同济大学
TONGJI UNIVERSITY

Unity RPG Project Refactoring

Group 8

2353120 陈柏熙　2351129 黄景胤

2351299 王雷　2354185 周达　2352609 林琪

2025/12/04

同济大学
TONGJI UNIVERSITY

Contents

同濟大學
TONGJI UNIVERSITY

1 Project Introduction

Project Introduction

This project is a **Unity-based 2D Action Role-Playing Game** featuring a fully interactive combat system, enemy AI, skill mechanics, inventory and equipment management, player progression, and persistent save/load functionality. It follows professional Unity architecture with component-based entities, finite state machines, and event-driven systems to ensure modularity and scalability.

Major Functionalities

Player System

- FSM-driven player controller (movement, jump, attack, defend, use skills, death)
- Diverse skills: ranged attacks, dash, clone, AoE ultimate
- Inventory, equipment, attributes, and elemental effects

Enemy & Boss AI

- Hierarchical FSM AI
- Various enemy types (Skeletons, Slimes)
- Multi-phase boss with melee, ranged, healing, teleport, and AoE magic

Combat System

- Supports both physical and magical damage types
- Comprehensive status effect system featuring enemy guarding, conditional damage windows, and player counter mechanics.
- Fluid combat flow allowing chaining of attacks, movement, and skills for fast-paced action gameplay.

UI & Gameplay Systems

- Includes HUD (HP, stamina, currency), inventory & equipment UI, skill tree interface, and more.
- Crafting system, resource management, and audio settings with separate SFX/BGM controls.
- Persistent save/load system covering player progress, items, skills, and user settings.

同济大学
TONGJI UNIVERSITY

Overview of Refactoring Work

2

Motivation for Refactoring

1. Code Quality Issues

- God classes (Too many responsibilities)
- Extensive duplicated logic
- Frequent direct access to multiple singletons, causing tight coupling.
- Event handling scattered across scripts, leading to lifecycle issues.

2. Design Problems

- Multiple violations of the Single Responsibility Principle.
- Inconsistent abstraction boundaries; many systems rely on concrete classes rather than interfaces.
- Enemy state instantiation duplicated in each subclass.
- Tight coupling between UI and gameplay logic.

```
/// <summary>
/// 敌人类 - 继承自Entity
/// 实现敌人的AI行为、状态机、攻击系统和奖励机制
/// 包含敌人特有的眩晕、格挡、发现玩家等功能
/// </summary>
0 个引用
public class Enemy : Entity
{
    1 个引用
    [SerializeField] protected LayerMask whatIsPlayer;       // 玩家层级掩码

    [Header("Stunned info")]
    0 个引用
    public float stunnedDuration;                            // 眩晕持续时间
    0 个引用
    public Vector2 stunnedDistance;                          // 眩晕时的击退距离
    3 个引用
    protected bool canBeBlocked;                             // 是否可以被格挡
    1 个引用
    public bool isStunned;                                   // 是否处于眩晕状态
    6 个引用
    private Coroutine temporaryFreezeCoroutine;              // 临时冰冻协程

    2 个引用
    [SerializeField] protected GameObject counterImage;      // 格挡提示图片

    [Header("Move info")]
    5 个引用
    public float moveSpeed;                                  // 移动速度
    0 个引用
    public float idleTime;                                   // 空闲时间
    0 个引用
    public float battleTime;                                 // 战斗时间
    3 个引用
    private float defaultMoveSpeed;                          // 默认移动速度

    [Header("Flip info")]
    1 个引用
    [SerializeField] private float flipCooldown = 0.5f;      // 翻转冷却（秒）
    2 个引用
    private float lastFlipTime;                              // 上次翻转时间

    [Header("Attack info")]
    1 个引用
    public float attackDistance;                             // 攻击距离
    0 个引用
    public float attackCooldown;                             // 攻击冷却时间
    0 个引用
    public float minAttackCoolDown;                          // 最小攻击冷却时间
    0 个引用
    public float maxAttackCoolDown;                          // 最大攻击冷却时间
    0 个引用
```

god class
(original_src/Enemy/Enemy.cs)

```
0 个引用
protected override void Awake()
{
    base.Awake();

    idleState = new SkeletonIdleState(this, stateMachine, "Idle", this);
    moveState = new SkeletonMoveState(this, stateMachine, "Move", this);
    battleState = new SkeletonBattleState(this, stateMachine, "Move", this);
    attackState = new SkeletonAttackState(this, stateMachine, "Attack", this);
    stunnedState = new SkeletonStunnedState(this, stateMachine, "Stunned", this);
    deadState = new SkeletonDeadState(this, stateMachine, "Die", this);
    blockedState = new SkeletonBlockedState(this, stateMachine, "Blocked", this);
}
```

Enemy state instantiation duplicated
(original_src/Enemy/Skeleton/Enemy_Skeleton.cs)

```
// 获取UI组件
inventoryItemSlot = inventorySlotParent.GetComponentsInChildren<UI_ItemSlot>();
stashItemSlot = stashSlotParent.GetComponentsInChildren<UI_ItemSlot>();
equipmentSlot = equipmentSlotParent.GetComponentsInChildren<UI_EquipmentSlot>();
statSlot = statSlotParent.GetComponentsInChildren<UI_StatSlot>();

AddStartItem();

// 初始化冷却时间
lastTimeUseWeapon = -100;
lastTimeUseArmor = -100;
lastTimeUseAmulet = -100;
lastTimeUseFlask = -100;

// 绑定技能解锁按钮事件
dashUseAmuletUnlockButton.GetComponent<Button>().onClick.AddListener(UnlockDashUseAmulet);
jumpUseAmuletUnlockButton.GetComponent<Button>().onClick.AddListener(UnlockJumpUseAmulet);
swordUseAmuletUnlockButton.GetComponent<Button>().onClick.AddListener(UnlockSwordUseAmulet);
```

violation of SRP
(original_src/Item/Inventory.cs)

Motivation for Refactoring

3. High Coupling & Low Cohesion

- Cross-module direct references create dependency cycles.
- Classes (e.g. CharacterStats, GameManager) mix unrelated responsibilities.

4. Scalability & Maintainability Limitations

- Adding new skills or enemy types requires modifying multiple core files.
- Hard to perform unit testing due to inline side effects (UI, audio, save).
- No object pooling in key managers, causing performance spikes.

Refactoring Goals & Solutions

- Introduce Service Locator / Facade to centralize service access and reduce coupling.
- Use Factory Method to unify and simplify player/enemy state creation.
- Apply EventBus to decouple UI from gameplay logic.
- Use the Command pattern to separate input handling from skill execution.
- Apply Bridge, Composite, Decorator, and Strategy to modularize equipment, UI, and combat logic.

同济大学
TONGJI UNIVERSITY

3 Refactoring Work & Design Pattern Application

Overview of All Applied Design Patterns

Creational Pattern

- Factory Method

Structural Pattern

- Bridge
- Composite
- Facade

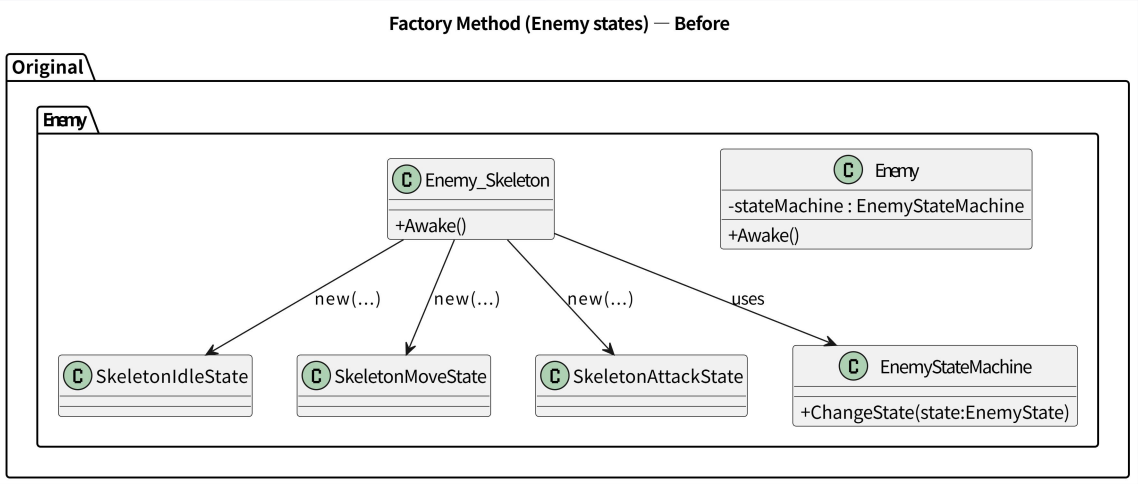
Behavioral Pattern

- Observer
- Command
- Strategy

Additional Pattern

- Service Locator

Creational Pattern: Factory Method



Orignal UML class diagram

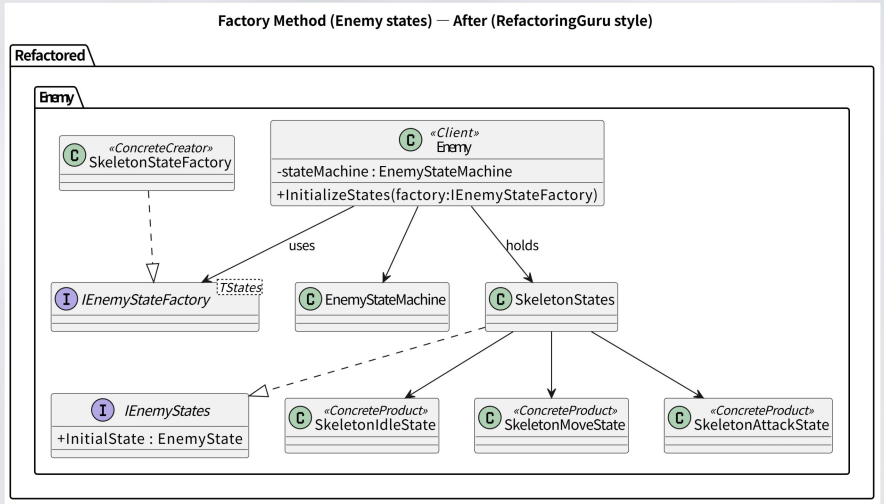
```
public class Enemy_Skeleton : Enemy
{
    private void Awake()
    {
        // Before: directly new each state (duplicated across enemies)
        idleState   = new SkeletonIdleState(this, stateMachine);
        moveState   = new SkeletonMoveState(this, stateMachine);
        attackState = new SkeletonAttackState(this,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

Related source code

Before

- Enemy classes duplicated state instantiation logic
- State construction mixed into Enemy implementation.

Creational Pattern: Factory Method



Updated UML class diagram

```
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. */ }
}
```

- Key refactored code excerpt

After

How it was applied

- Factory creates all states for each enemy type
- Enemy retrieves initial state from factory instead of instantiating directly

Comparison

- Before: Enemy manually constructed states → duplicated & tightly coupled
- After: Enemy only depends on factory → highly extensible

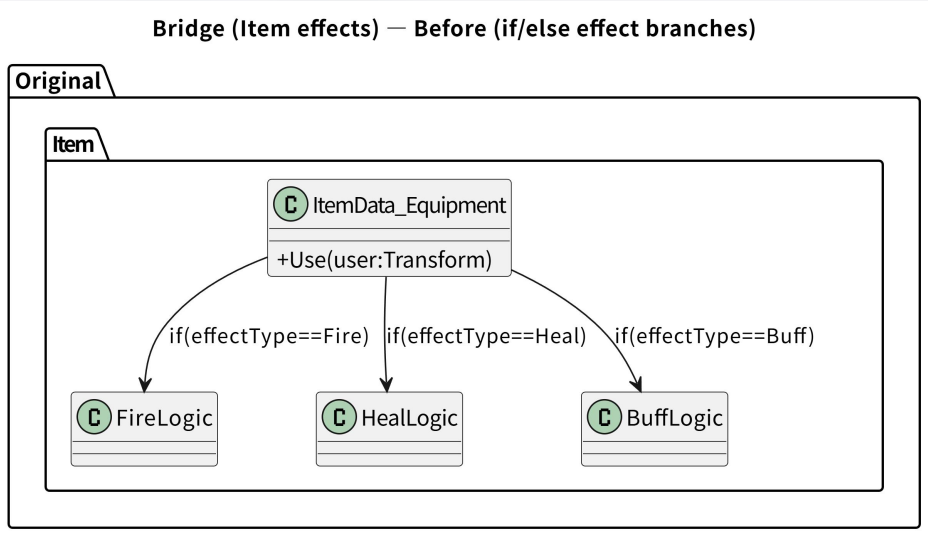
Benefits

- Reduced coupling
- Clearer responsibilities
- Easier testing and adding new enemy types

Structural Pattern: Bridge

Before

- Each equipment-effect pairing could cause class explosion
- Unity ScriptableObject workflow required flexible composition.

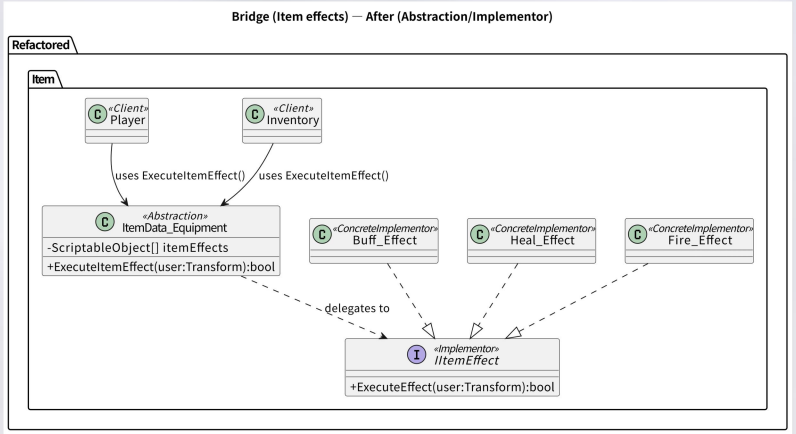


Orignal UML class diagram

```
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
    }
}
```

Related source code

Structural Pattern: Bridge



Updated UML class diagram

```
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;
    }
}
```

Key refactored code excerpt

After

How it was applied

- ItemData_Equipment uses a list of IItemEffect ScriptableObjects to represent item effects.
- Each effect implements IItemEffect and encapsulates its own logic.
- Effects are composed and executed at runtime through ExecuteItemEffect().

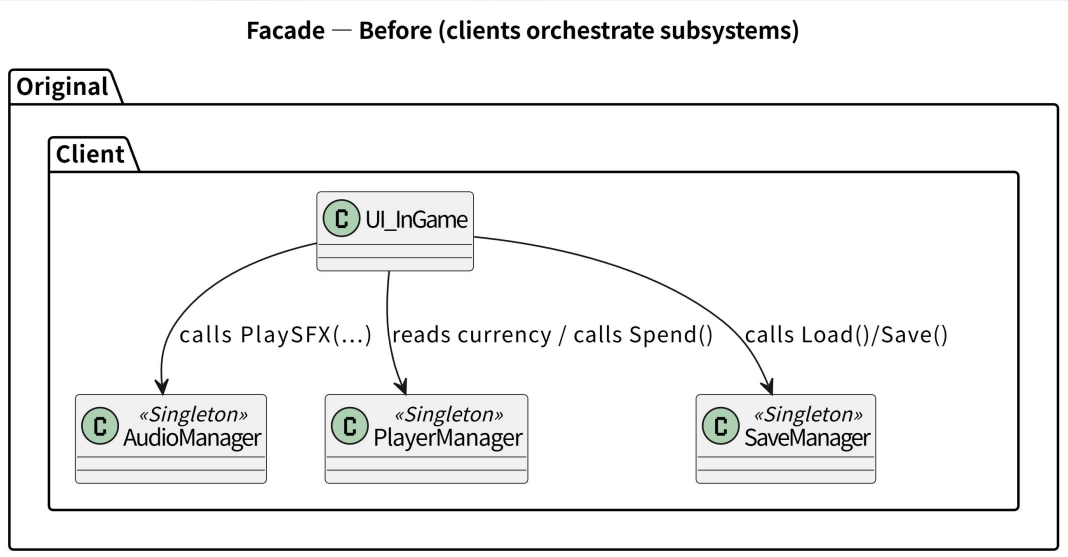
Comparison

- Before: One class per combined effect → class explosion, low reusability.
- After: Small, focused effect classes → reusable, composable, easier to extend.

Benefits

- Clearer modular structure
- High reusability of effect logic
- Better alignment with Unity's asset workflow
- Simplified addition of new effects

Structural Pattern: Facade



Orignal UML class diagram

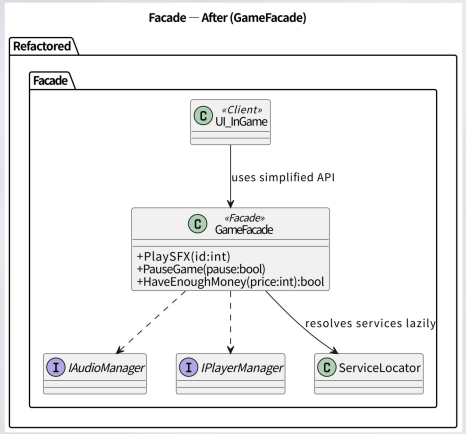
Before

- Repeated multi-service lookups clutter client code
- Callers orchestrate multiple service calls directly.

```
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); }
}
```

Related source code

Structural Pattern: Facade



Updated UML class diagram

```
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; }
}
```

Key refactored code excerpt

After

How It Was Applied

- Introduced GameFacade to provide a small set of unified operations (e.g., PlaySFX, PauseGame, HaveEnoughMoney).
- GameFacade lazily retrieves underlying services through the Service Locator, reducing direct service access across the codebase.

Comparison

- Before: Client code accessed multiple managers directly → scattered, repetitive, tightly coupled.
- After: A single, clean façade handles common operations → centralized orchestration and clearer interactions.

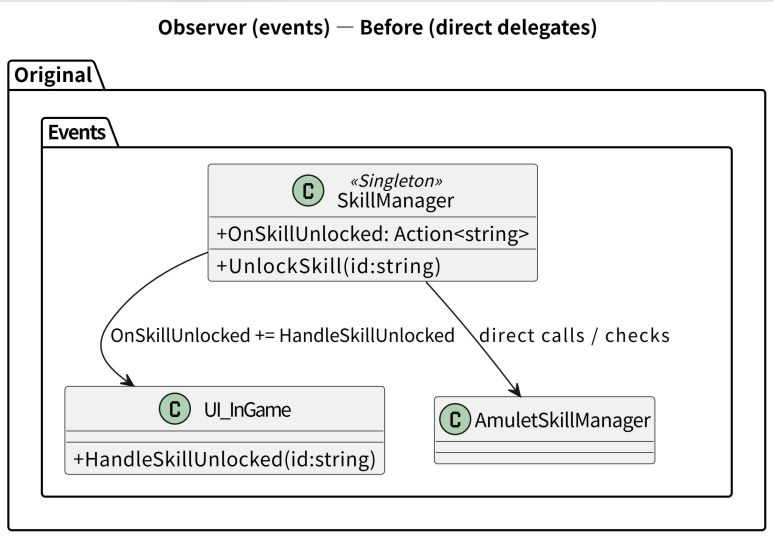
Benefits

- Lower coupling between gameplay code and service managers
- Improved maintainability and readability
- More consistent access patterns across modules

Behavioral Pattern: Observer

Before

- UI and managers tightly coupled via direct calls
- skill unlock and equipment events mixed with UI logic.



Orignal UML class diagram

```csharp
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 */ }
}
```

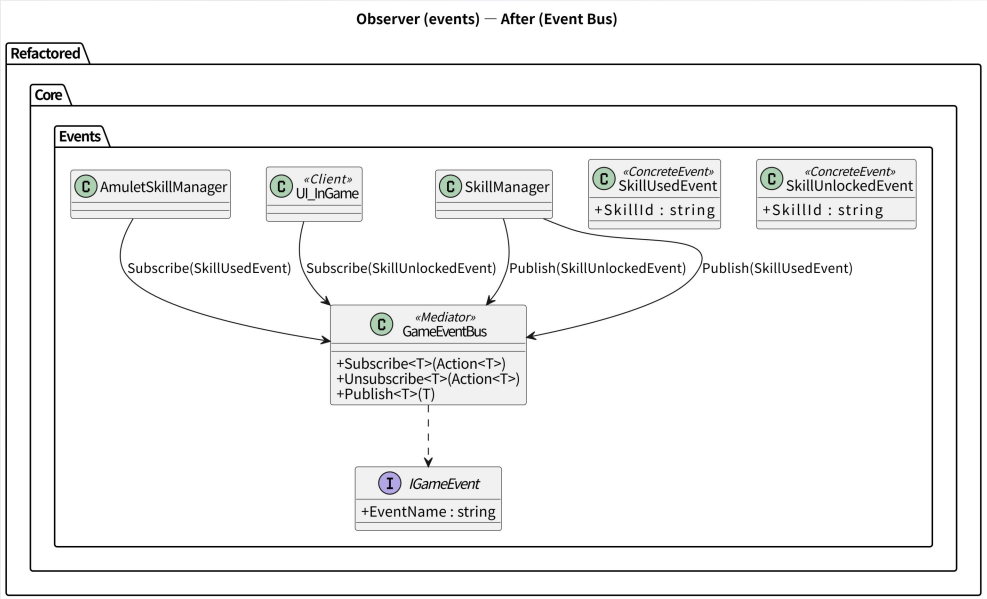
Related source code

Behavioral Pattern: Observer

After

How It Was Applied

- Added a typed GameEventBus with IGameEvent and concrete event types.
- Skills, player states, and equipment logic publish events.
- UI and managers subscribe to relevant events.



Updated UML class diagram

Comparison

- Before: Direct calls and tight coupling between systems.
- After: Decoupled publish/subscribe flow; easier to add new listeners and test with mocks.

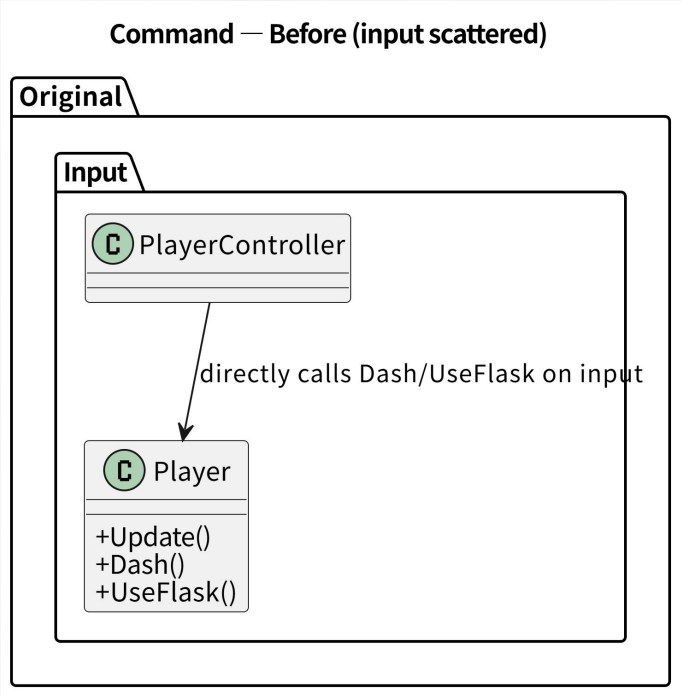
Benefits

- Lower coupling
- Better modularity
- Easier system integration and maintenance

```
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);
    }
}
```

Key refactored code excerpt

Behavioral Pattern: Command



Orignal UML class diagram

Before

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

```
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 */ }
}
```

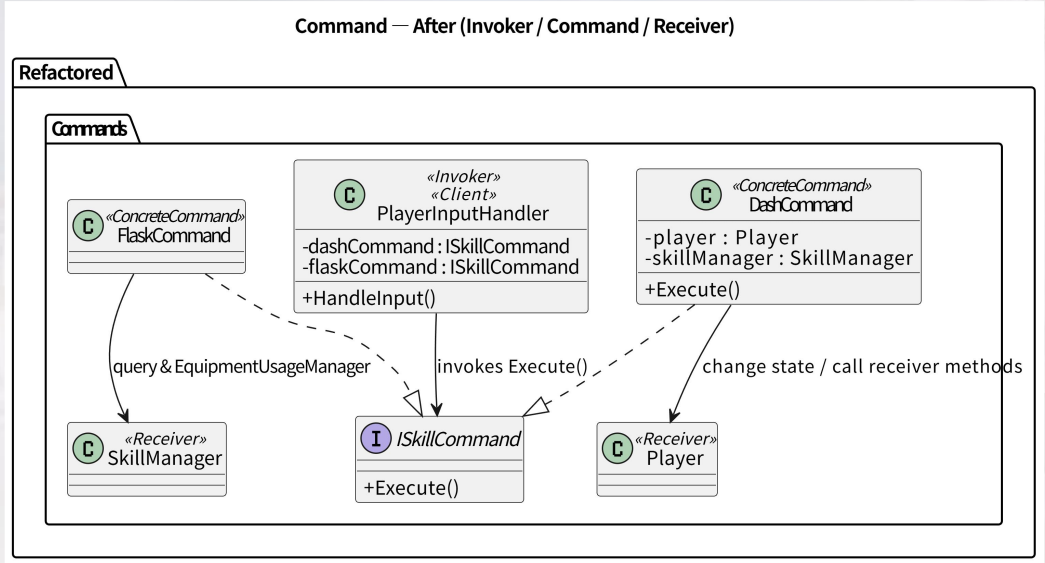
Related source code

Behavioral Pattern: Command

After

How It Was Applied

- Added ISkillCommand and concrete command classes (Dash, Flask, Crystal, Assassinate).
- PlayerInputHandler works as the Invoker, triggering Execute() on assigned commands.
- Execution logic and required managers are encapsulated inside each command.



Updated UML class diagram

Comparison

- Before: Input handling tightly coupled with skill/action logic.
- After: Clean separation—input maps to commands, not direct gameplay code; commands can be reused by AI or replay systems.

Benefits

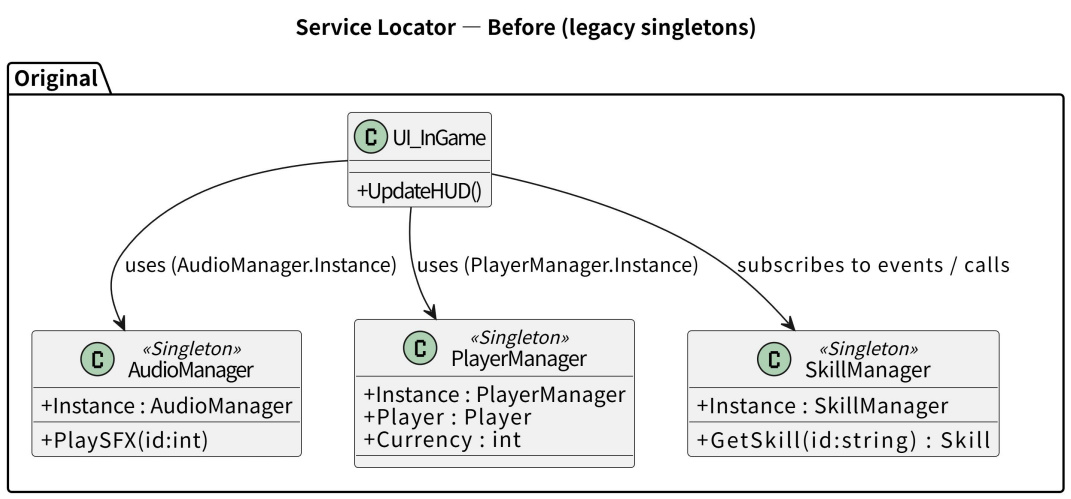
- Increased reusability
- Clear separation of responsibilities

```
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);
    }
}
```

Key refactored code excerpt

Additional Pattern: Service Locator



Orignal UML class diagram

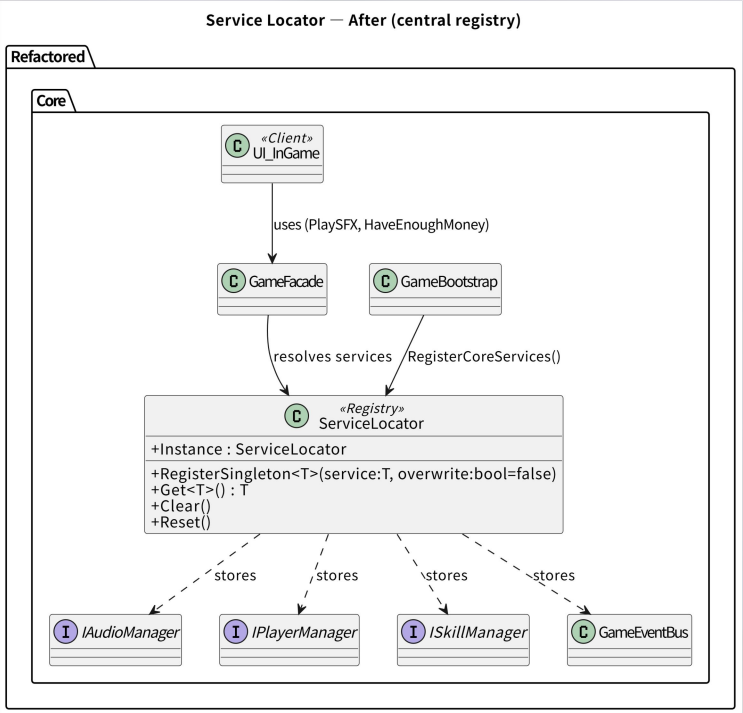
```
// Before: direct global/singleton access (tight coupling)
private void Start()
{
    // direct access to managers before refactor
    var audio = AudioManager.Instance;
    audio.PlaySFX(1);

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

Related source code

Before

- widespread static/global access and implicit dependencies made unit testing and substitution difficult.
- service lifecycle and reset across scenes was inconsistent.

Additional Pattern: Service Locator



Updated UML class diagram

```
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; } }
}
```

Key refactored code excerpt

After

How It Was Applied

- Added a generic, centralized Service Locator for service registration and lookup.
- GameBootstrap handles unified service registration during scene initialization.

Comparison

- Before: Services accessed through scattered singletons and ad-hoc lookups.
- After: Centralized lifecycle control, easier mocking, and flexible service replacement.

Benefits

- Reduced coupling
- Clearer separation of concerns
- Increased flexibility and testability

同济大学
TONGJI UNIVERSITY

Use of AI Tools

4

AI Tools in Refactoring

ChatGPT

Design pattern advice
UML generation
Draft documentation

Cursor

Contextual code refactoring
Batch edits
Code style consistency

GitHub Copilot

Template completion
Rapid prototyping
Alternative Approaches

PlantUML / Diagram AI

Design pattern advice
Class diagram generation
Before/after comparison

- **Identifying Refactoring Opportunities:** God classes, long methods, duplicated logic, tightly coupled modules
- **Design Pattern Recommendations:** Strategy, Factory, Observer, Composite / Decorator
- **UML Diagram Generation:** original diagrams, refactored diagrams, comparison diagrams
- **Code Refactoring:** modularization, consistent renaming, interfaces & abstract classes, method splitting
- **Code Quality Evaluation:** linting, cohesion improvement, complexity detection

AI Tools in Refactoring

Best Practices

- Provide minimal reproducible code + expected outcome
- Specify constraints (language, Unity version, performance, API)
- Small iterative steps → review → next step
- Pair with unit tests for validation

Limitations & Challenges

- AI may produce non-compilable code → compile first, fix manually
- Over-generic or textbook suggestions → provide concrete constraints
- Limited understanding of domain logic → give context & tests

When Not to Rely

- Performance-critical paths (collision, physics loops)
- Core gameplay logic (Boss phases, frame-exact mechanics)
- Proprietary design decisions (gameplay rules)

Lessons Learned

- Rapid identification of refactoring targets
- Accelerated documentation (UML and report drafts)
- More consistent code style
- Faster onboarding of new team members

同济大学 TONGJI UNIVERSITY

THANKS

Group 8

2353120 陈柏熙　2351129 黄景胤

2351299 王雷　2354185 周达　2352609 林琪

2025/12/04