五个原则-SOLID principles

Single-responsibility principle

```
[RequireComponent(typeof(PlayerAudio), typeof(PlayerInput),
typeof(PlayerMovement))]
public class Player : MonoBehaviour
{
     [SerializeField] private PlayerAudio playerAudio;
     [SerializeField] private PlayerInput playerInput;
     [SerializeField] private PlayerMovement playerMovement;


     private void Start()
     {
          playerAudio = GetComponent<PlayerAudio>();
          playerInput = GetComponent<PlayerInput>();
          playerMovement = GetComponent<PlayerMovement>();
     }
}


public class PlayerAudio : MonoBehaviour
{
     …
}


public class PlayerInput : MonoBehaviour
{
     …
}


public class PlayerMovement : MonoBehaviour
{
     …
}
```

Open-closed principle

- open for extension but closed for modification

```
public abstract class Shape
{
     public abstract float CalculateArea();
}
```

```
public class Rectangle : Shape
{
     public float width;
     public float height;

     public override float CalculateArea()
     {
        return width * height;
```

```
    }
}


public class Circle : Shape
{
    public float radius;

    public override float CalculateArea()
    {
       return radius * radius * Mathf.PI;
    }
}
```

```
public class AreaCalculator
{
    public float GetArea(Shape shape)
    {
        return shape.CalculateArea();
    }
}
```

Liskov substitution principle

- 派生类（子类）对象可以在程序中代替其基类（超类）对象
- 子类不应该丢弃父类的特性
- 想要传递函数时，用接口而不是继承(Composition over inheritance)

```
public interface ITurnable
{
    public void TurnRight();
    public void TurnLeft();
}

public interface IMovable
{
    public void GoForward();
    public void Reverse();
}
```

```
public class RoadVehicle : IMovable, ITurnable
{
    public float speed = 100f;
    public float turnSpeed = 5f;

    public virtual void GoForward()
    {
        ...
    }

    public virtual void Reverse()
    {
        ...
    }

    public virtual void TurnLeft()
    {
        ...
    }
```

```
    public virtual void TurnRight()
    {
        ...
    }
}

public class RailVehicle : IMovable
{
    public float speed = 100;

    public virtual void GoForward()
    {
        ...
    }

    public virtual void Reverse()
    {
        ...
    }
}
```

```
public class Car : RoadVehicle
{
  ...
}

public class Train : RailVehicle
{
  ...
}
```

Interface segregation principle

- no client should be forced to depend on methods it does not use
- 避免过大的接口

```
//AVOID
public interface IUnitStats
{
    public float Health { get; set; }
    public int Defense { get; set; }
    public void Die();
    public void TakeDamage();
    public void RestoreHealth();
    public float MoveSpeed { get; set; }
    public float Acceleration { get; set; }
    public void GoForward();
    public void Reverse();
    public void TurnLeft();
    public void TurnRight();
    public int Strength { get; set; }
    public int Dexterity { get; set; }
    public int Endurance { get; set; }
}
```

```
public interface IMovable
{
    public float MoveSpeed { get; set; }
    public float Acceleration { get; set; }
```

```
    public void GoForward();
    public void Reverse();
    public void TurnLeft();
    public void TurnRight();
}

public interface IDamageable
{
    public float Health { get; set; }
    public int Defense { get; set; }
    public void Die();
    public void TakeDamage();
    public void RestoreHealth();
}

public interface IUnitStats
{
    public int Strength { get; set; }
    public int Dexterity { get; set; }
    public int Endurance { get; set; }
}
```

Dependency inversion principle

- high-level modules should not import anything directly from low-level modules
- 高内聚、低耦合
- 下例中：用 `ISwitchable` 接口替代 `Door` 在 `Switch` 中的位置

```
public interface ISwitchable
{
    public bool IsActive { get; }
    public void Activate();
    public void Deactivate();
}
```

```
public class Switch : MonoBehaviour
{
    public ISwitchable client;

    public void Toggle()
    {
        if (client.IsActive)
        {
            client.Deactivate();
        }
        else
        {
            client.Activate();
        }
    }
}
```

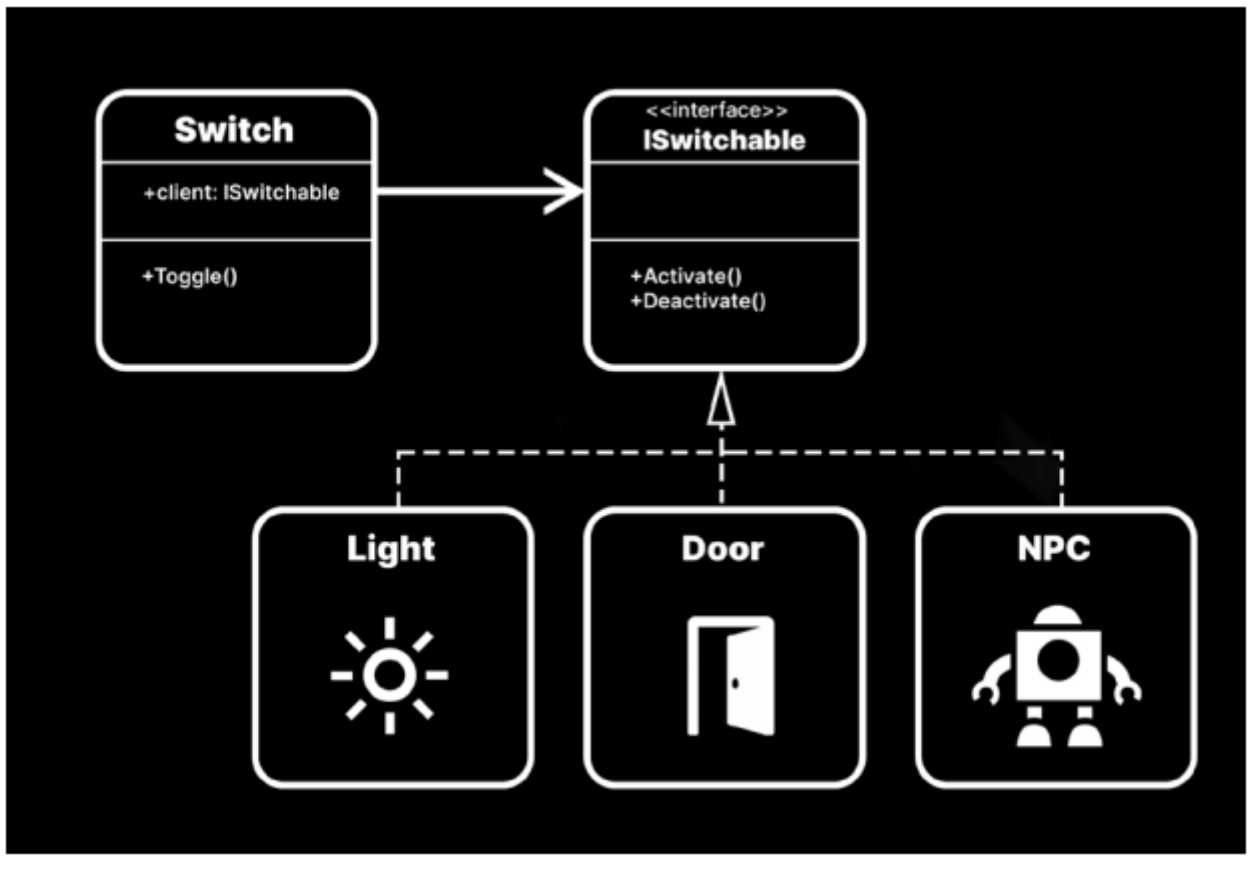
```
public class Door : MonoBehaviour, ISwitchable
{
    private bool isActive;
    public bool IsActive => isActive;

    public void Activate()
```

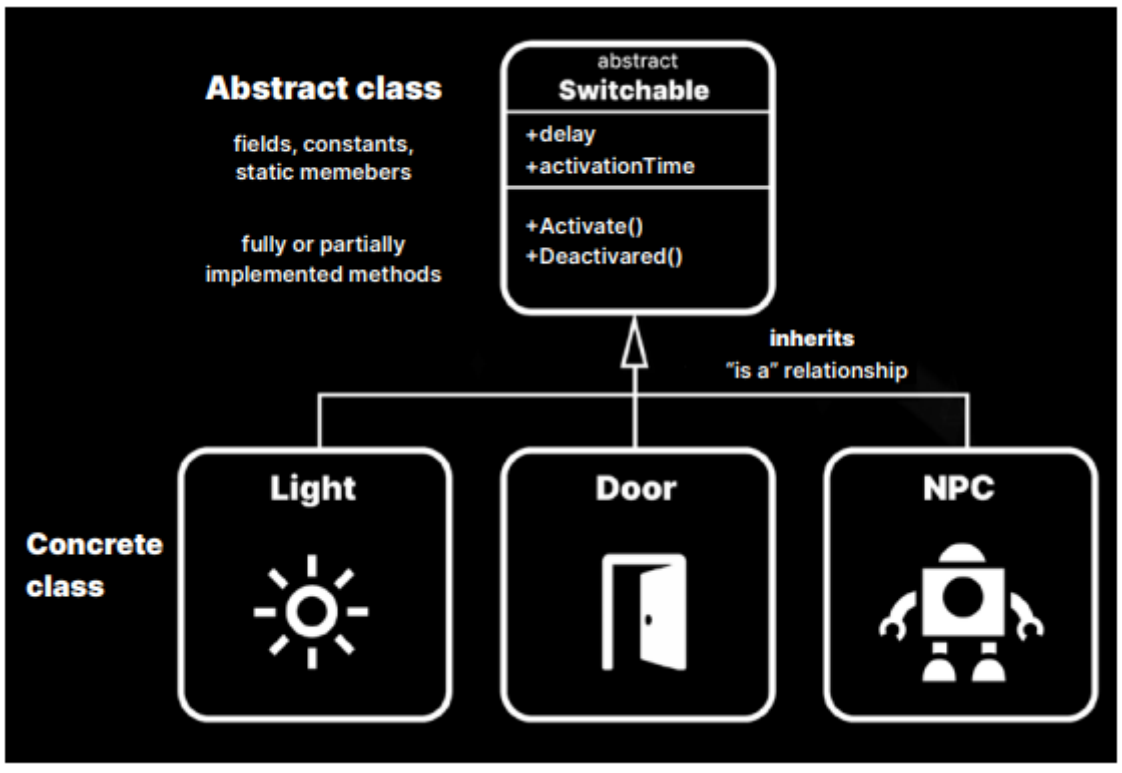
```
    {
        isActive = true;
        Debug.Log("The door is open.");
    }
    public void Deactivate()
    {
        isActive = false;
        Debug.Log("The door is closed.");
    }
}
```

interface vs abstract

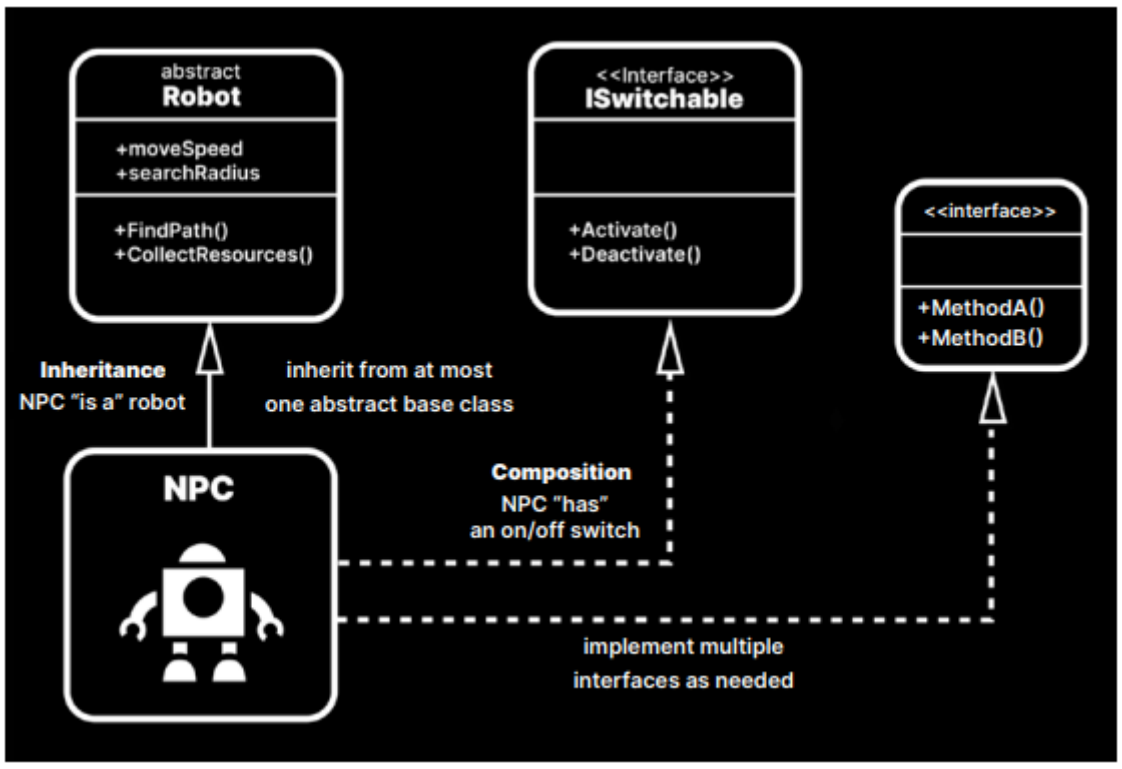
> The advantage of abstract classes is they can have fields and constants as well as static members. They can also apply more restricted access modifiers, like protected and private. Unlike interfaces, abstract classes let you implement logic that enables you to share core functionality between your concrete classes.

- 继承的缺陷：C#不能继承多个基类





- 一般，继承和接口一起使用，继承一个主要基类，并实现多个接口



- 对比

Abstract class	Interface
Fully or partially implements methods	Declares methods but can't implement them
Declares/uses variables and fields	Declares only methods and properties (but not fields)
Has static members	Can't declare/use static members
Uses constructors	Can't use constructors
Uses all access modifiers (protected, private, etc.)	Can't use access modifiers (all members are implicitly public)

Unity内置的模式

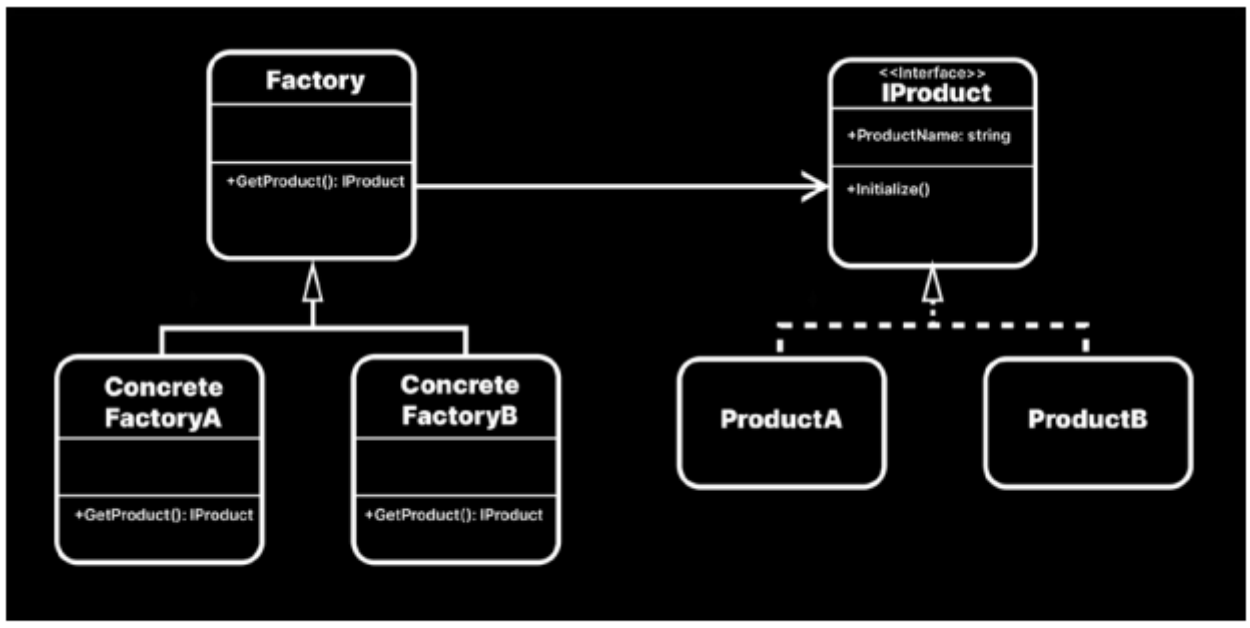
- `GameLoop`
- `Update`
- `Prototype：Unity的Prefab系统`
- `Component`

Factory Pattern

代码示例

- 用一个类创建其他对象

```
public interface IProduct
{
     public string ProductName { get; set; }
     public void Initialize();
}

public abstract class Factory : MonoBehaviour
{
     public abstract IProduct GetProduct(Vector3 position);
     // shared method with all factories
     …
}
```



Using an interface to define shared properties and logic between your products

```
public class ProductA : MonoBehaviour, IProduct
{
     [SerializeField] private string productName = "ProductA";
     public string ProductName { get => productName; set => productName = value ; }
     private ParticleSystem particleSystem;

     public void Initialize()
     {
          // any unique logic to this product
          gameObject.name = productName;
          particleSystem = GetComponentInChildren<ParticleSystem>();
          particleSystem?.Stop();
          particleSystem?.Play();
     }
}

public class ConcreteFactoryA : Factory
{
     [SerializeField] private ProductA productPrefab;

     public override IProduct GetProduct(Vector3 position)
     {
          // create a Prefab instance and get the product component
          GameObject instance = Instantiate(productPrefab.gameObject, position, Quaternion.identity);
          ProductA newProduct = instance.GetComponent<ProductA>();
          // each product contains its own logic
          newProduct.Initialize();
          return newProduct;
     }
}
```

评价

- 利弊
  - 添加多种product时优点明显，每种product内部有自己的初始化逻辑，添加新的product不需要更改以前的代码
  - 缺点是创建许多类和子类

进阶使用（对象池）

- 使用对象池的进阶方法
  - Make it static or a singleton: If you need to generate pooled objects from a variety of sources, consider **making the object pool static**. This makes it accessible anywhere in your application but precludes use of the Inspector. Alternatively, combine the object pool pattern with the singleton pattern to make it globally accessible for ease of use.
  - Use a **dictionary** to manage multiple pools: If you have a number of different Prefabs that you want to pool, store them in separate pools and store a key-value pair so you know which pool to query (the InstanceID of the Prefab can work as the unique key).
  - **Remove unused** GameObjects creatively: Part of utilizing an object pool effectively is hiding unused objects and returning them to the pool. Use every opportunity to deactivate a pooled object (e.g., offscreen, hidden by explosions, etc.)
  - **Avoid releasing** an object that is **already in the pool**.
  - **maximum size/cap**
- Unity自2021版本后内置了对象池
  - Unity - Scripting API: ObjectPool (unity3d.com)
  - 案例：枪发射子弹

```
using UnityEngine;
using UnityEngine.Pool;

namespace DesignPatterns.ObjectPool
{
    public class RevisedGun : MonoBehaviour
    {
        [Tooltip("Prefab to shoot")]
        [SerializeField] private RevisedProjectile projectilePrefab;
        [Tooltip("Projectile force")]
        [SerializeField] private float muzzleVelocity = 700f;
        [Tooltip("End point of gun where shots appear")]
        [SerializeField] private Transform muzzlePosition;
        [Tooltip("Time between shots / smaller = higher rate of fire")]
        [SerializeField] private float cooldownWindow = 0.1f;

        // stack-based ObjectPool available with Unity 2021 and above
        private IObjectPool<RevisedProjectile> objectPool;

        // throw an exception if we try to return an existing item, already in the pool
        [SerializeField] private bool collectionCheck = true;

        // extra options to control the pool capacity and maximum size
        [SerializeField] private int defaultCapacity = 20;
        [SerializeField] private int maxSize = 100;

        private float nextTimeToShoot;

        private void Awake()
        {
            objectPool = new ObjectPool<RevisedProjectile>(CreateProjectile,
                OnGetFromPool, OnReleaseToPool, OnDestroyPooledObject,
                collectionCheck, defaultCapacity, maxSize);
        }
```

```
// invoked when creating an item to populate the object pool
private RevisedProjectile CreateProjectile()
{
    RevisedProjectile projectileInstance = Instantiate(projectilePrefab);
    projectileInstance.ObjectPool = objectPool;
    return projectileInstance;
}

// invoked when returning an item to the object pool
private void OnReleaseToPool(RevisedProjectile pooledObject)
{
    pooledObject.gameObject.SetActive(false);
}

// invoked when retrieving the next item from the object pool
private void OnGetFromPool(RevisedProjectile pooledObject)
{
    pooledObject.gameObject.SetActive(true);
}

// invoked when we exceed the maximum number of pooled items (i.e. destroy the pooled object)
private void OnDestroyPooledObject(RevisedProjectile pooledObject)
{
    Destroy(pooledObject.gameObject);
}

private void FixedUpdate()
{
    // shoot if we have exceeded delay
    if (Input.GetButton("Fire1") && Time.time > nextTimeToShoot && objectPool != null)
    {

        // get a pooled object instead of instantiating
        RevisedProjectile bulletObject = objectPool.Get();

        if (bulletObject == null)
            return;

        // align to gun barrel/muzzle position
        bulletObject.transform.SetPositionAndRotation(muzzlePosition.position,
muzzlePosition.rotation);

                // move projectile forward
                bulletObject.GetComponent<Rigidbody>().AddForce(bulletObject.transform.forward *
muzzleVelocity, ForceMode.Acceleration);

                // turn off after a few seconds
                bulletObject.Deactivate();

                // set cooldown delay
                nextTimeToShoot = Time.time + cooldownWindow;

            }
        }
    }
}
```

```
using System.Collections;
```

```
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Pool;

namespace DesignPatterns.ObjectPool
{
    // projectile revised to use UnityEngine.Pool in Unity 2021
    public class RevisedProjectile : MonoBehaviour
    {
        // deactivate after delay
        [SerializeField] private float timeoutDelay = 3f;

        private IObjectPool<RevisedProjectile> objectPool;

        // public property to give the projectile a reference to its ObjectPool
        public IObjectPool<RevisedProjectile> ObjectPool { set => objectPool = value; }

        public void Deactivate()
        {
            StartCoroutine(DeactivateRoutine(timeoutDelay));
        }

        IEnumerator DeactivateRoutine(float delay)
        {
            yield return new WaitForSeconds(delay);

            // reset the moving Rigidbody
            Rigidbody rBody = GetComponent<Rigidbody>();
            rBody.velocity = new Vector3(0f, 0f, 0f);
            rBody.angularVelocity = new Vector3(0f, 0f, 0f);

            // release the projectile back to the pool
            objectPool.Release(this);
        }
    }
}
```

Singleton Pattern

代码示例

```
using UnityEngine;
public class SimpleSingleton : MonoBehaviour
{
     public static SimpleSingleton Instance;
     private void Awake()
     {
         if (Instance == null)
         {
             Instance = this;
         }
         else
         {
             Destroy(gameObject);
         }
     }
}
```

改进

考虑`场景切换后销毁`，`未添加脚本却直接调用Instance`两个问题

前者通过`DontDestroyOnLoad`，后者通过`Lazy initialization`Lazy initialization - Wikipedia

```
public class Singleton : MonoBehaviour
{
    private static Singleton instance;
    public static Singleton Instance
    {
        get
        {
            if (instance == null)
            {
               SetupInstance();
            }
           return instance;
        }
    }

    private void Awake()
    {
        if (instance == null)
        {
            instance = this;
            DontDestroyOnLoad(this.gameObject);
        }
        else
        {
            Destroy(gameObject);
        }
    }
    private static void SetupInstance()
    {
        instance = FindObjectOfType<Singleton>();
        if (instance == null)
        {
            GameObject gameObj = new GameObject();
            gameObj.name = "Singleton";
            instance = gameObj.AddComponent<Singleton>();
            DontDestroyOnLoad(gameObj);
        }
    }
}
```

改进（泛型编程）

泛型编程版本，可以转换任意类为Singleton

```
public class Singleton<T> : MonoBehaviour where T : Component
{
     private static T instance;
     public static T Instance
     {
         get
         {
             if (instance == null)
             {
                 instance = (T)FindObjectOfType(typeof(T));
```

```
            if (instance == null)
            {
                SetupInstance();
            }
        }
        return instance;
    }
}

public virtual void Awake()
{
    RemoveDuplicates();
}

private static void SetupInstance()
{
    instance = (T)FindObjectOfType(typeof(T));
    if (instance == null)
    {
        GameObject gameObj = new GameObject();
        gameObj.name = typeof(T).Name;
        instance = gameObj.AddComponent<T>();
        DontDestroyOnLoad(gameObj);
    }
}

private void RemoveDuplicates()
{
    if (instance == null)
    {
        instance = this as T;
        DontDestroyOnLoad(gameObject);
    }
    else
    {
       Destroy(gameObject);
    }
}
}
```

```
public class GameManager: Singleton<GameManager>
{
 // ...
}
```

评价

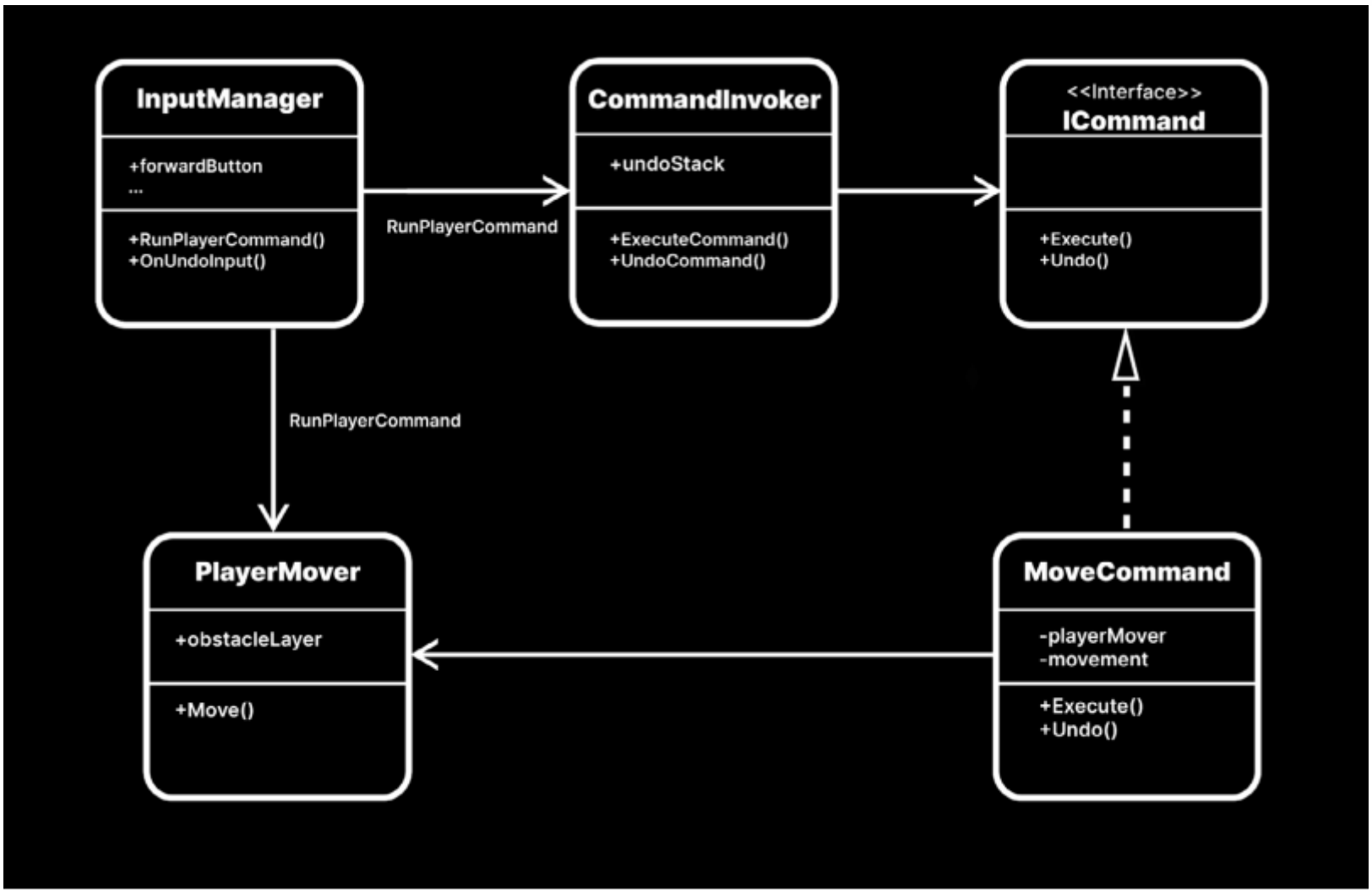
- Singleton可能带来的问题
  - 全局访问Singleton实例，可能会有很多隐藏的依赖关系
  - 单元测试困难
  - 要求紧耦合
- Singleton的优点
  - 全局访问，不必通过Find操作查找组件，访问快
  - 对于小型项目来说直接获取全局状态很方便

Command Pattern

示例

- 策略游戏中常使用，将命令存储到队列或栈中，可以延后执行或撤销这些指令

```
public interface ICommand
{
    void Execute();
    void Undo();
}
```



The CommandInvoker, ICommand, and MoveCommand

- `Input`不会直接控制角色移动，而是通过`CommandInvoker`执行`MoveCommand`

```
private void RunPlayerCommand(PlayerMover playerMover, Vector3 move•ment)
{
    if (playerMover == null)
    {
        return;
    }
    if (playerMover.IsValidMove(movement))
    {
        ICommand command = new MoveCommand(playerMover, movement);
        CommandInvoker.ExecuteCommand(command);
    }
}
```

评价

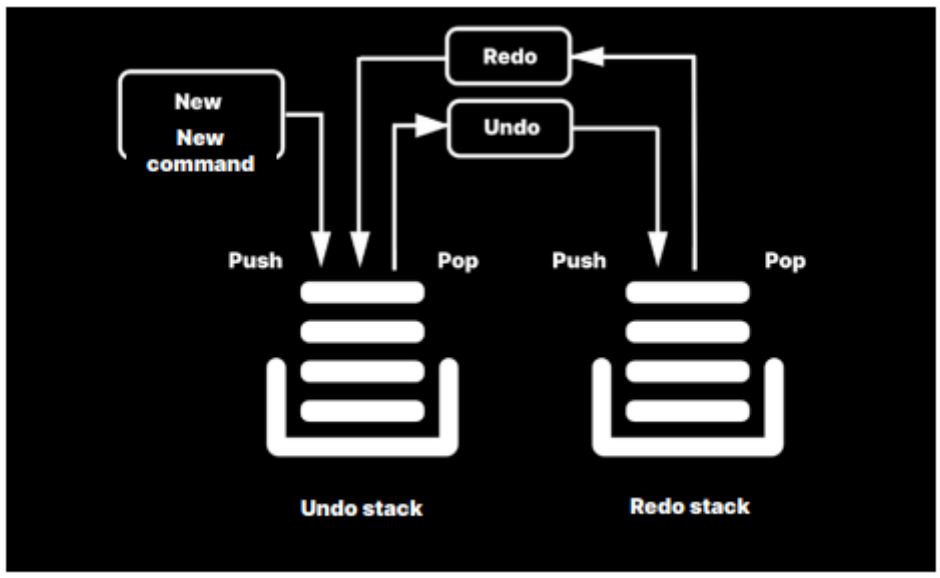
- 对于格斗/策略游戏，Command模式很合适
- 和其他设计模式一样，引入更多的结构（Command）

提高

- 创建更多的Command
- 限制栈的大小
- undo和redo的关系

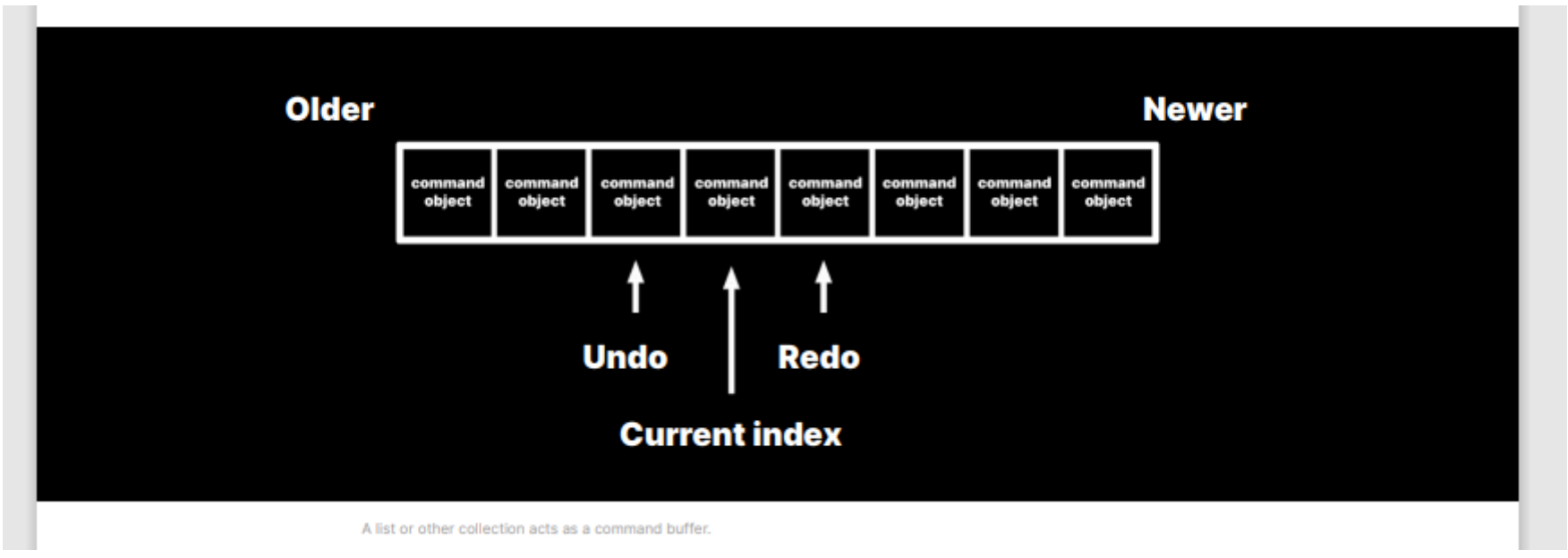
```
public static void ExecuteCommand(ICommand command)
```

```
{
    command.Execute();
    _undoStack.Push(command);

    // clear out the redo stack if we make a new move
    _redoStack.Clear();
}

public static void UndoCommand()
{
    if (_undoStack.Count > 0)
    {
        ICommand activeCommand = _undoStack.Pop();
        _redoStack.Push(activeCommand);
        activeCommand.Undo();   //undo需要逆向逻辑处理
    }
}

public static void RedoCommand()
{
    if (_redoStack.Count > 0)
    {
        ICommand activeCommand = _redoStack.Pop();
        _undoStack.Push(activeCommand);
        activeCommand.Execute();   //redo不需要逆向逻辑处理
    }
}
```



Undo and redo stacks

- 用队列或链表也可以实现，下图为链表示例



A list or other collection acts as a command buffer.

- `CommandInvoker` 不需要知道 `Command` 的具体实现细节

State Pattern

使用示例

- 有限状态机（Finite State Machine）：动画系统中使用
  - 每个状态需实现 `enter` `update` `exit` 三种方法

```
namespace DesignPatterns.State
{
    public interface IState: IColorable
    {
        public void Enter()
        {
            // code that runs when we first enter the state
        }

        public void Update()
        {
            // per-frame logic, include condition to transition to a new state
        }

        public void Exit()
        {
            // code that runs when we exit the state
        }
    }
}
```

- state machine的设计
  - `Update` `Initialize` `TransitionTo`
  - `CurrentState`
  - `event:` stateChanged

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

namespace DesignPatterns.State
{
    // handles
    [Serializable]
    public class StateMachine
    {
        public IState CurrentState { get; private set; }

        // reference to the state objects
        public WalkState walkState;
        public JumpState jumpState;
        public IdleState idleState;

        // event to notify other objects of the state change
        public event Action<IState> stateChanged;

        // pass in necessary parameters into constructor
        public StateMachine(PlayerController player)
        {
```

```
        // create an instance for each state and pass in PlayerController
        this.walkState = new WalkState(player);
        this.jumpState = new JumpState(player);
        this.idleState = new IdleState(player);
    }

    // set the starting state
    public void Initialize(IState state)
    {
        CurrentState = state;
        state.Enter();

        // notify other objects that state has changed
        stateChanged?.Invoke(state);
    }

    // exit this state and enter another
    public void TransitionTo(IState nextState)
    {
        CurrentState.Exit();
        CurrentState = nextState;
        nextState.Enter();

        // notify other objects that state has changed
        stateChanged?.Invoke(nextState);
    }

    // allow the StateMachine to update this state
    public void Update()
    {
        if (CurrentState != null)
        {
            CurrentState.Update();
        }
    }
  }
}
```

- 使用state machine
  - 主要为了侦听状态机发出的stateChanged通知

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

namespace DesignPatterns.State
{
    // a user interface that responds to internal state changes
    [RequireComponent(typeof(PlayerController))]
    public class PlayerStateView : MonoBehaviour
    {
        [SerializeField] private Text labelText;

        private PlayerController player;
        private StateMachine playerStateMachine;

        // mesh to changecolor
        private MeshRenderer meshRenderer;
```

```
    private void Awake()
    {
        player = GetComponent<PlayerController>();
        meshRenderer = GetComponent<MeshRenderer>();

        // cache to save typing
        playerStateMachine = player.PlayerStateMachine;

        // listen for any state changes
        playerStateMachine.stateChanged += OnStateChanged;
    }

    void OnDestroy()
    {
        // unregister the subscription if we destroy the object
        playerStateMachine.stateChanged -= OnStateChanged;
    }

    // change the UI.Text when the state changes
    private void OnStateChanged(IState state)
    {
        if (labelText != null)
        {
            labelText.text = state.GetType().Name;
            labelText.color = state.MeshColor;
        }

        ChangeMeshColor(state);
    }

    // set mesh material to the current state's associated color
    private void ChangeMeshColor(IState state)
    {
        if (meshRenderer == null)
        {
            return;
        }

        meshRenderer.sharedMaterial.color = state.MeshColor;
    }
}
```

评价

- 可以很好的遵循solid原则
- 但也有额外的结构负担

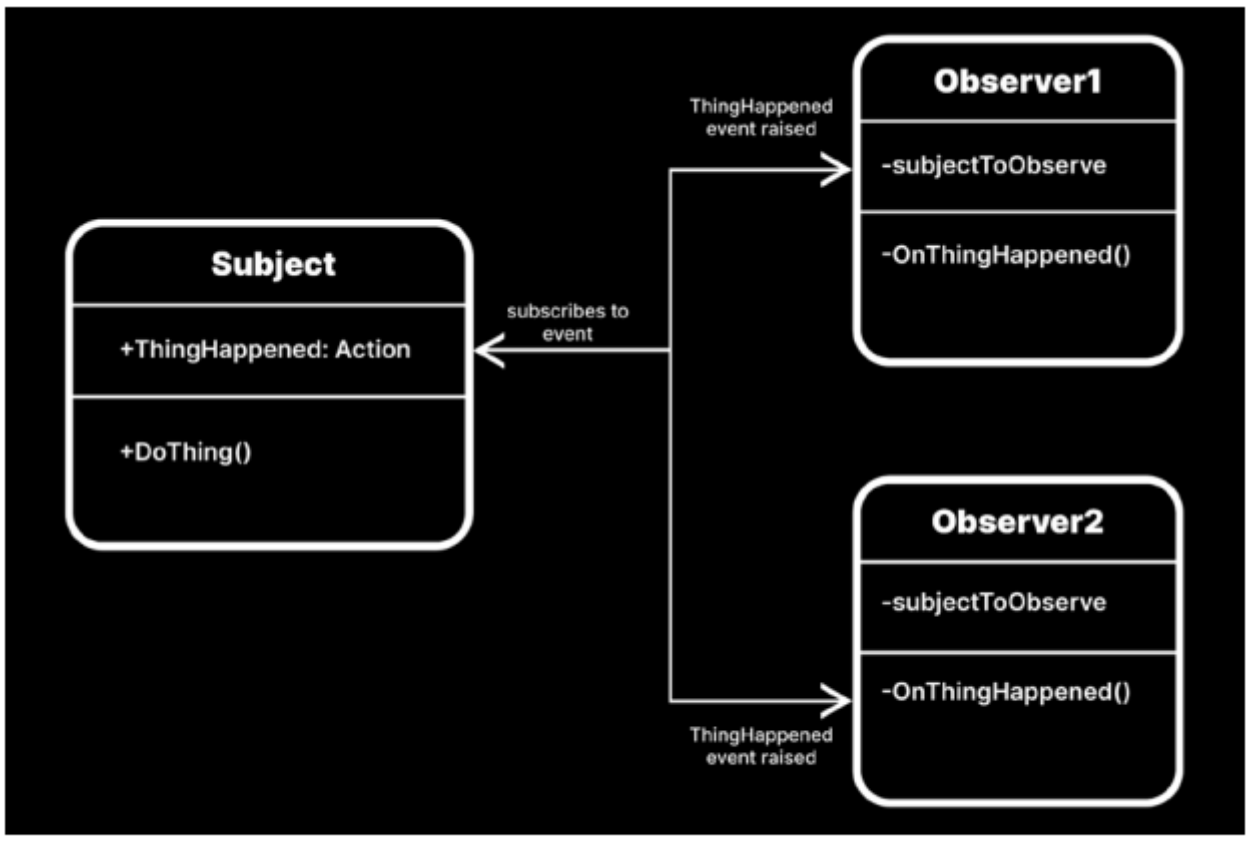
提高

- 用于animator
- 添加event（观察者模式）
- 使用分层的状态继承，比如Walk和Run都是OnGround的子状态
- 用于AI

Observer Pattern

使用示例

- C#可以通过event通知某件事发生了
- event基于委托实现（Delegates - C# Programming Guide | Microsoft Learn）



The subject raises the event to notify the observers.

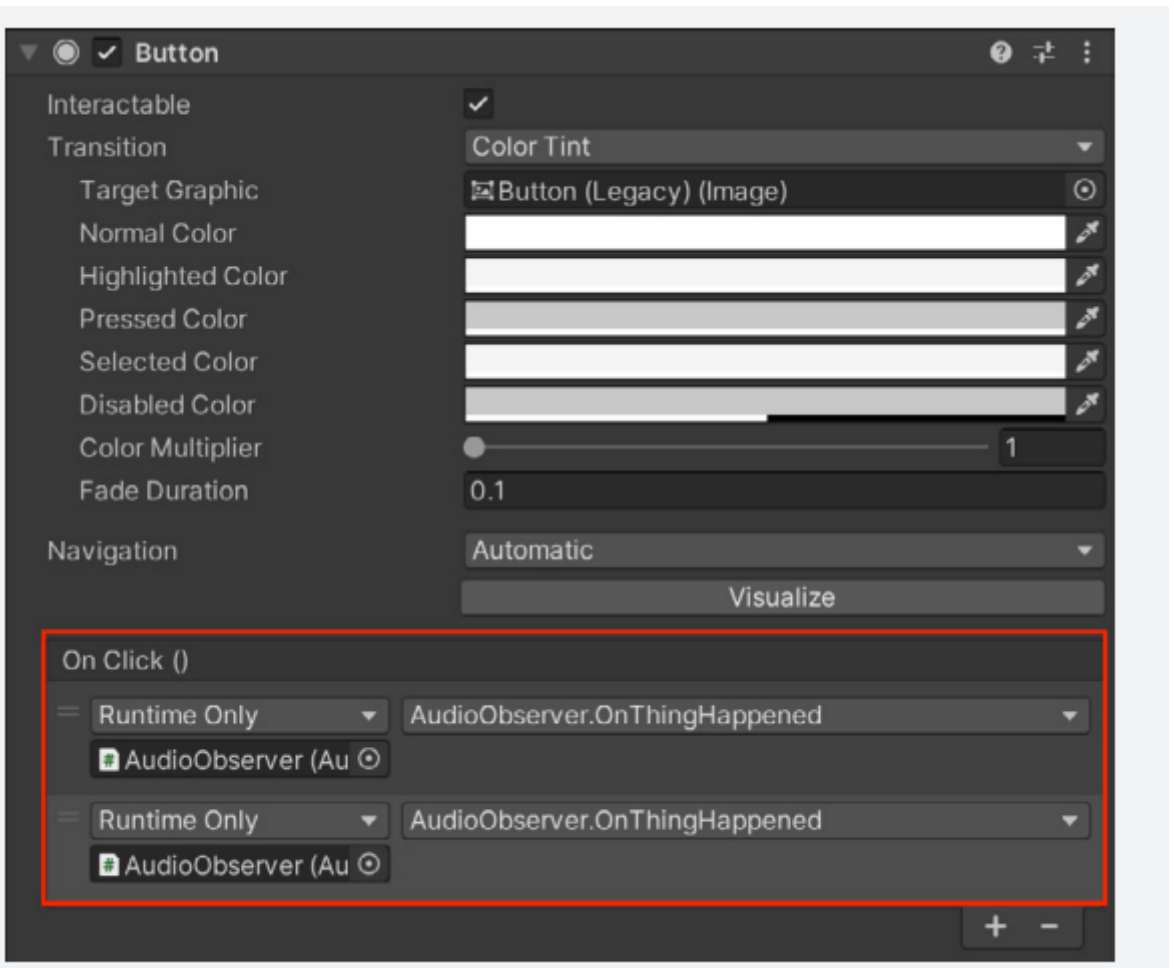
```
namespace DesignPatterns.Observer
{
    public class Subject: MonoBehaviour
    {
        // define an event with your own delegate
        //public delegate void ExampleDelegate();
        //public static event ExampleDelegate ExampleEvent;

        // ... or just use the System.Action, System.Action
        public event Action ThingHappened;

        // invoke the event to broadcast to any listeners/observers
        public void DoThing()
        {
            ThingHappened?.Invoke();
        }
    }
}
```

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

namespace DesignPatterns.Observer
{
    public class ExampleObserver : MonoBehaviour
    {
        // reference to the subject that we are observing/listening to
        [SerializeField] Subject subjectToObserve;

        // event handling method: the function signature must match the subject's event
```

```
    private void OnThingHappened()
    {
        // any logic that responds to event goes here
    }

    private void Awake()
    {
        // subscribe/register to the subject's event
        if (subjectToObserve != null)
        {
            // observer订阅通知：subject.Event += InvokeFunction
            subjectToObserve.ThingHappened += OnThingHappened;
        }
    }

    private void OnDestroy()
    {
        // unsubscribe/unregister if we destroy the object
        if (subjectToObserve != null)
        {
            subjectToObserve.ThingHappened -= OnThingHappened;
        }
    }
}
```

- 一般来说`System.Action`就够用了，但若想要自定义委托，参考资料（Action Delegate (System) | Microsoft Learn）

- 事件的添加场景：
  - 目标达成/任务达成
  - 胜利/失败 判断条件
  - 角色死亡/敌人死亡/造成伤害
  - 添加物品
  - UI界面
- Unity有自己的Event系统和Action系统



UnityEvents have graphical components for your setup

评价

- 很实用，对解耦帮助很大
- 很方便，因为是内置的通过event实现
- 很适合UI，帮助GamePlay代码与UI隔离，UI只需要做个Observer即可
- 需要注意在销毁GameObject时 `unregister`
- observer对Subject的event有一定依赖，可以用EventManager解决
- 对性能有影响

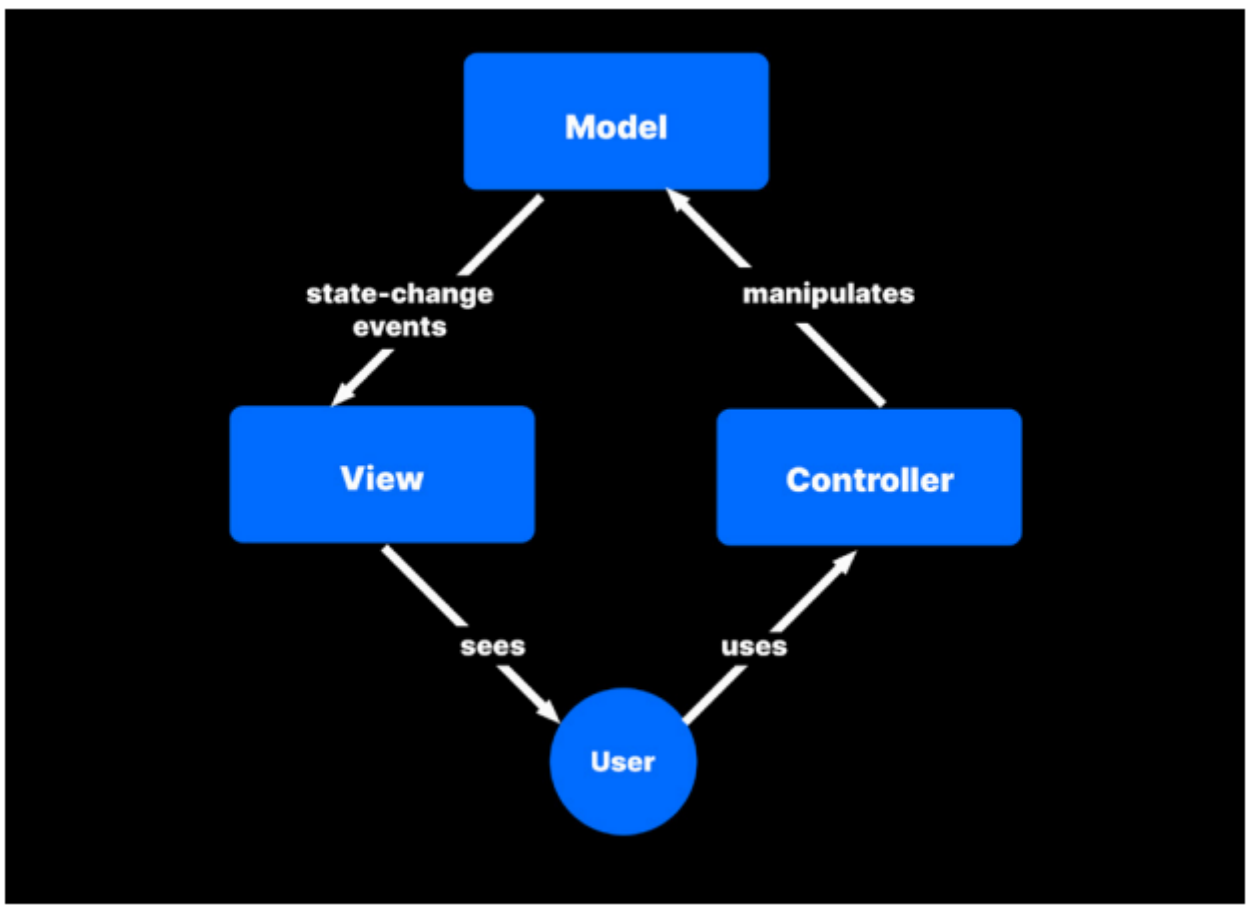
提高

- 使用ObservableCollection Class (System.Collections.ObjectModel) | Microsoft Learn
- 传递唯一的实例ID作为参数（`Action<int>`），借此只让某个observer执行
- `static EventManager`：统一管理，参见教程FPS Microgame - Unity Learn
- `event queue`：配合`Command模式`，使用`command buffer`存储`event`，或者限制最大值并忽略超出界限后的处理函数
- observer模式与mvp模式紧密相连

MVP（Model View Presenter）

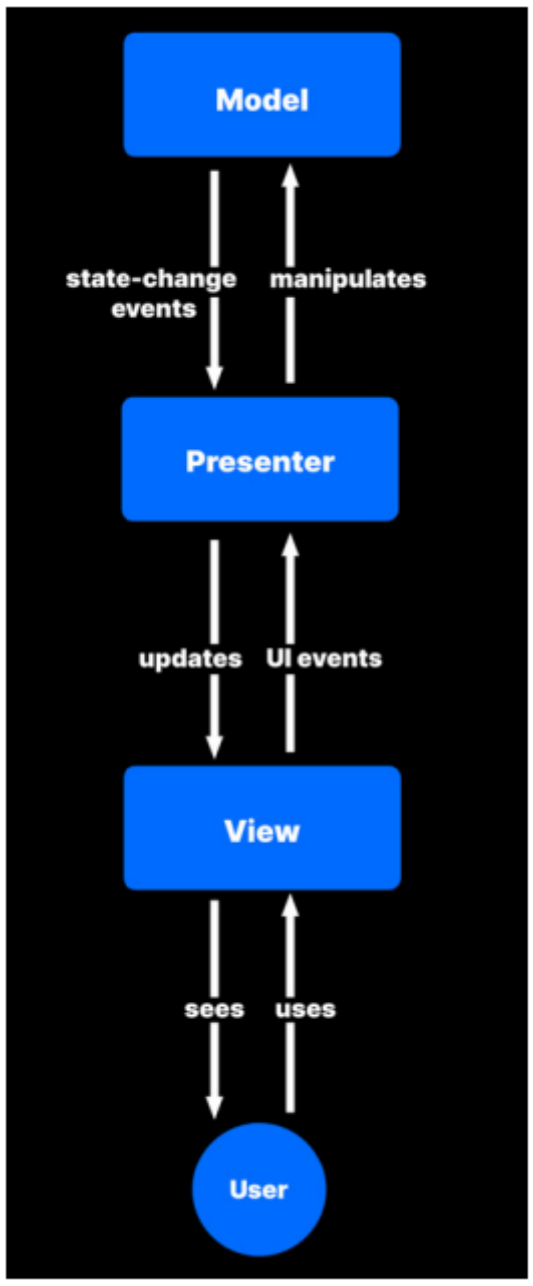
MVC(Model View Controller)

- 思想：分离数据与逻辑
  - model：存储数据
  - view：展示界面
  - controller：逻辑控制



Model View Presenter (MVP) and Unity

- Unity的UI系统就是View
- MVC的变体：MVP
  - 改变了输入处理层（Controller->View）
  - View 通过UI events 传递数据给 Presenter，Presenter 修改 Model.
  - Model中的数据更新通过event传递给presenter,presenter 更新 View



MVP: A variation on MVC

- model示例如下

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System;

namespace DesignPatterns.MVP
{
    // The Model. This contains the data for our MVP pattern.
    public class Health : MonoBehaviour
    {
        // This event notifies the Presenter that the health has changed.
        // This is useful if setting the value (e.g. saving to disk or
        // storing in a database) takes some time.
        public event Action HealthChanged;

        private const int minHealth = 0;
        private const int maxHealth = 100;
        private int currentHealth;

        public int CurrentHealth { get => currentHealth; set => currentHealth = value; }
        public int MinHealth => minHealth;
        public int MaxHealth => maxHealth;

        public void Increment(int amount)
        {
            currentHealth += amount;
            currentHealth = Mathf.Clamp(currentHealth, minHealth, maxHealth);
            UpdateHealth();
        }
```

```
        public void Decrement(int amount)
        {
            currentHealth -= amount;
            currentHealth = Mathf.Clamp(currentHealth, minHealth, maxHealth);
            UpdateHealth();
        }

        // max the health value
        public void Restore()
        {
            currentHealth = maxHealth;
            UpdateHealth();
        }

        // invokes the event
        public void UpdateHealth()
        {
            HealthChanged.Invoke();
        }
    }
}
```

- presenter如下(大多数类不应该自己控制血量):

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

namespace DesignPatterns.MVP
{
    // The Presenter. This listens for View changes in the user interface and the manipulates the Model
(Health)
    // in response. The Presenter updates the View when the Model changes.

    public class HealthPresenter : MonoBehaviour
    {
        [Header("Model")]
        [SerializeField] Health health;

        [Header("View")]
        [SerializeField] Slider healthSlider;
        [SerializeField] Text healthText;

        private void Start()
        {
            if (health != null)
            {
                health.HealthChanged += OnHealthChanged;
            }

            Reset();
        }

        private void OnDestroy()
        {
            if (health != null)
            {
                health.HealthChanged -= OnHealthChanged;
```

```
    }
}

// send damage to the model
public void Damage(int amount)
{
    health?.Decrement(amount);
}

public void Heal(int amount)
{
    health?.Increment(amount);
}

// send reset to the model
public void Reset()
{
    health?.Restore();
}
```

```
public void UpdateView()
{
    if (health == null)
        return;

    // format the data for view
    if (healthSlider ≠null && health.MaxHealth ≠ 0)
    {
        healthSlider.value = (float) health.CurrentHealth / (float)health.MaxHealth;
    }

    if (healthText ≠ null)
    {
        healthText.text = health.CurrentHealth.ToString();
    }
}

// listen for model changes and update the view
public void OnHealthChanged()
{
    UpdateView();
        }
    }
}
```

评价

- 分工明确，适合团队合作
- 方便单元测试
- 可读性高，便于维护
- 小项目优势不明显
- 通过测试决定是否用，如果能加快测试，就用

其他的模式

> **Adapter**: **wrapper** between two unrelated entities so they can work together

> **Flyweight**：共有属性放基类

> **Double buffer**:maintain two sets of array data while your calculations finish. You can then **display one set of data while you process the other,** which is useful for procedural simulations (e.g., cellular automata) or just rendering things to screen.

> **Dirty flag**：在执行费时间操作前，检查是否修改，根据结果决定是否需要执行(e.g., saving to disk or running a physics simulation).

> **Interpreter/Bytecode**: If you want to add modding support or allow non-programmers to extend your game, you can create a simplified language that users can edit in an external text file. The bytecode component can then translate that interpreted language into C# game code.

> **Subclass sandbox**: If you have similar objects with **varying behaviors**, you can define those **behaviors as protected in a parent class**. Then the child classes can mix and match to create new combinations

> **Type object**: **many varieties of a GameObject**, instead of making subclasses for each one, **define all possible behaviors in a single abstract or parent class. Differentiate the special characteristics of individual objects in a separate data file** (such as a **ScriptableObject**) that can be customized without changing the code. For example, this allows you to create an inventory of seemingly different items that all derive from the same class. A game designer can customize the data file to make each item unique (e.g., weapons for an RPG), all without the assistance of a programmer.

> **Data locality**: If you optimize data so that it's stored efficiently in memory, you can reap the rewards of performance. **Replacing classes with structs** can make your data **more cache-friendly**. Unity's **ECS and DOTS** architecture implement this pattern.

> **Spatial partitioning**: With large scenes and game worlds, use special structures to **organize your GameObjects by position**. The Grid, Trie (Quadtree, Octree), and Binary search tree are all techniques to help you divide and search more efficiently.

> **Decorator**: This allows you to add responsibilities to an object without changing its existing structure. A decorator could imbue special abilities or modify a GameObject, e.g., adding perks to a weapon without needing to change the base weapon class

> **Facade**: This provides a simple, unified interface to a more complex system. If you have a GameObject with separate AI, animation, and sound components, you might **add a wrapper class around those components** (imagine a Player controller class managing PlayerInput, PlayerAudio, and so on). This facade hides details of the original components and simplifies usage.

> **Template method**: This pattern **defers the exact steps of an algorithm into a subclass**. For example, you could define a rough skeleton of algorithm or data structure in an abstract class but allow the subclasses to override certain parts without changing the algorithm's overall structure.

> **Strategy**: This behavioral pattern (also called a policy pattern) helps you **define a family of algorithms and encapsulate each one inside a class**. This makes each algorithm (a strategy) **interchangeable at runtime**. For example, if you created a pathfinding system, you could use the strategy pattern to define multiple algorithms (A*, Dijkstra's shortest path, and so on) that you could swap during gameplay, depending on context

> **Composite**: Use this structural design pattern to **organize objects into tree structures** and then treat the resulting structure like you would individual objects. You construct the tree from both simple and composite elements (a leaf and a container). Every element implements the same interface so you can run the same behavior recursively on the entire tree

参考资料

- 设计模式：可复用面向对象软件的基础 - 维基百科，自由的百科全书 (wikipedia.org)
- Game Programming Patterns
- Unity-Technologies/game-programming-patterns-demo: A repo of small demos that assemble some of the well-known design patterns in Unity development to support the ebook "Level up your code with game programming patterns" (github.com)
- Unity - Scripting API: ObjectPool (unity3d.com)

- FPS Microgame - Unity Learn