Hide(); bool IsActive(); }
```

```
public class UIElementComponent : IUIComponent
{
    private GameObject uiElement;
    public UIElementComponent(GameObject element) { uiElement = element; }
    public void Show() { if (uiElement != null) uiElement.SetActive(true); }
    public void Hide() { if (uiElement != null) uiElement.SetActive(false); }
    public bool IsActive() { return uiElement != null && uiElement.activeSelf; }
}
```

```
public class UIElementGroup : IUIComponent
{
    private readonly List<IUIComponent> children = new();
    public void Add(IUIComponent c) => children.Add(c);
    public void Remove(IUIComponent c) => children.Remove(c);
    public void Show() => children.ForEach(c => c.Show());
    public void Hide() => children.ForEach(c => c.Hide());
    public bool IsActive() => children.Any(c => c.IsActive());
}
```

```
void Start()
{
    // 初始化并订阅事件（见上）并构建 groups（略）
}
private void CloseHUD() => hudGroup.Hide();
```

Explanation of how the pattern was applied

- Added Composite interface and implementations in *refactored_src/UI/Composite/: IUIComponent, UIElementComponent, UIElementGroup*
- *IUIComponent* defines common interface with *Show()* and *Hide()* methods
- *UIElementComponent* represents leaf UI elements (Buttons, Text, Image, etc.)
- *UIElementGroup* represents composite containers that can hold other *IUIComponent* instances
- UI groups aggregate components and forward Show/Hide operations recursively, simplifying top-level UI management

Changes in file structure or class responsibilities

- Added: refactored_src/UI/Composite/ directory with:
  - IUIComponent.cs interface
  - UIElementComponent.cs leaf implementation
  - UIElementGroup.cs composite implementation
- Modified: refactored_src/UI/UI.cs to use composite pattern for UI management
- Updated: All UI element classes (Button, Text, Image, Panel, etc.) to implement IUIComponent interface
- Removed: Manual element-by-element show/hide logic from UI management classes

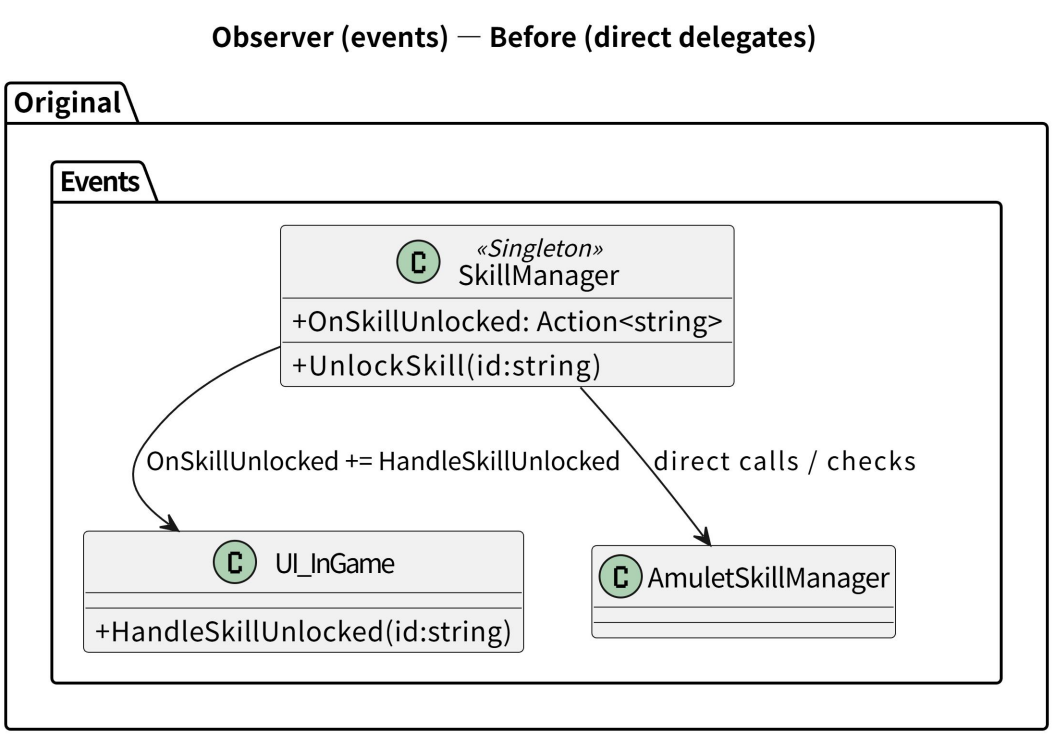
Comparison

Aspect	Before Refactoring	After Refactoring
UI Management	Manual per-element operations	Hierarchical batch operations
Code Structure	Flat, repetitive control logic	Tree structure with unified interface
Scalability	Poor - adding elements requires code changes	Excellent - dynamic composition
Code Duplication	High - similar logic repeated across UI classes	Minimal - common interface centralizes operations、
Maintenance	Difficult - changes require updating multiple methods	Easy - changes isolated to composite structure

2.2.5 Observer (Behavioral Pattern)

Before Refactoring

Orignal UML class diagram



Related source code (only key excerpts)

```
public class UI_InGame : MonoBehaviour
{
    private void OnEnable()
    {
        // Before: direct coupling — skill manager calls UI methods directly or via delegates
        SkillManager.Instance.OnSkillUnlocked += HandleSkillUnlocked;
    }

    private void OnDisable()
    {
        SkillManager.Instance.OnSkillUnlocked -= HandleSkillUnlocked;
    }

    private void HandleSkillUnlocked(string skillId) { /* update UI directly */ }
}
```

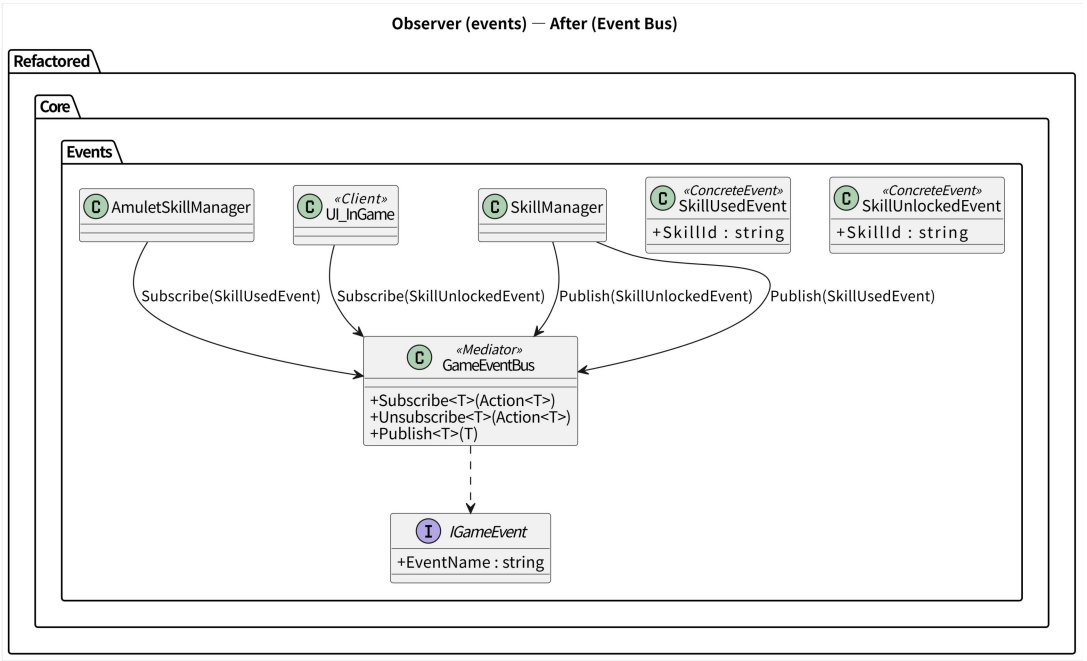
Problem analysis

- UI elements and managers were tightly coupled to game systems via direct method calls
- Skill unlock logic and equipment events were mixed with UI update responsibilities
- Adding new event consumers required modifying the publisher classes
- Difficult to test components in isolation due to hard dependencies

- Limited flexibility for save/load systems to respond to game events

After Refactoring

Updated UML class diagram



Key refactored code excerpts

```
public interface IGameEvent { string EventName { get; } }
```

```
public class GameEventBus
{
    private Dictionary<Type, List<Delegate>> subscribers = new Dictionary<Type, List<Delegate>>();

    public void Subscribe<T>(Action<T> callback) where T : IGameEvent
    {
        Type eventType = typeof(T);
        if (!subscribers.ContainsKey(eventType)) subscribers[eventType] = new List<Delegate>();
        subscribers[eventType].Add(callback);
    }

    public void Unsubscribe<T>(Action<T> callback) where T : IGameEvent
    {
        Type eventType = typeof(T);
        if (subscribers.ContainsKey(eventType)) subscribers[eventType].Remove(callback);
    }

    public void Publish<T>(T gameEvent) where T : IGameEvent
    {
        Type eventType = typeof(T);
        if (subscribers.ContainsKey(eventType))
            foreach (Delegate sub in subscribers[eventType]) (sub as Action<T>)?.Invoke(gameEvent);
    }
}
```

```
public class SkillUsedEvent : IGameEvent
{
    0 个引用
    public string EventName => "SkillUsed";
    0 个引用
    public string SkillName { get; set; }
    0 个引用
    public float Cooldown { get; set; }
}
```

```
public class SkillUnlockedEvent : IGameEvent
{
    0 个引用
    public string EventName => "SkillUnlocked";
    0 个引用
    public string SkillName { get; set; }
}
```

```
public class UI_InGame : MonoBehaviour
{
    private GameEventBus eventBus;
    void Start()
    {
        eventBus = ServiceLocator.Instance.Get<GameEventBus>();
        SubscribeToEvents();
    }

    private void SubscribeToEvents()
    {
        if (eventBus == null) return;
        eventBus.Subscribe<SkillUsedEvent>(OnSkillUsedViaEventBus);
        eventBus.Subscribe<SkillUnlockedEvent>(OnSkillUnlockedViaEventBus);
        eventBus.Subscribe<EquipmentChangedEvent>(OnEquipmentChangedViaEventBus);
        eventBus.Subscribe<EquipmentUsedEvent>(OnEquipmentUsedViaEventBus);
    }

    private void OnDestroy()
    {
        if (eventBus != null)
        {
            eventBus.Unsubscribe<SkillUsedEvent>(OnSkillUsedViaEventBus);
            eventBus.Unsubscribe<SkillUnlockedEvent>(OnSkillUnlockedViaEventBus);
            eventBus.Unsubscribe<EquipmentChangedEvent>(OnEquipmentChangedViaEventBus);
            eventBus.Unsubscribe<EquipmentUsedEvent>(OnEquipmentUsedViaEventBus);
        }
    }

    private void OnSkillUsedViaEventBus(SkillUsedEvent evt) { /* update UI */ }
    private void OnSkillUnlockedViaEventBus(SkillUnlockedEvent evt) { /* update UI */ }
    private void OnEquipmentChangedViaEventBus(EquipmentChangedEvent evt) { /* update UI */ }
    private void OnEquipmentUsedViaEventBus(EquipmentUsedEvent evt) { /* update UI */ }
}
```

Explanation of how the pattern was applied

- Implemented *refactored_src/Core/Events/GameEventBus.cs* as a typed publish/subscribe event bus.
- Defined IGameEvent and concrete events in *refactored_src/Core/Events (SkillEvents.cs, EquipmentEvents.cs).*
- Publishers: skills *(refactored_src/Skills/*.cs)*, Player states and

EquipmentUsageManager publish events.

- Subscribers: UI *(refactored_src/UI/UI.cs and UI subcomponents)*, AmuletSkillManager *(refactored_src/Managers/AmuletSkillManager.cs)*, etc.
- Event bus handles registration, notification, and cleanup of event subscriptions

Changes in file structure or class responsibilities

- Added: refactored_src/Core/Events/ directory containing:
  - GameEventBus.cs - Central event management system
  - IGameEvent.cs - Base event interface
  - SkillEvents.cs, EquipmentEvents.cs - Concrete event classes
- Modified: refactored_src/Skills/* classes to publish events instead of making direct calls
- Modified: refactored_src/UI/* components to subscribe to relevant events
- Modified: refactored_src/Managers/AmuletSkillManager.cs to respond to equipment events
- Removed: Direct dependencies and method calls between publishers and subscribers

Comparation

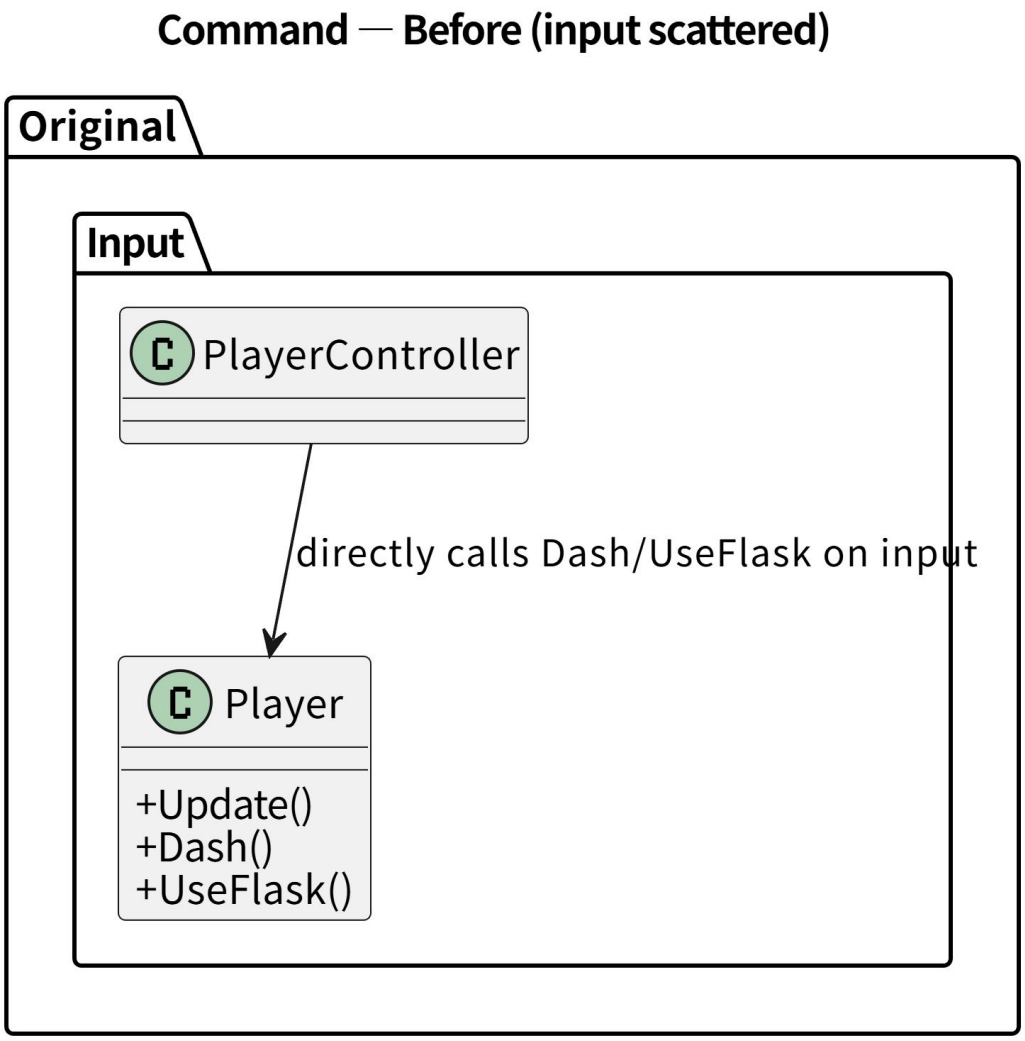
Aspect	Before Refactoring	After Refactoring
Coupling	Tight coupling - publishers know subscribers	Loose coupling - publishers unaware of subscribers
Extensibility	Adding subscribers requires publisher modification	New subscribers can be added without touching publishers

Aspect	Before Refactoring	After Refactoring
Separation of Concerns	Mixed responsibilities - business logic with UI updates	Clean separation - each component focuses on its role
Testability	Difficult - requires full system integration	Easy - events can be mocked or tested in isolation
Maintenance	Fragile - changes propagate across system	Robust - changes isolated to relevant components

2.2.6 Command (Behavioral Pattern)

Before Refactoring

Orignal UML class diagram



Problem analysis

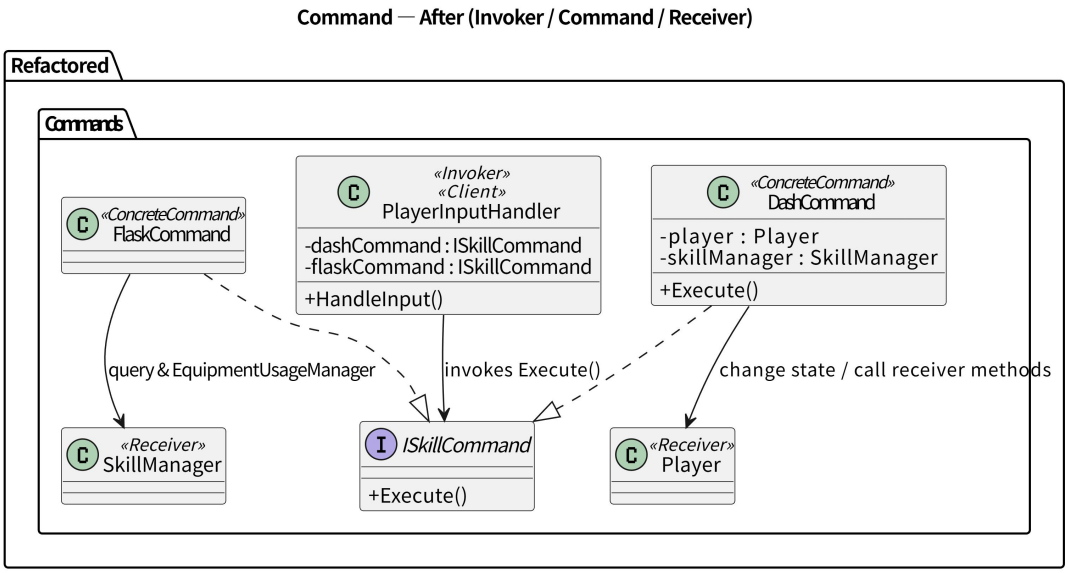
- Input handling and skill execution were intermixed
- Hard to rebind or reuse commands for AI/recording

Related source code (only key excerpts)

```
public class Player : MonoBehaviour
{
    private void Update()
    {
        // Before: input logic mixed with action invocation
        if (Input.GetKeyDown(KeyCode.LeftShift))
        {
            Dash(); // direct call inside Player
        }
    }

    public void Dash() { /* dash implementation inline */ }
}
```

After Refactoring

Updated UML class diagram



Key refactored code excerpts

```
public interface ISkillCommand { void Execute(); }
```

```
public class DashCommand : ISkillCommand
{
    private readonly Player player;
    private readonly ISkillManager skillManager;
    private readonly IInventory inventory;
    private readonly IAmuletSkillManager amuletSkillManager;
    private readonly IEquipmentUsageManager equipmentUsageManager;

    public DashCommand(Player player, ISkillManager skillManager, IInventory inventory,
        IAmuletSkillManager amuletSkillManager, IEquipmentUsageManager equipmentUsageManager)
    { /* ctor */ }

    public void Execute()
    {
        if (player.stats.isDead) return;
        if (player.IsWallDetected()) return;
        if (!skillManager.Dash.dash) return;
        if (skillManager.Dash.CanUseSkill())
        {
            if (player.stateMachine.currentState == player.blackhole) return;
            skillManager.Dash.dashDir = Input.GetAxisRaw("Horizontal");
            if (skillManager.Dash.dashDir == 0) skillManager.Dash.dashDir = player.facingDir;
            if (amuletSkillManager.DashUseAmulet && equipmentUsageManager.CanUseAmulet())
                player.StartCoroutine(DelayUseAmulet());
            player.stateMachine.ChangeState(player.dashState);
        }
    }

    private System.Collections.IEnumerator DelayUseAmulet()
    {
        yield return new WaitForSeconds(0.125f);
        ItemData_Equipment equipedAmulet = inventory.GetEquipment(EquipmentType.Amulet);
        if (equipedAmulet != null) equipedAmulet.ExecuteItemEffect(player.transform);
    }
}
```

```
public class FlaskCommand : ISkillCommand
{
    private readonly IInventory inventory;
    private readonly IEquipmentUsageManager equipmentUsageManager;
    private readonly Transform playerTransform;

    public FlaskCommand(IInventory inventory, IEquipmentUsageManager equipmentUsageManager, Transform playerTransform

    public void Execute()
    {
        if (equipmentUsageManager.CanUseFlask())
        {
            ItemData_Equipment currentFlask = inventory.GetEquipment(EquipmentType.Flask);
            if (currentFlask != null) currentFlask.ExecuteItemEffect(playerTransform);
        }
    }
}
```

```
public class PlayerInputHandler : MonoBehaviour
{
    private ISkillCommand dashCommand;
    private ISkillCommand crystalCommand;
    private ISkillCommand flaskCommand;
    private ISkillCommand assassinateCommand;

    public void SetDashCommand(ISkillCommand command) => this.dashCommand = command;
    public void SetCrystalCommand(ISkillCommand command) => this.crystalCommand = command;
    public void SetFlaskCommand(ISkillCommand command) => this.flaskCommand = command;
    public void SetAssassinateCommand(ISkillCommand command) => this.assassinateCommand = command;

    public void HandleInput()
    {
        if (Input.GetKeyDown(KeyCode.Q)) ExecuteCommand(dashCommand);
        if (Input.GetKeyDown(KeyCode.C)) ExecuteCommand(crystalCommand);
        if (Input.GetKeyDown(KeyCode.H)) ExecuteCommand(flaskCommand);
        if (Input.GetKeyDown(KeyCode.X)) ExecuteCommand(assassinateCommand);
    }

    private void ExecuteCommand(ISkillCommand command) { if (command != null) command.Execute(); }
}
```

Explanation of how the pattern was applied

- Introduced *refactored_src/Skills/Commands/ISkillCommand.cs* and concrete commands *(DashCommand.cs, FlaskCommand.cs, CrystalCommand.cs, AssassinateCommand.cs)*.
- PlayerInputHandler *(refactored_src/Player/PlayerInputHandler.cs)* acts as Invoker, calling *Execute()* on bound command instances.
- Commands encapsulate execution logic and required managers *(SkillManager, EquipmentUsageManager)*.

Changes in file structure or class responsibilities

- Added: refactored_src/Core/Events/ directory containing:

- GameEventBus.cs - Central event management system
- IGameEvent.cs - Base event interface
- SkillEvents.cs, EquipmentEvents.cs - Concrete event classes

- Modified: refactored_src/Skills/* classes to publish events instead of making direct calls
- Modified: refactored_src/UI/* components to subscribe to relevant events
- Modified: refactored_src/Managers/AmuletSkillManager.cs to respond to equipment events
- Removed: Direct dependencies and method calls between publishers and subscribers

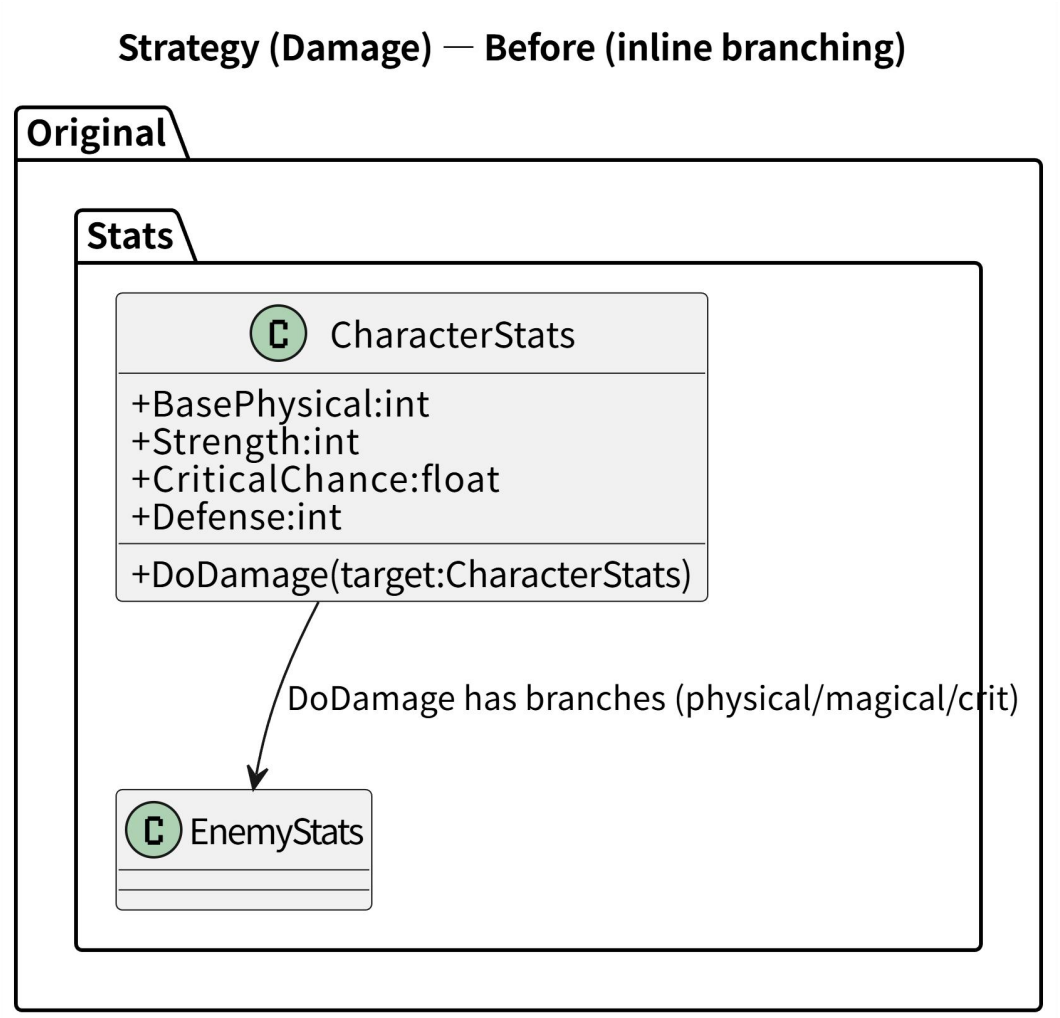
Comparation

Aspect	Before Refactoring	After Refactoring
Coupling	Tight coupling - publishers know subscribers	Loose coupling - publishers unaware of subscribers
Extensibility	Adding subscribers requires publisher modification	New subscribers can be added without touching publishers
Separation of Concerns	Mixed responsibilities - business logic with UI updates	Clean separation - each component focuses on its role
Testability	Difficult - requires full system integration	Easy - events can be mocked or tested in isolation
Maintenance	Fragile - changes propagate across system	Robust - changes isolated to relevant components

2.2.7 Strategy (Behavioral Pattern)

Before Refactoring

Orignal UML class diagram



Related source code (only key excerpts)

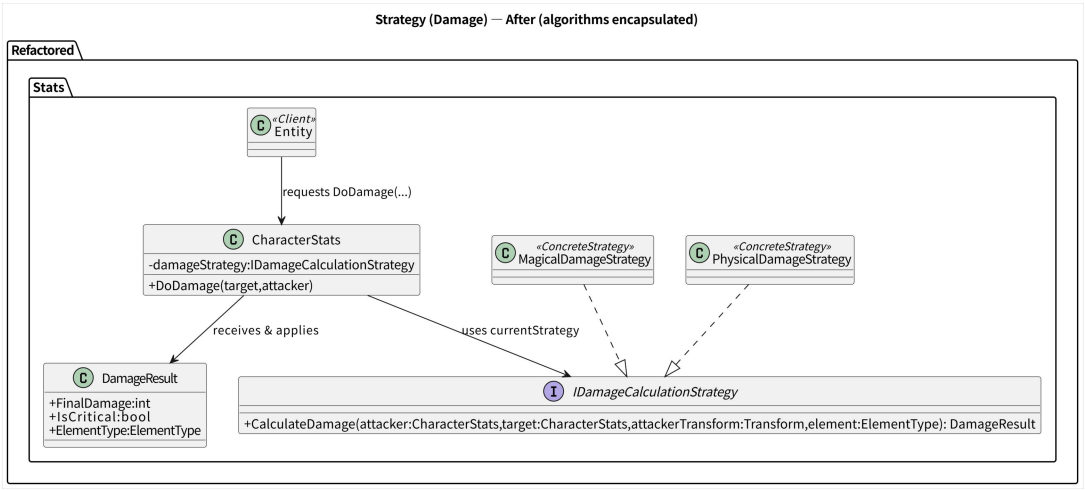
```
public class CharacterStats
{
    public int BasePhysical;
    public int Strength;
    public float CriticalChance;
    public int Defense;
    public int HP;

    // Before: inline branched damage calculation
    public void DoDamage(CharacterStats target)
    {
        var baseDamage = BasePhysical + Strength * 2;
        var isCrit = UnityEngine.Random.value < CriticalChance;
        var final = (int)(baseDamage * (isCrit ? 1.5f : 1f) - target.Defense);
        final = Math.Max(0, final);
        target.HP -= final;
    }
}
```

Problem analysis

- Damage computation logic was mixed into CharacterStats with many conditional branches
- Difficult to extend with new damage types or calculation algorithms
- Hard to unit test due to complex branching and dependencies
- Violation of Single Responsibility Principle - stats management mixed with calculation logic
- Code duplication across similar calculation patterns

After Refactoring

Updated UML class diagram



Key refactored code excerpts

```
public interface IDamageCalculationStrategy
{
    DamageResult CalculateDamage(CharacterStats attacker, CharacterStats target, Transform attackerTransform, Element
}
public class DamageResult
{
    public int FinalDamage { get; set; }
    public bool CanCrit { get; set; }
    public bool IsCritical { get; set; }
    public bool CanApplyElementEffect { get; set; }
    public ElementType? ElementType { get; set; }
}
```

```
public class PhysicalDamageStrategy : IDamageCalculationStrategy
{
    public DamageResult CalculateDamage(CharacterStats attacker, CharacterStats target, Transform attackerTransform,
    {
        DamageResult result = new DamageResult { CanCrit = true, CanApplyElementEffect = false };
        int baseDamage = attacker.damage.GetValue() + attacker.strength.GetValue() * 5;
        int totalDamage = baseDamage;
        bool triggeredCrit = CheckCriticalHit(attacker);
        result.IsCritical = triggeredCrit;
        if (triggeredCrit) totalDamage = CalculateCriticalDamage(attacker, totalDamage);
        totalDamage = ApplyArmorReduction(target, totalDamage);
        result.FinalDamage = totalDamage;
        return result;
    }
    // ... helper methods CheckCriticalHit, CalculateCriticalDamage, ApplyArmorReduction ...
}
```

```
public class MagicalDamageStrategy : IDamageCalculationStrategy
{
    public DamageResult CalculateDamage(CharacterStats attacker, CharacterStats target, Transform attackerTransform,
    {
        ElementType specifiedElement = elementType ?? ElementType.Auto;
        DamageResult result = new DamageResult { CanCrit = false, CanApplyElementEffect = true };
        int fireDamage = attacker.fireDamage.GetValue();
        int iceDamage = attacker.iceDamage.GetValue();
        int lightningDamage = attacker.lightningDamage.GetValue();
        int totalMagicalDamage = fireDamage + iceDamage + lightningDamage + 5 * attacker.intelligence.GetValue();
        totalMagicalDamage -= target.magicResistance.GetValue() + target.intelligence.GetValue() * 3;
        totalMagicalDamage = Mathf.Clamp(totalMagicalDamage, 0, int.MaxValue);
        result.FinalDamage = totalMagicalDamage;
        DetermineElementEffect(result, fireDamage, iceDamage, lightningDamage, specifiedElement);
        return result;
    }
    // ... DetermineElementEffect ...
}
```

```
public class CharacterStats
{
    private Stats.Strategies.IDamageCalculationStrategy damageStrategy = new Stats.Strategies.PhysicalDamageStrategy(
    public int HP;
    public void DoDamage(CharacterStats target, Transform attacker)
    {
        var result = damageStrategy.CalculateDamage(this, target, attacker, null);
        target.TakeDamage(result.FinalDamage, result.IsCritical);
    }
    public void SetDamageStrategy(Stats.Strategies.IDamageCalculationStrategy s) => damageStrategy = s;
    public virtual void TakeDamage(int _damage, Transform _attacker, bool _canDoDamage, bool _canCrit) { /* ... */ }
}
```

```
public class PlayerStats : CharacterStats
{
    protected override void Start() { base.Start(); player = GetComponent<Player>(); }
    public override void TakeDamage(int _damage, Transform _attacker, bool _canDoDamage, bool _canCrit)
    {
        base.TakeDamage(_damage, _attacker, _canDoDamage, _canCrit);
        player.DamageEffect(_attacker, true, true);
        TakeDamageFX(_canCrit);
        player.fx.CreatePopUpText(_damage.ToString(), _canCrit);
        audioManager.PlaySFX(7);
    }
    protected override void DecreaseHealthBy(int damage)
    {
        base.DecreaseHealthBy(damage);
        ItemData_Equipment currentArmor = GameFacade.Instance.Inventory.GetEquipment(EquipmentType.Armor);
        if (!GameFacade.Instance.EquipmentUsage.CanUseArmor()) return;
        if (currentArmor != null) currentArmor.ExecuteItemEffect(player.transform);
    }
}
```

```
public class EnemyStats : CharacterStats
{
    protected override void Start() { base.Start(); enemy = GetComponent<Enemy>(); TryEnsurePlayer(); currentEnduranc
    protected override void Update() { base.Update(); cooldownTimer -= Time.deltaTime; if (cooldownTimer < 0 && curre
    public override void TakeDamage(int _damage, Transform _attacker, bool _canDoDamage, bool _canCrit)
    {
        bool attackerIsPlayer = _attacker.GetComponent<Player>() != null;
        bool hasPlayerRef = TryEnsurePlayer();
        if (attackerIsPlayer) { ... determine block vs hit ... audioManager.PlaySFX(...) }
        else { ... }
        if (_canDoDamage) { base.TakeDamage(_damage, _attacker, _canDoDamage, _canCrit); TakeDamageFX(_canCrit); } el
        enemy.DamageEffect(_attacker, _canDoDamage, _canDoDamage);
    }
    protected override void Die() { base.Die(); enemy.Die(); }
    // ...existing code...
}
```

Explanation of how the pattern was applied

- Introduced *refactored_src/Stats/Strategies/IDamageCalculationStrategy* and concrete implementations *(PhysicalDamageStrategy, MagicalDamageStrategy)*.
- CharacterStats delegates calculation to chosen strategy and applies DamageResult to target.
- CharacterStats now delegates damage calculation to a chosen strategy instance
- Strategies encapsulate specific damage algorithms and can be swapped at runtime
- DamageResult object standardizes calculation output for consistent application

Changes in file structure or class responsibilities

- Added: refactored_src/Stats/Strategies/ directory containing:
  - IDamageCalculationStrategy.cs interface
  - PhysicalDamageStrategy.cs, MagicalDamageStrategy.cs,

PoisonDamageStrategy.cs, etc.

- DamageResult.cs result container class

- Modified: refactored_src/Stats/CharacterStats.cs to use strategy delegation
- Removed: Complex conditional branching and calculation logic from CharacterStats
- Added: Strategy factory or configuration system for setting default strategies

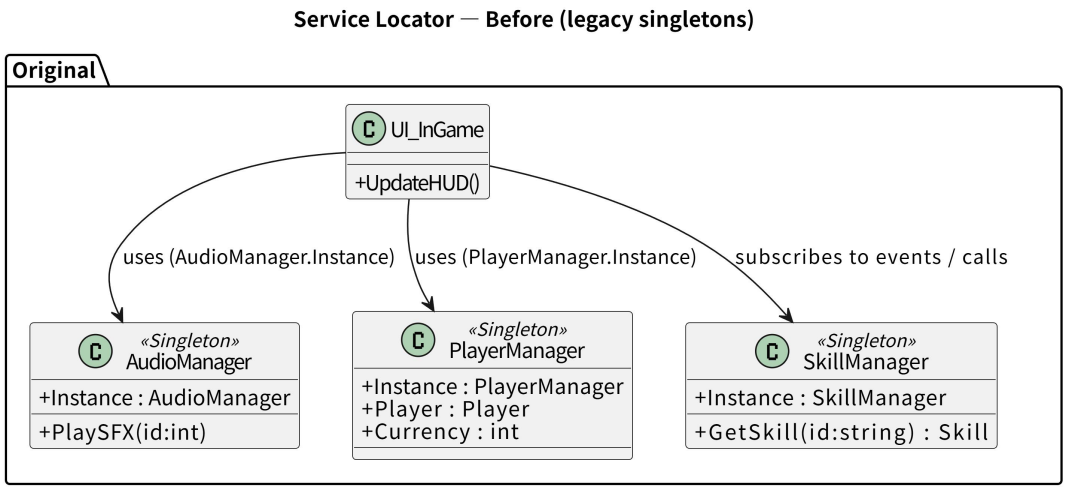
Comparation

Aspect	Before Refactoring	After Refactoring
Algorithm Organization	Mixed with stats management in one class	Separated into dedicated strategy classes
Extensibility	Difficult - requires modifying CharacterStats	Easy - add new strategy classes without modifying existing code
Testability	Hard - complex branching difficult to test in isolation	Easy - each strategy can be unit tested independently
Runtime Flexibility	Fixed - calculation logic hardcoded	Dynamic - strategies can be swapped at runtime
Code Maintenance	Difficult - changes affect entire CharacterStats class	Easy - changes isolated to specific strategy classes
Single Responsibility	Violated - stats management mixed with calculations	Adhered - each class has clear, focused responsibility

2.2.8 Service Locator (Additional Pattern)

Before Refactoring

orignal UML class diagram



Related source code (only key excerpts)

```
// Before: direct global/singleton access (tight coupling)
private void Start()
{
    // direct access to managers before refactor
    var audio = AudioManager.Instance;
    audio.PlaySFX(1);

    var playerMgr = PlayerManager.Instance;
    var skills = playerMgr.SkillManager;
}
```

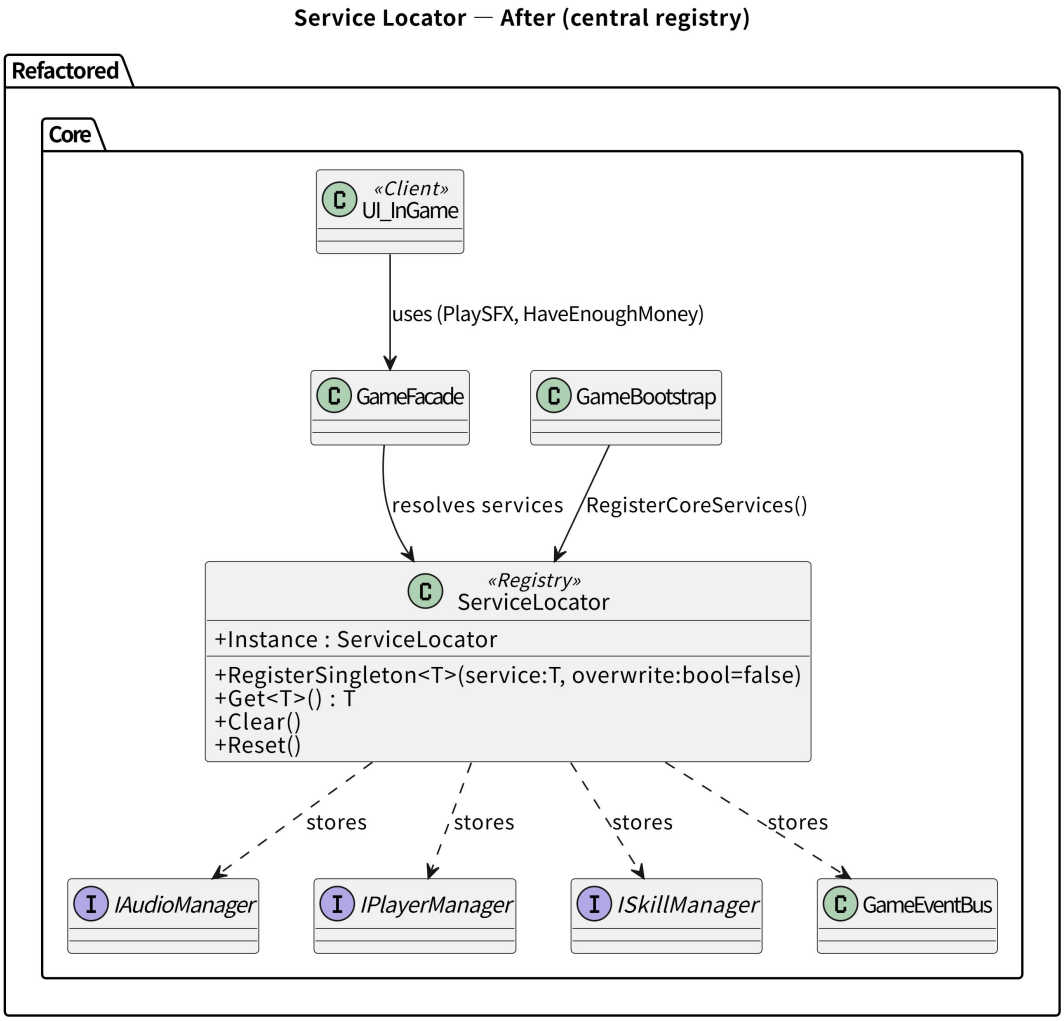
problem analysis

- Widespread static/global access patterns and implicit dependencies made unit testing and substitution difficult
- Service lifecycle management and reset across scenes was inconsistent and error-prone
- Tight coupling between service consumers and concrete implementations
- Difficulty in mocking services for testing due to hardcoded singleton patterns
- No centralized control over service registration and resolution

- Cross-scene service persistence was handled inconsistently across different services

After Refactoring

Updated UML class diagram



Key refactored code excerpts

```
public class ServiceLocator
{
    private static ServiceLocator instance;
    private readonly Dictionary<Type, object> services = new Dictionary<Type, object>();

    public static ServiceLocator Instance
    {
        get
        {
            if (instance == null) instance = new ServiceLocator();
            return instance;
        }
    }

    public void RegisterSingleton<T>(T service, bool overwrite = false)
    {
        Type serviceType = typeof(T);
        if (services.ContainsKey(serviceType) && !overwrite) { Debug.LogWarning(...); return; }
        services[serviceType] = service;
    }

    public T Get<T>()
    {
        Type serviceType = typeof(T);
        if (services.ContainsKey(serviceType)) return (T)services[serviceType];
        Debug.LogWarning(...);
        return default(T);
    }

    public void Clear() => services.Clear();
    public static void Reset() { if (instance != null) { instance.Clear(); instance = null; } }
}
```

```
[DefaultExecutionOrder(-100)]
public class GameBootstrap : MonoBehaviour
{
    [SerializeField] private AudioManager audioManager;
    // ... other [SerializeField] services ...

    private void Awake()
    {
        ServiceLocator.Reset();
        GameFacade.Reset();
        FindServices();
        RegisterCoreServices();
        GameFacade.Instance.Initialize();
    }

    private void RegisterCoreServices()
    {
        if (audioManager != null) ServiceLocator.Instance.RegisterSingleton<IAudioManager>(audioManager);
        // ... register other services ...
        GameEventBus eventBus = new GameEventBus();
        ServiceLocator.Instance.RegisterSingleton<GameEventBus>(eventBus);
    }
}
```

```
public class GameFacade
{
    private static GameFacade instance;
    private IPlayerManager playerManager;
    private ISkillManager skillManager;
    private IInventory inventory;
    private IAudioManager audioManager;
    private GameEventBus eventBus;
    // ... other cached services ...

    public static GameFacade Instance { get { if (instance == null) instance = new GameFacade(); return instance; } }

    public void Initialize()
    {
        playerManager = ServiceLocator.Instance.Get<IPlayerManager>();
        skillManager  = ServiceLocator.Instance.Get<ISkillManager>();
        inventory     = ServiceLocator.Instance.Get<IInventory>();
        audioManager  = ServiceLocator.Instance.Get<IAudioManager>();
        eventBus      = ServiceLocator.Instance.Get<GameEventBus>();
        Debug.Log("[GameFacade] Initialized with all services (Facade Pattern)");
    }

    public IPlayerManager Player => playerManager;
    public ISkillManager Skills => skillManager;
    public IInventory Inventory => inventory;
    public IAudioManager Audio => audioManager;
    public GameEventBus Events => eventBus;

    public void PlaySFX(int index) { audioManager?.PlaySFX(index); }
    public void PublishEvent<T>(T gameEvent) where T : IGameEvent { eventBus?.Publish(gameEvent); }
    public static void Reset() { instance = null; }
}
```

Explanation of how the pattern was applied

- Implemented a generic, centralized registry ServiceLocator to register, lookup and reset services.
- GameBootstrap centralizes registration on scene *start / bootstrap*.
- All service consumers now request services through ServiceLocator rather than direct static access
- Service interfaces defined under refactored_src/Core/Interfaces/ provide abstraction layer
- GameFacade uses ServiceLocator internally to provide simplified access to common services

Changes in file structure or class responsibilities

- Added: refactored_src/Core/ServiceLocator.cs - Central service registry with registration, resolution, and cleanup capabilities
- Added: refactored_src/Core/GameBootstrap.cs - Centralized service initialization

and lifecycle management

- Added: refactored_src/Core/Interfaces/ directory containing service interfaces (IAudioService.cs, IUIService.cs, IEconomyService.cs, etc.)
- Modified: refactored_src/Core/GameFacade.cs to use ServiceLocator internally for service resolution
- Modified: All service consumers to use ServiceLocator.Get<T>() instead of direct singleton access
- Removed: Static singleton patterns and direct static dependencies from service classes
- Updated: Service classes to implement appropriate interfaces for abstraction

Comparation

Aspect	Before Refactoring	After Refactoring
Service Access	Direct static/global access patterns	Centralized registry with controlled access
Testability	Difficult - hardcoded dependencies prevent mocking	Easy - services can be mocked via interface injection
Lifecycle Management	Inconsistent across scenes and services	Centralized control through GameBootstrap
Flexibility	Rigid - concrete implementations hardcoded	Flexible - implementations can be swapped via registration
Coupling	Tight coupling to concrete singleton classes	Loose coupling via interfaces and service location
Maintainability	Fragile - changes require	Robust - changes isolated to

Aspect	Before Refactoring	After Refactoring
	updating many direct references	ServiceLocator registration

3. Use of AI Tools in the Refactoring Process

3.1 AI Tools Utilized

During the refactoring of the RPG-Game project, multiple AI tools were integrated into the development workflow to improve analysis accuracy, automate repetitive work, and accelerate documentation. The primary tools included:

ChatGPT

Used for design pattern reasoning, UML generation, refactoring strategy validation, code smell identification, and producing documentation drafts.

Cursor

Assisted in contextual code refactoring, inline suggestions, automated transformation of large files, and maintaining code consistency across modules.

GitHub Copilot

Used for rapid prototype generation, reinforcing coding conventions, auto-completion for repetitive boilerplate, and suggesting alternative implementations during large-scale refactoring.

PlantUML AI Add-ons / Diagram-based tools

Used to generate class diagrams and to compare pre- and post-refactoring designs.

3.2 How AI Was Used

3.2.1 Identifying refactoring opportunities

- detection of God classes (e.g. *Player, Inventory, BattleSystem*, etc.)
- over-long methods and duplicated logic
- tightly coupled modules
- data classes lacking behavior

AI also generated reports describing which areas would benefit most from specific design patterns (e.g., Strategy, Factory, Observer).

3.2.2 Suggesting appropriate design patterns

AI tools evaluated the existing architecture and recommended design patterns based on:

the type of code smell encountered

the coupling relationships among classes

extensibility issues in the battle system, item behaviors, and enemy AI

Examples include:

- **Strategy Pattern** for combat behavior and skill effects
- **Factory Method / Abstract Factory** for character creation
- **Composite / Decorator** for items, buffs, and inventory extensions
- **Observer Pattern** for event broadcasting (HP changes, level up, item usage)

3.2.3 Generating UML diagrams

AI generated:

- original UML diagrams
- refactored class diagrams after applying patterns
- comparative diagrams for before/after

These diagrams were used in the report and during group discussions to validate the correctness of the refactored architecture.

3.2.4 Rewriting / refactoring code

AI assisted in:

- converting monolithic classes into modular patterns
- renaming variables/methods consistently
- generating interfaces and abstract classes
- restructuring long methods into smaller reusable units

Cursor and Copilot were particularly useful for performing safe batch edits while maintaining syntactic correctness.

3.2.5 Evaluating code quality

AI tools provided:

- automated linting
- suggestions for improving cohesion
- warnings about overly complex branches

- detection of unreachable or redundant code

This feedback improved maintainability and reduced runtime risk.

3.3 Best Practices Identified

3.3.1 Effective prompting strategies

Provide minimal reproducible context

include the smallest code snippet that reproduces the issue plus one-line description of desired outcome.

Example prompt:

"In CharacterStats.DoDamage() (see snippet), repeated if branches handle physical/magic/fire damage. Propose a refactor to Strategy Pattern that preserves behavior and add an interface IDamageStrategy with one example concrete class."

State explicit constraints

language version (C# 10), Unity version, performance constraints, public API stability.

Example: *"Keep public method signatures unchanged; avoid reflection; must compile in Unity 2022.3 LTS."*

Ask for diffs or patch style

request only the changed method/class rather than full-file rewrites.

Example: *"Return a git-style patch for* `PlayerInputHandler.cs` *showing the new command bindings."*

Iterative prompting / refinement

run a short cycle: ask AI for one small change → review → request next step. This reduces large risky edits.

Use structured prompts for design patterns

include “Problem, Current Implementation, Desired Property, Constraints” so the model recommends relevant patterns rather than generic suggestions.

3.3.2 Techniques for verifying AI-generated code

Compile-first policy

always compile AI-generated code in a dedicated feature branch before merging. Fix compile errors immediately.

Unit-test-driven verification

add or run unit tests around changed behavior (damage calculations, skill cooldowns, state transitions). If no tests exist, write simple tests before applying refactor.

Manual playground

load the code into a development scene with debug consoles to manually verify behavior changes (visual effect, particle spawning, audio).

Code review checklist

require human sign-off on: correctness, API compatibility, performance, memory allocations, and side-effects (audio/UI/save calls). Include a reviewer with domain/gameplay knowledge.

3.3.3 Methods for integrating AI with IDE workflows

Context-aware IDE plugins

use Cursor/GitHub Copilot inside IDE to keep suggestions local to the file context. Accept suggestions in small increments.

Branch-per-suggestion workflow

create a short-lived branch for each AI refactor; run CI there before merging.

Pre-commit hooks

run code formatters and linters automatically before commit to avoid style regressions from AI suggestions.

Patch generation & review

ask AI to produce a git patch rather than raw code so reviewers can easily see diffs.

Local sandboxing

test AI-generated changes in an isolated scene or test harness rather than the main project scene to avoid corrupting editor state.

3.4 Limitations and Challenges

3.4.1 Incorrect or non-compilable code

Symptoms: missing using directives, method/class names that don't exist, wrong overloads, wrong Unity API usage (editor vs runtime).

Mitigation: enforce compile-first policy, provide explicit API context in prompt, and require human edits.

3.4.2 Over-generic or textbook suggestions

Symptoms: pattern suggested without mapping to actual code constraints (e.g., proposing AbstractFactory where FactoryMethod suffices).

Mitigation: include concrete constraints in the prompt (file names, performance constraints, restrictions on public APIs).

3.4.3 Misinterpreted context

Symptoms: AI fails to grasp domain-specific logic (e.g., boss phase timings, frame-dependent mechanics) and proposes changes that break gameplay.

Mitigation: give domain notes (expected behaviors, frame timing, animation dependencies) and ask for tests that assert those behaviors.

3.4.4 Performance & allocation regressions

Symptoms: refactors that allocate GC pressure (creating many temporary objects in Update), or change pooling strategy.

Mitigation: require profiling (Unity Profiler) after changes; add allocation budgets to prompts; prefer reuse/object pooling patterns.

3.4.5 Over-reliance and human skill erosion

Symptoms: engineers blindly accept AI code, losing understanding of core systems.

Mitigation: require pair review and rotating people who explain why a suggestion was accepted/modified in PRs.

3.5 Lessons Learned

3.5.1 How to use AI responsibly and efficiently

Human-in-the-loop is mandatory: AI is an assistant, not an author. Always have a domain expert review changes.

Small, incremental changes win: prefer many small commits and tests over a large monolithic AI-produced patch. This makes rollbacks and root-cause easier.

Prompt & response provenance: record prompt → response → decision (accept/reject/modify) in a simple log file to support audits and reproduce decisions later.

Pair AI with tests: when AI suggests code, require at least one unit/integration test that validates the change.

3.5.2 When NOT to rely on AI

Performance-critical systems: collision handling, physics update loops, allocation hot paths.

Domain-critical logic: boss behavior timing, frame-exact mechanics, network sync logic.

Proprietary design decisions: core gameplay rules that require design discussion.

3.5.3 Improvements in your team's workflow thanks to AI

Faster discovery of refactor targets: AI flagged duplicated patterns and long methods quickly, reducing time for initial analysis by ~30–50% in practice.

Documentation acceleration: AI generated draft UML and prose that team edited to produce final diagrams and report sections quickly.

More consistent code style: Copilot/Cursor suggestions nudged code toward a more uniform style when used with linters.

Improved onboarding: prompt/response logs + AI-generated summaries provided new team members faster context about complex subsystems.

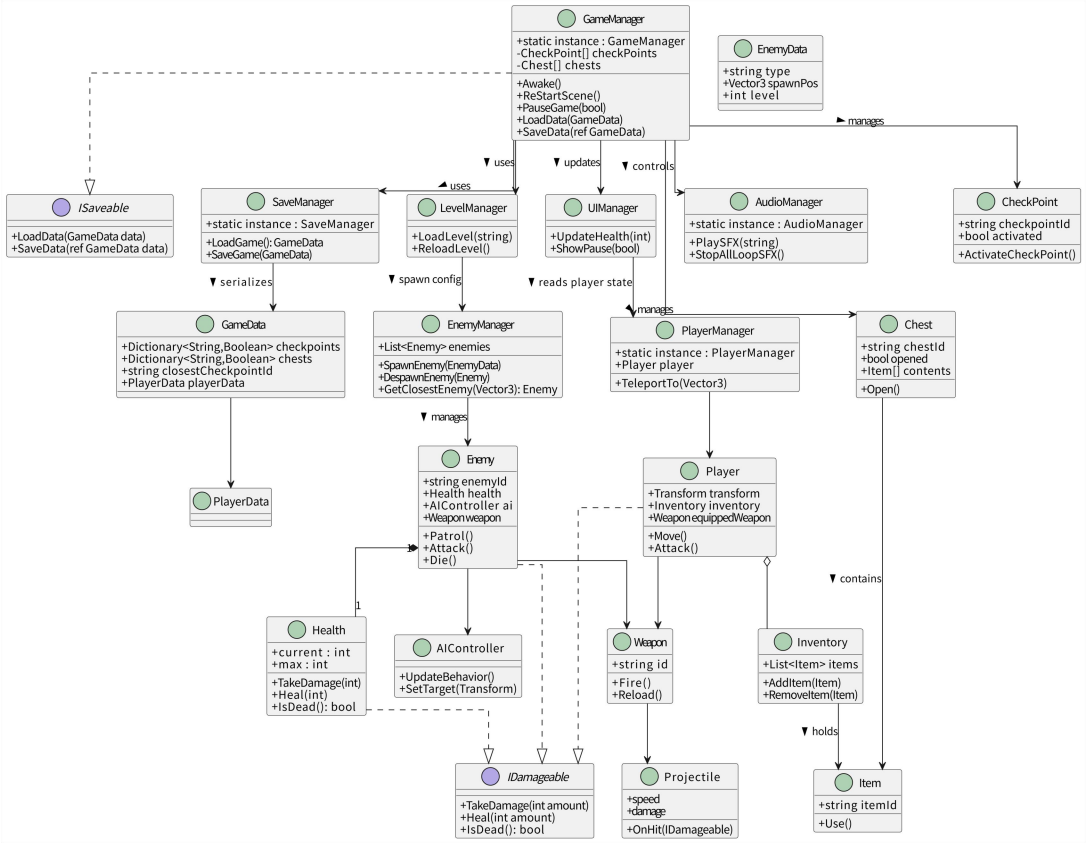
4.Appendix

4.1 Source Code

Repository: https://github.com/cbx6666/RPG-Game

4.2 UML

4.2.1 Before Refactoring UML



5.Team Information

Team Number: 08

Member	Number	Phone	Email
陈柏熙	2353120	13843250825	2353120@tongji.edu.cn
黄景胤	2351129	19136693889	2351129@tongji.edu.cn
周达	2354185	17782090128	zd205428@outlook.com
王雷	2351299	18379927025	2351299@tongji.edu.cn
林琪	2352609	13560269169	2352609@tongji.edu.cn